Improving Flash-based Disk Cache with Lazy Adaptive Replacement

Sai Huang*, Qingsong Wei†✉, Jianxi Chen*†, Cheng Chen†, Dan Feng*✉
*Wuhan National Lab for Optoelectronics
School of Computer, Huazhong University of Science and Technology, Wuhan, China
Email: seth.hg@gmail.com, dfeng@hust.edu.cn
†Data Storage Institute, A*STAR, Singapore
Email: {WEI_Qingsong, CHEN_Jianxi, CHEN_Cheng}@dsi.a-star.edu.sg

***Abstract*—The increasing popularity of flash memory has changed storage systems. Flash-based solid state drive(SSD) is now widely deployed as cache for magnetic hard disk drives(HDD) to speed up data intensive applications. However, existing cache algorithms focus exclusively on performance improvements and ignore the write endurance of SSD. In this paper, we proposed a novel cache management algorithm for flash-based disk cache, named Lazy Adaptive Replacement Cache(LARC). LARC can filter out seldom accessed blocks and prevent them from entering cache. This avoids cache pollution and keeps popular blocks in cache for a longer period of time, leading to higher hit rate. Meanwhile, LARC reduces the amount of cache replacements thus incurs less write traffics to SSD, especially for read dominant workloads. In this way, LARC improves performance and extends SSD lifetime at the same time. LARC is self-tuning and low overhead. It has been extensively evaluated by both trace-driven simulations and a prototype implementation in flashcache. Our experiments show that LARC outperforms state-of-art algorithms and reduces write traffics to SSD by up to 94.5% for read dominant workloads, 11.2-40.8% for write dominant workloads.**

***Index Terms*—Flash; Solid State Drive; Cache Algorithm; Endurance**

I. Introduction

Flash memory has been changing the paradigm of storage system over the past few years. It is now widely used in computer systems, not only in consumer devices, but also in data centers and enterprise-class storage systems. As an alternative for magnetic hard disk dirve(HDD), flash-based solid state drive(SSD) outperforms its counterpart by orders of magnitude for random I/Os. And SSD consumes less power than HDD as well. However, the price of SSD is much higher than HDD. Thus SSDs are often used as a cache tier between DRAM and HDD for cost efficiency [1].

Although SSD shows its attractive worthiness on improving performance, it however could suffer from endurance issue when deployed as disk cache. Flash memory can sustain only a finite number of erase/write cycles. Increased erase operations due to writes shorten the lifetime of a SSD. This inherent defect of SSD is a serious problem and should be taken care of by software.

Existing cache algorithms are primarily designed for buffer cache resident in RAM. They focused exclusively on hit rate improvement to maximize the utilization of cache device. There are a large number of algorithms proposed. The most widely used LRU algorithm exploits temporal locality and keeps the most recently used blocks for furture use. Sophisticated algorithms such as FBR [2], EELRU [3], 2Q [4], LIRS [5], MQ [6] and ARC [7] are proposed to overcome the weakness of LRU and cope with access patterns with weak temporal locality. They improves hit rate by identifying seldom accessed blocks and evicting them earlier from cache. However, direct application of these algorithms is inappropriate for SSD based disk cache because endurance issue is unfortunately ignored.

In this paper, we proposed a novel algorithm for SSD based disk cache, named *Lazy Adaptive Replacement Cache*(LARC). LARC is different from other algorithms because it tries to keep seldom accessed blocks out of cache. LARC uses a ghost cache to identify potentially popular blocks. Cache replacement is triggered only if such a block is identified. As a result, blocks in cache tend to be more popular and can be kept longer to improve hit rate. Meanwhile, LARC reduces the amount of cache replacements and hence incurs less write traffics to SSD, especially for read dominant workloads. LARC is self-tuning. It dynamically controls replacement rate to reduce as much write traffics as possible without decreasing hit rate. Trace-driven simulation shows that LARC outperforms state-of-art cache algorithms for different kinds of workloads. For read dominant workloads, LARC reduces a large portion of write traffics incurred by cache replacements. And it also reduces cache write traffics to some extent for write dominant workloads. Benchmarks on a prototype implementation in Facebook's flashcache [8] further validates the simulation results.

The rest of this paper is organized as follows. In section 2, we discuss the basics of SSD and problems of existing cache algorithms. Section 3 describes the LARC algorithm in detail. Experimental results of trace-driven simulation and benchmarks on a prototype implementation are in section 4 and 5. We discuss related work in section 5 and conclude this work in section 6.

Sai Huang is now an intern at Data Storage Institute, A*STAR, Singapore.

978-1-4799-0218-7/13/$31.00

II. Background and Motivation

A. Flash Memory and SSD

Flash memory is an electronic device which stores information in cells called floating-gate transistor. These cells can be progrommed to different states to represent one of more bits of data. There are two types of flash memory named NOR and NAND respectively. NOR flash supports byte or word level addressability and is faster for read, while NAND flash is cheaper and its storage density is higher.

SSDs use NAND flash memory as storage medium. NAND flash memory can be classified into Single-Level Cell (SLC) and Multi-Level Cell (MLC) flash. A SLC flash memory cell stores only one bit, whereas a MLC flash memory cell can store two bits or even more. NAND Flash memory is organized as blocks. Each block consists of 64 to 256 pages. Each page has a 2KB or 4KB data area and a metadata area (e.g. 128 bytes). Flash memory performs read and write in the unit of page and erase in block unit. Blocks must be erased before they can be re-written. In addition, each block can be erased only a finite number of times. A typical SLC flash memory has around 100,000 erase cycles, while MLC flash memory has around 10,000 erase cycles or even less.

As competing with hard disk, the SSD must retain its performance advantages even with low cost configurations, such as those using MLC flash. Unfortunately, both performance and endurance of flash memory decrease as the storage density increases [9]. The write speed of MLC flash is more than three times slower than that of SLC flash, making writes more expensive for MLC flash. And erase cycle of the MLC flash is ten times less than that of SLC flash. Therefore, the impact of write on MLC is more significant than on SLC. In addition to the writes directly requested by users, SSD's internal garbage collection can produce extra writes [10]. To this end, writes should be minimized through efficient design.

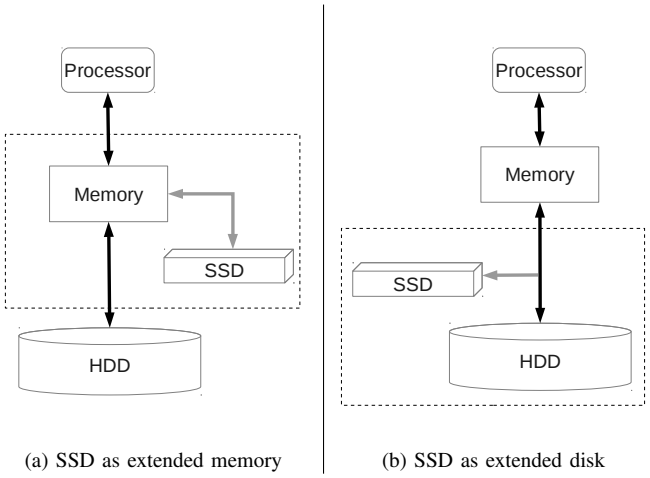
B. SSD Cache Models

There are basically two usage models for SSD based disk cache [11](see Figure 1). In the first model, SSD is used as an extension of system memory. In this model, RAM and SSD are managed as a single unified cache tier for HDD. In the other model, SSD is used as extended disk. SSD lies beneath the standard block interface and serves as a second level cache. It is transparent to other components in the system.

Due to the inclusive property of multi-level cache [12], the extended disk model is often inferior to the other one on aggregate hit rate. However, implementing the extended memory model usually involves modification of the operating system or the application itself [13]. This is expensive or even impossible. Therefore, the extended disk model is more common in production systems. And we focused on it in this research.

C. Problems with Cache Algorithms

Intensive study on cache management led to a large variety of algorithms in past years. To maxmize the utilzation of cache device, a competent algorithm should always keep the most popular blocks in cache. Generally speaking, all existing algrithms are based on two empirical assumptions. The first is temporal locality, i.e. recently used blocks are highly likely to be used again in near future. The other is skewed popularity of blocks, i.e. some blocks are more frequently accessed than others. Accordingly, two classical algorithms are proposed, known as LRU and LFU. LRU is widely used in production systems for its simplicity and $O(1)$ overhead. However, it suffers from poor performance for workloads with weak temporal locality. Here are two examples.



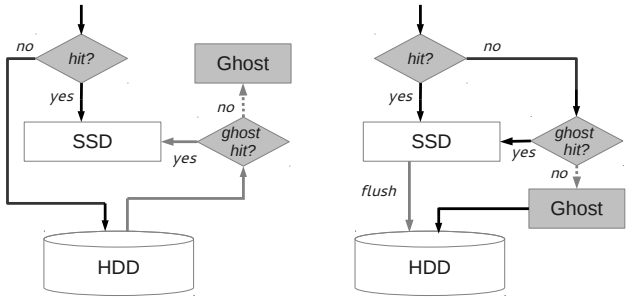
(a) SSD as extended memory (b) SSD as extended disk

Fig. 1. Two usage models of SSD based Disk Cache

- Scan through a large data set. This can easily purge cache space and populate it with one-time accessed blocks. LRU is extremely vulunerable to this. In a shared storage system, when one of the users starts a scan, blocks frequently accessed by other users will be pushed out and response time increases dramatically.
- Loop over a file that is slightly larger than cache size. In this case, the least recently used block will be re-accessed first in future. Nevertheless, LRU always evicts it and hence performs terribly.

LRU is insufficient to cope with these access patterns since it simply ignores the popularity of blocks. Frequently accessed blocks can be mistakenly replaced by seldom used ones. Several sophisticated algorithms have been proposed to solve this problem, such as FBR [2], EELRU [3], 2Q [4], LIRS [5], MQ [6] and ARC [7]. These algorithms can identify seldom accessed blocks and evict them earlier. Thus they improves hit rate by keeping popular blocks in cache for a longer period of time. Take ARC as an example. It divides cached blocks into two groups according their access frequency. One-time accessed blocks are stored in T_1 and other blocks in T_2. A threshold P is used to limit the length of T_1. When the length of T_1 is larger than P, blocks are evicted from T_1. Otherwise, blocks are evicted from T_2. The value of P is dynamically adjusted. As a result, blocks in T_2 roughly have a higher priority thus stay longer in cache.

However, none of these algorithms takes the write endurance of SSD into account. When applied to SSD based disk cache,



(a) Read: on a read miss, the block will not be cached if its identifer is absent from ghost cache.

(b) Write: on a write miss, the block will be written to SSD if its identifier exists in ghost cache. Otherwise, it is written back to HDD directly.

Fig. 2. Data Flow of *LARC*

the vulnerability of LRU not only decreases performance, but also incurs unnecessary write traffics to SSD, shortening its lifetime. As for ARC, evicting one-time accessed blocks from T_1 improves hit rate, but the write endurance issue remains. Since these blocks will not be accessed during their residence in cache, they should not be written to SSD at all. If these blocks can be identified and kept out of cache, we can improve hit rate and reduce SSD write traffics at the same time. This motivated us to push the research forward and design a new SSD friendly cache algorithm.

III. Lazy Adaptive Replacement Cache

We now describe the *Lazy Adaptive Replacement Cache* algorithm. LARC exploits the skewed popularity of blocks. Recently accessed blocks are divided into two groups, one-time accessed blocks and blocks accessed for two or more times. Blocks in the second group are more likely to be popular in future, thus should be cached on SSD. And the first group of blocks are kept out of cache to avoid cache pollution and reduce SSD write traffics.

A. The Basic Algorithm

The key idea of LARC is to identify seldom accessed blocks and keep them out of cache. To achieve this, LARC uses a ghost cache as filter. Ghost cache is an LRU queue(denoted as Q_r) which only stores block identifiers. When cache is full, first-time accessed blocks are stored in ghost cache as candidate. If it is accessed again in the ghost cache later, it is considered to be more popular than others in future. Then LARC moves it to physical cache on SSD. Physical cache is managed with an LRU queue, denoted as Q.

Figure 2 shows the data flow of LARC algorithm. When reading/writing a block B, LARC first lookups for B in Q. If it is found, the request is redirected to the corresponding block on SSD and B is moved to the MRU end of Q. Otherwise, LARC searches Q_r for B to determine whether a cache replacement should be triggered. If B is in Q_r, LARC replaces the LRU block in Q with B and moves B to the MRU end of Q. If B is absent from both Q and Q_r, LARC redirects the request to HDD and inserts B to the head of Q_r. If Q_r exceeds its capacity C_r, the tail block is removed.

TABLE I
An example of the basic LARC algorithm

Access	LARC	LRU
1	Q: {0, 1, 2, 3} Q_r: {}	{0, 1, 2, 3}
5	Q: {1, 0, 2, 3} Q_r: {}	{1, 0, 2, 3}
3	Q: {1, 0, 2, 3} Q_r: {5}	{5, 1, 0, 2}
4	Q: {3, 1, 0, 2} Q_r: {5}	{3, 5, 1, 0}
3	Q: {3, 1, 0, 2} Q_r: {4, 5}	{4, 3, 5, 1}
1	Q: {3, 1, 0, 2} Q_r: {4, 5}	{3, 4, 5, 1}
0	Q: {1, 3, 0, 2} Q_r: {4, 5}	{1, 3, 4, 5}
4	Q: {0, 1, 3, 2} Q_r: {4, 5}	{0, 1, 3, 4}
0	Q: {4, 0, 1, 3} Q_r: {5}	{4, 0, 1, 3}
3	Q: {0, 4, 1, 3} Q_r: {5}	{0, 4, 1, 3}
6	Q: {3, 0, 4, 1} Q_r: {5}	{3, 0, 4, 1}
1	Q: {3, 0, 4, 1} Q_r: {6, 5}	{6, 3, 0, 4}
hits	8	6
replacements	1	6

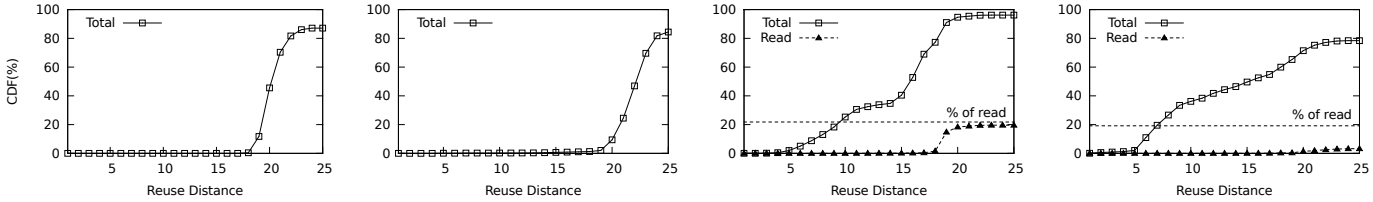
The first column lists an sequence of 12 block accesses. The rest two column shows the content of cache(and ghost cache) at that time. Cache size is 4 and ghost cache size is 2.

We use the example in Table I to explain how LARC works and why it achieves our dual objectives. The example contains 12 block accesses. The size of cache and ghost cache is 4 and 2 respectively. As shown in the first column, block 1 and 3 appeared 3 times in the access sequence. Thus they should always be kept in cache. However, LRU mistakenly replaced block 3 with block 5. By contrast, LARC can keep both block 1 and 3 in cache. Thus LARC achieves higher hit rate. Meanwhile, LARC reduces cache writes by filtering out one-time accessed block 5 and 6.

B. Self Tuning Policy

In the basic algorithm, the capacity of ghost cache C_r remains undetermined. As previously described, cache replacement is only triggered by a hit in ghost cache. Replacement rate in LARC equals to hit rate in ghost cache, thus depends on C_r. C_r indicates how long a block should be kept as candidate. The optimal value of C_r depends on the characteristics of workloads. Specifically, it depends on the minimal reuse distance.

Reuse distance is the number of blocks accessed between two consecutive accesses to a block. The distribution of reuse distance is often used to measure temporal locality of workloads. Figure 3 gives the cumulative distribution of reuse distance of 4 different I/O traces. Information of these traces are in section IV-B. For read dominant traces (a) (b) and read requests in write dominant traces (c) (d), the curve remains at around 0 until reuse distance exceeds a certain threshold, i.e. the minimal reuse distance. As we can see, minimal reuse distance of read requests is relatively large. This is due to the existence of an upper cache level, for instance, page cache of



(a) websearch (b) ads (c) webvm (d) homes

Fig. 3. Cumulative distribution of reuse distance: point (x, y) means that there are $y\%$ of requests whose reuse distance is no more than 2^x.

the operating system. Assuming page cache has a capacity of C_0 blocks and uses LRU algorithm, a block will be kept there for at least C_0 requests. Thus no block will be accessed twice within C_0 requests. As a result, minimal reuse distance of reads depends on C_0. This was also reported in [6]. Our trace analysis further validates this. In addtion, we found this only stands for read requests. There is no obvious minimal reuse distance for write requests. This is because dirty blocks in page cache are flushed back synchronously or periodically to ensure the consistency and reliability of data. Thus reuse distances of writes are relatively small and irrelevant to C_0. This also indicates that writes exhibit stronger temporal locality than reads.

According to this, C_r should be larger than the minimal reuse distance. Otherwise, there will be almost no hit in ghost cache for read dominant workloads and the cache content remains unchanged. However, preliminary results show that a fixed-sized ghost cache does not work well in many cases. Thus we introduced a self-tuning policy to solve this problem. In this policy, C_r is dynamically adjusted according to hit rate: C_r is decreased by $\frac{C}{C-C_r}$ on a cache hit and increased by $\frac{C}{C_r}$ on a cache miss. The increments $\frac{C}{C-C_r}$ and $\frac{C}{C_r}$ is chosen to make LARC more responsive. The smaller C_r is, the faster it increases. The larger C_r is, the faster it decreases.

Hit rate reflects the popularity of cached blocks. If hit rate is high, most popular blocks are already in cache. In this case, decreasing C_r will reduce replacement rate and keep popular blocks in cache for a longer period of time. On the other hand, a low hit rate probably means 1) a lot of popular blocks are out of cache or 2) the workload lacks locality and there are no popular blocks at all. In the first case, increasing C_r leads to higher replacement rate and cached blocks will be removed faster to make room for popular blocks. While in the other case, hit rate in Q_r remains low even if C_r is large. And most blocks are not admitted into cache. We set the upper and lower bound of C_r to 90% and 10% of the cache size. Experiments show that LARC is insensitive to these boudnaries. The pesudo code of LARC is in Algorithm 1.

Algorithm 1: Routine to access block B

```
Initilize: C_r = 0.1 * C
if B is in Q then
    move B to the MRU end of Q
    C_r = max(0.1 * C, C_r - C/(C-C_r))
    redirect the request to the corresponding SSD block
    return
end if
C_r = min(0.9 * C, C_r + C/C_r)
if B is in Q_r then
    remove B from Q_r
    insert B to the MRU end of Q
    if |Q| > C then
        remove the LRU block D in Q
        allocate the SSD block of D to B
    else
        allocate a free SSD block to B
    end if
else
    insert B to the MRU end of Q_r
    if |Q_r| > C_r then
        remove the LRU block of Q_r
    end if
end if
```

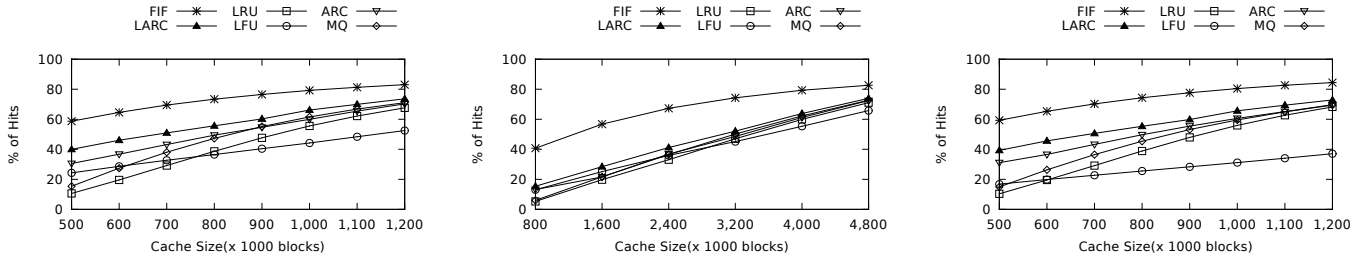
IV. Simulation Experiments

In this section, we evaluate LARC algorithm with trace-driven simulation. We ran several real-life workloads with our cache simulator and collected the results under various cache sizes. Cache size is smaller than the working set of the trace in all configurations.

We chose 4 other algorithms for comparision, including LRU, LFU, MQ and ARC. LRU and LFU are two classical algorithms. MQ is optimized for second level cache, which perfectly matches the model we discussed in this paper. ARC is the most advanced algorithm and have been deployed in several production systems. We also implemented Belady's offline optimal algorithm [14]. This algorithm is also known as Furthest In Future algorithm(FIF), since it always evict the block which will not be used for the longest time in future. It is used as upper bound of hit rate. We evaluate this algorithm to show the potential of hit rate improvements.

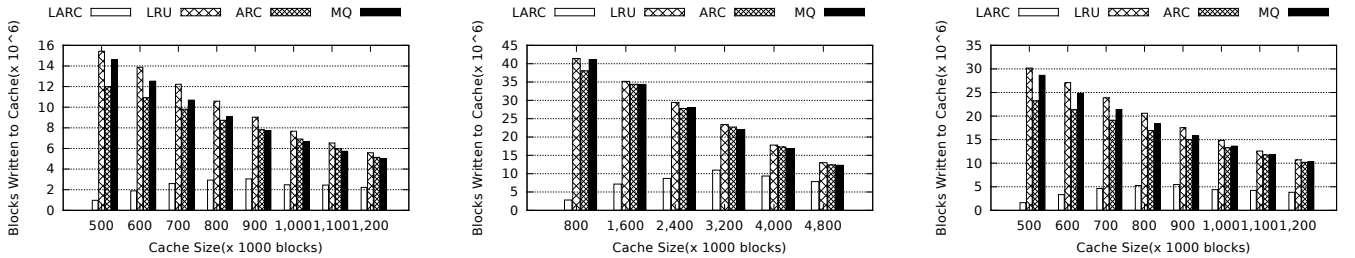
A. The Cachesim Framework

We have developed the cachesim framework to analyze I/O traces and evaluate cache algorithms. Cachesim consists of a preprocessor, a set of query APIs and a cache simulator. The preprocessor converts traces of different format into



(a) *websearch* (b) *ads* (c) *ws_con*

Fig. 4. Hit rate of different algorithms under read-dominate traces



(a) *websearch* (b) *ads* (c) *ws_con*

Fig. 5. SSD write traffics under read dominant traces

cachesim's customized format. I/O requests are normalized so that every single request accesses only one block. Requests spanning accross more than one blocks are broken up into multiple normalized requests with the same timestamp. Block size is selected according to the characteristics of the trace. The preprocessor also extracts statistical information of traces, including number of unique blocks accessed, number of blocks read/written, number of read/write requests, and so on. The query API is used to obtain trace information and facilitate trace analysis. The cache simulator implements the general workflow of replaying I/O operations, calling cache algorithms and collecting simulation results. Cache algorithms are implemented as modules. We've implemented several cache algorithms in this simulator. New algorithms can be easily added using the infrastructures provided by cachesim.

B. Workloads

We use 5 different I/O traces in the simulation. 4 of them are directly collected form real-life systems, denoted as *websearch*, *ads*, *webvm* and *homes* respectively. *Websearch* is available at UMass Trace Repositry [15]. It is collected from a popular web search engine. *Ads* trace is from a Windows server which serves as a cache tier between front-end servers and payload servers in an online advertisement system [16]. *Webvm* and *homes* traces are from a research on I/O deduplication [17]. *Webvm* is from a Linux server running webmail proxy and online course management system for a university department. *Homes* is from an NFS server which stores personal files of scientific researchers, whose daily activities include developing, testing, experiments, technical writing and plotting.

TABLE II
CHARACTERISTICS OF TRACES

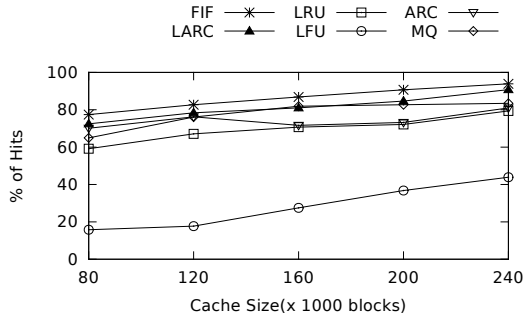
	Blocks($\times 1,000$)			Requests($\times 1,000$)		
	read	write	total	read	write	% of read
websearch	2,223	0.034	2,223	17,253	2	99.99
ads	5,408	129	5,535	14,089	348	97.59
webvm	353	248	549	3,116	11,177	21.80
homes	3,490	1,299	4,569	4,053	17,110	19.15
ws_con	3,622	0.082	3,622	33,660	4	99.99

The fifth trace is synthetic. We append another peice of trace to *websearch*. This piece of trace is collected from the same system but at a different time. Thus the subsets of popular blocks in two traces differ from each other. The resulting trace is denoted as *ws_con*. With this synthetic trace, we investigated how LARC behaves when the popularity of blocks change in a sudden.

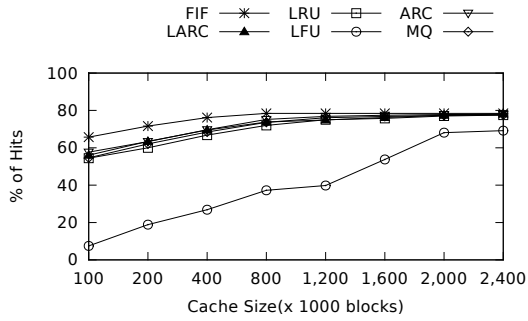
Table II gives detailed information of all 5 traces. Notice that block size of different traces may be different. The block size is 512B in *ads* and 4KB in other traces. Among these traces, *websearch*, *ads* and *ws_con* are read dominant, while *webvm* and *homes* are write dominant.

C. Simulation Results

Figure 4 shows the hit rates of different algorithms for 3 read dominant traces. LARC convincingly outperforms all other algorithms for *websearch*, *ads* and *ws_con*, especially when cache size is relatively small. Compared with LRU, LARC improves hit rate by up to 277.3%, 190.7% and 279.3% for 3 traces respectively. Compared to ARC, the closest competitor,

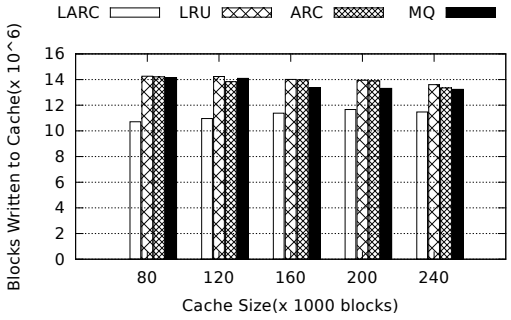


(a) *webvm*

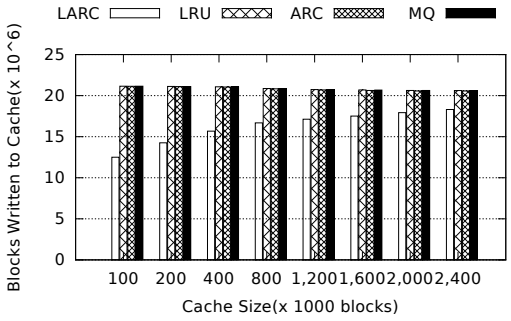


(b) *homes*

Fig. 6. Hit rate of different algorithms under write dominant traces



(a) *webvm*



(b) *homes*

Fig. 7. SSD write traffics under write dominant traces

LARC also improves hit rate by 30.8%, 16.5% and 26.4%. Besides, hit rates of all algorithms are much lower than the upper bound(FIF), indicating that there is still potential for improvements for these workloads.

For *webvm*, LARC also outperfoms all other algorithms significantly. Compared to LRU, LARC improves hit rate by 14-22%. For *homes*, LARC shows only marginal improvement against LRU. Hit rate of these algorithms are quite close to each other, except for LFU.

Hit rate of LFU is much lower than other algorithms in most cases, especially for *webvm* and *homes*. Since the time duration of *webvm* and *homes* is very long(21 days), popularity of blocks changes time to time. This leads to the poor performance of LFU.

As shown in Figure 3, read requests tend to have long reuse distance. LARC, MQ and ARC are capable of capturing these requests by keeping popular blocks longer in cache. Therefore, they improve hit rate significantly for read dominant traces. For *webvm*, read requests exhibits strong locality(most blocks will be reused). They can also capture these reads with long reuse distance and improves hit rate to some extent. On the contrary, read requests in *homes* lack locality. Thus all these algorithms shows very little improvements.

We further broke the synthetic trace *ws_con* into epochs of 100,000 requests and observed how LARC behaves when popularity of blocks changes in a sudden. As shown in Figure 8, hit rate drops dramatically after 17,100,000 requests, where the first piece of two traces ends. Owing to the self-tuning policy, replacement rate increases soon after the drop because the size of ghost cache is increased. This proves that LARC is capable of detecting the change and response to it. Due to its lazy property, the response is hysteretic. Nevertheless, LARC still achieves highest hit rate in the whole simulation.

Another goal of LARC is to extend SSD lifetime. The lifetime of an SSD can be estimated by the write traffics to it. We measure the write traffics by counting number of blocks written to cache. Since there are both read and write requests in the traces, write traffic is calculated as follows. W is the write traffics to cache. H_w is number of write hits. R is number of replacements.

$$W = H_w + R$$

For read dominant traces, the majority of write traffics to SSD are incurred by cache replacements. Since LARC avoids unnecessary replacements, it can reduce a large portion of cache write traffics. Figure 5 gives the cache write traffic of 3 read dominant traces. Compared to LRU, LARC reduces cache write traffic by up to 93.7%, 93.2% and 94.5% for these traces respectively.

For write dominant traces, write hits predominate write traffics. Thus we didn't expect much improvement from LARC. In spite of this, LARC can still reduce cache write traffics by 15.6-24.9% and 11.2-40.8% for *webvm* and *homes* respectively, as shown in Figure 7.

In LARC, cache replacement is only triggered by ghost hit. Therefore, R equals to the number of hits in ghost cache, denoted as H_g. Hence we have $W = H_w + H_g$. When cache size is small, both H_w and H_g increases with cache size.

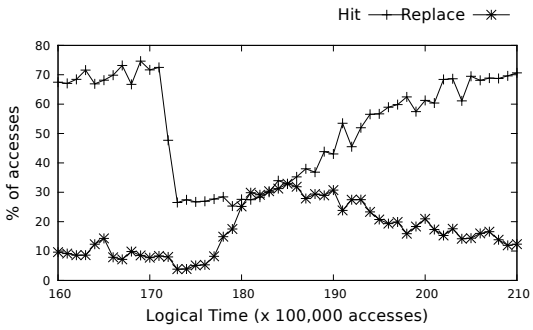


Fig. 8. Hit rate and replacement rate under the synthetic trace *ws_con*. The sudden decline in hit rate indicates that popularity of blocks have changed dramatically. LARC is aware of this change and increases replacement rate as response.

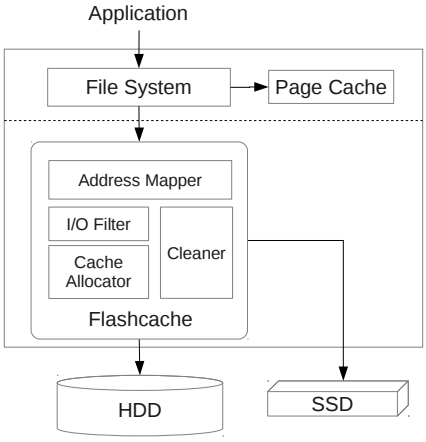


Fig. 9. Architecture of Flashcache

And W increases consequently. When cache size is large and hit rate is high, H_w stops increasing. Meanwhile, the capacity of ghost cache becomes smaller according to the self-tuning policy. Thus H_g becomes smaller and W decreases. In contrast, for other algorithms, $R = 1 - (H_w + H_r)$. Thus $W = 1 - H_r$. Accordingly, W exclusively depends on H_r. As cache size increases, H_r increases and W decreases, as shown in Figure 5 and 7.

V. Implementation and Evaluation

We implemented the LARC algorithm in flashcache [8] to validated our simulation results in a real system.

A. Flashcache Architecture

Flashcache is developed at Facebook to accelerate the storage system in their data centers. It is implemented as a pesudo device driver lying beneath the general block interface. Two replacement algorithms(LRU, FIFO) and 3 write policy(write-back, write-through and write-around) are supported. Flashcache consists of 4 major components. Figure 9 shows the architecture of flashcache.

The address mapper translates the logical address of a block to its location on SSD. Cache space is organized as a set associative hash. It is divided into a number of cache sets, each containing 512(configurable) blocks. A block can be mapped to any SSD block in a certain cache set. When looking for a block, flashcache calculates the cache set number using the hash function and then linearly search for the block within the set. As a result, cache space on SSD are often underutilized.

The I/O filter filters out requests from certain processes to avoid interference in a shared storage context. Administrators can configure a white list or black list to specify which processes should be cached. It is also responsible to detect sequential I/O streams.

Cache allocator tries to allocate a cache block from SSD on every cache miss. After allocated, the cache block is locked until its data is written to SSD. Dirty blocks and locked blocks can never be allocated. Thus the allocation sometimes fails. As a consequence, not all recently accessed blocks are cached, especially when the system is busy. This makes the LRU algorithm in flashcache different from the one we discussed in our simulation and may have significant impact on both performance and SSD write traffics.

The cleaner flushes dirty blocks back to HDD in the background. It is activated when the percentage of dirty blocks exceeds a configurable threshold. Flashcache also tries to clean idle dirty blocks which haven't been read or written for 15 minutes.

TABLE III
Hardware and software configurations of the testbed

CPU	Intel Core 2 Duo E6750 @ 2.66GHz
Memory	2 × 1GB DDR2-667MHz
HDD	Seagate ST373207LW 73GB
SSD	Intel SSDSA2SH064G1GC 64GB
Operating System	Scientific Linux 6.3
Kernel Verison	2.6.32-279.5.1.el6
File System	ext4
Benchmark Tool	Filebench-1.4.9.1

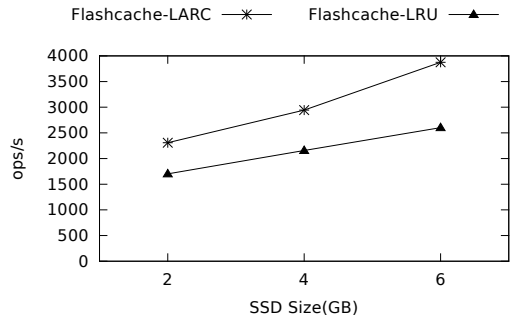
B. Implementation Issues

We implemented LARC in flashcache with about 200 lines of code and modifies the I/O routine to intergrate it. The prototype mantains an array of ghost block descriptors. Each descriptor contains a 64bit logical block number and pointers used for queueing. It takes only a few megabytes of memory to store these descriptors. Ghost cache is organized into multiple sets in the same way as physical cache. Each ghost cache set is assigned to the cache set with the same set number.

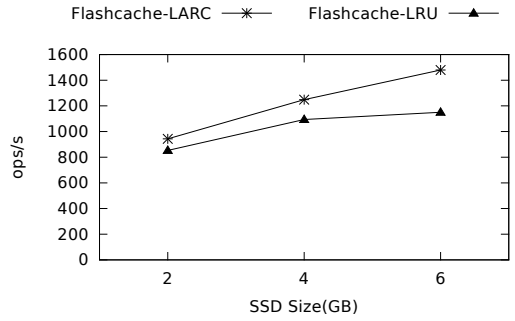
We also modified LARC to handle allocation failures. If LARC decides to cache a block(i.e. a hit in ghost cache) but no SSD blocks are available, the block is then put back to the ghost cache and moved to the LRU end of Q_r.

C. Evaluation Methodology

Detailed information of our testbed is in Table III. We configured a partition of the SSD as cache for the HDD. Cache

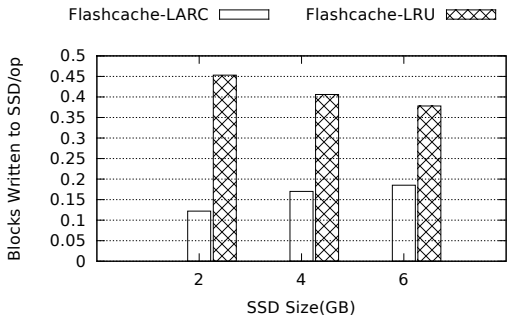


(a) *webserver*

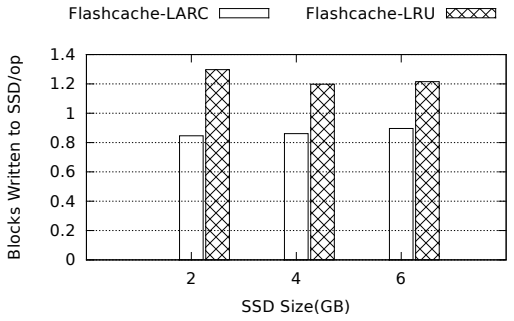


(b) *netsfs*

Fig. 10. IOPS delivered under Filebench's *webserver* and *netsfs* workloads



(a) *webserver*



(b) *netsfs*

Fig. 11. SSD write traffic under Filebench's *webserver* and *netsfs* workloads

size varies from 2GB to 6GB. Apart from those occupied by the operating system and benchmark tool, the amount of memory used as page cache is no more than 1GB in the benchmarks. Thus the second level SSD cache is always larger than the first level page cache. Write-back policy is used in all benchmarks. Our previous experience on flashcache shows this achieves highest performance. Block size of flashcache is 4KB, same as that of the file system.

We use Filebench to generate synthetic workloads for evaluation. Filebench is a flexible benchmark tool for file systems. It can be configured to mimic the behaviour of different kinds of applications. Using the Workload Model Language(WML), Filebench can generate a large variety of workloads. We used two workloads, *webserver* and *netsfs*, in our benchmarks. These workloads are based on predefined configurations in Filebench. We modified them slightly for our benchmark.

The *webserver* workload simulates file access behaviors of a multi-threaded web server. Each of the 100 worker threads performs a sequence of whole file read and log appending. Read-write ratio is about 10:1. We customized Filebench to create a file set of 250,000 files for the benchmark. The average file size is 32KB. The total amount of data is about 8.5GB, including meta-data. The average width and depth of directories are 20 and 4.1 repsectively.

The *netsfs* workload simulates the behaviour of a single threaded network file server. It performs operations including application launch, read-modify-write of files, file appending and stat etc. The ratio of read/write operations is 1:1. However, since page cache can absorb a lot of read requests, the read-write ratio observed at block level is about 1:5. We customized Filebench to create a file set containing 500,000 files for the benchmark. The average file size is about 2.4KB. The total amount of data is about 12GB, including meta-data. The average width and depth of directory is 20 and 3.8 respectively.

During the benchmark run, Filebench uses a random number generator to select a target file for each operation from pre-configured file set. There are 3 types of random number generator in Filebench, uniform generator, gamma generator and table-based generator. By default, the *webserver* workload uses a uniform generator. Thus I/O requests are distributed evenly across all files. To simulate locality of real-life workloads, we configured the *webserver* workload to use a gamma generator, with parameters $\alpha = 0.1$ and $\beta = 1.0$. We also modified Filebench to use a fixed number as seed for the generator so that the bechmark runs are reproducible. In both workloads, event_rate is set to 0(unlimited) to stress the storage system.

Before each benchmark run, we first ran Filebench once on a newly created file system on the HDD. This is to create an initial file set for the benchmark. We then attached the cache partition to the HDD and ran Filebench once again, using previously created files. Thus at the beginning of each benchmark run, the system contains approximately the same file set and an empty cache. This makes sure that all benchmark runs start from the same initial state so their results are comparable. Each benchmark run lasts for 30 minutes to minimize the affect of cache warm-up.

Operations per second is reported as metrics for performance. We also observe how many blocks were written to

SSD during benchmark runs. Since the number of operations completed varies among benchmark runs, we use the ratio of $\frac{blocks\ written}{operations\ completed}$ to compare SSD write traffics of different runs. Each benchmark was run 3 times. The reported result is the average value of three runs.

D. Results

Figure 10 compares the performance of LRU and LARC in flashcache. LARC outperforms LRU for both workloads. It improves throughput by 36-49% and 10-29% for *webserver* and *netsfs* respectively. Moreover, Figure 11 (a) shows that LARC eliminates a large portion(51-73%) of SSD write traffics for read dominant *webserver*. And Figure 11 (b) shows that LARC reduces 26-35% of SSD write traffics for write dominant *netsfs*. These experiments validated our simulation results.

The benchmark result is slightly different from simulation. The reason for this difference is twofold. First, we used a closed system model [18] in the benchmarks. In this model, a new request is issued right after previous one is completed. When hit rate is high, average response time is low and more requests are issued. Thus the workload becomes more intense. While trace-driven simulation uses an open system model. Arrivals of requests are independent to each other and workload intensity remains constant. Another reason is the LRU algorithm used in flashcache. As described in section V-A, cache block allocation can fail. In this case, the request will be uncached. This algorithm is different from the one we disscussed in simulation. As a consequence, replacement rate is usually lower than cache miss rate. This has significant impact on both hit rate and SSD write traffics.

VI. Related Work

With the increasing popularity of flash memory, both engineers and researhers have showed great interest in integreting them into storage stack. Hard disk manufacturers have been shipping hybrid disks for years [19]. Software implementations such as ReadyBoost [20] and Bcache [21] are also available for mainstream operating systems. Facebook uses SSDs as disk cache in their data centers [8]. Solaris ZFS has built-in support to use SSDs as a second level cache [22]. NetApp also uses SSDs to speed up their storage appliance [23].

Several previous studies focused on improving flash-based disk cache. Kgil et. al. [24] proposed to use flash as extended memory and disk cache to reduce power consumption. Flash memory is splitted into separated read/write regions to improve performance. They employ a programmable flash controller which can change the ECC stength and cell density of flash chips. With this controller, they show that flash lifetime can be extended with less than 5% of performance degradation.

Mesnier et al. [25] enhanced SSD based disk cache by leveraging semantic informantion from file system or DBMS. They designed an algorithm which selectively allocates cache space for blocks according to their priority. By assigning higher priority to meta-data blocks and small files, it achieves better performance than semanticially blind cache algorithms.

Pritchett et al. [26] uses SSD as ensemble-level cache for multiple servers to maximize its utilization. This achieves higher hit rate with less SSD drives and thus is more cost efficient than per-server caching. They also introduced a seive mechnism which only allocates cache space for blocks on their n^{th} access. It reduces allocation writes to SSD and thus extends it lifetime.

A recent research [27] exploits the internal garbage collection behaviour to improve SSD based disk cache. Garbage collection operations have great impact on SSD performance and lifetime. Hence they should be taken care of by the cache algorithm. The researchers divided SSD into 3 regions: unused, read and write. They designed an algorithms to dynamically tuning the size of these regions. This increases hit rate and decreses erase operations at the same time.

SSD is also used as extended buffer pool for DBMS. [28] proposed a temperature-based replacement policy(TAC). TAC maintains the temperature(access frequency) of disk extents(32 pages) and keeps blocks in warm region in SSD. If SSD is full of blocks from warm regions, cold blocks are denied from cache. [29] implemented the exclusive model in Microsoft SQL Server. They evaluated three different policies to handle dirty pages evicted from memory buffer pool.

In addition to caching, storage tiering [30] [31] [32] is another sought-after technology for hybrid storage. A tiered storage system dynamically selects a popular subset of data and moves them onto faster but smaller devices. This technology is widely used in enterprise storage arrays to meet the demand for both large capacity and high performance with low cost. Recently, Apple introduced storage tiering to consumer products with Fusion Drive [33].

Both cache and tiered storage belong to hierarchical storage. As an alertnative, a horizontal hybrid storage system uses SSD to store a specific part of data to make better use of both SSDs and HDDs. The idea behind this architecutre is that access pattern of a block depends on the data stored in it. For example, meta-data and small files are usually accessed with small and random I/Os and are likely to be accessed more frequently [34]. They account for only a small portion of data in a file system volume. Therefore, constantly storing them on SSD can boost overall performance without moving blocks to and fro. Examples of this architecture include Chunkstash [35] and I-CASH [36].

VII. Conclusion

In this paper, we proposed a novel cache algorithm LARC for flash-based disk cache. It adopts performance improvements and lifetime as dual objectives. LARC filters out seldom accessed blocks and keep them out of cache, thus reduces write traffic to SSD. Meanwhile, this keeps popular blocks longer in cache and achieves higher hit rate.

LARC has been evaluated by both trace-driven simulation and a prototype implementation in Facebook's flashcache. Simulation results show that LARC convincingly outperforms state-of-art algorithms such as MQ and ARC for read dominant workloads. And it reduces SSD write traffics by up to 94.5%

as well. On the other hand, for write dominant workloads, LARC also achieves higher hit rate than other algorithms, while reducing 11.2-40.8% of SSD write traffics. These results suggest that LARC is better to be used for read dominant applications or for read only disk cache. Benchmarks with synthetic workloads in a prototype implementation validate simulation results.

LARC is self-tuning and adapts to different kinds of workloads. It is easy to implement and incurs very little overhead. Although it is designed for flash-based disk cache, it is also applicable in other contexts such as local disk cache for a networked storage system.

ACKNOWLEDGMENT

This work was supported by Agency for Science, Technology and Research(A*STAR), Singapore under Grant No. 112-172-0010, the National Basic Research Program of China(973 Program) under Grant 2011CB302301, National Natural Science Foundation of China (NSFC) under Grant 61025008, 61232004, 61173043, The National Key Technology R&D Program No.2011BAH04B02, Natural Science Foundation of Hubei Province. No.2011CDB036.

REFERENCES

[1] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron, "Migrating server storage to SSDs: analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 145–158.

[2] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," *ACM SIGMETRICS Performance Evaluation Review*, vol. 18, no. 1, pp. 134–142, 1990.

[3] Y. Smaragdakis, S. Kaplan, and P. Wilson, "Eelru: simple and effective adaptive page replacement," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1. ACM, 1999, pp. 122–133.

[4] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 1994, pp. 439–450.

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

[6] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, 2001, pp. 91–104.

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

[8] Flashcache. [Online]. Available: https://github.com/facebook/flashcache

[9] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of NAND flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012, pp. 2–2.

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70.

[11] G. Graefe, "The five-minute rule twenty years later, and how flash memory changes the rules," in *Proceedings of the 3rd international workshop on Data management on new hardware*. ACM, 2007, p. 6.

[12] T. Wong and J. Wilkes, "My cache or yours? Making storage more exclusive," in *In Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[13] B. Gill, "On multi-level exclusive caching: offline optimality and why promotions are better than demotions," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. USENIX Association, 2008, p. 4.

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

[15] UMass Trace Repository. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[16] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 119–128.

[17] 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.

[18] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *Proceedings of the 3rd conference on Networked Systems Design & Implementation*, vol. 3, 2006, pp. 18–18.

[19] Seagate Momentus XT Solid State Hybrid Drives. [Online]. Available: http://www.seagate.com/internal-hard-drives/laptop-hard-drives/momentus-xt-hybrid/

[20] Understand ReadyBoost and whether it will Speed Up your System. [Online]. Available: http://technet.microsoft.com/en-us/magazine/ff356869.aspx

[21] bcache. [Online]. Available: http://bcache.evilpiepirate.org/

[22] A. Leventhal, "Flash storage memory," *Commun. ACM*, vol. 51, no. 7, pp. 47–51, Jul. 2008.

[23] M. Peters, "Netapps Solid State Hierarchy," *ESG White Paper*, 2009.

[24] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash Based Disk Caches," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 327–338.

[25] M. Mesnier, F. Chen, T. Luo, and J. B. Akers, "Differentiated Storage Services," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 57–70.

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

[27] Y. Oh, J. Choi, D. Lee, and S. Noh, "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems," in *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST12)*, 2012, pp. 25–25.

[28] M. Canim, G. Mihaila, B. Bhattacharjee, K. Ross, and C. Lang, "SSD bufferpool extensions for database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[29] J. Do, D. Zhang, J. Patel, D. DeWitt, J. Naughton, and A. Halverson, "Turbocharging DBMS buffer pool using SSDs," in *Proceedings of the 2011 international conference on Management of data*. ACM, 2011, pp. 1113–1124.

[30] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proceedings of the 9th USENIX conference on File and stroage technologies*. USENIX Association, 2011, pp. 20–20.

[31] 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*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 22–32.

[32] R. Appuswamy, D. van Moolenbroek, and A. Tanenbaum, "Integrating flash-based SSDs into the storage stack," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–12.

[33] Fusion Drive: An overview. [Online]. Available: http://www.macworld.com/article/2013805/fusion-drive-an-overview.html

[34] A. Wang, P. Reiher, G. Popek, and G. Kuenning, "Conquest: Better performance through a disk/persistent-RAM hybrid file system," in *Proceedings of the 2002 USENIX Annual technical Conference*, 2002.

[35] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: speeding up inline storage deduplication using flash memory," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIX Association, 2010, pp. 16–16.

[36] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of SSD and HDD," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 278–289.