WEC: Improving Durability of SSD Cache Drives by Caching Write-Efficient Data

Yunpeng Chai, Zhihui Du, *Member, IEEE*, Xiao Qin, *Senior Member, IEEE*, and David A. Bader, *Fellow, IEEE*

Abstract—Serving as cache disks, flash-based solid-state drives (SSDs) can significantly boost the performance of read-intensive applications. However, frequent data updating, the necessary condition for classical replacement algorithms (e.g., LRU, MQ, LIRS, and ARC) to achieve a high hit rate, makes SSDs wear out quickly. To address this problem, we propose a new approach—write-efficient caching (WEC)—to greatly improve the write durability of SSD cache. WEC is conducive to reducing the total number of writes issued to SSDs while achieving high hit rates. WEC takes two steps to improve write durability and performance of SSD cache. First, WEC discovers write-efficient data, which tend to be active for a long time period and to be frequently accessed. Second, WEC keeps the write-efficient data in SSDs long enough to avoid excessive number of unnecessary updates. Our findings based on a wide range of popular real-world traces show that write-efficient data does exist in a wide range of popular read-intensive applications. Our experimental results indicate that compared with the classical algorithms, WEC judiciously improves the mean hits of each written block by approximately two orders of magnitude while exhibiting similar or even higher hit rates.

Index Terms—Cache drives, flash, SSD, endurance, write-efficient data

1 Introduction

In recent years, flash-based solid-state drive (SSD) cache gradually attracts more and more attention due to the small capacity of RAM cache compared with the data explosion [3] and SSD cache's high storage density, remarkable performance boost, and low additional costs [10], [11], [12]. Because SSDs' read performance is extremely good compared with hard disk drives (HDDs), SSD cache is valuable especially for the systems mainly storing the fast accumulating write-once-read-many data (e.g., images, videos, and shared files). Ninety percent of network traffic in taobao.com—one of the largest e-commerce websites in China—is contributed by image browsing; the image data storage of the website increases by 200 percent each year [1]. Similarly, videos stored in Facebook's data centers grow by 239 percent per year [2]. However, SSDs have very limited write endurance in their entire life cycle [4], i.e. the off-the-shelf enterprise-class SSDs only allow writing 1.38∼10.05 times of its own capacity per day if a common five-year lifetime is expected (see Table 1).

SSD's limited write endurance makes the hardware assumption of cache replacement algorithms no longer applicable. Traditional cache algorithms (e.g. LRU, LFU, MQ [21], LIRS [17], and ARC [18]) assume cache device can be written infinitely, and then conduct a manner of frequent and unlimited cached data updating. This manner is not suitable for SSD any more. For example, Table 2 shows that powered by a widely adopted LRU cache replacement algorithm, the practical writing speed of a disk cache is 233∼154,134 times of SSD capacity per day. Such a high writing speed is far beyond the allowed endurance of commercial SSD products (see Table 1), meaning that SSDs will be rapidly worn out.

Therefore, each cached block should yield excessive hits while residing in SSD cache, where writing is an expensive operation; otherwise, it wastes one time of valuable and scarce SSD write limitation. We introduce *write efficiency* (WE) to quantify the rewards of write operations. Given an amount of data, its write efficiency (see Eq. (1)) can be defined as the data's total hits divided by the total number of writes to achieve the hits. Specially, the WE value of a single cached block is equal to its hits prior to its eviction; whereas the WE value of the entire SSD cache is equal to the total hits of all cached blocks divided by all SSD writes. Then we can say, unlike traditional RAM-oriented cache algorithms aiming at high hit rates, SSD-oriented cache algorithms should focus on caching data that can produce high write efficiency, i.e. many hits and few writes at the same time

$$WE = \frac{Hits}{Writes}. \tag{1}$$

Aiming at large write efficiency of cached data in SSD cache, two conditions must be satisfied:

1) *Selecting write-efficient data (WED), i.e. the long-term popular data that can produce many hits for writing one block, to enter SSD cache;*
2) *Keeping WED long in SSD cache to adequately explore their potential of achieving high write efficiency.*

- *Y. Chai is with the Key Laboratory of Data Engineering and Knowledge Engineering of Ministry of Education, School of Information, Renmin University of China, Beijing 100872, China. E-mail: ypchai@ruc.edu.cn.*
- *Z. Du is with the Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: duzh@tsinghua.edu.cn.*
- *X. Qin is with the Department of Computer Science and Software Engineering, Samuel Ginn College of Engineering, Auburn University, Suite 3101E, Auburn, AL 36849-5347. E-mail: xqin@auburn.edu.*
- *D.A. Bader is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: bader@cc.gatech.edu.*

Manuscript received 11 June 2013; revised 1 Feb. 2104; accepted 22 Jan. 2015. Date of publication 5 Feb. 2015; date of current version 9 Oct. 2015.
Recommended for acceptance by J. D. Bruguera.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2401029

TABLE 1
The Write Endurance of Off-the-Shelf Enterprise-Class SSDs is Limited and Insufficient for the Write Demands of Serving as a Cache

SSDs	Capacity	TBW	Endurance
Intel 910 [7]	400 GB	5 PB	**7.18x/day**
Micron P300 [8]	200 GB	3.5 PB	**10.05x/day**
Samsung PM830 [9]	512 GB	1250 TB	**1.38x/day**

*Note: Capacity and total Bytes Written (a.k.a., TBW) are from the producers' official documents [7], [8], [9]. Write endurance is measured in terms of the number of times an SSD can overwrite all its data in one day ensuring a five-year lifetime (i.e. TBW/Capacity/(5*365)).*

TABLE 2
Write Speed of a Disk Cache Is Far Beyond SSDs' Write Endurance in Typical Applications

Applications	CHR	TD	WA	WS
Search Engine [40]	63.51 percent	1 h	4,228.20x	**101,477x/day**
OLTP [40]	62.81 percent	3 h	19,266.79x	**154,134x/day**
File Server [42]	95.36 percent	0.5 h	92.56x	**4,443x/day**
VoD [43]	22.35 percent	5 h	48.56x	**233x/day**

*Note: CHR—cache hit rates; TD—trace duration; WA—write amounts: multiple of SSD capacity; WS—write speeds. WS = WA*24/TD. Cache capacity is one tenth of all the data volumes.*

Traditional cache replacement schemes (e.g. LRU) prefer to cache hot data in a short run, and of course, they cannot keep write-efficient data long in SSD cache. Some modified schemes are proposed in recent years to limit the writes to SSD cache through a data filter. For example, SieveStore [38] only allows the blocks with larger miss counts than a threshold to enter SSD. However, the cached data in SSDs remains to be evicted passively. These schemes cannot make write-efficient data stay long in SSD cache to satisfy the above condition 2. The following example in Table 3 shows the cached data changes in SSD for LRU and SieveStore, respectively. For the sake of simplicity, the cache capacity is only one block and the miss threshold of data filtering in SieveStore is set as 3. Another solution with a lazy data eviction scheme is also given in this example, and it obviously achieves higher write efficiency than the former two.

The above example suggests that the eviction manner of general cache replacement schemes has negative effect for write efficiency, because the new arrival data block often is not much better, or even worse than the evicted one, but one additional write operation to SSD indeed happens. Shown as Eq. (1), similar nominator and larger denominator often make write efficiency drop. The eviction manner of LRU and SieveStore can be summarized as *push mode*, where the cached blocks are passively evicted when new data blocks are pushed into SSD cache. Even though write-efficient data blocks are cached in SSD, they cannot fully realize their potentials to achieve high write efficiency due to early eviction far before the end of its active lifetime, as Fig. 1a shows. On the contrary, the other obviously better solution adopts a *pull mode* of caching, where the write-efficient data can be kept long in SSD cache to contribute much more hits in a much longer active lifetime in SSD. Only when it becomes unpopular, it leaves actively and pulls a new write-efficient data block, as Fig. 1b shows.

In order to address these issues, we designed a new cache algorithm called write-efficient caching (WEC) based on the idea of pull-mode caching. WEC can accurately and quickly identify write-efficient data blocks, and judiciously keep the write-efficient data in SSDs for a sufficiently long period to produce high write efficiency. Our experimental results show that WEC achieves writing efficiency one or two orders of magnitude higher than the existing cache algorithms.

The rest of this paper is organized as follows. In Section 2 we analyze the key features of write-efficient data based on real-world traces. Section 3 gives a detailed comparison between the pull cache mode and the push mode of existing cache schemes. Our proposed WEC is presented in Section 4, followed by the quantitative evaluation in Section 5 and related work in Section 6. Finally, Section 7 concludes this paper.

2 Features of Write-Efficient Data

Caching write-efficient data in SSDs to produce high write efficiency, which is a non-trivial process, raises the following three challenging questions.

I. *Does write-efficient data widely exist in various applications?*
II. *How to identify write-efficient data?*
III. *How to keep write-efficient data sufficiently long in SSD-based caches?*

In this section, we look for the answers to these three questions through analysis on some typical real-system traces. Workloads of five types of real-world read-intensive services and their mixed workload listed in Table 4 provide a detailed case study of the write-efficient data. The traces were respectively collected from a search engine [40], OLTP application for financial institutions [40], Windows build servers [41], file servers [42], and video-on-demand servers [43].

2.1 Identifying Write-Efficient Data

In order to discover write-efficient data in various applications, we analyze some key characteristics of all the block bins after ranking all the data blocks according to their access counts in each examined trace.

TABLE 3
An Example Comparing Write Efficiency among LRU, SieveStore, and an Obviously Better Solution with a Lazy Eviction Scheme

Requests	1	2	3	1	4	1_{3rd}	2	5	1	2_{3rd}	4	2	4_{3rd}	1	6	**Writes**	**Hits**	**WE**
LRU	[1]	[2]	[3]	[1]	[4]	[1]	[2]	[5]	[1]	[2]	[4]	[2]	[4]	[1]	[6]	**15**	**0**	**0**
SieveStore [38]	[ ]	[ ]	[ ]	[ ]	[ ]	[1]	[1]	[1]	**[1]***	[2]	[2]	**[2]***	[4]	[4]	[4]	**3**	**2**	**0.67**
A Better Solution	[ ]	[ ]	[ ]	[ ]	[ ]	[1]	[1]	[1]	**[1]***	[1]	[1]	[1]	[1]	**[1]***	[1]	**1**	**2**	**2.0**

Cache capacity is assumed to be one for simplicity. Note: R_{3rd} implies R can enter SSD now according to SieveStore; [X] means block X is in cache after this request; [X] means a cache hit.*

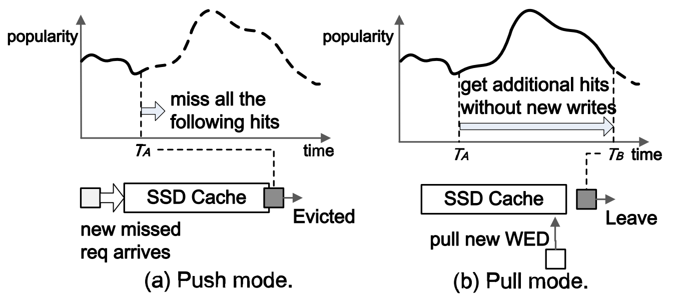


Fig. 1. *Push mode* makes hot data be evicted too early, with the result of low write efficiency, while *pull mode* can realize the potential of write-efficient data fully to contribute much more hits with the same cost of one write operation.

TABLE 4
Characteristics of Some Real-World Read-Intensive I/O Workload Traces

Trace Name	Application Type	Req Num	Read Ratio
websearch	Search Engine	5,047,596	99.98 percent
financial	OLTP	1,555,474	81.82 percent
buildserver	Build Server	957,596	87.34 percent
auspexserver	File Server	215,678	99.93 percent
cctvvod	VoD	550,310	100 percent
mix	Multiple App.	8,326,654	95.13 percent

Fig. 2 plots the average accesses per block against the ranked block bins for the five real-system traces and their mixed trace. We can conclude that the top data blocks are much more frequently accessed than the other blocks in the applications. For example, the top 0.1 percent blocks of the *mix* trace are accessed on average for more than 2,657 times during the time period of a few hours. Caching these popular blocks in SSD-based caches may potentially lead to a large number of cache hits, thereby achieving high writing efficiency.

Fig. 3 shows the average access interval length and the active lifespan of the ranked block bins for the *mix* trace. An access interval length quantifies the distance between two adjacent accesses to a block. If a block has a small access interval length, caching the block may yield a high hit rate. Note that the access interval of a block is not measured with clock time. Rather, the access interval is measured in inter-reference recency (IRR), which refers to the number of other blocks accessed between two consecutive references to the block [17]. For example, for a visiting sequence a, b, c, b, d, a, the IRR between the two adjacent request a is 3 (including b, c, d) rather than 4. In addition, the active lifespan—sum of a block's all access interval lengths—reflects the block's maximum possible staying time in cache.

Fig. 3 confirms that the most-accessed blocks not only have the longest active periods, but also are accessed in a relatively high density. Therefore, write-efficient data can be discovered by identifying these blocks that are most likely to be frequently accessed; this type of block can reduce the number of SSD writes after being cached in the SSD-based cache. Both high cache hit rate and a small count of written blocks can be simultaneously accomplished, leading to high write efficiency.

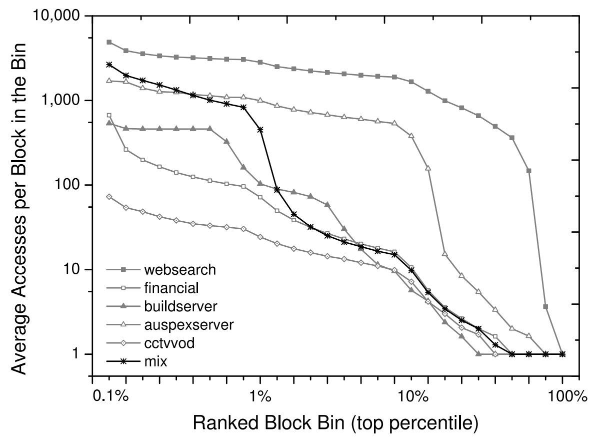


Fig. 2. Caching top blocks may lead to high write efficiency. For example, the top 1 percent blocks are accessed for more than 1,000 times within several hours.

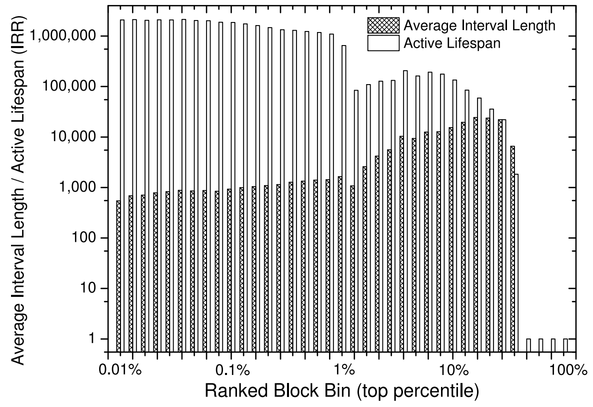


Fig. 3. The most-accessed data blocks can be considered as write-efficient data, because they not only have a long active lifespan, but also have shorter mean access intervals.

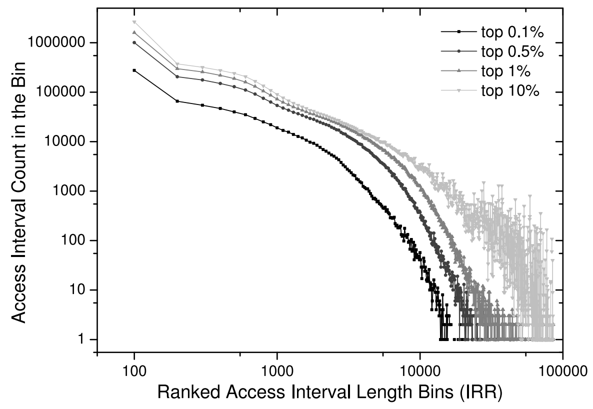


Fig. 4. Too early eviction of cached data and low write efficiency are consequent, because although most access intervals are very small, some very large access intervals are mixed up with them universally for most-accessed blocks.

2.2 Challenge of Preserving Write-Efficient Data

Although write-efficient data is widely found in read-intensive applications, caching the data in SSD-based disk cache raises challenging issues due to nonuniform access interval lengths.

Fig. 4 plots the access interval count against the ranked access interval length (measured in IRR) bins for the top 0.1, 0.5, 1 and 10 percent most-accessed blocks, respectively.

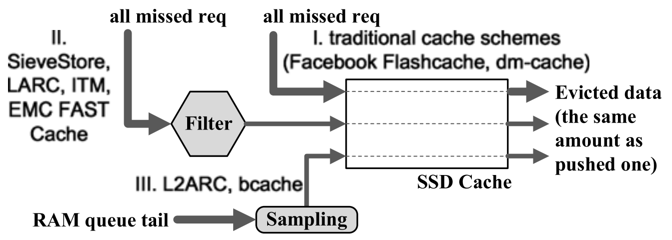

all missed req
all missed req
I. traditional cache schemes (Facebook Flashcache, dm-cache)
II. SieveStore, LARC, ITM, EMC FAST Cache
Filter
Evicted data (the same amount as pushed one)
III. L2ARC, bcache
SSD Cache
RAM queue tail
Sampling

Fig. 5. The three types of existing cache schemes all belong to the *push mode*, which prevents them to keep write-efficient data within SSD cache for long to produce high write efficiency.

A vast majority of the intervals have very short lengths; a few intervals are very long. This phenomenon is more pronounced when the statistics covers more top blocks. The drawback of the push-mode cache replacement algorithms is that a write-efficient data block can be easily evicted when a long interval occurs. In other words, write-efficient data will be swapped out of cache much before the end of the data's active lifespan and; therefore, low write efficiency becomes inevitable under the management of the traditional algorithms.

3 Pull Mode versus Push Mode

In this section, we summarize and category the existing SSD cache schemes into three types, shown as Fig. 5. But all of them fall into the same manner of *push-mode* caching. And then, a detailed comparison between the existing push mode and a new *pull mode*, which leads to much higher write efficiency, is given.

3.1 Existing Three Types of Push-Mode Caching

3.1.1 Traditional Cache Management

Traditional cache replacement algorithms, such as FIFO, LRU, LFU, MFU, FBR [19], LRU-k [20], MQ [21], LIRS [17], and ARC [18], ignore the erasure limits of SSDs, thereby issuing an excessive number of writes by updating cache contents nearly for every missed request. However, many industry SSD cache products still adopt the traditional algorithms for simpleness. For example, Facebook's Flashcache [13] employed FIFO and LRU as its cache schemes, while linux dm-cache [15] implemented MQ [21].

To address the issue of poor write efficiency of the traditional cache management for SSD-based cache disks [13], [15], [22], increasing efforts have been made to design the following two new types of caching algorithms to reduce the number of SSD writes.

3.1.2 Raising the Entry Threshold of Cached Data

One solution is to set a data filter on the data path entering SSD cache, with the results of reduced SSD writes. For instance, LARC [39] set a virtual LRU queue with limited length before SSD cache, and hitting the LRU queue is the condition of entering SSD. It means that only hot blocks with two consecutive close requests can be cached in SSDs.

Another kind of filtering condition are access counts. EMC's FAST Cache [10] and Intel's Turbo Memory (ITM) [36] only allow data blocks with an access count larger than a threshold to be fetched into SSD-based cache. To avoid cache pollution of high frequently accessed data, SieveStore [38] only allocates a cache to a block if the nth miss of the block is detected in a recent access window. SieveStore is the representative solution of this category due to its good effects. In industrial products, Netapp's Intellgent Caching [11] and Oracle Exadata's Smart Flash Cache [12] adopts similar cache schemes.

3.1.3 Periodical Sampling

A second kind of approach to reduce SSD writes is to periodically update cached data without handling each cache miss. For example, the Solaris ZFS file system adopts a technique called second-level ARC (L2ARC) [37], which periodically fills SSD with the to-be-evicted contents at the tail of RAM cache queue. L2ARC employs a periodical sampling module to select cached data to reduce the amount of written data to SSDs. In addition, Linux's bcache [14] adopts an analogous scheme. However, they may lead to an uncertainty in storage system performance due to the fact that hot contents may not be cached in a timely manner.

3.2 Comparison between Push and Pull Mode

Recall that the two conditions to achieve high write efficiency presented in Section 1 are: 1) selecting write-efficient data to cache, and 2) keeping them long in SSD cache. Therefore, in the following Table 5, we make a comparison on the two conditions among the three types of existing push-mode solutions and a new write-efficient pull mode (See Fig. 1).

3.2.1 Selecting Write Efficient Data to Cache

Among the traditional cache algorithms, the recency-based ones (e.g., LRU) are inadequate to identify write-efficient data, because these algorithms tend to cache short-term temporal hot data blocks rather than long-term hot ones. On the contrary, LFU and other frequency-based cache

TABLE 5
Unlike Pull Mode, Existing Push-Mode Solutions Can't Satisfy Both the Two Conditions to Achieve High Write Efficiency

Types		Selecting WED	Keeping WED	Representative Algorithm	Industrial Solutions
Push Mode	Traditional Cache Schemes	/	×	LRU, LFU, etc.	Facebook Flashcache, Linux dm-cache
	Raising Entry Threshold	√	–	SieveStore, LARC	EMC FAST Cache, Netapp Intelligent cache, Oracle Smart Flash Cache, Intel Turbo Memory
	Periodical Sampling	–	–	L2ARC	Solaris ZFS, Linux bcache
Pull Mode		√	√	Our proposed WEC (Write-Efficient Caching)	

Note: √ *, -,* × *, and / mean good, not so good, poor, and uncertain respectively.*

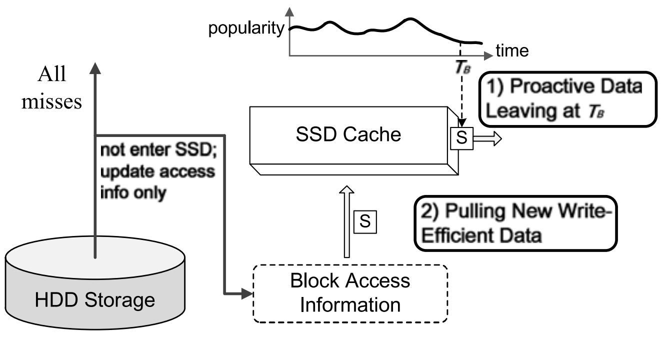

popularity
time
T_B
All misses
not enter SSD; update access info only
SSD Cache
S
1) Proactive Data Leaving at T_B
2) Pulling New Write-Efficient Data
Block Access Information
HDD Storage

Fig. 6. The design overview of WEC, which implements the *pull model*: a) WEC detaches SSD cache's data updating from the missed requests arrival, i.e. the pushed data source. b) Instead, SSD data updating process includes 1) proactive data leaving, and 2) pulling new write-efficient data.

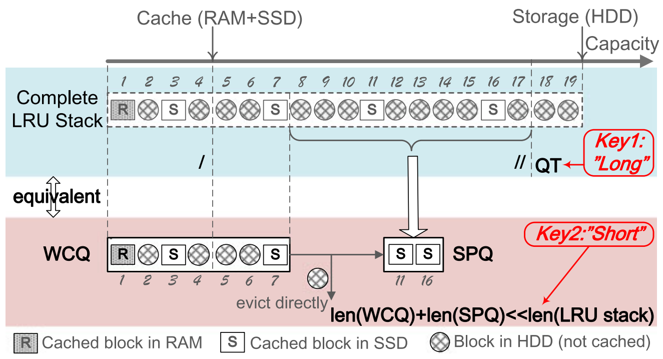


Fig. 7. The two keys of proactive data leaving: a) *Long*: putting QT off in the complete LRU stack long enough makes cached data in SSD tolerate occasionally large intervals and produce high write efficiency. b) *Short*: Complete LRU stack = WCQ (Write-efficient data Candidate Queue) + SPQ (SSD Protection Queue), and length (WCQ) + length (SPQ) ≪ length (LRU stack).

replacement schemes are focused on blocks with the highest access frequencies. Consequently, their performance is uncertain in condition 1.

The solutions with raised entry threshold of SSD cache usually select data blocks that have already been referenced for several times. These blocks usually have large possibility to be write-efficient data.

L2ARC periodically updates cached data in SSDs from the tail of the RAM queue managed by the ARC scheme. The cached data in SSDs are sampling points of ARC's preference; thus, L2ARC selects more short-term hot blocks as ARC does than write-efficient data for SSD.

Pull-mode solutions promote new data for SSD cache from many candidates with long access history records. Therefore, the probability of picking write-efficient data is high.

3.2.2 *Keeping Write-Efficient Data Long in SSD Cache*

Keeping write-efficient data in SSDs is much harder than condition 1, because the ubiquitous long access intervals of write-efficient data makes it easy be evicted much before its active lifetime ends (See Section 2.2).

In the push mode, a cached data block is forced to be evicted when a new data blocks enters SSDs. Therefore, the write-efficient data cached in the SSD are unable to be kept for a long time period due to early evictions. Although the solutions like SieveStore and L2ARC reduce SSD write amounts, these approaches, unfortunately, do not fundamentally change the push-mode behaviors of the traditional caching algorithms. For example, if there are excessive cache misses for SieveStore, the large number of SSD writes can quickly wear out the SSD cache disk. L2ARC lacks the flexibility in controlling the blocks' lifespan in SSDs. Such a deficiency makes it difficult for L2ARC to offer the improvement of write efficiency, because the fixed period may be too short for active blocks and too long for inactive ones.

On the contrary, in pull mode, the write-efficient data in SSD is protected to produce as many hits as possible. Only when a block loses its activity, it leaves proactively to make room for new pulled write-efficient data.

4 Write-Efficient Caching Algorithm

In this section, we propose a write-efficient caching algorithm dedicated to SSD-based caches to achieve high writing efficiency by adopting a new *pull mode* of SSD cached data updating to satisfy both of the two conditions summarized in Table 5. Fig. 6 illustrates that WEC's data updating in SSD is detached from the arrivals of newly pushed data sources (e.g., misses, filtered misses, or other requested data occurs). Rather, the SSD data are updated independently and periodically under the supervising of WEC.

Each round of SSD data updating has two steps: 1) If any write-efficient data block ends its active lifespan after many cache hits in the SSD cache, this block proactively leaves SSD itself in order to provide cache space for another write-efficient data block (see details on the "Proactive Data Leaving" module in Section 4.1). 2) WEC pulls new write-efficient data into SSD cache based on history block access information (see Section 4.2 for details on the "Pulling New Write-Efficient Data" module). The primary benefit of this pull model is protecting cached write-efficient data from being evicted in an early stage. WEC leverages the novel pull model to cache write-efficient data to achieve a high hit rate while reducing the amount of data written to SSD-based caches. The detailed workflow of WEC is summarized in Section 4.3.

4.1 Proactive Data Leaving

WEC prevents write-efficient data from being evicted in an early stage by introducing a loose quit condition of cached data in SSDs. As Fig. 7 shows, we assume a complete LRU stack containing all the data blocks in HDD storage, which arranges blocks in a recency order regardless of the block locations (i.e., RAM cache, SSD cache or HDD). Traditional cache schemes evict a cached block when it goes out of the boundary of cache capacity (see line I in Fig. 7).

Recall that it is a challenge to keep write-efficient data in cache for a long time due to occasional large access intervals (see more details in Section 2.2 and Fig. 4). We address this challenge by putting quit threshold (QT) off to line II in Fig. 7 so that SSD-based caches can tolerant large access intervals and keep write-efficient data for a longer time period.

In this case, the cached blocks that goes out of cache capacity will not be replaced by HDD blocks that has better recency, so you can see data blocks located in RAM cache, SSD cache and HDD are mixed in this complete LRU stack in Fig. 7.

Although this mechanism can improve write efficiency, it introduces the following two new challenges:

1) *The delayed eviction of cached data may leads to many unpopular data staying in SSDs, with the possible result of obvious drops of cache hit rates.*
2) *Maintaining such a huge complete LRU stack introduces a lot of overhead.*

The following two sections dedicate to solve the above two challenges.

4.1.1 Appropriate Quit Threshold

Simply setting a very large QT value, however, is not a satisfactory solution, because non-write-efficient data may be mistakenly treated as write-efficient one; overdue write-efficient data with terminated lifespan wastes valuable SSD resources for a very long time period. To avoid this problem, we must choose an appropriate QT value to keep active write-efficient data in SSDs while evicting garbage data in a timely manner.

The setting of QT needs to be automatically adjusted according to various cache capacity settings. WEC determines the QT value based on the capacity ratio (CR) between cache and storage. In other words, WEC relies on a mapping function to derive the QT value from the capacity ratio CR (see Eq. (2) where UBN means unique block number of the whole storage)

$$QT = UBN * f(CR). \tag{2}$$

Although it is a challenge to develop a scheme that can optimize SSD-based cache for a wide range of applications, designing a satisfactory mapping function for QT values is straightforward. For example, the implementation of WEC adopts a square root function to express the relation between QT and cache capacity ratio CR. Please refer to Section 5.4.2 for a discussion on various mapping functions used to adjust QT values. We observe that the effects of the tested functions are quite similar when the cache size is large (see Figs. 17 and 18).

4.1.2 $WCQ + SPQ =$ Complete LRU Stack

The downside of maintaining a huge LRU stack lies in heavy computing and metadata-storage overhead in RAM. To address this problem, we set the headmost small part of the complete LRU stack as a write-efficient data candidates queue (WCQ), which is used for selecting write-efficient data. For the other part of the LRU stack before QT, only the blocks located in SSD are meaningful for us, so we record these SSD blocks in a SSD protection queue (SPQ). The design of WCQ and SPQ queues help in reducing the metadata management overhead of WEC.

As Fig. 7 shows, SSD blocks' ranks in the complete LRU stack are also recorded and marked as "idle time" in SPQ. When an SSD block goes out of WCQ, it is placed at the head of SPQ, and set its initial idle time as the total number of blocks of WCQ, i.e. its current rank in the complete LRU stack. If a block is evicted from WCQ, the idle time of all the blocks in SPQ increases by 1, indicating its position in the LRU stack moves backward. In doing so, the idle time of a SPQ block equals to its position in the complete LRU stack. When the idle time of a block is larger than a length limit (a.k.a., QT of the complete LRU stack shown in Fig. 7), the block should leave proactively to make room for better ones.

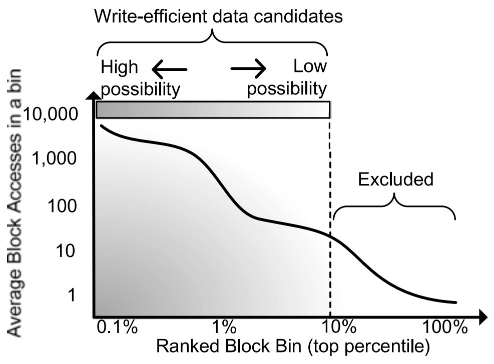


Fig. 8. Steps of pulling new write-efficient data in WEC: (1) Excluding non-popular data to guarantee that promoted data have good high write efficiency; and (2) selecting data with higher access counts as candidates prior to lower one to achieve high average quality of promoted data.

In addition, SPQ is implemented as a FIFO-like queue. When there is a hit to a block in SPQ, the block is deleted from SPQ and placed at the head of WCQ, indicating that the block is activated again.

4.2 Pulling New Write-Efficient Data

4.2.1 Selecting WED with Low Overhead

Recall that write-efficient blocks have the highest access rate with a long active lifespan (see Section 2). In the existing solutions (e.g., SieveStore, Intel's Turbo Memory, and EMC's FAST Cache), the selection of write-efficient blocks is triggered by access or miss thresholds. However, these solutions have high overhead due to the needs of maintaining access records of all the blocks in an entire storage system.

Write-efficient data not only has a high access rate, but also has a short interval length on average (i.e., a short time between two consecutive accesses). There is no need to maintain the historical access information of all the blocks in a storage system. Write-efficient data blocks can be standing out from a short cache queue. Therefore WEC only maintains the historical access information of WCQ rather than the entire storage system.

WEC takes a simple yet efficient approach (see Fig. 8) to discover write-efficient data candidates. Promoting blocks accessed more than a threshold from WCQ means that the block is continuously accessed within a small interval. A large number of non-popular blocks are excluded from WCQ. Furthermore, the high access rate of a block implies a good possibility of being repeatedly accessed within short periods of time. Thus, these blocks are more likely to be promoted as write-efficient data than the others. We do not by any means claim that WEC can accurately pick the best candidate blocks; nevertheless, a vast majority of candidates selected by WEC are write-efficient data.

4.2.2 Organization of WCQ

The write-efficient data candidates queue is a standard low-overhead LRU queue maintaining the hottest data in a storage system. WCQ provides candidates for the write-efficient data promotion module. To improve the accuracy of discovering write-efficient data, WCQ's capacity is usually larger than the sum of RAM and SSD to record historical access information. Except for all cached blocks in RAM and a portion of cached blocks in SSD (the rest part is managed by SPQ), the other blocks in WCQ are residing in HDDs (see, for example, Fig. 7).

When RAM is full, blocks with the least recency in RAM will be evicted for new arrivals. Note that a removed block is still kept in WCQ; the removed one is marked as a HDD block. The metadata of each block maintained in WCQ includes: ID, access counts, and location (i.e., RAM, SSD, or HDD).

WEC judiciously adjusts the length limit of WCQ. When a data promotion is performed, WCQ's length limit is reduced if the number of promoted write-efficient block candidates is more than what is needed by SSD; on the other hand, when there are few candidates, WCQ increases its length limit.

4.3 Workflow Summary

As Fig. 6 shows, the SSD cache's data updating is detached from normal request process. In this part, we summarize the workflow of these two processes.

4.3.1 SSD Data Updating

WEC's data updating in SSD cache is performed periodically. In each round, WEC first searches the whole SPQ for blocks with larger idle time than QT. If no such blocks exist, there is no need to update SSD cached data in this round; otherwise, the blocks with large enough idle time are removed from SPQ to make some free slots, and a process of pulling new write-efficient data from WCQ is invoked.

In this case, WEC searches blocks located in RAM or HDD in WCQ for the ones whose access count is larger than a threshold. When enough blocks are pulled to fill up the free slots in SPQ, or WCQ is fully scanned, the process is accomplished. In fact, the time complexity of WEC is $O(N)$, where N is the length of WCQ.

Furthermore, we can set the period of WEC's SSD data updating as a large value (e.g., 50,000 requests) to further reduce the overhead. Our evidence (see Section 5.4.1) confirms that setting a larger data updating period can lead to higher writing efficiency for WEC.

4.3.2 Request Processing

The pseudo code below outlines WEC's request processing procedure in various scenarios. Note that write requests do not impose any caching pressure, because WEC addresses the needs of read-intensive workloads. If the targets of writes are cached or exist in WCQ, the metadata and data will both be eliminated to avoid generating dirty blocks (i.e., few writes are directly served by disks).

```
0:  def REQ_PROCESS (Request $x$):
1:    case 1: $x$ is a read request.
2:      case 1.1: $x$ hits a RAM or SSD block in WCQ (cache hit).
3:        promote the block to WCQ's head.
4:      case 1.2: $x$ hits a HDD block in WCQ (cache miss).
5:        mark the block as a RAM block in WCQ;
6:        promote it to WCQ's head;
7:        RAM_REPLACE().
8:      case 1.3: $x$ hits a block in SPQ (cache hit).
9:        move the block from SPQ to WCQ's head.
10:       WCQ_EVICT().
11:     case 1.4: $x$ does not hit any queue (cache miss).
12:       add a new RAM block related to $x$ at WCQ's head;
13:       RAM_REPLACE();
14:       WCQ_EVICT().
15:   case 2: $x$ is a write request.
16:     if $x$ hits a block in WCQ or SPQ.
17:       delete the block from WCQ or SPQ.
18: def RAM_REPLACE ():
19:   if RAM blocks in WCQ is beyond RAM capacity:
20:     mark the last RAM block in WCQ as a HDD block.
21: def WCQ_EVICT():
22:   assuming $y$ is the evicted block.
23:   all SPQ blocks' IdleTime + 1.
24:   if $y$ is an SSD block.
25:     put $y$ at SPQ's head, $y$'s IdleTime=WCQ's length.
```

5 Evaluation

The performance of our WEC is evaluated driven by real-world workloads (see Section 5.1). We compare WEC with the traditional caching algorithms in terms of performance and impacts on SSD endurance in Section 5.2. We also examine the sensitivity of WEC under various cache capacity settings (see Section 5.3), and various parameter settings (see Section 5.4). Finally, an SSD inner writes estimation is made in Section 5.5.

5.1 Experimental Settings

Performance metrics. There are three performance metrics used in our experiments. (1) Write efficiency measured as SSD hits divided by the amount of data written to SSD is a primary performance metric to quantify the efficiency of turning SSD writes into hits, which is the dominate ability for the SSD-oriented cache schemes. (2) Total cache hit rate including both RAM-based and SSD-based cache is a traditional indicator to evaluate cache replacement algorithms. (3) The number of blocks written to SSD-based cache can be used to measure the amount of data written to SSDs.

Alternative solutions. We compare WEC against the three existing solution families:

- The two-layer cache hierarchy comprised of RAM and SSDs [22] are managed by the classical cache replacement algorithms like LRU, LFU, MQ [21], LIRS [17] and ARC [18], which offer high performance without addressing the endurance issue of SSDs.
- The SieveStore system makes use of selective cache allocation to reduce allocation-writes [38].
- The Solaris ZFS file system powered by the L2ARC scheme [37] applies ARC [18] for RAM data management. L2ARC limits the number of writes to SSDs by a periodical data updating scheme.

5.2 Overall Performance Evaluation

Let us compare the overall performance of WEC with that of the traditional cache algorithms (i.e., LRU, LFU, LIRS, MQ and ARC). Figs. 9, 10, and 11 plot the write efficiency, the

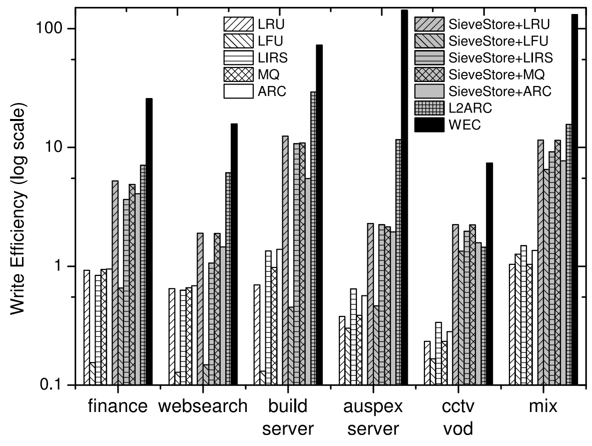


Fig. 9. WEC's write efficiency is about one to two orders of magnitude higher than those of the existing cache algorithms.

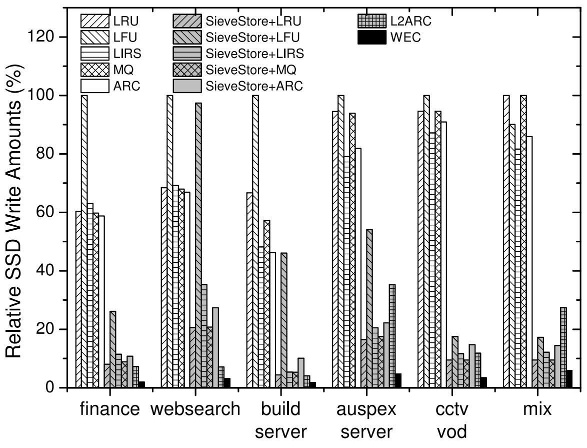


Fig. 11. WEC significantly reduces the amount of SSD writes.

total hit rates of an entire cache comprised of both RAM and SSD, and the amount of SSD writes, respectively. In this group of experiments, the capacity ratio between cache (including RAM and SSD) and storage is set as 1/10.

Fig. 9 shows that the write efficiency of WEC is about an order of magnitude higher than that of SieveStore and L2ARC, and is around two orders of magnitude higher than the traditional cache algorithms. Taking the *mix* trace for example, the write efficiency of WEC is as high as 131.03 (i.e., writing one block into SSD can bring 131.03 cache hits as a reward); the write efficiency values of L2ARC, SieveStore, and the traditional algorithms are 15.62, 6.52~11.54, and 1.04~1.50, respectively. The low write efficiency of the traditional algorithms is caused by their aggressive push mode. A large number of blocks in SSDs have not been frequently accessed before being evicted from the SSDs, leading to about 1 hit per written block on average. The high write efficiency of WEC can be demonstrated through high total cache hit rate (see Fig. 10) and reduced amount of SSD writes (see Fig. 11).

The high write efficiency of WEC confirms that the pull mode is very effective in terms of keeping high-quality contents in SSD-based caches all the time unless the popularity has changed. L2ARC and SieveStore, on the other hand, tend to evict the high-quality data blocks far before their active lifespan terminates. Even worse, the traditional algorithms evict write-efficient data in an earlier stage. The data in SSDs stays for a short time period due to the evictions of popular data induced by an excessive number of misses.

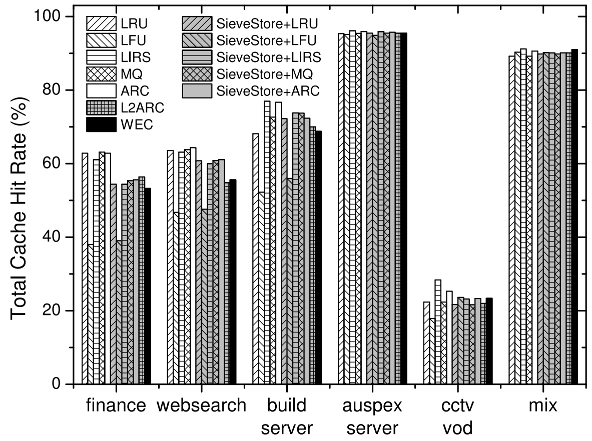


Fig. 10. WEC achieves similar or even higher total cache hit rates compared with existing solutions.

Among the traditional algorithms, LIRS achieves the highest write efficiency in most cases, whereas SieveStore +LRU outperforms the others in the SieveStore series. This result reveals that integrating simple algorithm with SieveStore may outperform the complicated algorithms. In addition, L2ARC always has higher write efficiency than SieveStore even when L2ARC's amount of SSD writes is larger than that of SieveStore, because SSDs in L2ARC's cache layer can deliver more cache hits.

Fig. 10 plots the total cache hit rates of RAM augmented with SSDs. In most cases, one of the traditional algorithms can achieve the highest cache hit rates because of no SSD write limitation. However, the caching solutions with SSD write limitation can also lead to very good performance thanks to write-efficient blocks cached in SSDs. This trend is more pronounced when it comes to the *mix* workload. For example, the total cache hit rate of WEC is 90.99 percent, whereas that of the traditional algorithms are in a range between 89.23 to 91.20 percent. Except for LIRS, WEC outperforms all the other traditional schemes.

In modern data centers, it is common to have multiple applications sharing a storage system. The *mix* workload examined here resembles real storage systems in data centers. WEC significantly improves the lifespan of SSDs without noticeably downgrading I/O performance. In many cases, WEC's cache hit rate is higher than those of the widely used algorithms like LRU.

5.3 Cache Capacity Sensitivity

5.3.1 Cache Sizes

A good caching algorithm needs to achieve high performance in a variety of system configurations including various capacity ratios between cache and storage. An ideal caching scheme for SSDs should exhibit high cache hit rates and maintain high write efficiency under various cache sizes.

In this group of experiments, we focus on the *mix* workload. In addition to WEC and L2ARC, LIRS and SieveStore +LRU are evaluated, because they achieve the best performance and the highest write efficiency in their own algorithm families (see Section 5.2). Figs. 12 and 13 compare the

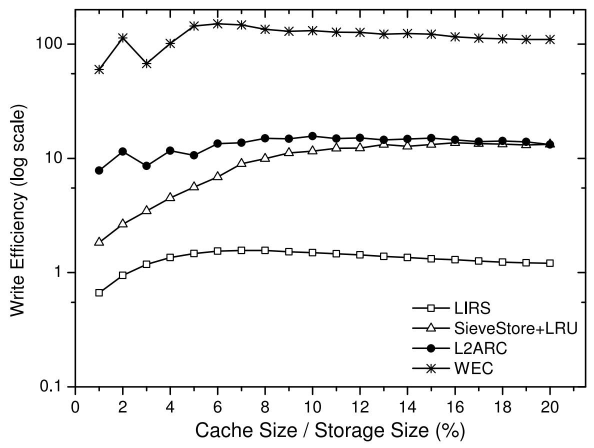


Fig. 12. WEC achieves higher and stable SSD write efficiency than the other three approaches under various cache sizes.

write efficiency and total cache hit rates of the four algorithms when the ratio between cache and storage ranges from 1 to 20 percent.

Fig. 12 indicates that the write efficiency of SieveStore +LRU is relatively low with small cache sizes. The write efficiency values of WEC, L2ARC, and LIRS are not very sensitive to cache size. The write efficiency gap between WEC and the other solutions is consistent with the results plotted in Fig. 9. For example, when cache size is relatively large, WEC achieves over 100 hits by fetching one block into SSD, whereas L2ARC and SieveStore+LRU can benefit as few as 10 hits, and the traditional algorithms only experience on average 1 hit.

SieveStore has low write efficiency under small cache sizes, because there are many misses satisfying the sieve condition caused by low cache hit rate. These large number of misses make SSD cache evict many active hot blocks before they contribute more hits.

Fig. 13 plots the total cache hit rate when the capacity ratio between cache and storage various from 1 to 20 percent. The cache hit rate of WEC, in most cases, is close to that of LIRS. In a few cases, WEC has better hit rates than LIRS. SieveStore+LRU is the worst one with perspective of hit rate.

We conclude that WEC reduces SSD writes and achieves high write efficiency without any adverse performance impact under various cache settings.

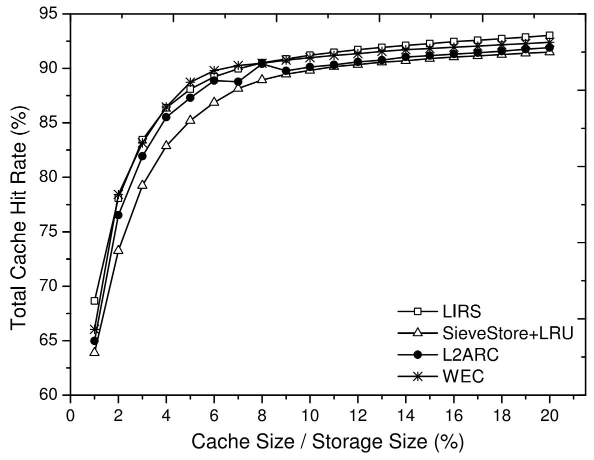


Fig. 13. WEC achieves the best or suboptimal total cache hit rate among the four algorithms under various cache sizes.

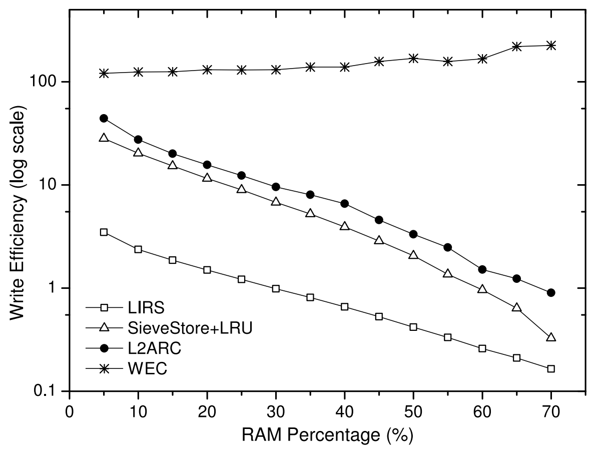


Fig. 14. The write efficiency of WEC is not sensitive to the RAM percentage configuration, whereas the RAM percentage significantly affects the write efficiency of the other three solutions.

5.3.2 Capacity Ratio Between RAM and SSD

Given a hybrid cache layer composed of RAM and SSD, we investigate the impact of the capacity ratio between RAM and SSD on overall system performance. Large RAM leads to low latency; a large SSD offers large cost-effective cache capacity, which in turn gives rise to a high cache hit rate. Ideally, a good cache replacement scheme should deliver high performance under various capacity ratio of RAM and SSD.

Again, we pay attention to the *mix* workload. Figs. 14 and 15 show the write efficiency and the cache hit rates when the percentage of RAM ranges from 5 to 70 percent of the entire cache layer. The capacity ratio between cache and storage is set to 1/10.

Fig. 14 reveals that the write efficiency of WEC is not sensitive to the RAM percentage configuration, whereas the RAM percentage significantly affects the write efficiency of the other three solutions. For the traditional caching algorithms and SieveStore, a small SSD capacity shortens each block's time spent in SSD-based cache, resulting in few hits per SSD write. In the L2ARC case, a large RAM capacity increases the number of blocks cached into SSDs, thereby leading to premature and frequent data evictions. On the contrary, the proactive-data-leaving mechanism implemented in WEC keeps popular blocks in the SSD cache

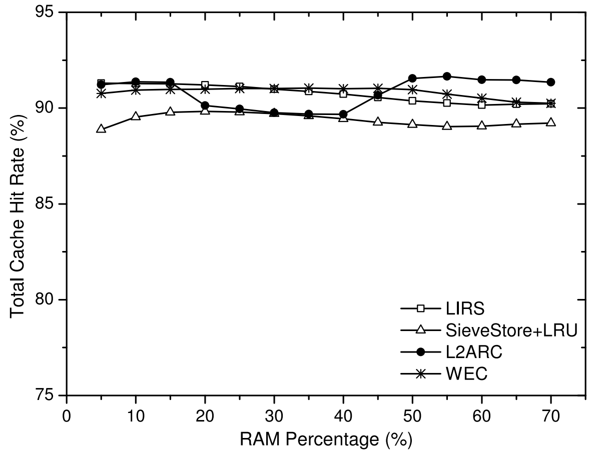


Fig. 15. The cache hit rate of WEC is insensitive to RAM percentage. WEC's cache hit rate is close to those of LIRS and L2ARC under various RAM settings.

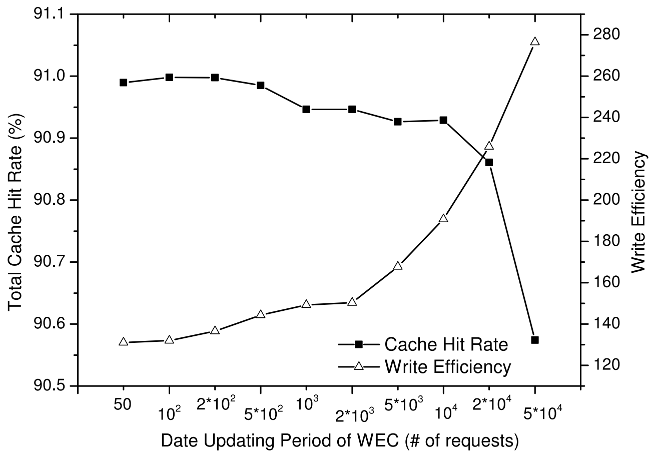


Fig. 16. Increasing WEC's SSD data updating periods from 50 to 50,000 requests leads to lower overhead and higher write efficiency. Large updating period has marginal impacts on WEC's hit rate.

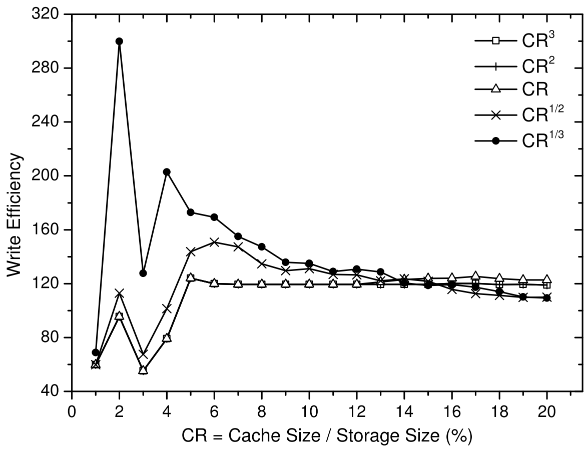


Fig. 17. When the capacity ratio of cache and storage is large than 9 percent, the five CR mapping functions deliver a similar write efficiency.

almost for the blocks' entire active life cycles under a variety of system settings.

We observe from Fig. 15 that the total cache hit rate of SieveStore is the worst one among all the four evaluated schemes. Unlike the other three solutions, the cache hit rate of WEC is insensitive to RAM percentage. This result suggests that WEC can achieve high cache hit rate regardless of the settings of RAM cache.

In most cases, WEC's write efficiency and hit rate are constantly higher than those of the traditional caching algorithms, SieveStore, and L2ARC under various RAM percentage settings.

5.4 Parameter Sensitivity

5.4.1 SSD Data Updating Period

WEC stores write-efficient data in SSDs for a long time; the long lifetime of cached data makes frequent SSD updates unnecessary. This feature allows us to set a large SSD data updating period (i.e., the period of checking if some cached blocks should be evicted) in order to reduce WEC's overhead without affecting performance. Fig. 16 exhibits the write efficiency and the cache hit rate of WEC when the data updating periods ranges from 50 to 50,000 requests.

Fig. 16 indicates that increasing WEC's updating period has marginal impacts on its cache hit rate (e.g., hit rate only decreases from 90.99 to 90.93 percent). Even when the updating period increases high up to 50,000 requests, the cache hit rate is still as high as 90.57 percent. On the other hand, a large updating period leads to high write efficiency. For example, WEC's write efficiency increases from 131.03 to 276.45 when the updating period goes up to 50,000 from 50 requests. When it comes to storage systems with no strict performance constraint, setting a very large data updating period not only helps to achieve high write efficiency, but also significantly reduces WEC's overhead.

In summary, it is recommended to minimize WEC's overhead by increasing its data updating period, which helps in improving write efficiency. Please note that in most of our experiments, WEC's updating period is set as short as 50 requests. In practice, we can increase the updating period to further reduce WEC's overhead.

5.4.2 Proactive Data Leaving Threshold Functions

A critical parameter of the proactive data leaving module is the quit threshold, which is used to distinguish active write-efficient blocks from to-be-evicted cold blocks residing in SSD-based caches. To achieve good performance under various cache capacity settings, WEC dynamically and judiciously sets the QT value according to the capacity ratio (CR) between cache and storage. The QT value is derived from CR using a proper mapping function. For a variety of cache capacities (i.e., CR ranges from 1 to 20 percent) under the *mix* workload, Figs. 17 and 18 show the write efficiency and cache hit rates of WEC, in which five different mapping functions $f(CR)$ (see Eq. (2)) are implemented. The five functions express super-linear, linear, and sub-linear increments of QT with the increasing value of the CR ratio

$$f(CR) = CR^3/CR^2/CR/CR^{1/2}/CR^{1/3}. \quad (3)$$

Fig. 17 shows that $CR^{1/3}$ achieves the highest write efficiency when the ratio of cache and storage is smaller than 14 percent; $CR^{1/2}$ is highest when the ratio is 14 percent; and CR is best when the ratio is larger than 14 percent. Nevertheless, the write efficiency discrepancy among the five mapping functions is very small. When the capacity ratio of cache and storage is large than 9 percent, the five functions deliver a similar write efficiency.

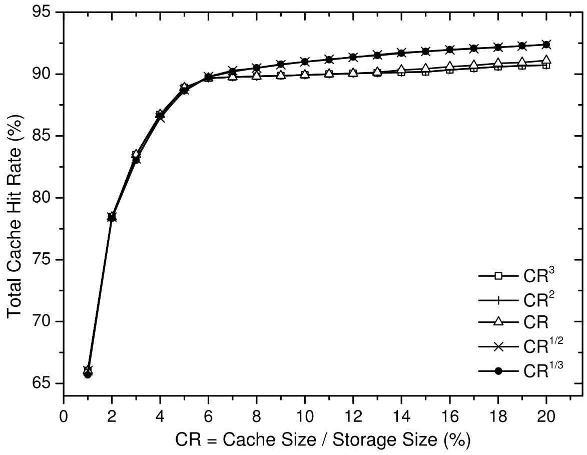
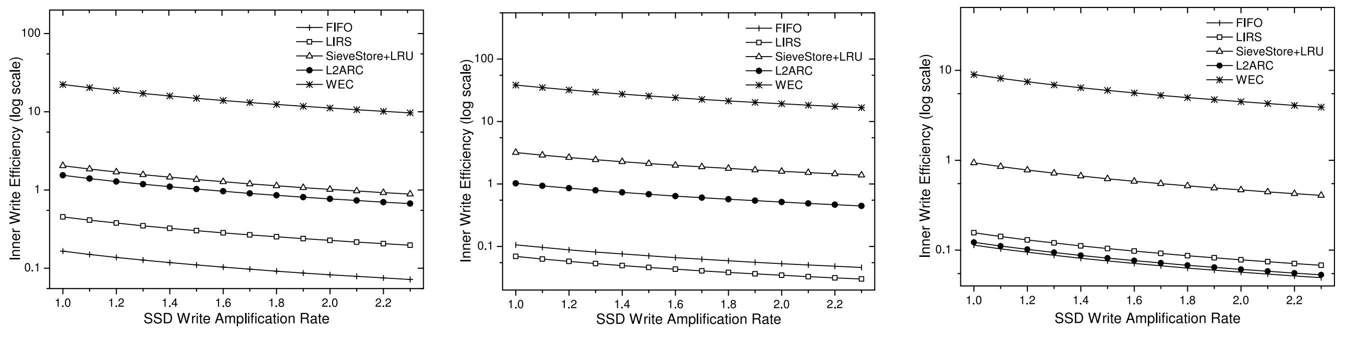


Fig. 18. The discrepancy in cache hit rate among the five CR mapping functions is negligible.



Filebench - *randomfileaccess.* Filebench - *networkfs.* Filebench - *randomread.*

Fig. 19. Inner write efficiency of three typical workloads of Filebench based on write ampflication rate estimation ranging from 1.0 to 2.3. The IWE of WEC is far ahead of all the others, even making the worst estimation.

Fig. 18 confirms that the discrepancy in cache hit rate among the five functions is negligible. When the cache/storage ration is larger than 6 percent, the hit rates of $CR^{1/3}$ and $CR^{1/2}$ are slightly higher than those of the other functions.

WEC's performance is insensitive to the CR mapping function implemented in the proactive eviction module. Generally, we recommend to implement $CR^{1/3}$ and $CR^{1/2}$ in the proactive eviction module; these two mapping functions allow WEC to exhibit good write efficiency and high cache hit rate in most cases. Please note that the $CR^{1/2}$ function is employed as a proactive eviction threshold function throughout all the other experiments in our study.

5.5 SSD Inner Writes Estimation

When it comes to SSDs, write amounts on physical flash chips (a.k.a., inner writes) are usually larger than the size of requests issued by users (a.k.a., external writes). This phenomenon is called write amplification of SSD (See Section 6 for details). Importantly, such inner write amounts on physical flash chips directly determine the endurance of SSD products.

In general, a random write pattern leads to a relative large write amplification rate (WAR) (i.e., inner writes divided by external writes) caused by many extra inner valid data migrations during garbage collection. For example, Saxena et al. [35] discovered that the SSD cache's write amplification rate of typical read-intensive applications has a value ranging from 2.03 to 2.23; their proposed a new technique to reduce WAR down to 2.02$\sim$2.18. In contrast, a sequential write pattern gives rise to a small WAR close to 1.

Expect for FIFO and L2ARC performing a sequential write pattern to SSD cache, all the other caching schemes lead to a random write pattern. In previous experiments, accurate SSD external write amounts are used to calculate write efficiency of all caching schemes. Accurate SSD inner writes can not be obtained here; nevertheless, we apply typical WAR of sequential and random write patterns to estimate inner write amounts. We make use of the inner write amounts to derive inner write efficiency (IWE), which is equal to total hits divided by total estimated inner write amounts.

Fig. 19 shows the IWE values of FIFO, LIRS, SieveStore +LRU, L2ARC and WEC under three workloads of Filebench [44] (i.e., *randomfileaccess*, *networkfs*, and *randomread*). In this group of experiments, SSD write amplification rate ranges from 1.0 to 2.3. Thanks to the sequential-write nature of FIFO and L2ARC, these two schemes achieve high IWEs near the left side of the curves. The other random-write based schemes obtain high IWEs near the right side of the curves. These results indicate that FIFO outperform LIRS in IWE in the *networkfs* and *randomread* workload; L2ARC outperforms SieveStore under the *randomfileaccess* workload. The results also show that the IWE value of WEC remains approximately one order of magnitude higher than those of SieveStore and L2ARC, and about two orders of magnitude higher of those of FIFO and LIRS.

Furthermore, much attention has been paid to reducing the write amplification rate of inner SSDs [32], [34], [35]. The IWE advantage of WEC over FIFO and L2ARC is expected to be more pronounced when WEC is seamlessly integrated with these write-amplification reduction techniques applied in SSD products.

6 Related Work

Flash-Based SSD Caching. Flash-based SSD caching is a promising solution for boosting today's storage systems. A classical technique proposed by Kgil et al. [22] is a two-level cache composed of RAM and a flash memory secondary cache, where the flash-based disk cache is spitted into separate read and write regions to improve I/O performance.

In addition to the cache replacement schemes discussed earlier in this paper, many other approaches are focused on a wide range aspects of flash cache. For example, Koller et al. [23] developed host-side flash-based cache for optimizing accesses to network storage through new cache policies to obtain a better tradeoff point among performance, data consistency and staleness than traditional write-back and write-through polices. Bonfire [24] was designed to significantly reduce large flash cache's warmup time. In addition, Lee et al. proposed a new in-place commit technique that merges write buffer and journal into one, thereby reducing the time and space overhead of logging by utilizing the non-volatile feature of new cache devices [25].

The Endurance Challenge of SSDs. Flash-based SSDs have very limited total bytes written (TBW) in their entire life cycle [4]. This deficiency is mainly caused by the following two factors. First, to pursue high storage density and reducing manufacturing cost, SSD vendors have widely adopted the technique of storing multiple bits in one unit, forcing erase cycles of per cell to drop from 100,000 to 5,000 or even lower [5]. Second, due to the inherent write amplification phenomenon of flash chips, actual write sizes are likely to be much larger than requested ones. Write amplification is triggered by the mismatch of erase and read/write operation units [6] as well as the extra migrations of valid data on to-be-erase blocks.

System-Level Enhancement on Endurance. Apart from cache replacement schemes aiming to reduce the amount of written data (see, for example, SieveStore [38] and L2ARC [37]), there are approaches to improve the endurance of SSDs that play other types of roles in storage systems. For instance, FlashVM [26] and SSDAlloc [27] aim to reduce the number of writes issued to SSDs in flash-based virtual memory systems. I-CASH [28] arranges SSDs to store seldom-changed and mostly read reference data, whereas a HDD stores a log of changed deltas of SSD data. In this case, SSDs does not handle random writes. Ren et al. [29] introduced a new caching algorithm that exploits content locality of I/O requests to reduce SSD cache's write pressure.

There are also some designs focusing on improving the reliability of the whole storage systems with a full consideration of the limited endurance of SSDs. Balakrishnan et al. [30] proposed Diff-RAID, in which SSDs do not wear out at the same time in SSD-based RAIDs. An age differential in an array of SSDs is created in Diff-RAID to ensure data safety by replacing old devices with new ones stepwise. Lee et al. [31] introduced a dynamic throttling technique called READY to enlarge SSDs' lifetimes by directly limiting the number of writes.

Device-Level Enhancement on Endurance. Optimizing the architecture and algorithms inner SSD devices is another important family of techniques to reduce write amplification and improve endurance. Oh et al. [32] implemented a method to balance cache space and over-provisioned space reserved for garbage collection of inner SSDs, thereby extending SSD lifetimes. A hybrid storage device that uses disks as a write buffer for SSD was designed by Soundararajan et al. [33] to reduce the number of writes issued to SSDs. CAFTL [34] reduces redundant data in similar pages at the FTL level to decrease the amount of written data in SSDs. And FlashTier [35] was proposed as a dedicated flash device architecture with an interface purposely designed for caching; FlashTier improves I/O performance and reduces erase cycles.

7 Conclusions

It is a great challenge to deploy SSDs as cache disks in storage systems, because frequently updating cached data can quickly wear these SSDs out. In this study, we designed a new solution—write-efficient caching—to improve write durability of SSD-based caches. At the heart of WEC is a new *pull mode* that proactively discards useless contents and pulls new write-efficient data into SSD-based caches. The pull mode not only guards against cached write-efficient data's too early eviction, but also prevents low-value-added data from being frequently pushed into SSD caches. The pull mode allows WEC to achieve high hit rates and to reduce the amount of data written to SSDs.

Our extensive experiments using real-world workloads demonstrate that (1) WEC's write efficiency is one or two orders of magnitude higher than those of the existing caching schemes; and (2) WEC maintains similar or sometimes even higher cache hit rates than those of the existing solutions. One of the salient features of WEC is that it can be applied in a wide range of computing environments to significantly extend SSD lifespans. For example, when it comes to video-on-demand applications powered by LIRS and L2ARC, WEC extends the lifespan of Intel 910 enterprise-class SSDs from 60.9 and 447.3 days to 1571.6 days (i.e., 4.3 years).

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (Nos. 61202115, 61440057, 61363019, 61272087 and 61073008), the open research program of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Science (No. CARCH201302), the National High-tech R&D Program of China (863 Program) (Grant No. 2013AA013204), Beijing Natural Science Foundation (Nos. 4082016 and 4122039), and the Sci-Tech Interdisciplinary Innovation and Cooperation Team Program of the Chinese Academy of Sciences. Xiao Qin's work is supported by the U.S. National Science Foundation under Grants CCF-0845257 (CAREER), CNS-0917137 (CSR), CNS-0757778 (CSR), CCF- 0742187 (CPA), CNS-0831502 (CyberTrust), CNS-0855251(CRI), OCI-0753305 (CI-TEAM), DUE-0837341 (CCLI), and DUE-0830831 (SFS). The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001 and NSF grant ACI-1339745. The corresponding author is Zhihui Du (E-mail: duzh@tsinghua.edu.cn).

References

[1] W. Zhang. (2011, Nov.). The Object Storage and CDN System of Taobao.com. [Online]. Available: http://www.slideshare.net/wensongzhang/taobaocdn-v2-6532753

[2] J. Yarow. (2010, Jun.). Videos on Youtube grew 123% year over year, while Facebook grew 239%. [Online]. Available: http://www.strangelove.com/blog/2010/06/videos-on-youtube-grew-123-year-over-year-while-facebook-grew-239

[3] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from internet services," in *Proc. 20th IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2014, pp. 488–499.

[4] S. Boboila and P. Desnoyers, "Write endurance in flash drives: Measurements and analysis," in *Proc. 8th USENIX Conf. File Storage Technol.*, Feb. 2010, p. 9.

[5] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proc. 8th USENIX Conf. File Storage Technol.*, Feb. 2012, p. 2.

[6] A. Rajimwale, V. Prabhakaran, and J. D. Davis, "Block management in solid-state devices," in *Proc. 2009 Conf. USENIX Annu. Tech. Conf.*, 2009, p. 21.

[7] Intel Corporation. (2012, Jun.). Intel solid-state drive 910 series: Product specification. [Online]. Available: http://www.intel.com/content/www/us/en/solid-state-drives/ssd-910-series-specification.html

[8] Micron. (2012, Aug.). Maximizing throughput-micron realssdtm p300 solid state drives. [Online]. Available: http://www.micron.com/~/media/Documents/Products/Product%20Flyer/ssd_p300_flyer.pdf

[9] Samsung. (2012, Aug.). Samsung PM830 solid state drive. [Online]. Available: http://www.samsung.com/us/business/oem-solutions/pdfs/PM830_MCE_Product%20Overview.pdf

[10] EMC. (2011, Oct.). EMC fast cache: A detailed review. [Online]. Available: http://www.emc.com/collateral/software/white-papers/h8046-clariion-celerra-unified-fast-cache-wp.pdf

[11] Mark Woods. (2010, Aug.). Optimizing storage performance and cost with intelligent caching. [Online]. Available: http://www.netapp.com/us/system/pdf-reader.aspx?m=wp-7107.pdf&cc=us

[12] Oracle. (2013, Jan.). Exadata smart flash cache features and the oracle exadata database machine. [Online]. Available: http://www.oracle.com/technetwork/server-storage/engineered-systems/exadata/exadata-smart-flash-cache-366203.pdf

[13] Facebook Flashcache. (2013, Feb.). [Online]. Available: https://github.com/facebook/flashcache
[14] Linux bcache. (2013, Feb.). [Online]. Available: http://bcache.evilpiepirate.org
[15] Linux dm-cache. (2013, Feb.). [Online]. Available: http://en.wikipedia.org/wiki/Dm-cache
[16] S. Podlipnig and L. Boszormenyi, "A survey of web cache replacement strategies," *ACM Comput. Surveys*, vol. 35, no. 4, pp. 374–398, Dec. 2003.
[17] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. ACM SIGMETRICS*, Jun. 2002, pp. 31–42.
[18] N. Megiddo and D. Modha, "ARC: A self-tuning, low over-head replacement cache," in *Proc. 2nd USENIX Symp. File Storage Technol.*, Mar. 2003, pp. 115–130.
[19] J. T. Robinson and N. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.*, 1990, pp. 134–142.
[20] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proc. ACM SIGMOD Conf.*, 1993, pp. 297–306.
[21] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 7, pp. 505–519, Jul. 2004.
[22] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," in *Proc. 35th Int. Symp. Comput. Arch.*, Jun. 2008, pp. 327–338.
[23] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proc. 11th USENIX Conf. File Storage Technol.*, Feb. 2013, pp. 45–58.
[24] Y. Zhang, G. Soundararajan, M. Storer, L. Bairavasundaram, S. Subbiah, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Warming up storage-level caches with bonfire," in *Proc. 11th USENIX Conf. File Storage Technol.*, Feb. 2013, pp. 59–72.
[25] E. Lee, H. Bahn, and S. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol.*, Feb. 2013, pp. 73–80.
[26] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2010, pp. 187–200.
[27] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM memory management made easy," in *Proc. 8th USENIX Conf. Netw. Syst. Design Implementation*, 2011, pp. 211–224.
[28] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of SSD and HDD," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 278–289.
[29] J. Ren and Q. Yang, "A new buffer cache design exploiting both temporal and content localities," in *Proc. IEEE 30th Int. Conf. Distrib. Comput. Syst.*, Jun. 2010, pp. 273–282.
[30] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD reliability," *ACM Trans. Storage*, vol. 6, no. 2, p. 4, Jul. 2010.
[31] S. Lee, T. Kim, K. Kim, and J. Kim, "Lifetime management of flash-based SSDs using recovery-aware dynamic throttling," in *Proc. 10th USENIX Conf. File Storage Technol.*, Feb. 2012, pp. 327–340.
[32] 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 *Proc. 10th USENIX Conf. File Storage Technol.*, Feb. 2012, pp. 313–326.
[33] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," in *Proc. 8th USENIX Conf. File Storage Technol.*, Feb. 2010, pp. 101–114.
[34] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. 9th USENIX Conf. File Storage Technol.*, Feb. 2011, pp. 77–90.
[35] M. Saxena, M. Swift, and Y. Zhang, "FlashTier: A lightweight, consistent and durable storage cache," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, Apr. 2012, pp. 267–280.
[36] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems," *ACM Trans. Storage*, vol. 4, no. 2, p. 4, May 2008.
[37] Oracle Corporation. (2011, Aug.). Deploying hybrid storage pools with oracle flash technology and the oracle solaris ZFS file system—An oracle white paper. [Online]. Available: http://www.oracle.com/technet-work/server-storage/archive/o11-077-deploying-hsp-487445.pdf
[38] T. Pritchett and M. Thottethodi, "SieveStore: A highly-selective, ensemble-level disk cache for cost-performance," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 163–174.
[39] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng, "Improving flash-based disk cache with lazy adaptive replacement," in *Proc. IEEE 29th Int. Conf. Massive Storage Syst. Technol.*, May 2013, pp. 1–10.
[40] University of Massachusetts. (2009, Dec.). UMASS Trace Repository. [Online]. Available: http://traces.cs.umass.edu/index.php
[41] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Proc. IEEE Int. Symp. Workload Characterization*, Sep. 2008, pp. 119–128.
[42] M. Blaze, "NFS tracing by passive network monitoring," in *Proc. USENIX Winter 1992 Tech. Conf.*, 1992, pp. 333–334.
[43] J. Luo, Q. Zhang, Y. Tang, and S. Yang, "A trace-driven approach to evaluate the scalability of P2P-based video-on-demand service," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 1, pp. 59–70, Jan. 2009.
[44] Filebench. (2013, Nov.). [Online]. Available: http://sourceforge.net/projects/filebench

Yunpeng Chai received the BE and PhD degrees in computer science and technology from Tsinghua University in 2004 and 2009, respectively. Since 2009, he has been an assistant professor at the College of Information, Renmin University of China. His research interests include emerging storage, cloud storage, and energy conservation of storage systems.

Zhihui Du received the BE degree from the Computer Department, Tianjian University, in 1992, and the MS and PhD degrees in computer science from Peking University, in 1995 and 1998, respectively. From 1998 to 2000, he worked at Tsinghua University as a postdoctoral researcher. Since 2001, he has been an associate professor at the Department of Computer Science and Technology, Tsinghua University. His research interests include high-performance computing and grid computing. He is a member of the IEEE.

Xiao Qin (S'00-M'04-SM'09) received the BS and MS degrees in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 1996 and 1999, respectively, and the PhD degree in computer science from the University of Nebraska, Lincoln, in 2004. He is currently an associate professor of computer science at Auburn University. His research interests include parallel and distributed systems, real-time computing, storage systems, fault tolerance, and performance evaluation. He received an NSF CAREER Award in 2009. He is a senior member of the IEEE and the IEEE Computer Society.

David A. Bader received the PhD degree from the University of Maryland in 1996. He received a US National Science Foundation (NSF) Postdoctoral Research Associateship in Experimental Computer Science. He is a professor at the School of Computational Science and Engineering and executive director of high performance computing at the Georgia Institute of Technology. He has coauthored more than 100 articles in journals and conferences. His main areas of research include parallel algorithms, combinatorial optimization, and computational biology and genomics. He is a fellow of the IEEE and a member of the ACM.