Information Systems 58 (2016) 24–42

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

A fully persistent and consistent read/write cache using flash-based general SSDs for desktop workloads

Sung Hoon Baek [a], Ki-Woong Park [b,*]

[a] Department of Computer System Engineering, Jungwon University, Republic of Korea
[b] Department of Computer Hacking and Information Security, Daejeon University, Republic of Korea

ARTICLE INFO

Article history:
Received 6 April 2015
Received in revised form
10 December 2015
Accepted 4 February 2016
Recommended by: B. Kemme
Available online 12 February 2016

Keywords:
Secondary storage

ABSTRACT

The flash-based SSD is used as a tiered cache between RAM and HDD. Conventional schemes do not utilize the nonvolatile feature of SSD and cannot cache write requests. Writes are a significant, or often dominant, fraction of storage workloads. To cache write requests, the SSD cache should persistently and consistently manage its data and metadata, and guarantee no data loss even after a crash. Persistent cache management may require frequent metadata changes and causes high overhead. Some researchers insist that a nonvolatile persistent cache requires new additional primitives that are not supported by general SSDs in the market. We proposed a fully persistent read/write cache, which improves both read and write performance, does not require any special primitive, has a low overhead, guarantees the integrity of the cache metadata and the consistency of the cached data, even during a crash or power failure, and is able to recover the flash cache quickly without any data loss. We implemented the persistent read/write cache as a block device driver in Linux. Our scheme aims at virtual desktop infra servers. So the evaluation was performed with massive, real desktop traces of five users for ten days. The evaluation shows that our scheme outperforms an LRU version of SSD cache by 50% and the read-only version of our scheme by 37%, on average, for all experiments. This paper describes most of the parts of our scheme in detail. Detailed pseudo-codes are included in the Appendix.

1. Introduction

The advent of flash-based solid-state drive (SSD) has spurred a proliferation of studies on new storage architectures. Flash memory is widely used in mobile phones and embedded systems because it is smaller and more resistant to shock than mechanical storage devices such as hard disk drives (HDD). In addition, SSD that utilizes tens or hundreds of independent flash chips is superior to HDD in terms of bandwidth and response time. Thus, SSD has been replacing HDD in devices ranging from desktop computers to enterprise servers.

The capacity per price of SSD is greater than that of RAM and lower than that of HDD. In the aspect of performance, SSD is slower than RAM but faster than HDD. Especially, HDD exhibits much longer latency for non-sequential requests than SSD due to its mechanical components, while SSD provides a short constant response time regardless of request patterns.

In terms of the performance, the capacity, and the price, SSD is in between RAM and HDD. Hence, various tiered architectures where SSD is used as a second level cache between RAM and HDD have been studied [1,2].

Many researchers have studied the nonvolatile memory (NVRAM) such as phase-change memory and magnetoresistive

* Corresponding author.
Tel.: +82 10 9165 1624; fax: +82 42 280 2404.
E-mail addresses: shbaek@jwu.ac.kr (S.H. Baek), woongbak@dju.kr (K.-W. Park).

http://dx.doi.org/10.1016/j.is.2016.02.002
0306-4379/

RAM as an intermediate tier between RAM and HDD [3–6]. The byte-addressing feature of NVRAM makes persistent cache management easy but SSD is not byte-addressable. SSD, however, is superior to NVRAM in terms of performance per price and has been used as a cache below RAM in commercial storage systems [7–9].

Approaches to use the raw flash memory as a tiered cache have been introduced [10,11]. Consequences of the drive to lower the price and increase the capacity of flash memory, have been that the reliability of flash cells continues to diminish and a variety of error patterns has arisen. The flash architecture turned into 3D NAND from planar NAND [12], and its error types greatly changed. The manufacturers have investigated various solutions for these errors and keep them secret for market advantage. It is difficult for a third party company to manage the low-level flash memory. However, SSD provides us an error-free interface; thus, it can easily be managed as a tiered cache.

Many studies on databases that use flash SSDs as a mid-tier cache have been carried out [13–15]. Canim et al. introduced a read cache replacement algorithm that estimates I/O costs between SSD and HDD for each region to determine which region is best to be promoted to SSD [14]. Do et al. showed that a write cache can give a significant gain to database systems [15]. However, a cache metadata management scheme that has no data staleness is needed for database systems to adopt a write cache.

1.1. Persistency and consistency

The conventional second-level cache schemes for flash-based SSDs are variations of traditional RAM-based cache policies. Hence, they do not utilize the nonvolatile feature of SSD. Cached data in SSD cannot be used at the next restart because they store metadata in RAM for fast processing. Data stored in SSD does not volatilize but the data cannot be used after a restart because the metadata in RAM is lost and we cannot know which sector the cached data is related to [1,2,8,16–18].

The capacity of SSD as the second level cache is much bigger than the main memory by two or three orders of magnitude. The conventional second cache schemes ignore all data stored in SSD at a restart and require too long time until the SSD cache is filled with data at the rate of application I/O. This process takes several hours or even days to fill the SSD.

A warm-start scheme was proposed [19] that would shorten the restart time of the storage-class second level cache. It makes a log of warmup data to fill the second level cache at the next boot time. However, this scheme still requires at least several hours to fill the storage-class cache.

A cache is persistent if its cache data is immediately reusable after a power failure. The proposed system is persistent, does not need a warming process, and has a low overhead. The cache metadata and data are stored in a nonvolatile device such as SSD and is consistent without data loss even after a crash or a power failure.

1.2. Write cache and overhead

“Writes are significant, or often dominant, fraction of storage workloads” [20–23], thus, a write cache can greatly improve the write performance. The traditional caches do not safely manage their cache metadata during a crash, so they cache only read requests and cannot retain dirty data, even though the cache device is nonvolatile.

Most tiered cache technologies use a write-through policy that prohibits dirty data in SSD. In other words, a write request invalidates a block that is cached in SSD and is directly delivered to HDD, thus all the newest data are stored in HDD. This means that conventional technologies utilize SSD as a read-only cache that cannot improve the write performance.

Even though SSD is a nonvolatile device, it is very difficult to apply the write-back policy to the tiered cache. To persistently maintain dirty data in SSD, cache metadata must be stored in a nonvolatile device and consistently updated whenever a block is evicted or cached. In addition, cached data and cache metadata must not be lost and be consistent even at a crash. A persistent cache that employs dirty data may require a high real-time overhead for consistent metadata management.

A persistent read/write cache improves both read and write performance but it must also guarantee the integrity of the cache metadata and the consistency of the cached data even during a crash or a power failure.

Saxena et al. [24] proposed a durable (same as persistent) write cache with a small overhead. They insisted that storage provide new primitives (write-dirty, write-clean, evict, clean, exists) to consistently manage a write cache with a low overhead. However, general SSDs, though they support only the basic primitives, read and write; have the benefits of fast development time, low cost, and popularity.

An approach using general SSDs was introduced [25]. It can cache write data and does not return stale data but may restart with an empty cache if a crash occurred while updating its cache metadata.

The system proposed herein is a fully persistent read/write cache with low overhead, it never return stale data, and it utilizes general SSDs that are now available in the retail market.

1.3. Lossless recovery and no-write-back

Koller et al. [20] introduced a write-back policy for an SSD cache, which is journaled write-back that makes a block-level journal that aggregates multiple write requests with a header block and a commit block, while limiting the amount of data loss. The journaling cache quickly recovers by replaying the last completed transaction to its home location. However, it cannot provide a recovery point objective (RPO) of zero (i.e., no data loss). It loses the last journaled data if a crash occurs in the middle of journaling.

Holland et al. [26] investigated flash write policies, which are write-through, asynchronous write-through, and periodic write-back. The write-through policy does not permit dirty data on flash. The other write policies may return stale data after a crash or a power failure. Some SSD caches use the write-back policy for the write performance by sacrificing a risk of data loss [8,23,27]. However, our solution never loses dirty data for highly available systems.

The traditional write-back policy flushes dirty data by a periodic write-back scheduler [28] or a write-barrier [29] of journaling file systems. The non-persistent write cache cannot retain many dirty data for a long time because it has a possibility of data loss.

In contrast, the proposed persistent read/write cache can adopt a no-write-back policy, which can permanently retain dirty data in the cache and does not flush dirty data because it does not lose dirty data even if a crash occurs. This policy flushes a dirty block to the backing storage only when the dirty block is evicted from the persistent cache.

If a crash or power failure occurs, the write cache has to recover the data and cache metadata. Recoverability is measured by the amount of data loss and recovery time that a recovery process takes until the whole data is available again.

Fig. 1 shows trade-offs in write policies. A write-back policy will incur data staleness [20]. The write-back policy accumulates the dirty pages in the SSD cache and writes these pages to the HDD in sequential order [14]. This policy loses all accumulated dirty pages when a crash occurs. If it accumulates fewer dirty pages, it returns less stale data after a crash, but its performance falls toward that of the write-through policy.

The performance of the write-back policy would be better or worse than that of the no-write-back-policy depending on the number of accumulated dirty pages. The journaled write-back policy [20] is better than the normal write-back policy in terms of consistency, but it also loses dirty data in an incomplete transaction that has no commit block.

The most important advantage of the proposed no-write-back policy against the write-back policy is the write cache with *zero-staleness*.

The proposed system quickly recovers the flash cache without any data loss and employs the no-write-back policy that does not flush dirty data to the backing storage until the dirty data is evicted for capacity reasons. This provides sustained high write performance and reduces the bandwidth consumed by writing back dirty data.

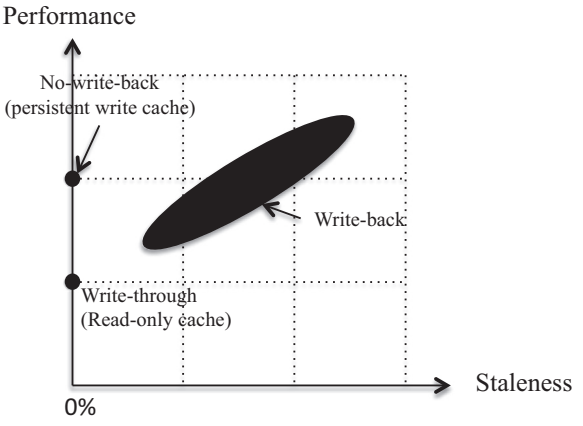

Fig. 1. Trade-offs in write caching: a write-back policy will incur data staleness. The no-write-back (persistent write cache) and write-through policies support zero staleness.

1.4. Memory usage and workload

The memory usage of the cache management is proportional to the cache size. The memory usage of the storage-class cache cannot be ignored. A page-based LRU scheme uses 512 MiB of memory for 64 GiB SSD. The proposed scheme adopted segment-based tables that require 21.5 MiB in our implementation.

The proposed system focuses on desktop workloads. Anti-virus scan, disk defragmentation, and copying video files can invalidate most of the cache. Recent activities of such processes may not predict the future desktop I/O. Our system considers desktop workloads and analyzes recency and frequency for a long period. Our scheme is suitable for desktop computers and virtual desktop infra servers [30], which run virtual machines to deliver the desktop environment to the remote clients.

2. Persistent read write cache

Our cache management policy is quite different from traditional cache policies. We propose a storage-class, persistent, and consistent read/write cache (PRWC), which (1) can retain dirty data in SSD with zero RPO (no data loss) even after a crash or a power failure; (2) has a small overhead for cache consistency; (3) does not need a warm-start; (4) does not need a new SSD interface; (5) considers both recency and frequency in its cache replacement policy; (6) requires a small amount of main memory for cache management; and (7) recovers the cache metadata a short time after a crash, thereby supporting a storage-class cache.

PRWC learns and analyzes I/O accesses for a long-term period. Updating the cache content is periodically performed during idle time after long-term learning. Hence, this policy may decrease the hit rate in comparison with on-demand cache policies; however on-demand cache replacement policies consume a considerable portion of the SSD bandwidth. The SSD bandwidth is much less than that of RAM by an order of magnitude (SATA III is 600 MB/s; PC3-17000 is 17066 MB/s). The second-level storage cache must consider not only hit rate but also throughput, which include bandwidth, fetching cost, eviction cost, and access pattern. The bandwidth of memory-level SSDs such as PCIe SSDs or memory channel storage devices (ULLtraDIMM SSD, eXFlash DIMM) is very close to that of RAM. However, we focus on storage-level SSD. Our evaluation shows that the reduced fetching cost (SSD bandwidth) compensates for the decreased hit rate in PRWC.

PRWC does not change the cache metadata during most of the running time because it completes cache updates during idle time after long-term learning. For perfect persistency and consistency during the cache-update procedure, PRWC employs a journaling scheme for the cache metadata. The changed parts of the cache metadata are appended as a log in SSD. When the log region is full or the cache-update procedure finishes, the main cache metadata is combined with all of the logs and is written as the latest one.

PRWC also does not use a special primitive (e.g., atomic write or transactional write). We established consistency using only the basic read and write primitives that are

supported by general SSDs. The cache update policy of PRWC can aggregate multiple logs into a single write, thus it has a low update overhead.

PRWC manages the SSD cache in segment unit that divides the storage volume, is the minimum unit for the cache management, and ranges tens of KiB to several MiB. If a segment becomes hot, the segment is brought from the backing storage to SSD.

There are many cache replacement policies that considers both recency and frequency for RAM cache, including LRFU [31], ARC [32], CAR [33], and L2ARC [18]. However, these schemes make persistency management very complicated. A way to easily maintain consistency and persistency is needed. PRWC is managed with minimal information (an indicating value for recency and frequency, mapping information) in a form of table. Each segment has a value that is an indicator representing the recency and frequency of I/O accesses to it.

If a crash occurs in the middle of the cache metadata update, the next restart reads only the metadata area and reconstructs the cache metadata in a short time without any data loss.

In this paper, we present the detailed algorithm, structure, and codes of the persistent read/write cache. We implemented PRWC as a module driver in Linux. The Appendix includes the detailed pseudo-codes of the implementation.

2.1. Addressing

Fig. 2 shows the proposed structure of the tiered storage that uses an SSD or SSD RAID [34–36] as the second level storage cache. A single HDD or HDD RAID [37,38] is the backing storage device.

The storage volume is divided by segment unit, which is the minimum unit for cache management and ranges from 128 KiB to 1 MiB in the evaluation. The frequently and recently used segments (hot segments) of the main storage are mapped to SSD.

Each segment in the HDD has an RF value that indicates the level of recency and frequency. The HDD segment whose RF value is greater than a threshold value is cached to SSD. A mapping table that maps SSD segments to HDD segments is managed in both SSD and main memory. A radix tree in main memory provides mapping from HDD segments to SSD segments.

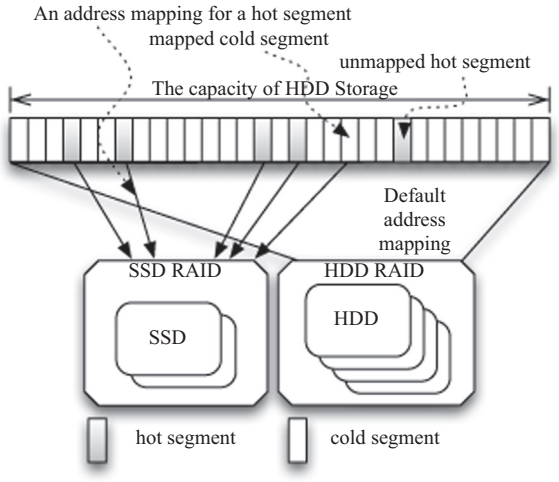

Fig. 2. Segment mapping.

If an I/O request is delivered, the system investigates whether the requested HDD segment is mapped to an SSD segment using the radix tree. If the requested HDD segment is mapped to an SSD segment, the request is redirected to the SSD segment; otherwise, the request is delivered to the HDD.

2.2. Write cache policy – no-write-back

A segment where reads and writes are frequently and recently requested becomes a hot segment and is promoted to SSD at the next cache-update procedure. The deferred caching may be a shortcoming, but is compensated by the following strong points.

Requests for cold segments are not cached in SSD. Requests only for cached segments are redirected to SSD. If all write requests are cached regardless of the segment temperature, data written in cold segments must be evicted in the near future, which requires one more write and read, thereby consuming the limited SSD bandwidth.

This write-back policy causes data synchronization from SSD to HDD in the near future, to reduce the amount of data loss [20]. Our write policy is no-write-back. PRWC is fully persistent and flushes dirty data only when the dirty data is evicted, because this guarantees no data loss. In addition, it does not cause frequent evictions because hot segments are determined by long-term learning.

A read-only cache invalidates a cached region where a write is delivered, whereas PRWC just redirects a write or read request to SSD when a write or read request for a hot segment is delivered.

The proposed cache is resistant to cache pollution. A recently accessed data segment is valuable in the least recently used (LRU) cache, but a recently first accessed data segment is not hot and is not cached in PRWC. A big data backup pollutes the LRU cache, whereas PRWC protects hot data against cache pollution.

The write-back scheme of the RAM cache buffers all write requests, aggregates multiple requests into a single request, and reorders them for an optimal seek time. However, buffering all write requests is less beneficial to the SSD cache because the aggregation and reordering is already performed in the RAM cache.

2.3. Caching all writes

PRWC does not buffer all writes. Write requests only for cached segments are redirected to SSD. Writes for cold segments are directly written to HDD. If a system supports a write cache for all writes and long and continuous I/Os are requested, the write cache capacity become full eventually and a new cold data should evict the oldest cold data. Caching a cold data and eviction another cold data consume considerable SSD bandwidth. Therefore, caching all writes is worse than bypassing writes for cold data.

To cache all writes, we may add the write-back journaling for uncached segments as in Koller's scheme [20]. This would be useful for a sporadically busy system. Journaling may slowdown a long-term busy system after the journaling region becomes full. A very busy system may not have idle time sufficient to move the cold data in a journal region to their home location, and would consume the limited SSD bandwidth for cold data. PRWC does not presently include the write-back journaling.

2.4. Cache metadata

PRWC should preserve data even after a power failure or a crash, and it does this by storing the cache metadata in a nonvolatile device such as an SSD. If a system modifies the metadata whenever the cache is updated, it suffers severe performance degradation. Hence, we need a completely different cache policy.

To reduce the update overhead of the cache metadata, the proposed scheme learns the recency and frequency of segment accesses for a given period, which could be a couple of hours or a day. Updating the contents of the cache is scheduled for idle time and is suspended any time the system is not idle.

A long-term cache update may be adequate for a storage-class cache for the following two reasons. First, there are two common access patterns across various applications: daily re-accesses and hourly re-accesses [19]. Second, it takes several hours or days for workloads to exceed the capacity of the SSD cache. Even a rate of one update per day was superior to an on-demand cache policy in our performance evaluation.

Fig. 3 shows the metadata structure of the proposed system. The metadata is stored in SSD. There are two metadata regions for integrity. The metadata is stored alternatively in the two metadata regions. If a crash occurs while writing a main metadata to one region, the latest main metadata is corrupted. However, we can then use the metadata in the other metadata region, which still contains intact metadata. A metadata region consists of a metadata header, main metadata, and log area.

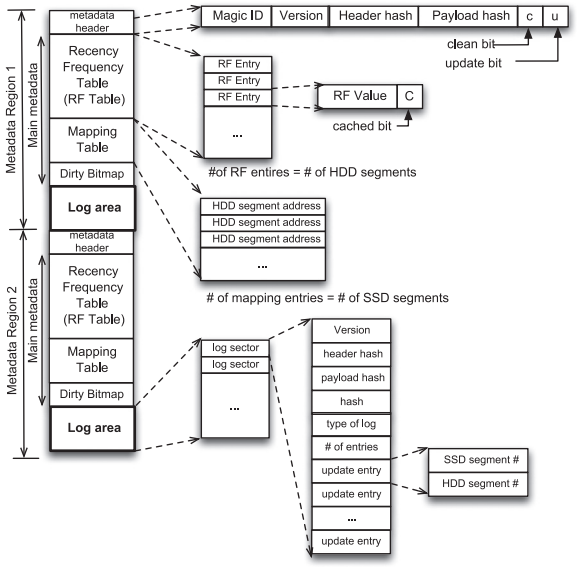

Fig. 3. Metadata structure: the cache data is persistently managed by the metadata saved in a nonvolatile device (SSD).

The version field of the metadata header indicates the latest main metadata and increases by one after every main metadata update. The header hash field guarantees the integrity of the metadata header. The payload hash checks errors of the main metadata. The cleared clean bit indicates a crash. The update bit informs whether the crash occurred in the middle of the cache update.

The recency/frequency table (RF table) consists of RF entries allocated to each HDD segment. The *RF value* is the index of recency and frequency. The greater *RF value* indicates that the HDD segment has been used more recently and frequently. The cached bit indicates if the corresponding HDD segment is cached in SSD.

Each entry of the mapping table corresponds to a SSD segment. If the corresponding SSD segment caches an HDD segment, the entry contains the address of the cached HDD segment. Otherwise, it contains NULL.

Each bit in the dirty bitmap is allocated to an SSD segment to indicate whether the cached data in the SSD segment is dirty.

Another copy of the mapping table and the RF table is in memory to lookup the metadata quickly (called the in-memory metadata). A radix tree and a lock table are only in memory. Fig. 4 shows the in-memory metadata.

The radix tree is used to search for the SSD segment where a given HDD segment is cached. Namely, the radix tree is a reverse mapper of the mapping table. The radix tree can be rebuilt from the mapping table, thus is not stored in the metadata region.

The lock table is used for exclusive access to the metadata. Each lock entry corresponds to an HDD segment. When an I/O is requested to a segment, the corresponding lock-counter increases by one. When the request finishes, the counter decreases by one. While caching or evicting a segment, the corresponding update lock bit is set. For each lock entry, all I/Os to the segment are blocked while the update lock bit is set. Caching and evicting the segment is prohibited while the lock counter is non-zero.

The cache update procedure locks a segment that is being promoted or evicted. Then, any I/O access to the locked segment is blocked until the segment is unlocked. While an I/O to a segment is being processed, the cache update procedure is blocked from accessing the segment until the I/O finishes.

I/Os can be requested during the cache update procedure even though this is very rare. Hence, we need these locks to prevent a race condition in which I/O requests and the cache update procedure compete with each other for the cache metadata.

The cache update procedure locks only one segment at a time, so the possibility that an I/O request is for the locked segment is very low. If the I/O request has a uniform distribution and there are 1 million of segments, the possibility of blocking is 0.00001% only when the cache update procedure runs. In addition, there is no locking in most cases.

SSD is better to store the metadata region with high performance than HDD but the metadata region may reduce the effective cache size a little. Table 1 shows the

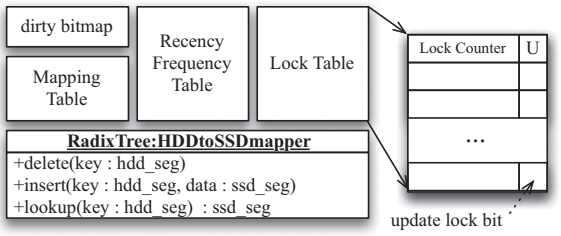

Fig. 4. In-memory metadata: it is used to quickly lookup the metadata.

Table 1
The size of the metadata region (MiB).

HDD size (TiB)	SSD size (GiB)			
	8	16	32	64
1	6.2	6.3	6.4	6.6
2	8.3	8.4	8.5	8.8
4	12.6	12.6	12.8	13.0
8	21.1	21.1	21.3	21.5

size of the metadata region. The HDD size dominates the size of the metadata region but the SSD size lightly affects it. We need to avoid from frequently updating the metadata region because it is not small. In our proposed policy, the metadata region updated only once after the completion of the cache update procedure which runs hourly or daily.

2.5. Recency and frequency

The cache replacement policy of PRWC considers both recency and frequency, and is a straightforward way to maintain consistency and persistency. Many researches on recency and frequency have been proposed [31–33,39]. The traditional cache replacement policies consider various aspects of I/Os such as not only recency and frequency but also marginal gain [32,33], and access-pattern-based cost [39,14]. However, these schemes make persistency management very difficult. The traditional cache replacement policies modify their cache metadata for every I/O request. If the metadata is managed in a secondary storage device, it causes a high overhead.

The proposed cache analyzes hourly or daily workloads and gives each HDD segment a recency/frequency value (RF value) that is a combination of the level of recency and the level of frequency. The greater the RF value, the hotter the HDD segment.

For every hit to a HDD segment, the RF value increases by a predefined value (20 is used in the implementation). Successive hits on the same segment are considered a single hit. If the RF value exceeds a predefined hot threshold value (50 is used in the implementation), the corresponding HDD segment becomes a *hot segment*. Otherwise, it is a *cold segment*. The optimal hot threshold value varies according to workloads. Finding the optimal value is not trivial. We plan to study it in future work.

For recency, all RF values decrease by multiplying a decay value (less than one) to them in a decay process that is invoked when the number of hot segments exceeds the capacity of the SSD.

The decay process requires the time complexity of $O(N)$. However, it is rarely called and runs in a dedicated thread that is independent of I/O threads. Thus it does not affect the I/O response time and causes unnoticeable performance degradation.

At the beginning of the empty SSD cache, it takes too long to fill the big SSD cache with hot segments (perhaps several days). To boost the filling process for an empty cache, any HDD segment that is accessed at least once becomes a hot segment until the SSD cache is first fully filled.

2.6. Cache replacement policy

The storage-class cache has properties different from traditional caches. The storage-class cache does not need to frequently evict for a cache miss because it has a huge amount of cache space, and it takes a long time to fill the cache. Various applications exhibit the common patterns of hourly or daily re-accesses in a storage-class cache [19].

The proposed caching and eviction occur only in a cache-update procedure, which is scheduled to run hourly or daily with a low priority at an idle time (e.g., midnight or lunchtime). Any busy I/O can suspend the cache-update procedure at any time.

Traditional cache policies that need an eviction and caching on every cache miss are better in terms of hit rate; however, caching and eviction consumes a considerable portion of the SSD bandwidth unlike for RAM. Our experiment shows that the reduced consumption of SSD bandwidth by the proposed periodic cache update compensates for the decreased hit rate.

The cache-update procedure caches all of *uncached hot segments* to SSD, and evicts *cached cold segments*. Fig. 5 shows exemplary metadata for each step of the decay procedure, and the cache-update procedure with four SSD segments and eight HDD segments.

The decay procedure decreases all RF values in the RF table so that the number of hot segments is less than or equal to the SSD capacity. The system always keeps the number of hot segments below the number of SSD segments.

The RF table and the mapping table form circles like CLOCK. Each entry of the RF table contains its RF value and a cached bit (denoted by '/' in Fig. 5) for each HDD segment. Each entry of the mapping table contains the address of an HDD segment that is cached to the corresponding SSD segment.

At the beginning of the cache-update procedure, the HDD clock starts at zero, rotates clockwise, and then stops at an uncached hot segment. The SSD clock starts from the last position, rotates clockwise, and stops at a victim SSD segment that contains a cached cold segment. The HDD segment that is cached in the victim SSD segment is evicted, and its data is flushed to its original HDD segment, if the segment is dirty. The HDD segment pointed to by the HDD clock is cached to the SSD segment pointed to by the SSD clock. If the HDD clock rotates one revolution, the cache-update procedure saves current in-memory metadata as the main metadata in the next metadata region.

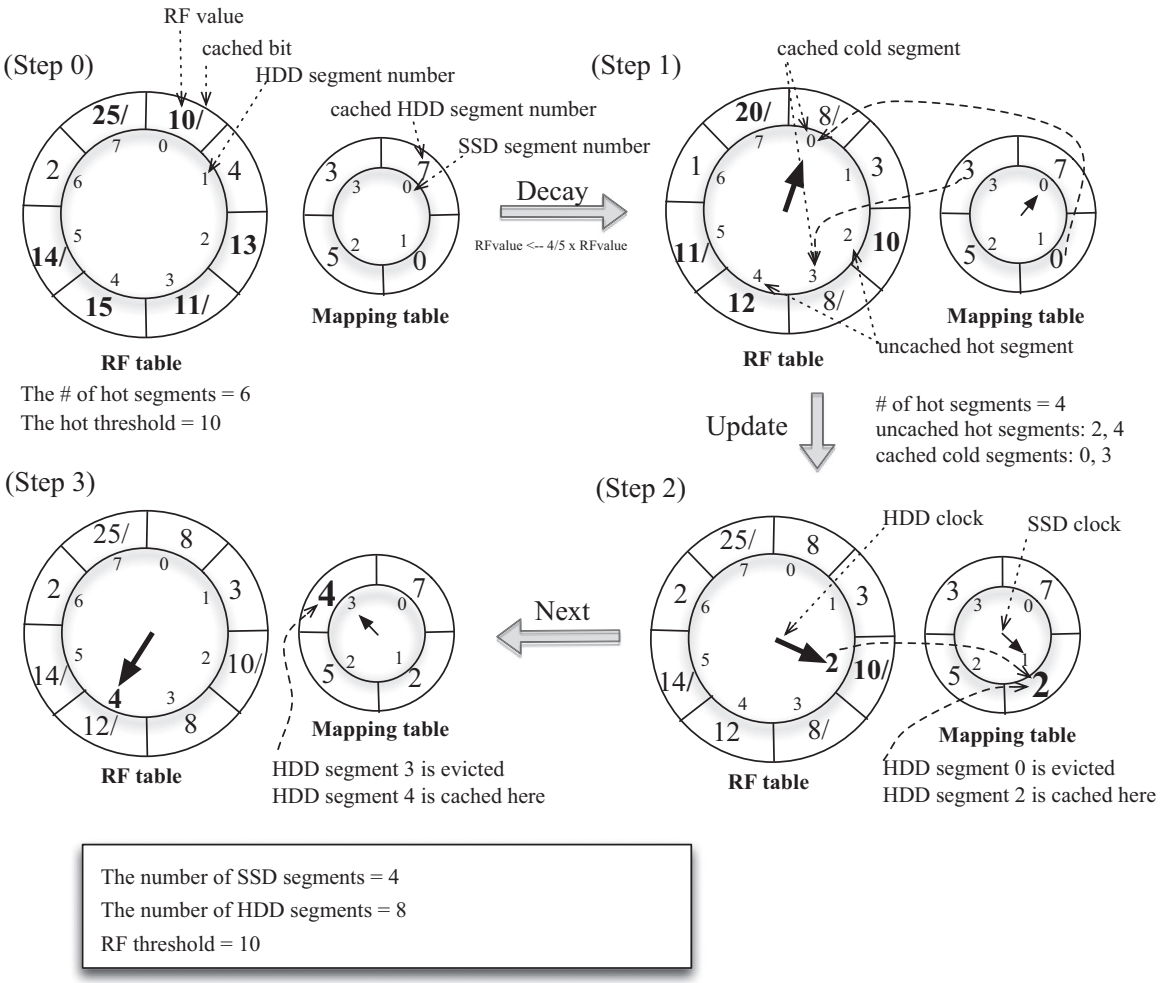

Fig. 5. Exemplary metadata for the decay and cache-update procedures with four SSD segments and eight HDD segments.

In Step 0 of Fig. 5, the number of hot segment (=6) exceeds the number of SSD segments (=4). To reduce the number of hot segments below the number of SSD segments, the decay procedure multiplies every RF values by the decay value (=4/5).

In Step 1 of Fig. 5, the HDD segments '2' and '4' are the uncached hot segments, and the HDD segments '0' and '3' are cached cold segments.

Step 2 of Fig. 5 shows the cache metadata after evicting HDD segment '0' and caching HDD segment '2' in SSD segment '1'. The SSD clock points to '1'. SSD segment '1' contains HDD segment '0'; which is evicted. The HDD clock points to '2'. HDD segment '2' is cached to SSD segment '1'. Step 3 of Fig. 5 shows the cache metadata after evicting HDD segment '3' and caching HDD segment '4' in SSD segment '3'.

2.7. Metadata log and dual metadata regions

A journaling for the metadata is used for persistency and consistency. The logs are produced only during the cache-update procedure. Even if a crash occurs during the cache-update procedure, the metadata can be reconstructed by combining the last saved main metadata with the logs.

PRWC utilizes a logging scheme like database write-ahead logging [15,40,41] and general file system journaling [42–46]. However, PRWC has two metadata regions for higher reliability. Reliability is much important in the cache metadata because a failure on even a single sector in a cache metadata leads to a tremendous amount of corrupted data. If an error occurs in the latest main metadata, we can recover it from the alternate metadata region. In addition, the dual metadata regions make the recovery procedure simple.

Fig. 6 shows how the two metadata regions are used. The main metadata and logs are recorded alternatively in two metadata regions. If a crash occurs while the main metadata is being written, the last metadata is corrupt, but the system can rebuild the latest metadata from the other metadata region.

The main metadata that is in SSD is not changed during the cache-update procedure but the in-memory metadata changes in real time. The changed parts from the main metadata against the in-memory metadata are recorded as logs during the cache-update procedure (Steps 2 and 3 in Fig. 6).

The final main metadata (in-memory metadata) is equal to the combination of the last saved main metadata

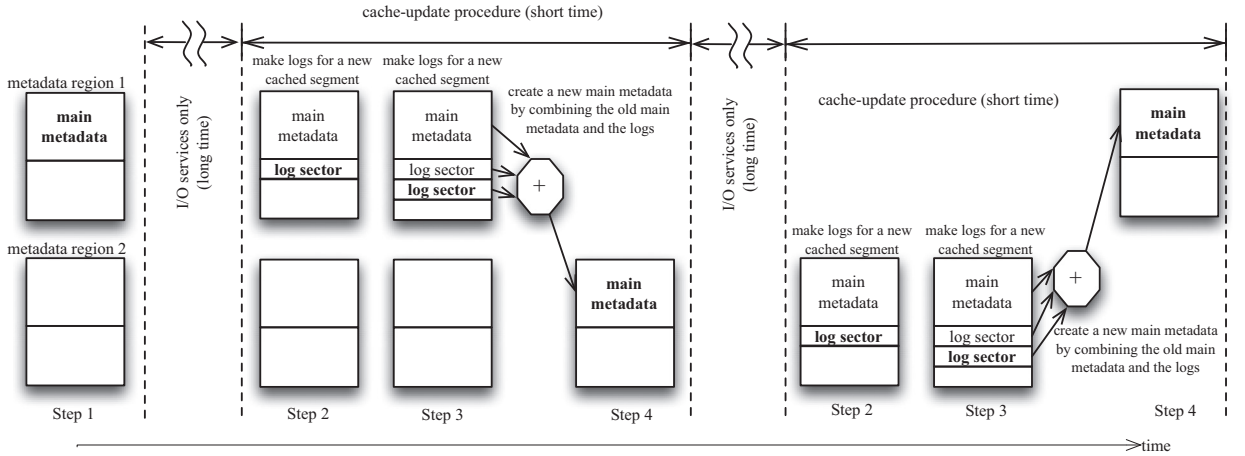

Fig. 6. The main metadata and logs are alternatively recorded in two metadata regions. If a crash occurs in Step 4, the partially written metadata is unusable but the latest main metadata is rebuilt with the other main metadata and the logs from the previous step (Step 3).

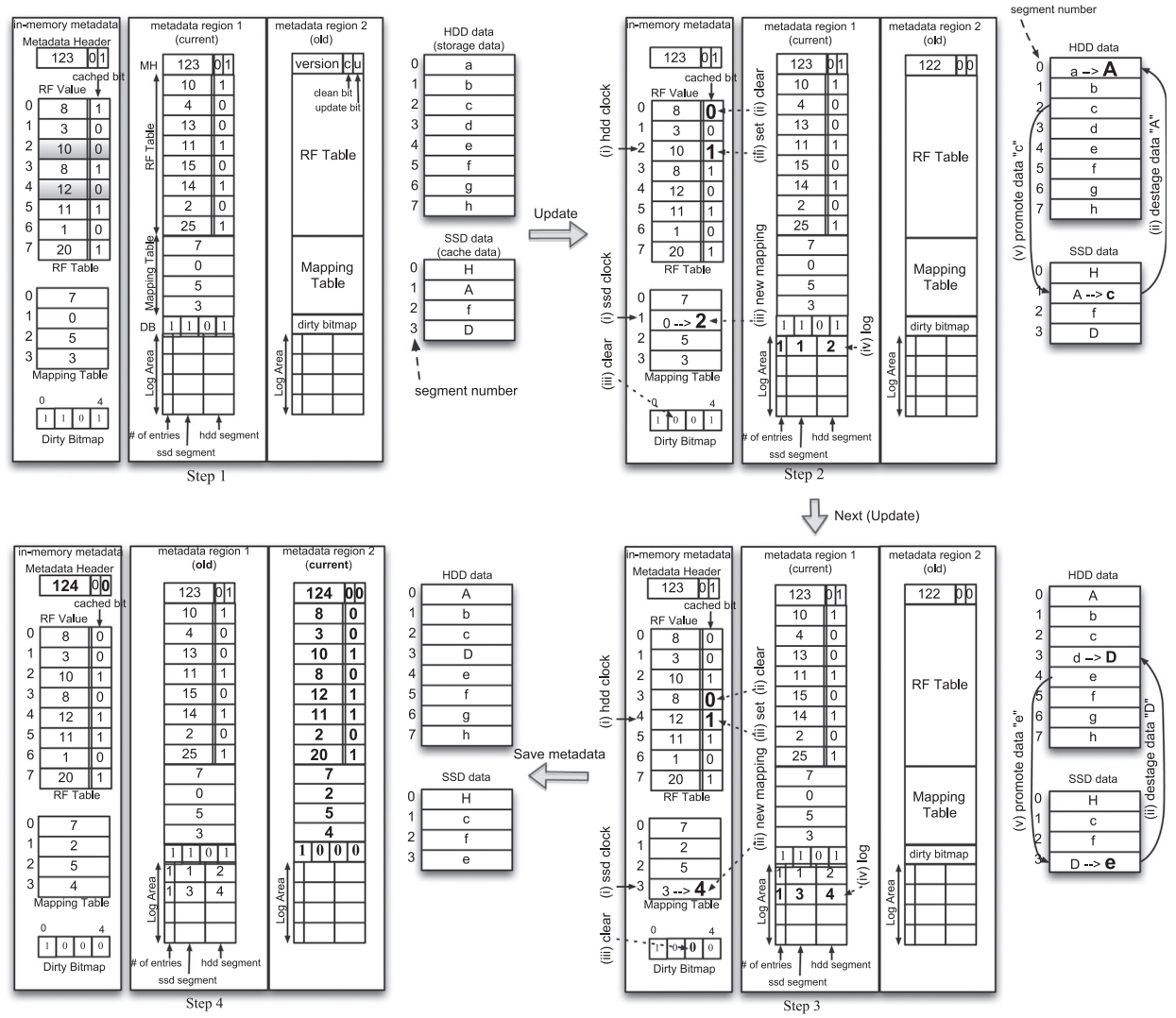

Fig. 7. Overview of the cache-update procedure including in-memory metadata, metadata regions, storage data, and cache data. Steps 1–3, and the exemplary values of this figure are the same as those in Fig. 5.

and the logs, and it is saved in the other metadata region at the end of the cache-update procedure (Step 4 in Fig. 6). If a crash occurs in Step 4, the partially written metadata is unusable; however, the latest main metadata can be rebuilt using the alternate main metadata and the logs of the previous step (Step 3 in Fig. 6).

A log consists of a mapping pair (SSD segment number and HDD segment number), which means that the SSD segment caches the HDD segment. The log sector size is 512 B that is minimum I/O unit of storage devices, but the pair is much smaller than the sector size. Thus we can reduce the logging overhead by aggregating multiple mapping pairs into a single log sector.

Fig. 7 shows the in-memory metadata, metadata regions, storage data, and cache data during the cache-update procedure. Steps 1–3, and the exemplary values of Fig. 7 are the same as those of Fig. 5.

In Step 1 (decay process) of Fig. 7, the metadata region 1 contains the metadata before the decay, but the in-memory metadata is the latest one after the decay process.

The more detailed sub-steps in Step 2 or Step 3 is as follows:

- *Step* i: The system determines an uncached hot HDD segment that is pointed by the HDD clock, and an SSD segment that contains a cold segment (pointed by the SSD clock). This step is safe from a crash.
- *Step* ii: The system copies (flushes) the data in the SSD segment to its home location, if the data in the SSD segment is dirty. Then it deletes the caching information (cached bit, dirty bit, radix tree) related with the SSD segment in the in-memory metadata. This step is safe from a crash.
- *Step* iii: The system makes a new mapping (cached bit, radix tree) from the hot HDD segment (HDD clock) to the SSD segment (SSD clock) in the in-memory metadata. This step is safe from a crash.
- *Step* iv: The system appends a log, ⟨SSD clock, HDD clock⟩, to the log area of the current metadata region.
- *Step* v: The data of the hot HDD segment is copied to the SSD segment.

In Step 2 (caching process), the cached cold HDD segment '0' is evicted from SSD segment '1' and the uncached hot HDD segment '2' is cached to the SSD segment. In the metadata region, the system appends only a log $\langle 1, 2\rangle$, which means SSD segment '1' is caching HDD segment '2'. We do not need to make a record that HDD segment '0' is evicted because we know the evicted segment number (HDD segment '0') from the old mapping table of the metadata region.

In Step 3 (caching process), the cached cold HDD segment '3' is evicted from SSD segment '3' and the uncached cold HDD segment '4' is cached to SSD segment '3'. In the metadata region, the system appends only a log $\langle 3, 4\rangle$, which means SSD segment '3' is caching HDD segment '4'.

In Step 4 (main metadata update process), the cache-update procedure finishes, the version of the metadata header of the in-memory metadata increases, and the in-memory metadata is written to the other metadata region.

For consistency between I/O request services and the cache-update procedure, an I/O request for a segment is blocked while the cache-update procedure is processing the segment of the I/O request. The cache-update procedure that processes a segment is blocked while an I/O request for the segment is being processed.

2.8. Recovery and consistency

PRWC recovers the cache metadata without any data loss in a short time after a crash. It does not need any warm-up process. Most of the cached data are retained.

Fig. 8 illustrates a simplified recovery process. At the beginning of a restart, the highest version of the main metadata between the two metadata regions is loaded into the in-memory metadata. If the latest metadata region has valid log sectors, the in-memory metadata is combined with its logs. At the end of recovery, we should rollback with the last log sector because we cannot guarantee that the last log sector is valid.

2.8.1. Verification of lossless recovery

This section describes that the recovery process is correct in each crash step. Neither cache data nor metadata changes most of the time except for the cache update procedure that runs at an idle time. A recovery process is needed only if a crash occurs in the middle of the cache update procedure, which consists of the decay process (Step 1), the caching process for each uncached hot segment (Steps 2 and 3), and the main metadata update process (Step 4). Steps 1–4 are illustrated in Figs. 5 and 6.

The following describes how metadata is recovered and why there is no data loss when a crash occurs in each step.

1. *Decay process* (Step 1): Only the RF values in the in-memory metadata are changed. Nothing changes in the metadata and data in the storage.
2. *Caching process* (Step 2 or 3): It consists of the more detailed sub-steps (Steps i–v) that are described in Section 2.7.
   - *Step* i: The system changes the HDD clock and SSD clock, which are not the part of the cache metadata.
   - *Step* ii: The system copies the dirty data of a SSD segment to a HDD segment but none of the cache metadata in the storage changes (not in-memory). After a crash, the metadata indicates that the SSD segment is still dirty but it is actually clean. However, this error does not lose any data.
   - *Step* iii: Only the in-memory metadata is modified and the main metadata in SSD is unchanged; thus, there is no consistency problem caused by a crash.
   - *Step* iv: A log is appended. The hot segment that is indicated by the log is not yet cached to SSD, which means that the SSD segment still contains the cold HDD segment of the old mapping. Hence, the recovery process deletes the cache mapping for the last log. In other words, the SSD segment that is indicated by the last log is set to be uncached. We can erase the mapping of the SSD segment without data loss because the cold HDD segment that is contained in the SSD segment was clean in Step ii.
   - *Step* v: This sub-step makes the metadata and data consistent by copying the data of the hot HDD

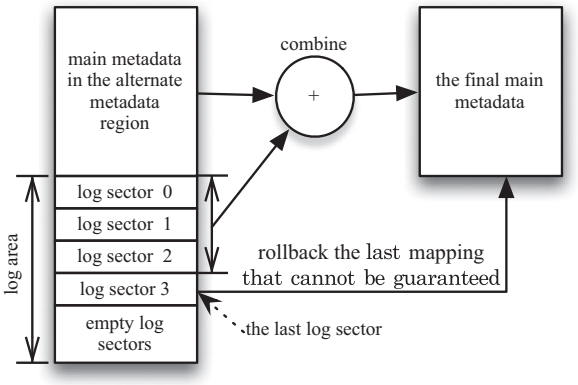

Fig. 8. The in-memory metadata is rebuilt by combining the main metadata with its logs. At the end of recovery, we should rollback with the last log sector because we cannot guarantee that the last log sector is valid.

segment to the SSD segment according to the log that is stored in Step iv. By the way, we cannot determine whether a crash occurred in Step iv or Step v because the last metadata of Step iv is the same as that of this sub-step. Hence, we just erase the mapping of the SSD segment that is indicated by the last log, even though the metadata and data are consistent. However, there is no data loss because all related data are clean before this sub-step. All the other logs except for the last log are not discarded.

3. *Main metadata update process* (Step 4): The system writes the in-memory metadata to the next metadata region. The mapping information of the latest in-memory metadata is equal to the combination of the main metadata and the logs. If a crash occurs while the newest main metadata is being written, the system can notice an error by the payload hash of the newest main metadata and reconstruct the final main metadata from the previous main metadata and the logs in the previous version of the metadata region.

2.8.2. Recovery from repeated crashes

This section investigates what happens if a crash occurs during the recovery process. Fig. 8 illustrates a simplified recovery process. If the latest valid metadata region has valid log sectors in metadata region X, a new main metadata is built by combining the latest main metadata with logs and written in the alternate metadata region Y. If a crash occurs during writing the new main metadata, the system will find a corrupted main metadata in metadata region Y and a valid main metadata and logs in metadata region X at the next restart. The system takes the metadata region X as the latest region and again builds a new main metadata in the same way of the recovery at the first crash.

2.8.3. Recovery example

Fig. 9 shows cases of recovery using a main metadata and logs. There exist two different data sets in the cache shown in Fig. 9(a) and (b) with the same metadata shown in Fig. 9(c).

Fig. 9 (a) shows the storage data in Step iv of Step 3. Fig. 9(b) shows the storage data in Step v of Step 3. SSD segment '3' that is pointed at by the last log ⟨3,4⟩ contains different data between Step iv and Step v. SSD segment '3' in Step iv contains the data of old mapping (HDD segment '3') but SSD segment 3 in Step v contains the data of new mapping (HDD segment '4'). The two cases have the same metadata and logs because the log is added in Step iv.

The SSD segment pointed at by the last log (SSD segment '3') is invalidated in the recovery process because its cached data is different between Step iv and Step v. Fig. 9(e) shows how this is done. The old mapping that SSD segment '3' caches HDD segment '3' is removed (clear the cached bit of HDD segment '3', erase the mapping entry of SSD segment '3' in the mapping table, and clear the dirty bit of SSD segment '3'). There is no consistency problem because the dirty data of the deleted segment is already flushed to its home location before Step iv.

Logs, except for the last one, are applied to the in-memory metadata in their order. Fig. 9(d) shows an example in which a log is applied to the in-memory metadata. The log indicates that SSD segment '1' caches HDD segment '2'. The old main metadata indicates that SSD segment '1' caches HDD segment '0'. Applying the log is as follows: (1) The cached bit of the old mapping (HDD segment 0) is cleared. (2) The mapping table entry of the SSD segment is updated with the new mapping (HDD segment '2'). (3) The cached bit of the new mapping is set (HDD segment 2). (4) The dirty bit of the cache is cleared (SSD segment 1).

2.8.4. Miscellaneous concerns

The combination of the main metadata and logs in the previous version of the metadata region is always equal to the latest main metadata in the alternate metadata region; thus, there is a kind of redundancy. Therefore, if the latest main metadata is corrupted due to a bad sector, it can be rebuilt from the alternate metadata region. If a 512 B sector in the mapping table is bad, then 64 segments are lost. If the segment size is 1 MiB, we lose 64 MiB of data due to the 512 B error. That is why we maintain the redundant metadata in the two regions.

An error may occur if the same SSD segment numbers or the same HDD segment numbers appear among logs in the same metadata region. However, the clock-based cache policy does not cause such an error because SSD clock and HDD clock rotate, at most, one revolution.

When a write request is delivered to a clean SSD segment, the SSD segment becomes dirty and the SSD segment number is journaled as a dirty log in the log area before the write request is processed. After a crash, the final dirty bitmap is rebuilt by combining the saved old dirty bitmap and the dirty logs. The dirty log may degrade the performance of normal I/O services.

We can choose another policy without the dirty log for faster write response time. After a crash, the dirty bitmap must be ignored and all SSD segments must be treated as dirty. No-dirty-log policy may make the cache update procedure longer after a crash (longer Step ii). Evaluation of the dirty log is left as a further work.

3. Performance evaluation

3.1. Experimental environment

There are two major concerns of this evaluation.

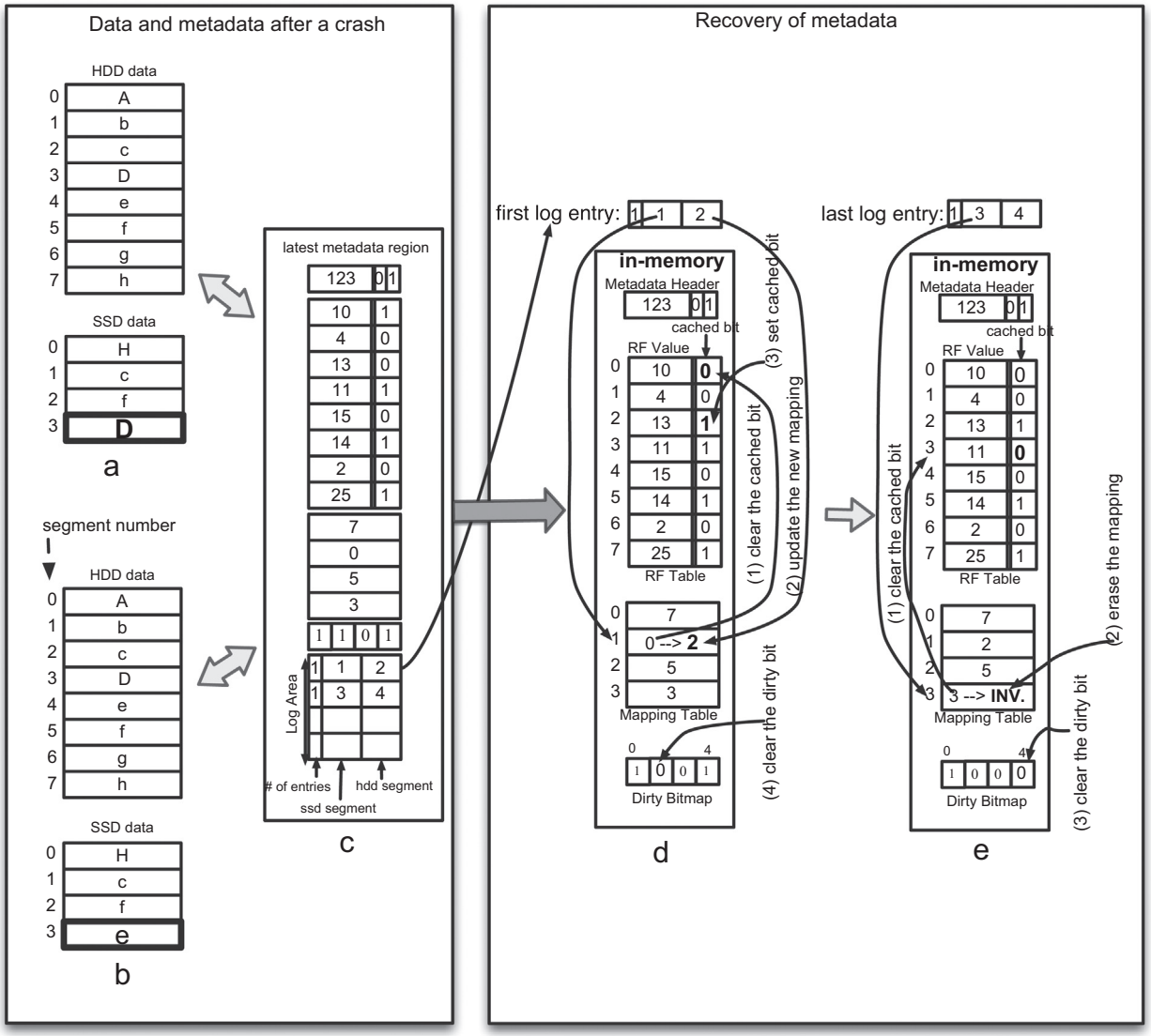

Fig. 9. Cases of recovery using a main metadata and logs. There exist two different data sets in the cache with the same metadata. (a) Storage data in Step iv of Step 3. (b) Storage data in Step v of Step 3. SSD segment '3' that is pointed at by the last log ⟨3, 4⟩ contains different data in Step iv and Step v. SSD segment '3' in Step iv contains the data of the old mapping (HDD segment '3') but the SSD segment in Step v contains the data of the new mapping (HDD segment '4').

First, is the periodic cache update better than an on-demand cache update? To resolve this issue, we implemented an SSD cache with the LRU policy, which caches a segment for each a cache miss, and evicts the least recently used segment on the fly. This LRU implementation does not persistently manage the cache metadata and is a read-only cache because it loses data in a crash. Depending on the crash, the LRU cache may restart with an empty cache. Thus, we evaluated two types of LRU: SLRUr (SSD-LRU reset everyday) and SLRUn (SSD-LRU never reset). SLRUr resets the SSD data every day. SLRUn retains its cached data without any power-off.

The second issue involves how much the performance differs between a read-only cache and a read–write cache. We evaluated our proposed scheme, PRWC (Persistent Read/Write cache) and an additionally implemented PROC (Persistent Read Only Cache, a read-only version of PRWC), PROC is almost same as PRWC, except that PROC does not make any dirty data by forwarding write requests for cached segments to both SSD and HDD.

In our experiment, PRWC and PROC executed the cache-update procedure only once a day. SLRUr, SLRUn, PROC, and PRWC were implemented as a block driver in Linux kernel 2.6.35. A Samsung MZ-7PD512 SSD and a Seagate ST2000DM001 HDD were used for storage. The storage devices' interface was SATA III (6 Gbps). The default capacity of the SSD is 32 GB. The capacity of the HDD was 2 TB throughout the experiments. The achievable maximum bandwidths of the SSD and the HDD are 566 MB/s and 210 MB/s, respectively. The maximum throughputs of the SSD and the HDD are 100,000 and 600 IOPS, respectively.

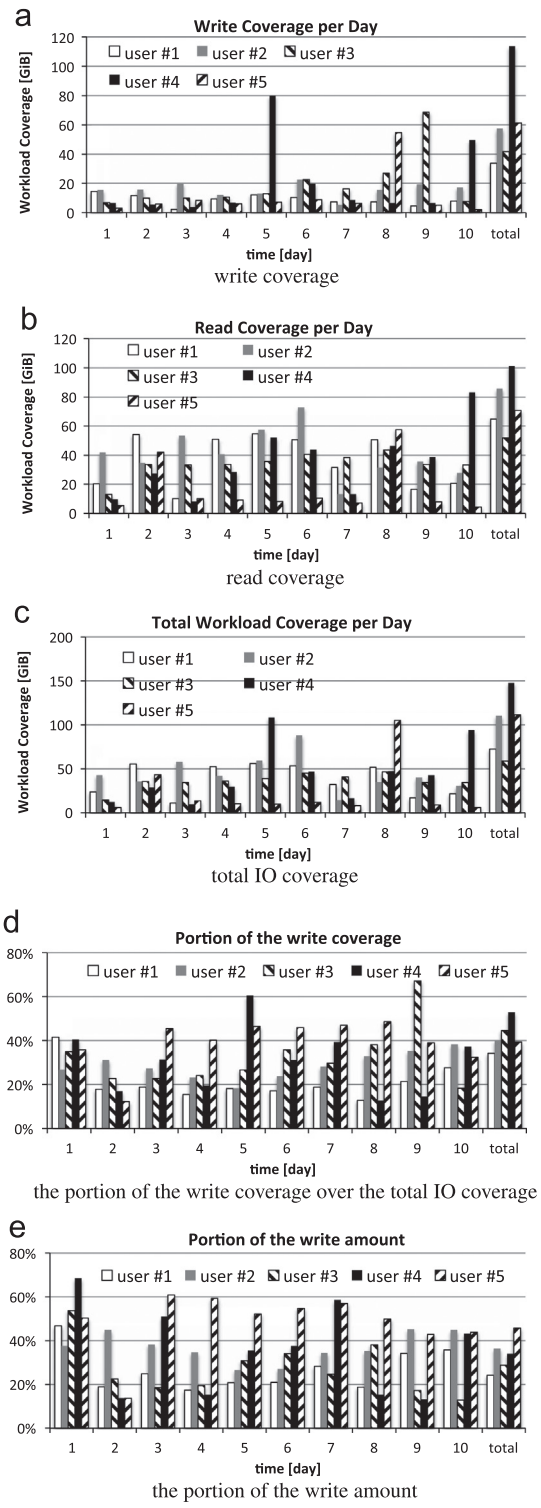

Fig. 10. Workload coverage.

3.2. Workload

This evaluation used a large amount of traces that were extracted using Xperf [47] from five users during ten days. The users were a director, two programmers, and two system engineers. They ran MS-Windows 7. Traces were replayed with Direct IO, which bypasses the RAM cache. Hence, the main memory size had no effect on this evaluation.

Fig. 10 shows the workload coverage per day. The coverage was analyzed in 1 MiB units as the segment size. The coverage is defined as a set of segments that are requested at least once. Fig. 10(a), (b), and (c) shows the write coverage, read coverage, and total coverage, respectively, for the workloads. The workload of User 1 consists of writes on 33 GB of segments and reads on 65 GB of segments. The workload coverage of User 4 consists of writes on 113 GB of segments and reads on 101 GB of segments.

The total (read and write) workload coverage is 73 GB, 111 GB, 59 GB, 148 GB, and 112 GB for Users 1 to 5, respectively.

Fig. 10 (d) is the portion of the write coverage over the total coverage. The portion of the write coverage was 42% on average, for all users. The portions of the write amount are shown in Fig. 10(e). The portions of the write amount were 24%, 36%, 29%, 34%, and 46%, for Users 1 to 5, respectively, whereas the portions of the write coverage were 34%, 40%, 45%, 53%, and 39% for each user. That means that we can expect more spatial locality in writes than in reads.

3.3. Bandwidth and latency

Fig. 11 shows the total (read and write) bandwidth. PRWC had no cache hit on the first day because its cache was updated after one day. The performance of PRWC significantly increased from the first day to the fourth day because the cache was not full until the fourth day. PRWC outperformed SLRUn after the third day. PRWC showed the best performance because only PRWC can cache write requests. The bandwidth of PRWC was 54% better than that of SLRUn by 54% and 48% better than PROC, on average.

The bandwidth of PROC was similar to that of SLRUn. However, PROC never lost its cached data but SLRUn is an artificial scenarios. A practical system using SLRU clears its cache by a crash or power failure and thus exhibits the same performance as SLRUr after a crash. SLRUr is the performance after SLRU experiences a crash. The bandwidth of SLRU was degraded by 41% at the first day after a crash, on average, for all users.

Fig. 12 shows the write-part bandwidth, where both read and write coexist in the workload. The write performance of PRWC was significantly superior to the others because only PRWC can retain dirty data. The write performance of SLRUn and PROC was similar to HDD. PRWC showed 3 times better write bandwidth than did the PROC.

Fig. 13 shows the read-part bandwidth. In read bandwidth, PRWC was best for User 1 and User 4, PROC was best for User 2, and SLRUn was best for User 3. The three schemes, therefore, have similar read performance.

PRWC exhibited the shortest total latency as shown in Fig. 14. This evaluation shows that latency tends to be inversely proportional to bandwidth. This evaluation shows that caching writes can significantly reduce latency.

PRWC provides zero-staleness, which means that it does not return any stale data after a crash. Many write

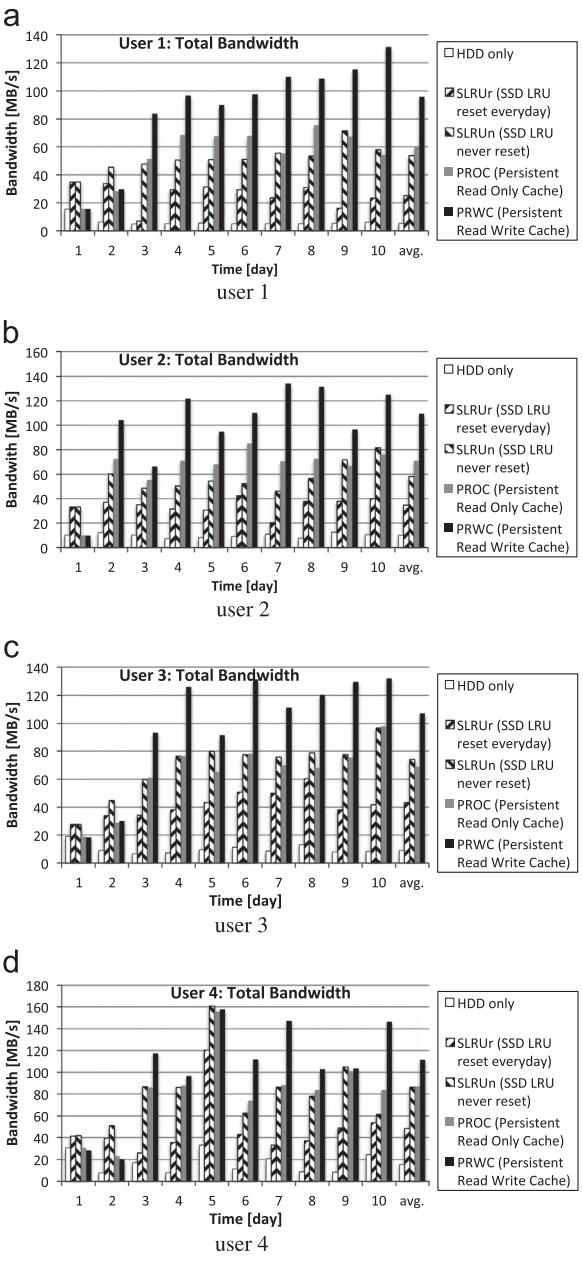

Fig. 11. Total bandwidth.

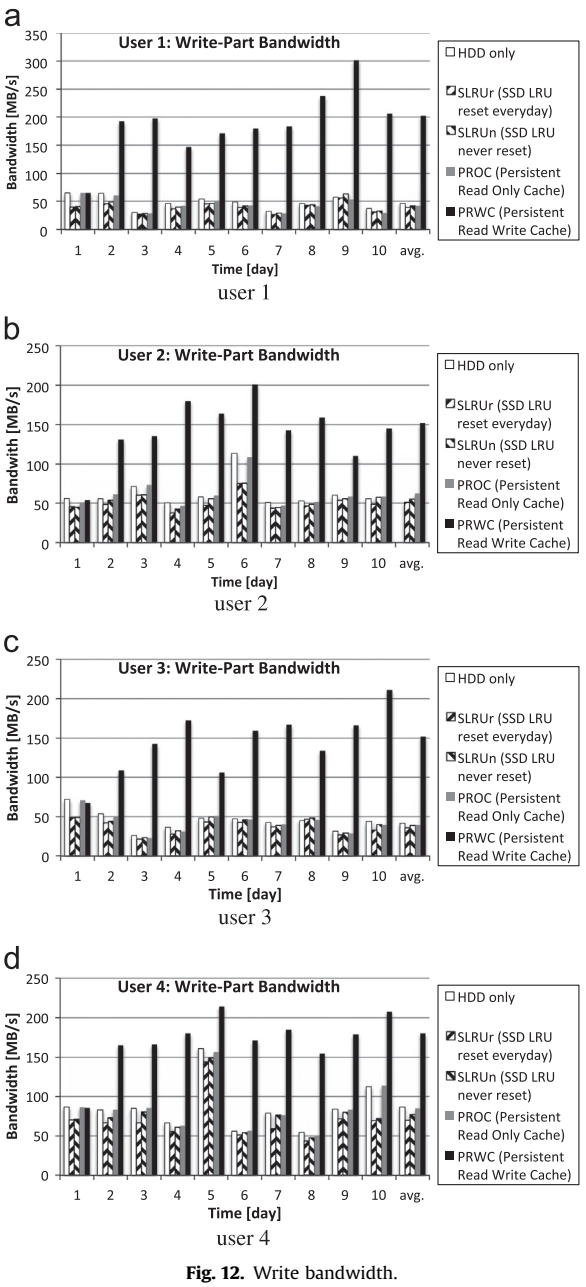

Fig. 12. Write bandwidth.

caches with write-back policies cannot support zero-staleness. Hence, we evaluated PRWC with read-only caches that support zero-staleness.

3.4. SSD size

The bandwidth of User 5 is shown in Fig. 15 as the SSD size varies. For all SSD sizes in this experiment, PRWC worked best and PROC was second. PRWC outperformed PROC by 30%, 35%, 34%, and 39% for 8, 32, and 128 GB of SSD, respectively. PRWC outperformed SLRUn by 49%, 67%, and 33% for 8, 32, and 128 GB of SSD, respectively.

The total workload coverage (see Section 3.2) of User 5 is 112 GB. Hence, all data can be cached to 128 GB of SSD. In many cases, SLRUn is outperformed by PROC because SLRUn needs more SSD traffic for on-demand caching. However, if the cache size is greater than the workload coverage, the on-demand cache update (SLRU) could outperform the periodic cache update (PROC) as shown in Fig. 15(c).

On the eighth day, User 5 executed a big data backup that produced sequential reads and writes, thus the cache hit rate was low and the performance of 'HDD only' was high. The big data backup polluted the LRU cache with blocks that were not used again. SLRUn consumed a significant portion of the SSD bandwidth, thus it was worse

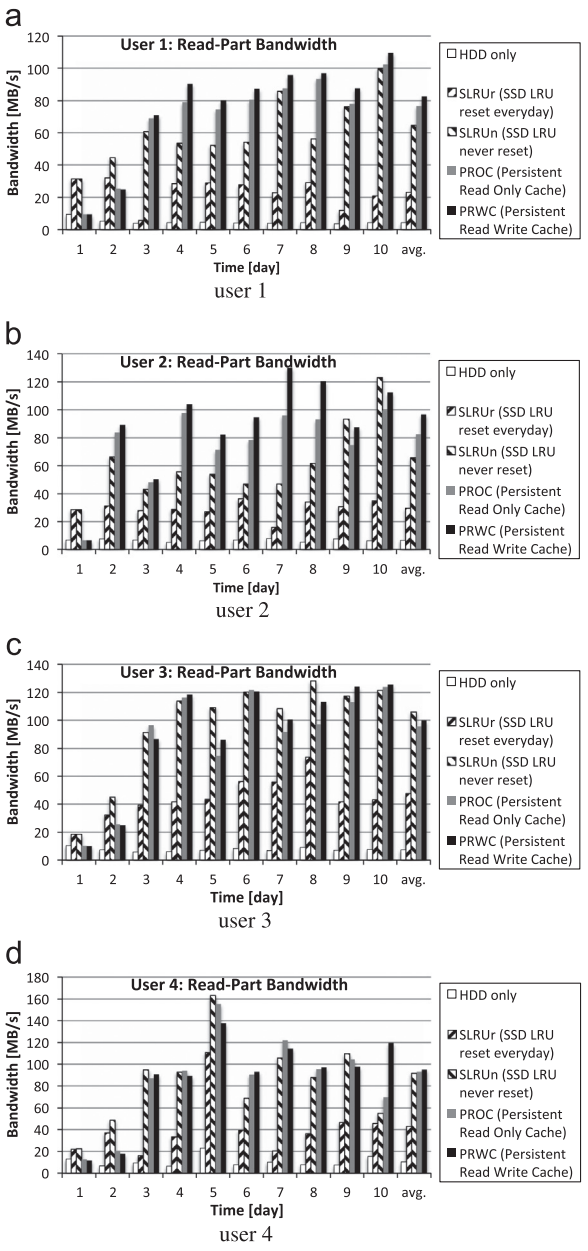

Fig. 13. Read bandwidth.

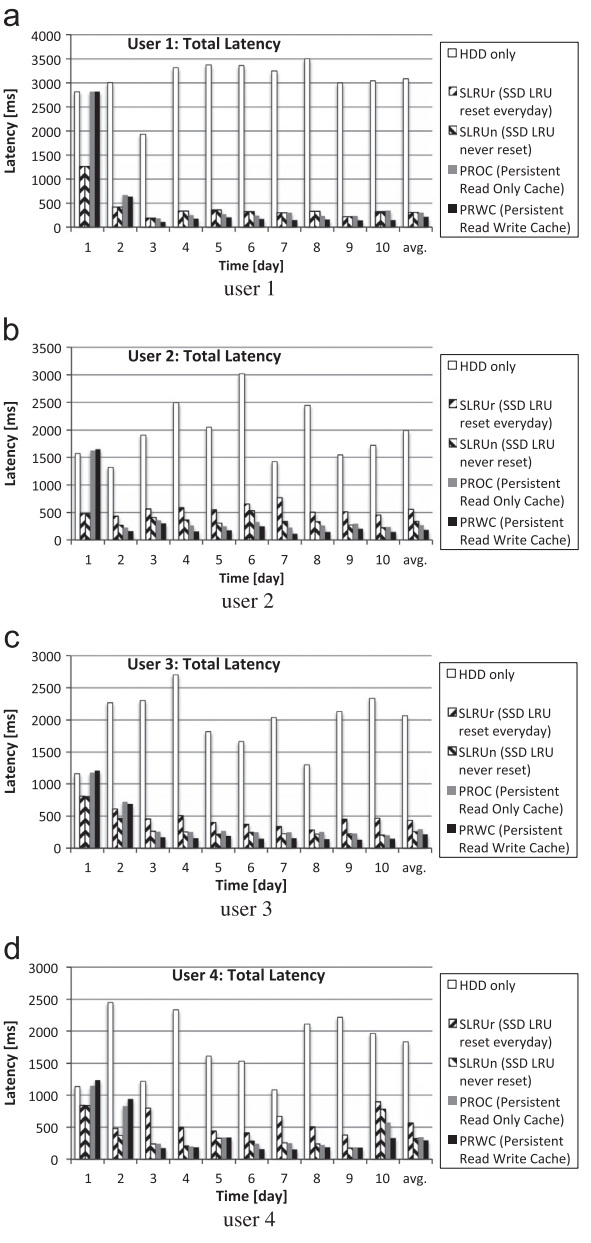

Fig. 14. Total latency.

than 'HDD only' on the eighth day. PRWC and PROC are resistant to sequential I/Os, which are usually considered as requests on cold segments.

3.5. Hit rate

The write hit rate is the number of write requests that are for the cached segment over the total number of write requests from the upper layer. The total hit rate is the number of read and write requests that are for the cached segmented over the total number of read and write requests.

The hit rate of PRWC was zero on the first day as shown in Figs. 16 and 17. This was because it executed the cache-update procedure at midnight and had no cached data at the first day. The hit rate of PRWC increased on the second day but it was lower than that of SLURn. The read hit rate of PWRC was a little smaller than SLRUn after the third day but the total hit rate of PRWC was 23% better than SLRUn (on average) because PRWC can cache write requests.

SLRUn updates its cached data on the fly, whereas PROC updates the cache contents only once a day. Hence, the read hit rate of PROC was smaller than SLRUn by 0.06% point, as shown in Fig. 16. PROC, however, provided 4% more bandwidth than SLRUn (on average).

Hit rate does not consider the limited bandwidth of the cache. Unlike RAM, the SSD bandwidth is a limited resource. Promoting data from HDD to SSD in SLRU consumes the SSD bandwidth and makes SSD more congested.

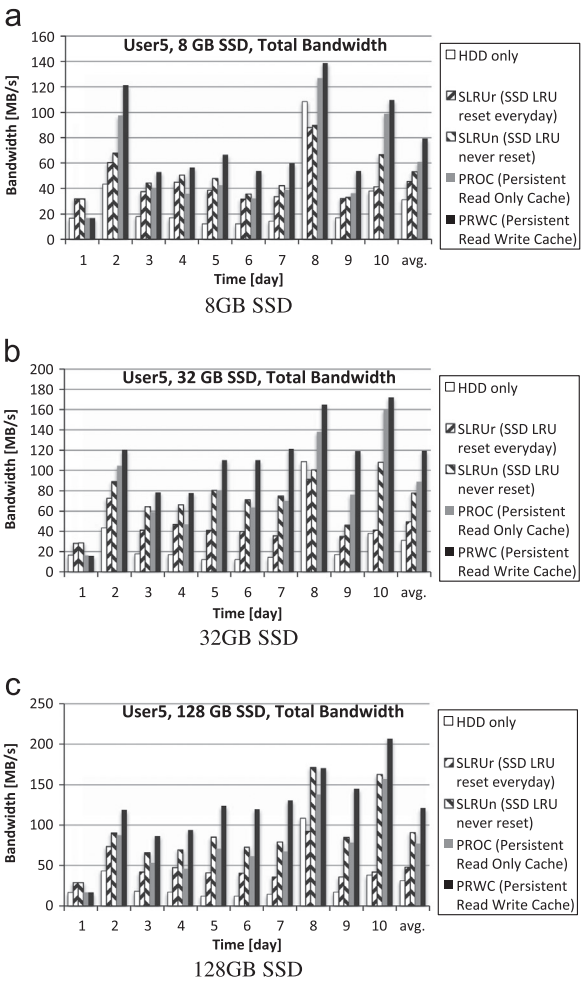

Fig. 15. Total bandwidth as the SSD size varies.

PROC and PRWC do not promote data while they are serving I/O requests, thus PROC can provide greater throughput with less hit rate than SLRU.

3.6. Segment size

The storage volume is divided by segment unit, which is the minimum unit for cache management. Fig. 18 shows the bandwidth of User 5 as the segment size varied from 128 KiB to 1 MiB. The 1 MiB of segment showed the best performance. Due to an implementation restriction, we could not evaluate segments larger than 1 MiB. A big segment is beneficial for spatial locality and total bandwidth, despite wasting cache capacity.

PRWC outperformed SLRUn by 29%, 35%, 54%, and 67% for 128, 256, 512, and 1024 KiB of the segment size, respectively (on average). PRWC outperformed PROC by 19%, 29%, 31%, and 34% for 128, 256, 512, and 1024 KiB of the segment size, respectively (on average).

The greater the segment size was, the greater the bandwidth was in this evaluation. When a cache miss on a page occurs, PRWC promotes the segment that includes the requested page. This means that PRWC caches a lot of pages

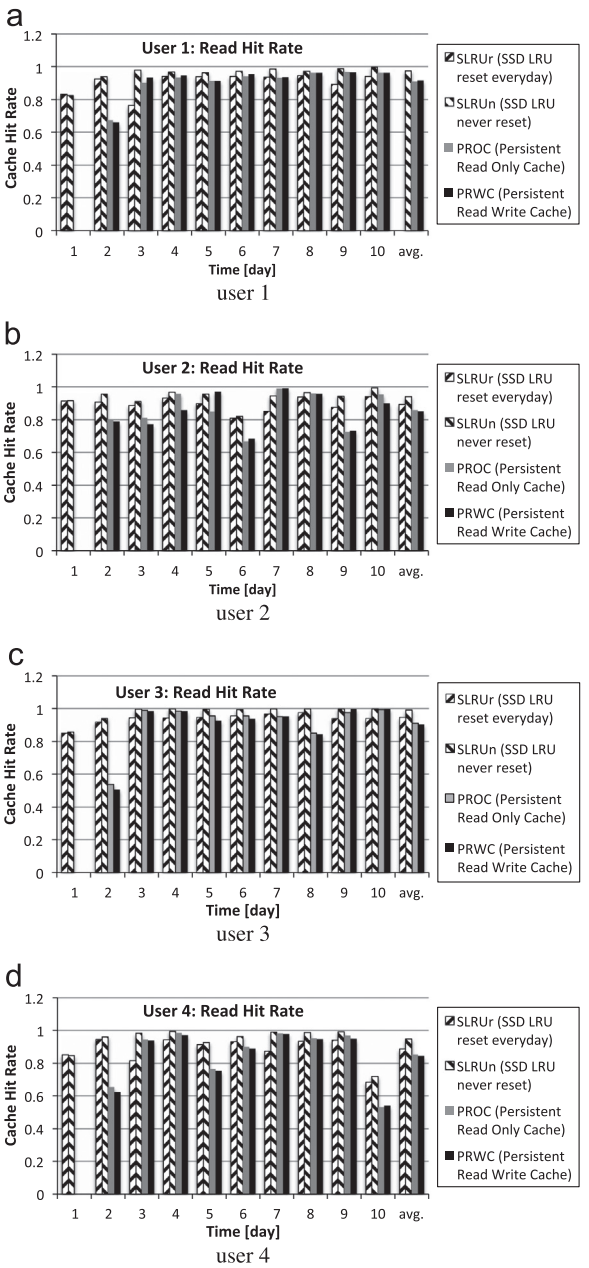

Fig. 16. Read hit rate.

that are located near the requested page. Hence, this evaluation shows that desktop workloads exhibit great spatial locality.

The hit rate for various segment sizes is shown in Fig. 19. The greater the segment size is, the higher the hit rate is. The hit rates of PRWC are 0.46, 0.49, 0.51, and 0.58, respectively, for 128, 256, 512, and 1024 KiB of the segment sizes (on average). This evaluation attests that a group of blocks, as a cache management unit, is better than a small block for managing a storage-class cache.

3.7. No-write-back vs. write-back

PRWC is a no-write-back policy, which does not flush dirty pages in SSD until the pages are evicted, and it

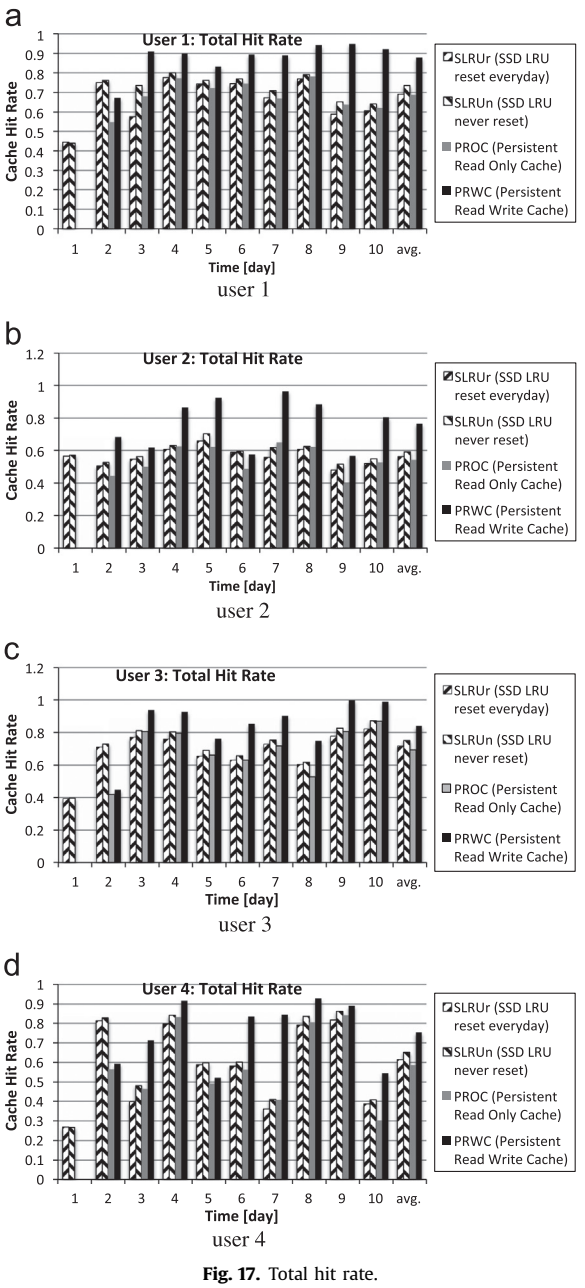

Fig. 17. Total hit rate.

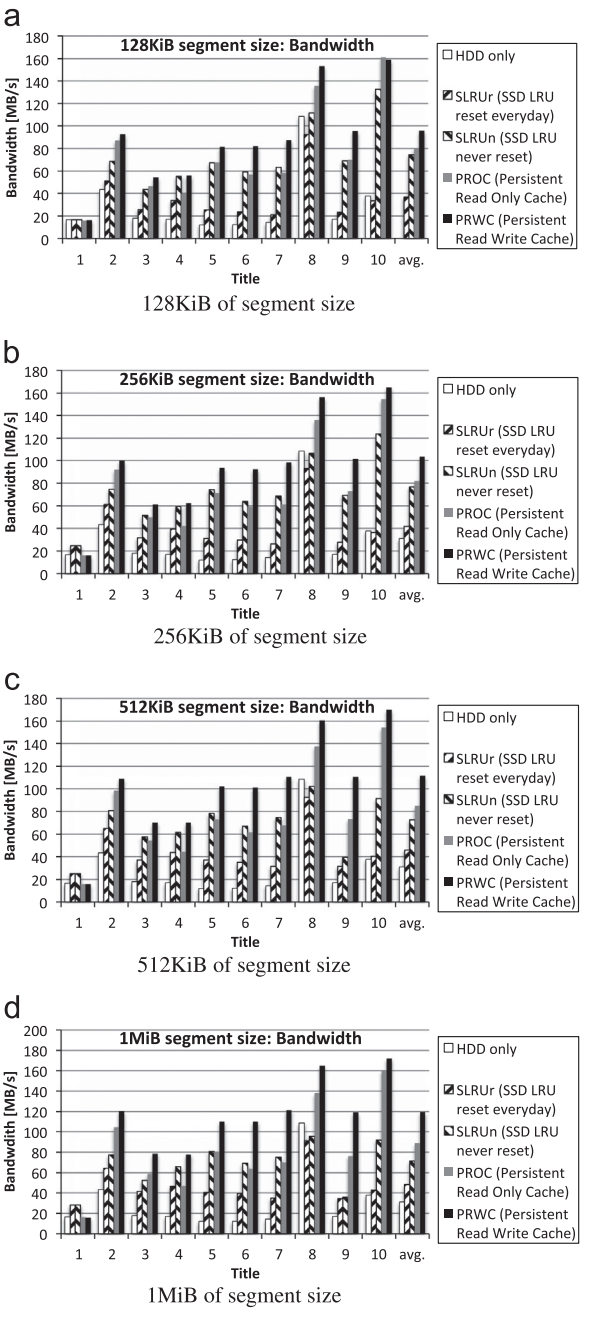

Fig. 18. Bandwidth as the segment size varies.

redirects write requests only for hot segments. The write-back policy buffers all write requests but it needs to evict the buffered pages when a write buffer becomes full. Therefore, this eviction process requires one more SSD read and one more HDD write and consumes the limited SSD bandwidth. PRWC does not require the eviction process while user I/Os are being serviced.

The write-back policy loses dirty pages when a crash occurs, but there is no data loss in PRWC. If the write-back policy reduces the amount of data loss by decreasing the write buffer size or flushing dirty pages more frequently, its performance approaches that of the write-through policy. Hence, it is difficult to determine the reliability of the write-back policy.

The write-back policy is completely different from the structure of PRWC. Hence, we chose another implementation that supports the write-back policy. Fig. 20 compares the write-back policy and the write-through policy that are implemented by flashcache [10] with the workload of User 1 and the 32 GiB of cache size.

The read bandwidth of the write-back is similar to that of the write-through, and the read bandwidth of PROC is similar to that of PRWC as shown in Fig. 20(a). Fig. 20(b) compares the write performance. The write bandwidth of the write-back was greater than that of the write-through by 48% on average. However, the write bandwidth of PRWC was

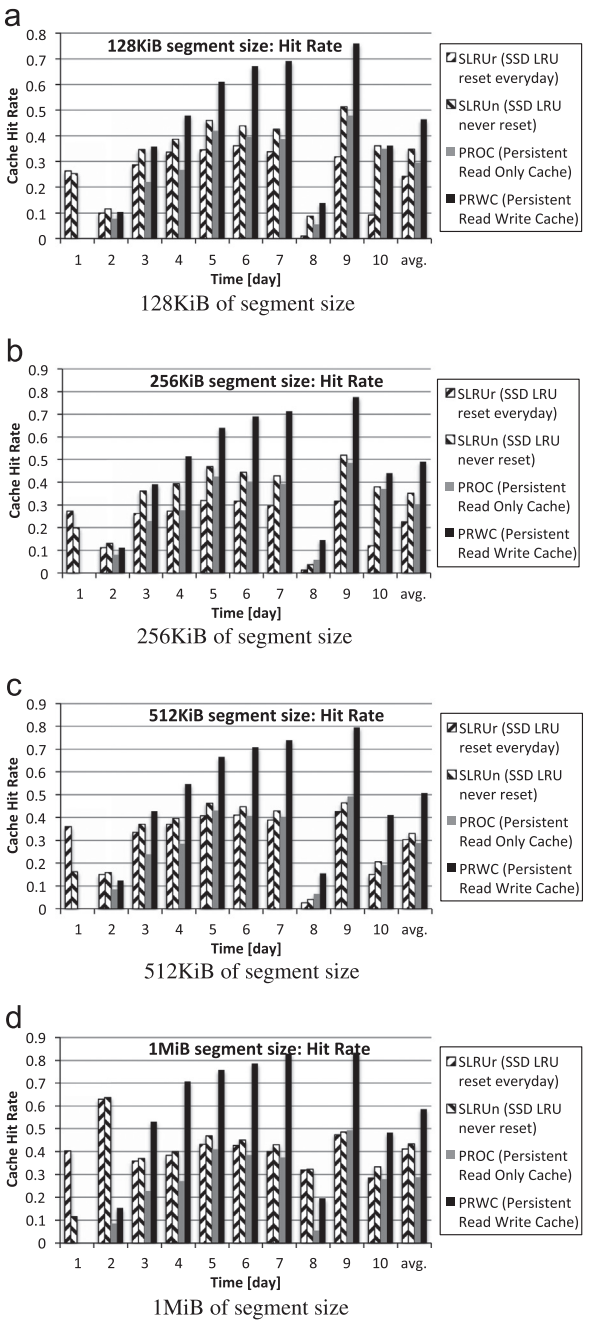

Fig. 19. Cache hit rate as the segment size varies.

4.2 times better than that of PROC on average. This evaluation shows that the no-write-back policy can improve more write performance than the write-back policy for the given workload. The great performance difference between PRWC and flashcache is due to the caching unit, such as page and segment.

3.8. Recovery time

We evaluated the recovery time at various crash points as shown in Table 2. After the first day of User 1, we generated a crash during the cache update procedure. The percentage of the crash point indicates that a crash occurred after a given percentage of the cache update procedure had finished. A zero percent crash point indicates a clean shutdown.

A clean shutdown required 0.74 s to load the metadata. In the worst case, it took 7.15 s to recover the metadata in this experiment. In comparison with a warm-start [19], which requires several hours, the recovery time of PRWC is inconsiderable.

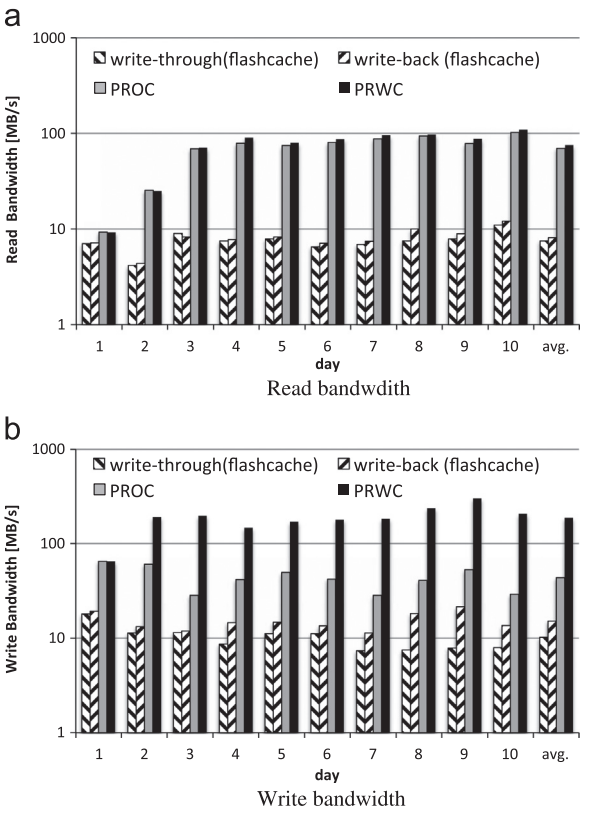

Fig. 20. Write-back vs. no-write-back: the write bandwidth of the write-back outperforms that of the write-through by 48% on average. However, the write bandwidth of PRWC is 4.2 times better than that of PROC.

Table 2
Recovery time at various crash points.

Crash point (%)	0	25	50	75	99
Recovery time (s)	0.74	2.41	4.05	5.62	7.15

4. Conclusions

We proposed a fully persistent read/write cache that improves both read and write performance; guarantees the integrity of the cache metadata and the consistency of the cached data even at a crash or a power failure; does not requires a special primitive, and has a low overhead. It quickly recovers the flash cache without any data loss, and considers both recency and frequency in its cache replacement policy. It requires only a small amount of main

memory for cache management, and employs a no-write-back policy. This provides sustained high write performance, and reduces the bandwidth consumption that is required to write back dirty data. Our system relates to desktop workloads and analyzes recency and frequency over a long period. Our scheme is suitable for desktop computers and virtual desktop infra servers.

The evaluation was performed with massive real desktop traces. The experiments verify the following facts. Our persistent read/write cache is significantly superior to read-only caches. A big sequential workload pollutes the LRU cache with blocks that are not used again, but the proposed scheme is resistant to sequential I/Os. The proposed long-term cache-update procedure decreases the read hit rate but increases bandwidth in comparison with on-demand caches because normal I/O services do not share SSD bandwidth with the cache update in our system. A large segment is beneficial for spatial locality and shows better performance, despite wasting cache capacity.

We implemented this persistent read/write cache as a block device driver of Linux. The evaluation with real desktop workloads for 10 days shows that our scheme outperforms a basic SSD cache (SLRU) by 50% and the read-only version of our scheme by 37%, on average of all experiments. Thus, its use would significantly benefit virtual desktop infra servers. This paper describes most parts of our scheme in detail. The metadata structure, the cache replacement policy, a metadata management for consistency and persistency, and the recovery procedure are fully disclosed in this paper.

Acknowledgment

This research was supported by the Daejeon University research fund (2014).

Appendix A. Supplementary data

Supplementary data associated with this paper can be found in the online version at http://dx.doi.org/10.1016/j.is.2016.02.002.

References

[1] Adam Leventhal, Flash storage memory, Commun. ACM 51 (7) (2008) 47–51. http://dx.doi.org/10.1145/1364782.1364796.
[2] J. Guerra, H. Pucha, J.S. Glider, W. Belluomini, R. Rangaswami, Cost effective storage using extent based dynamic tiering, In: Proceedings of the 9th USENIX Conference on File and Storage Technologies, USENIX Association, San Jose, CA, 2011, pp. 273–286.
[3] E. Lee, H. Bahn, Caching strategies for high-performance storage media, ACM Trans. Storage 10 (3) (2014) 11.
[4] Z. Liu, B. Wang, P. Carpenter, D. Li, J.S. Vetter, W. Yu, Pcm-based durable write cache for fast disk i/o, In: 2012 IEEE 20th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, San Francisco, CA, 2012, pp. 451–458.
[5] E. Lee, H. Bahn, S.H. Noh, A unified buffer cache architecture that subsumes journaling functionality via nonvolatile memory, ACM Trans. Storage 10 (1) (2014) 1.
[6] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, Y. Xie, Hybrid cache architecture with disparate memory technologies, ACM SIGARCH Comput. Archit. News 37 (3) (2009) 34–45.
[7] M. Peters, Netapp's Solid State Hierarchy with a Focus on Flash Cache, Technical Report, NetApp, September 2009.
[8] M. Srinivasan, Flashcache: A Write Back Block Cache for Linux, 2011. URL ⟨https://github.com/facebook/flashcache/blob/master/doc/flashcache-doc.txt⟩.
[9] EMC, Vnx Fast Cache—A Detailed Review, Technical Report H8046.9, EMC, December 2013.
[10] T. Kgil, T. Mudge, Flashcache: a nand flash memory file cache for low power web servers, In: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, ACM, Seoul, Korea, 2006, pp. 103–112.
[11] Y. Ou, J. Xu, T. Härder, Towards an efficient flash-based mid-tier cache, In: Database and Expert Systems Applications, Springer, 2012, pp. 55–70.
[12] J. Jang, H.-S. Kim, W. Cho, H. Cho, J. Kim, S.I. Shim, Y. Jang, J.-H. Jeong, B.-K. Son, D.W. Kim, et al., Vertical cell array using tcat (terabit cell array transistor) technology for ultra high density nand flash memory, In: 2009 Symposium on VLSI Technology, Kyoto, Japan, 2009, pp. 192–193.
[13] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, S.-W. Kim, A case for flash memory ssd in enterprise database applications, In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, Vancouver, Canada, 2008, pp. 1075–1086.
[14] M. Canim, G.A. Mihaila, B. Bhattacharjee, K.A. Ross, C.A. Lang, Ssd bufferpool extensions for database systems, Proc. VLDB Endow. 3 (1–2) (2010) 1435–1446.
[15] C. Mohan, F. Levine, ARIES/IM: an efficient and high concurrency index management method using write-ahead logging, ACM SIGMOD Record 21 (2) (1992) 371–380.
[16] S. Byan, J. Lentini, A. Madan, L. Pabón, M. Condict, J. Kimmel, S. Kleiman, C. Small, M. Storer, Mercury: Host-side flash caching for the data center, In: 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, San Diego, CA, 2012, pp. 1–12. http://dx.doi.org/10.1109/MSST.2012.6232368.
[17] EMC, Introduction to emc vfcache, Technical Report H10502.2, EMC, August 2012.
[18] B. Gregg, Zfs l2arc, July 2008. URL ⟨https://blogs.oracle.com/brendan/entry/test⟩.
[19] Y. Zhang, Gokul Soundararajan, Mark W. Storer, L.N. Bairavasundaram,
S. Subbiah, A.C. Arpaci-Dusseau, R.H. Arpaci-Duesseau, Warming up storage-level caches with bonfire, In: Proceedings of the 11th USENIX Conference on File and Storage Technologies, USENIX Association, San Jose, CA, 2013, pp. 59–72.
[20] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, M. Zhao, Write policies for host-side flash caches., In: Proceedings of the 11th USENIX Conference on File and Storage Technologies, USENIX Association, San Jose, CA, 2013, pp. 45–58.
[21] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, V. Hristidis, Borg: Block-reorganization for self-optimizing storage systems, In: FAST, vol. 9, USENIX, San Francisco, CA, 2009, pp. 183–196.
[22] D.S. Roselli, J.R. Lorch, T.E. Anderson, et al., A comparison of file system workloads, In: USENIX Annual Technical Conference, General Track, USENIX, San Diego, CA, 2000, pp. 41–54.
[23] D. Arteaga, M. Zhao, Client-side flash caching for cloud systems, In: Proceedings of International Conference on Systems and Storage, ACM, Haifa, Israel, 2014, pp. 1–11.
[24] M. Saxena, M.M. Swift, Y. Zhang, Flashtier: a lightweight, consistent and durable storage cache, In: Proceedings of the 7th ACM European Conference on Computer Systems, ACM, Bern, Switzerland, 2012, pp. 267–280.
[25] S.H. Baek, K.-W. Park, A persistent read/write cache using general flash-based ssds, J. Korean Inst. Next Gener. Comput. 10 (4) (2014) 41–54.
[26] D.A. Holland, E. Angelino, G. Wald, M.I. Seltzer, Flash caching on the storage client, In: Proceedings of 2013 USENIX Annual Technical Conference, USENIX Association, San Jose, CA, 2013, pp. 127–138.
[27] E.V. Hensbergen, M. Zhao, Dynamic policy disk caching for storage networking, URL: ⟨http://visa.cs.fiu.edu/ming/dmcache⟩.
[28] D.P. Bovet, M. Cesati, Writing Dirty Pages to Disk in Understanding the Linux Kernel, 3rd edition, . O'Reilly, 2005.
[29] T. Heo, I/O barriers, Linux Kernel Archives, 2005.
[30] K. Miller, M. Pegah, Virtualization: virtually at the desktop, In: Proceedings of the 35th Annual ACM SIGUCCS Fall Conference, ACM, Orlando, USA, 2007, pp. 255–260.

[31] D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S.L. Min, Y. Cho, C.S. Kim, Lrfu: a spectrum of policies that subsumes the least recently used and least frequently used policies, IEEE Trans. Comput. 50 (12) (2001) 1352–1361.
[32] N. Megiddo, D.S. Modha, ARC: a self-tuning, low overhead replacement cache, In: Proceedings of USENIX Conference on File and Storage Technologies, USENIX Association, San Francisco, CA, 2003, pp. 115–130.
[33] S. Bansal, D.S. Modha, CAR: Clock with adaptive replacement, In: Proceedings of USENIX Conference on File and Storage Technologies, USENIX Association, San Francisco, CA, 2004, pp. 187–200.
[34] M. Balakrishnan, A. Kadav, V. Prabhakaran, D. Malkhi, Differential raid: rethinking raid for ssd reliability, ACM Trans. Storage 6 (2) (2010) 4.
[35] S. Im, D. Shin, Flash-aware raid techniques for dependable and high-performance flash memory ssd, IEEE Trans. Comput. 60 (1) (2011) 80–92.
[36] Intel, Intel solid-state drives in server storage applications, white paper, Order Number: 330037-001US, February 2014.
[37] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson, RAID: high-performance, reliable secondary storage, ACM Comput. Surv. 26 (2) (1994) 145–185.
[38] S.H. Baek, K.H. Park, Matrix-stripe-cache-based contiguity transform for fragmented writes in RAID-5, IEEE Trans. Comput. 56 (8) (2007) 1040–1054.
[39] X. Ding, S. Jiang, F. Chen, A buffer cache management scheme exploiting both temporal and spatial localities, ACM Trans. Storage 3 (2), 2007.
[40] D.S. DeLorme, M.L. Holm, W.D. Lee, P.B. Passe, G.R. Richard, George D. Timms, Jr., L.W. Youngren, Database index journaling for enhanced recovery, US Patent 4,819,156, April 4, 1989.
[41] K. Rothermel, C. Mohan, ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions, IBM Thomas J. Watson Research Division, 1989.
[42] M. Rosenblum, J. Ousterhout, The design and implementation of a log-structured file system, ACM Trans. Comput. Syst. 10 (1) (1992) 26–52.
[43] R.H. Arpaci-Dusseau, A.C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Arpaci-Dusseau Books, 2014.
[44] S. Tweedie, EXT3, journaling filesystem, In: the 2000 Ottawa Linux Symposium, Ottawa, Ontario, Canada, 2000, pp. 24–29.
[45] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, The new EXT4 filesystem: current status and future plans, In: Proceedings of the 2007 Ottawa Linux Symposium, Ottawa, Ontario, Canada, 2007, pp. 21–33.
[46] D. Robbins, Common Threads: Advanced Filesystem Implementers Guide, Part 9, Introducing XFS, IBM, 2002.
[47] M. Corp., Windows on/off transition performance analysis, April 2011. ⟨http://msdn.microsoft.com/en-us/windows/hardware/gg463386⟩.