RFLRU: A Buffer Cache Management Algorithm for Solid State Drive to Improve the Write Performance on Mixed Workload

Arul Selvan Ramasamy, and Porkumaran Karantharaj

Abstract—**Flash memory based Solid State Drives (SSD) acquiring greater attention in enterprise storage computing environment; this is primarily due to its high I/O speed. SSDs use multiple NAND flash memory chips as a storage media and deploy internal RAM to maintain the flash translation layer (FTL) mapping table. The rest portion of the inner RAM is used as a buffer. This buffer absorbs the write requests and thus the resulting write requests to the NAND flash memory is determined by the SSD buffer cache management algorithm. Many of the previously proposed write buffer cache management algorithms concentrate on improving the random write performance either by reordering the writes, addressing the temporal locality or evicting the clean pages etc. And they have not concentrated enough to exploit the sequential locality in the write pattern. Because of this, the input sequential write patterns are not completely utilized by the Flash Translation Layer and that reduces the number of switch merges and increases full and partial merges in log block based FTL. In this paper, a novel algorithm called RFLRU: Random First Least Recently Used is proposed to improve the performance of SSD write operation on mixed workload. The algorithm identifies the interleaved sequential writes, constructs various cache eviction policies and the write sequence is constructed by contemplating the SSD characteristics. Essentially this new technique reduces number of erase and writes operations on SSD. An efficient RFLRU implementation is developed and tested in a trace driven simulation environment and compared to the previously proposed LRU FAST, BPLRU and REF buffer management schemes. The results show that RFLRU reduces the number of merge; erase and write operations and the overall write performance is improved.**

Index Terms—**sequential write, random write, switch merge, full merge, page replacement, buffer management, log block, erase-before-write, write amplification, de-stage.**

I. Introduction

SSD's consists of multiple flash memory blocks, and each block is composed of multiple pages. The scope of this paper is limited to NAND flash memory. There are three basic types of operations in NAND based flash memory; read, program and erase. Page is the basic unit of read/write operation in flash memory. Flash memory does not support in-place update. After a page write, entire block has to be erased before the subsequent write operation on the same page or any of the page that belongs to the specific block, this block is called erasure block. Thus, flash memory based SSDs poses a well-known challenge, that is, the erase-before-write problem.

Manuscript received Jan 07, 2014; revised August 11, 2014.

F.A. Arul Selvan Ramasamy (corresponding author) is a doctoral research scholar, associated with the department of information and communication engineering, Anna University, Chennai, India. (e-mail: connectwitharul@gmail.com).

S.A. Porkumaran Karantharaj is a principal and professor, associated with the Electrical and Electronics Engineering Department at Dr NGP Institute of Technology, Coimbatore, affiliated to Anna university, Chennai India (e-mail: porkumaran@gmail.com).

In SSD write procedure, the free space in flash memory array is scanned. In general the free space can be an un-programmed blocks of memory cells and individual pages. In case if the space is not enough for data storage in one location, the flash memory array data is rearranged by erasing, rewriting and moving the data to new place within the same memory array. This inner generated write traffic is called write amplification [1]. And NAND flash memory can incur only a finite number of erase operations for a given erasure block. Therefore, increased erase operation will impact the write performance negatively and reduce the life span of SSD.

Random write causes lot of erase-before-write operation. In flash delete operation, the data is obsoleted, not deleted. Obsolete data still occupies the storage, and cannot be deleted alone in the same erasure block. Therefore a garbage collector is required to clean and erase the block by moving all valid data into a free erase block, obsoleting old erase block. Random write involves high overhead in the garbage collection than sequential write. Random write causes fragmentation which results in large number of live page migration during garbage collection. To handle this special feature of flash memory, most systems use flash translation layer (FTL). The primary role of FTL [2] is to emulate the functionality of block device by abstracting the erase before write characteristics of the NAND flash memory.

In the past, the flash memory usage was uncomplicated. For example, MP3 player and digital camera used the flash memory to read and write only large-sized multimedia files. And most of the read or write operations are sequential in nature. However, recent applications for flash memory are complex and diverse. In a real world enterprise storage system, there are many sequential writes gets manifested as random write because of write interleaving across applications, this results in a mixed write workload which is again random in nature. Random write [3] is a critical problem for write performance and lifetime of flash memory; this constrains the SSD's widespread acceptance in enterprise storage system [4]. Write amplification [1,3] triggered by random write can be addressed from two different perspectives.

Firstly, design the FTL address mapping scheme to postpone and minimize block erasures. Primarily three types of FTL mapping schemes are developed based on the granularity of the address mapping: page-level, block-level, and hybrid or log buffer. The hybrid mapping uses both page mapping and block mapping. In this scheme, all erasure blocks are separated into log blocks and data blocks, this is also called log buffer based FTL [5]. For a write request, the log block based FTL first sends the data to the log block and invalidates the corresponding data in the data block. Once the log space is utilized, the victim blocks are selected and all the valid pages in the victim block are

migrated to the data blocks, this migration process is called block merge. There are three kinds of block merge techniques [7] namely; full merge, partial merge and switch merge. The full merge allocates a free block that is erased beforehand, and then copies the most up-to-date pages either from the data block or from the log block, into the new free block. After copying all the valid pages, the free block becomes the data block and the former data block and the log block are erased. Therefore, a single full merge requires read and write operations as many as the number of valid pages in a block and two erase operations. Partial and switch merges are special cases of the merge operation. The partial merge takes place when all the valid pages in the data block can be copied to the rest of the log block. Partial merge copies only the valid pages in the data block and one erase operation can be saved compared to the full merge. On the other hand, if all the pages in the data block are already invalidated, simply switch the log block to the new data block and erase the old data block. This case is called switch merge. Garbage collector handles the merging process. For a better write performance on random workload, the merging complexities have to be reduced, which requires a sophisticated FTL address mapping technique.

Secondly, in addition to address mapping and garbage collection algorithms, FTL has page replacement algorithm [6]. Flash memory write performance is enhanced by incorporating DRAM-backed buffers inside SSD in which write requests are buffered by the SSD buffer cache. The write operation is expensive on SSD device and most of the time, applications demonstrates good temporal and spatial locality. It is of advantage to incorporate a write-back cache, building on principles used in conventional file system cache to address the temporal and spatial locality in the write workload. The basic policy used is write-behind, i.e., always data is first written into the buffer cache, and later propagated into flash memory. The page replacement policy of buffer cache is critical for the write performance and device endurance as it determines the flash memory write pattern.

The rest of the paper is organized as follows. In section2, the problem is explored in detail, and section 3 defines the contribution to address the defined problem. In section 4, the related work is explored in detail, and section 5 introduces the RFLRU scheme which is an enhanced version of previously proposed buffer cache algorithms. Section 6 describes the prototyping environment. Experimented results are presented in section 7 and finally the results are concluded.

II. THE PROBLEM

SSD's internal RAM is used as a buffer to absorb a portion of the read and write requests by the application such as file system. The write performance of NAND flash memory is heavily influenced by the FTL page replacement algorithm. SSD exhibits good read performance on sequential and random workloads and good write performance on sequential workload. However, SSD suffers from random write workload. For example, most write requests from MP3 player and movie files are sequential. However, typically file server write request pattern exhibits combined request of sequential and random, many times the random writes are interleaved between sequential writes. Firstly, SSD random write is much slower than sequential write. Secondly, NAND flash memory can incur only a finite number of erases for a given erasure block. Therefore, increased erase operation due to random write shortens the lifetime of flash memory. Finally, random writes result in higher overhead of internal garbage collection than sequential writes. This is due to the expensive full merge triggered by random write. Full merge is a type of merge operation executed by the garbage collection, which causes extra read, erase and write operation. If the incoming writes are randomly distributed, sooner or later all the erasure blocks will be fragmented which results in large number of live page migration during the garbage collection.

Though several page replacement algorithms proposed, they have not entirely addressed the performance degradation of mixed work load of random and sequential write pattern. To improve the flash write performance on mixed write workload, it is indispensable to detect the locality of the incoming write request and build a sophisticated FTL page replacement algorithm to absorb the mixed workload in alignment to the flash memory characteristics.

III. CONTRIBUTIONS

So it is very essential to contemplate the flash memory characteristics and build a high performance FTL for the evolving general purpose storage system based on flash memory. This research primarily makes the following contributions: Firstly, detect the locality type more precisely on the write workload. Secondly, random and sequential write requests are segregated and different write policies are applied on them. Finally, different write enlargement techniques are applied to reduce the merge cost during the garbage collection process.

For these contributions, a new page replacement scheme called a Random First Least Recently Used (RFLRU) is developed. This scheme is an improvement to the previously proposed block-level replacement schemes. The proposed algorithm reduces the full and partial merge count and increases the switch merge count during the garbage collection process. Also the proposed algorithm improves and springs up the best practices from the previously proposed schemes to reduce the overall cost of page replacement on write. Trace driven experiment is conducted on a SSD simulation environment, and the result shows that RFLRU performs better than BPLRU, FAST and REF page replacement algorithms for the mixed workload of sequential and random write pattern. Effectively, RFLRU brings up the following benefits on flash write operation:

- ✓ Reduces the number of write and erase operations
- ✓ Reduces the costlier merge in garbage collection which in turn reduces the number of live page migration.

IV. RELATED WORK

There have been many researches on the log buffer-based FTLs on write workload. These researches are aimed to improve the write performance and address the merging complexities in the garbage collection process.

FAST [5] keeps a single sequential log block dedicated for sequential update and the rest of the log blocks are used for random writes. FAST scheme shows a significant difference between the worst case block merge time and the best block merge time due to its high block associativity. This has been known for superior performance for random write operations. The key idea of FAST is to rely on the full associativity between data blocks and log blocks in order to

avoid the log block thrashing problem and this increases the log block utilization. SuperBlock FTL [7] scheme demonstrated that the temporal locality can be exploited by allowing the page level mapping in a SuperBlock which is set of consecutive blocks. Then, the cold and hot data are separated automatically into different blocks within a superblock, thus the garbage collection efficiency is improved by reducing the number of full merge operations. However, this approach does not efficiently distinguish the cold pages from the hot pages and addresses the temporal the locality in write workload. LAST [8] scheme tries to alleviate the shortcomings of FAST by providing multiple sequential log blocks to exploit spatial locality in workloads. It further separates random log blocks into hot and cold regions to reduce full merge cost. In order to provide this dynamic separation, LAST depends on an external locality detection mechanism that determines the locality type by comparing the size of each request with a threshold value. However, Lee et al. themselves realize that the proposed locality detector cannot efficiently identify sequential writes when the small-sized write has sequential locality. FAST does not take full advantage of temporal locality of page writes, which results in excessive costly merge operation. FASTer [9] was proposed to address the temporal locality write workloads. It adopts a new hot-cold separation strategy to exploit the dissymmetry in the online transaction processing write requests. When a log block is chosen as a victim for reclamation, by carrying the valid pages from the log block over to a new log block, these pages are given a second chance to be invalidated before being merged to the corresponding data blocks, the log block is virtually extended. S-FTL [10] exploits the spatial locality in the SSD's workloads and significantly reduce page mapping table size without imposing any restriction on page mapping method. By doing so, S-FTL can benefit from the page-level mapping with a low garbage collection cost. It effectively addresses the challenge of high garbage collection cost experienced by the block-level mapping and hybrid mapping such as BAST and FAST. And overcome the challenge of limited cache size experienced by the page-level mapping such as DFTL [11]. Unlike log buffer based FTLs, DFTL is purely page mapped, which exploits temporal locality in enterprise workload. Selective portion of the mapping table is cached for the address translation and the rest of the table is loaded on demand. DFTL improves the locality detection proposed by LAST and identifies the sequential writes when the small-sized write has sequential locality. WAFTL [12] explores page mapping block to store random data and handle large number of partial updates, and block mapping block to store sequential data and lower overall mapping table. Additionally WAFTL explores buffer zone to log the data sequentially and partition data based on threshold.

There are not many researches on the buffer cache in flash based storage system. Jo et al. [13] proposed a flash-aware buffer (FAB) management scheme. In this scheme, the buffers of the same erase blocks are grouped together on LRU (least recently used) sequence and victim groups are larger which are based on the page count on every group. This buffer management policy evicts all the pages of a block at a time. In sequential writes FAB is very effective. PUD-LRU [14] differentiates blocks and judiciously de-stages the blocks based on their frequency and recency so as to avoid the unnecessary erasures due to repetitive updates. PUD-LRU maximizes the number of valid pages in the de-staged block in each erase operation. Kimet al. presented BPLRU [6] buffer management scheme especially for improving the performance of random writes. BPLRU employs a page padding scheme where the log block is padded with some clean pages from the data block to reduce the number of full merges. BPLRU groups the buffer from the same erasure block, and replace them together on SSD. Compared to fully block padding technique used in BPLRU, the partial block padding technique in BPCLC [15] reduces the page padding overhead and therefore improves the I/O performance. REF [16] enforces the buffer cache to evict only the pages that belongs to the victim block. The victim block remains unchanged as far as possible to reduce the block thrashing and block associativity. The pages are reordered to reduce the number of block merges. Since REF uses padding and reordering, this outperforms FAB and BPLRU polices. Reordering the write sequence [17] was proposed to address the out-of-place update problems and the sequential write constraint within a block. The reorder space is inside the SSD and the writes are reconfigured which prohibits write amplification. In BA-GC [18], the garbage collection process considers the contents of a buffer cache. This process prevents the unnecessary page migrations as there are corresponding dirty pages in the buffer cache for pages to be moved by the block merge at the garbage collection time, thus improves the write efficiency. The buffer cache is consulted during the garbage collection process and the garbage collection takes place whenever the log blocks are exhausted or explicitly triggered by the host file system through the firmware TRIM [19] command. TRIM command is designed to enable the operating system to notify the SSD which pages no longer contain valid data due to erases either by the user or operating system itself. During a delete operation, the operating system will both mark the sectors as free for new data and send a Trim command to the SSD to be marked as no longer valid. After that the SSD knows not to relocate data from the affected blocks during garbage collection. This results in fewer writes to the flash, reducing write amplification and increasing flash memory life.

Most of the research works on the page replacement algorithms have addressed the temporal locality in the write workload. In some research works, random writes and write reordering is exploited [17, 18] but insignificantly. They are not optimized for write sequence and the write buffer is mostly used to decrease the write traffic. As write requests are reduced the number of invalid pages are reduced, this extends the SSD lifespan. However, SSD life span is exhausted by the inner-generated write amplification. And the proposed buffer management scheme improves the SSD performance and endurance by erasing less physical block for the mixed workload of random and sequential write.

In summary, current FTL address mapping schemes and page replacement algorithms can show poor performance and device endurance for the mixed workload of sequential and random. This is due to SSD's inner generated write traffic or writes amplification.

V. RFLRU – Random First Least Recently Used Algorithm

In this section, Random First Least Recently Used (RFLRU) algorithm is discussed along with the write sequence analysis in real world enterprise storage system. RFLRU algorithm employs sequence detection mechanism

and three new write techniques are proposed, namely buffering the spatial locality; write random ahead and different write enlargements. By employing these techniques, sequential write workload is segregated from the mixed (random and sequential) workload. And from the segregated sequential workload, NAND friendly write pattern is generated. NAND friendly write pattern is a sequential pattern whose request size aligns with SSD erasure block boundaries. NAND friendly write pattern increases the probability for switch merge to improve the overall I/O performance in flash memory based SSD.

A. Write sequence analysis

In various data storage applications, a storage device receives data from multiple data sources in an interleaved fashion. For example, the Serial Attached SCSI (SAS) [20] protocol allows a host to send data associated with multiple write commands in an interleaved manner. SSD operating in accordance with the SAS protocol will receive multiple data streams from the host operating system that are interleaved with each other. As yet another example, SSD may receive multiple data streams concurrently over multiple different physical ports, so that the data of the different streams may be interleaved with one another when arriving in the SSD. And among the write data streams, some might be random and others might be sequential in nature. The inefficiency of the SSD write performance is attributed from different write access patterns. Random writes are harmful to SSD [4] as it incurs more live page migration and erases during the garbage collection process. Most write request in media player are sequential in nature with small amount of random pattern in it. Whereas in general purpose computing has write pattern of high temporal locality, sequential locality and many times there are many random writes interleaved between sequential writes. By processing the de-interleaved data streams, write performance and subsequent read performance can be improved significantly.

B. Sequence Identification - Defragmentation and Reconstruction

In reality, most random write workload exhibits certain level of sequential pattern. A sequential workload is generally a host workload that includes request(s) of data at logical addresses that are substantially sequential. Many times the sequential work load is interleaved by the multitasking nature of the operating systems. The idea is to observe the characteristics of the I/O request pattern from the operating system. And de-fragment the various write streams, by identifying the data source with which each data item is associated. In other words, the proposed algorithm reconstructs the successive stream of data items associated with each data source or program. Figure 1 shows the high level work flow in the proposed RFLRU algorithm. Multiple programs generate sequential and random write workload. Host file system receives this in a pseudo random stream and the sequential sequence is identified and reconstructed. And the write operation is processed on these data streams individually.

An effective way to detect the sequential patterns is to maintain the recent history of write. The write buffers contains the data for which write requests are issued by the application program and yet to be written into the flash memory. Upon arrival of the new write request, RFLRU refers to the recent write history to decide if the current write is within the proximity of the previous write to determine sequential nature. In order to separate sequential streams from intermingled pseudo random stream, RFLRU uses a Pre-write Context (PWC). This PWC represents the run-time object associated with each detected sequential stream.

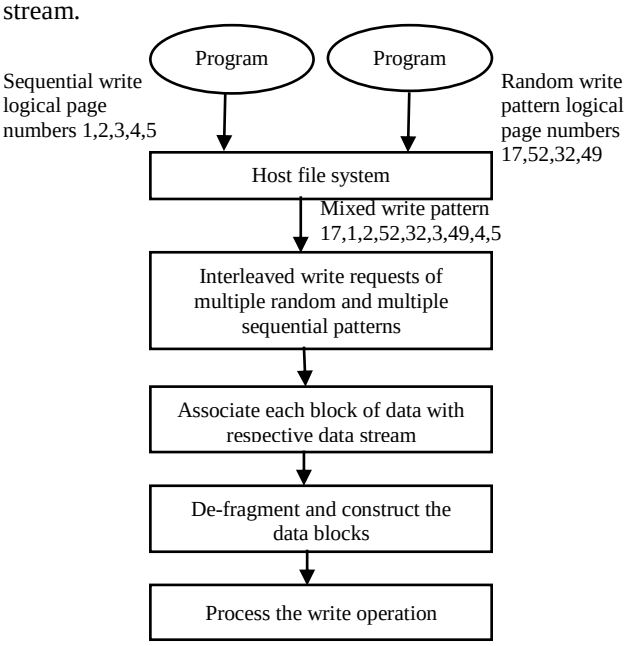

Fig. 1. RFLRU-Different stages in the write sequence

Each PWC includes the critical parameters to identify the sequential locality in the write workload. The locality type of each write request is related to the size and frequency. Usually the small writes have high temporal locality and large writes have high spatial locality. And other parameter is closeness in space and time of the logical addresses in the write requests. Closeness in space is measured as a distance between the write requests. If the requests are continuing in time or space without interruption, then there involved a sequential write. How close the request with respect to time is another parameter. Though the workload exhibits sequential locality, the write requests cannot be buffered for a long duration which is not close in time. Finally, PWC contains pointers for maintaining itself in different data structures such as indexing trees and PWC queues. When a new write request comes, the request address is compared against the most recent pre-write addresses of the existing streams to see if there is an established write sequence. If an extension is found, the total sequence length of the stream is increased by current request's length. Then statistics of this stream are updated with attributes of this new write request. When no PWC is available, new PWC is created. And the locality type is determined by comparing the write size with the predefined threshold value. Requests larger than the threshold value are directed to the sequential buffer and the rest of the requests are directed to the random buffer. At times, even though the incoming write data size is less than the threshold value and has an established write sequence, the random buffer is moved to sequential area. PWC keeps a record of all write requests in this stream for prediction of the next write request's closeness in space, e.g. recent request's length and timing interval.

Finally, a PWC contains address pointers for maintaining itself in different data structures such as indexing trees and

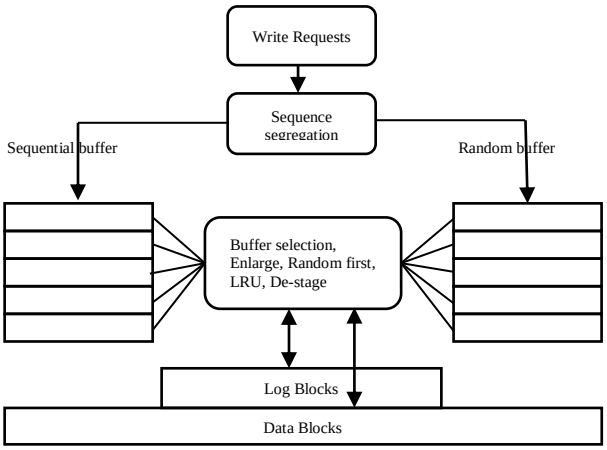

Fig. 2. The overall architecture of RFLRU scheme

queues. When a new write request arrives, the request address is compared against the pre-write addresses of the existing streams to see if it extends any of them. Figure 2 shows the typical behavior of write requests which are generated by multiple applications and received by the FTL and several stages in the proposed RFLRU buffer replacement scheme. The sequence identification module creates a new PWC if the incoming write cannot extend to any of the existing PWC's.

The primary complexity of this algorithm is to design an efficient data structure to search the existing PWCs and remove outdated PWCs. This is needed to remove the PWCs created and kept for non-sequential requests. Indexing of PWCs is done using a balanced tree. This decreases the search time to locate the appropriate PWCs and the memory consumption on the data structures. Purging process runs in the background to delete useless PWCs including both obsolete sequential ones and non-sequential ones.

As shown in figure 3, PWC's are maintained in two queues: active and passive queues. And PWC are managed in these queues using furtherance and demotion policies. On every new write request, a new PWC is created and it is added to the furtherance queue. As PWCs sequential pattern is found it is retained in furtherance queue, if not found it is moved to the demotion queue. These two queues are ordered based on the LRU (least recently used) policy.

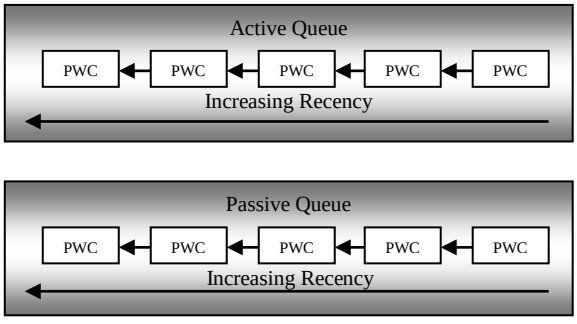

Fig. 3. PWC data structures. Each PWC is linked to either active or passive queue. And the PWC's are indexed by a balanced tree data structure for efficient maintenance

When the upper bound of number of active PCs is reached, the algorithm moves the PWCs in batches from active queue into passive queue. So that PWCs that are not sequential or very slow sequential are purged earlier from passive queue. For a write request, PWC is found in the passive queue, it is moved to the active queue. To search a PWC, RFLRU uses a balanced index tree to organize all the PWCs. The balanced tree ensures that operations, such as deletion, insertion, and search, can complete in O(log n) time. Since there is a constant upper bound of the number of PCs, the indexing and purging process involves only a small overhead.

C. Write Random Ahead

In this section the random ahead feature of RFLRU page cache replacement scheme is described. Write sequence segregation will output two different workload buffers namely, sequential and random. In the flash memory write operation (de-stage), random blocks are given more priority than the partially occupied sequential blocks. That is random buffer is written into the SSD log block ahead of the partially occupied sequential block. And the partially occupied sequential write block is queued until the block is completely occupied with write data or the threshold timer expired. The write buffers are evicted based on the LRU policy. By queuing up the sequential work load, the write bandwidth is utilized for random data. Once the sequential buffer reaches the erasure block size, the data is written into the flash memory. LRU page cache replacement policy is not effective for mixed write workload of sequential and random. In RFLRU, sequential write is transformed into buffered write and random write is preferred for write. Fully occupied sequential write on the flash memory log lock will produce switch merge during the FTL garbage collection process. Write random ahead feature of RFLRU is depicted in figure 4. In addition to the above described random ahead write scheme, the RFLRU algorithm has additional polices for the eviction. This prevents the partial merge triggered by the partially occupied sequential blocks. The proposed algorithm works on random buffers, at times the sequential buffers is not ready for de-stage. This random ahead feature is primarily useful on the slow and interleaved sequential writes. Applying the proposed de-stage polices on sequential and random write queue reduces the number of SSD merge operation in the garbage collection process, thus enhancing the SSD write performance.

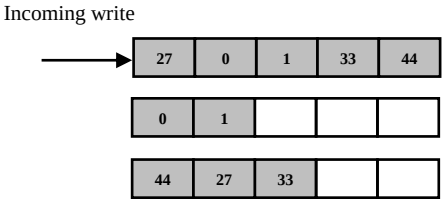

Fig. 4a. RFLRU- Sequential pages 0&1 are buffered in sequential buffer and the random pages 44,27&33 are buffered in random buffer

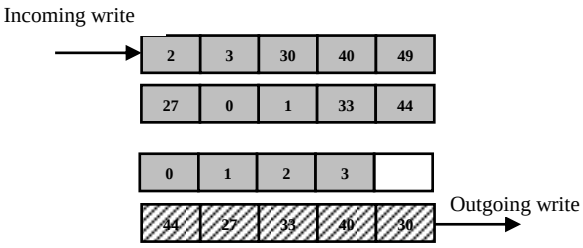

Fig. 4b. RFLRU-Sequential pages 0,1,2&3 are buffered and deferred for write. And the random block which has pages 44, 27,33,40&30 are preferred for write

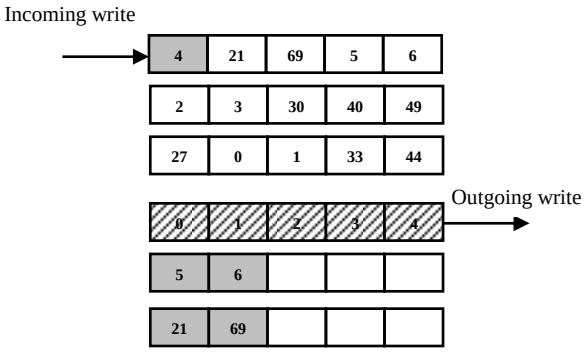

Fig. 4c. RFLRU-The deferred sequential block which has pages 0,1,2,3&4 are preferred for write. And the subsequent sequentil and random pages are deferred for write.

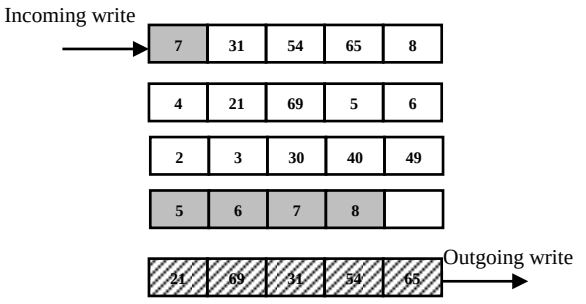

Fig. 4d. Sequential pages 5, 6, 7 & 8 are deferred and random block which has pages 21,69,31,54&65 are preferred for write

D. Comparison among LRU FAST, REF and RFLRU

Fig. 5 compares the behaviors of buffer cache under LRU, REF and RFLRU policies using an example write sequence. More pragmatic comparisons are done in the experimental results in section 7. In figure 5a LRU triggers full merge, though REF reorders the write it triggers partial merge. Reorder page numbers 0,1,2,3,27 triggers partial merge because page 27 belongs to different data block and RFLRU triggers switch merge. In figure 5b, LRU triggers full merge. Here REF triggers switch merge through reordering. However RFLRU will also trigger the switch merge either through padding or utilizing the next write request from host operating system

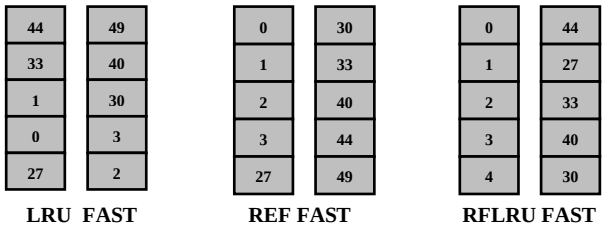

Fig. 5a. Comparison of LRU FAST, REF and RFLRU schemes

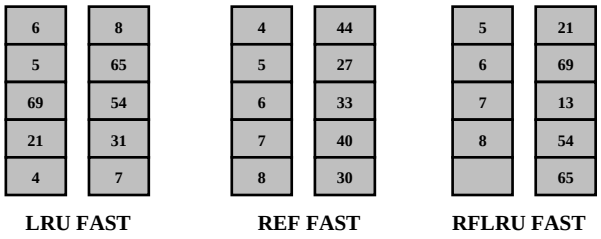

Fig. 5b. Comparison of LRU FAST, REF and RFLRU schemes

E. RFLRU Buffer Cache De-stage Control

RFLRU employs policies for de-stage sequence and de-stage interval for effective write buffer cache utilization and enhanced write performance. Sequential buffers are de-staged based on multiple parameters namely average accumulation interval, buffer size, threshold time, number of valid pages or load factor and cache utilization ratio as shown in figure 6. Every block has maximum wait time or threshold time. Completely occupied sequential blocks are de-staged with high priority. The average block accumulation interval is measured by the average update time of the accumulated data. If the data accumulation time is 50% above the average block accumulation time, that sequential buffer is ready for de-stage. Moreover, the sequential blocks are de-staged based on the cache utilization ratio. This utilization ratio is calculated by the number of valid data to block capacity. The block which has the maximum number of valid data is chosen for de-stage. Every block has a maximum wait time, when time elapsed, it is de-staged irrespective of other de-stage parameters. The above said de-stage policies are applied on the LRU list. The de-stage policies effectively utilizes the cache and assists to reduce the complexities in the garbage collection. RFLRU page replacement scheme is devised for the SSD write buffer. That is RFLRU allocates and manages buffer cache memory only for write requests from the host operating system. And for the read request, the data is directly read from SSD as the read operation is a low time consuming compared write operation.

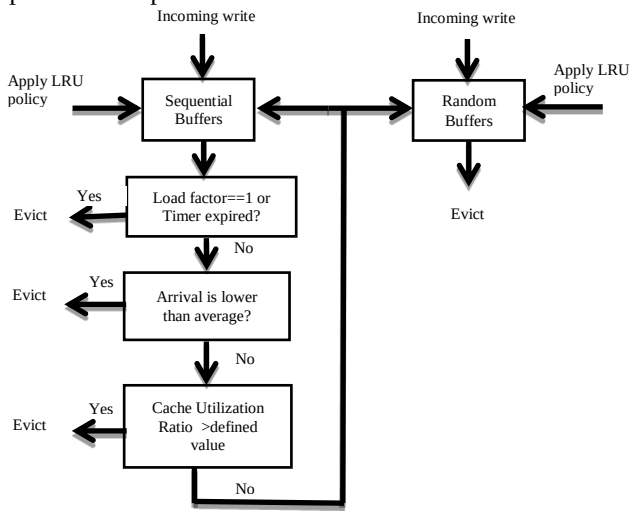

Fig. 6. RFLRU buffer de-stage flow Diagram

F. Merging Techniques for Switch Merges in Garbage Collection

In RFLRU, the LRU policy is applied on entire block rather than on individual page. A victim sequential buffer may be completely occupied or partially occupied. The blocks are aligned with SSD erasure block boundaries. That is the buffer page offset and SSD erasure block offsets are aligned. LRU policy is applied to the dirty pages on the block and not on the clean pages. If a dirty page is added to the block all pages in the same block range are placed at the head of the LRU list. Completely occupied sequential buffers are written into SSD, this triggers the switch merge. However, enlargement techniques are applied on the partially occupied sequential buffers to merge the buffer with the clean pages from the SSD log and data block. There

are primarily three techniques are used namely Early, Late and Hybrid enlargements.

1) **Early Enlargement:** Partially occupied sequential buffers are enlarged through padding the clean pages from the SSD data block; we call this as an early write enlargement. For the reason that the sequential buffer's data is padded with SSD data block before the buffer cache data is written into SSD as shown in figure 7. A bit similar method called block padding [8] is proposed to improve the block utilization. Block padding manages all the pages in buffer cache by the block level LRU policy in FTL. The block which is not accessed for the longest time is selected as a victim block. BPLRU invokes switch merge through block padding. In REF [23] the padding is done for the blocks which has more than 80% load factor. However in our approach the enlargement is done intelligently based on the following three important criteria

- Enlarge the suffix pages in the write sequence if the incoming write is attenuated for that particular sequence
- Enlarge the prefix pages in the write sequence if no write sequence preceded
- Discard the enlargement, if the block eviction is forced due to high cache utilization

These criteria's identifies the page sequence for which the write is either not expected or the current write sequence is terminated. This gives the determination that the write is not expected in the near future. Unlike in BPLRU and REF, the enlargement is done using the above mentioned rationale. For the clean page addition, the data is brought from the SSD and the whole erasure block is written into SSD. This significantly improves the block utilization compared to LAST [8] and FAST [5] scheme and hence enhanced write performance. Early enlargement prevents the garbage collection for partially occupied write blocks.

2) **Late Enlargement:** In contrary to early enlargement, in this technique the clean pages are brought from the log block into the buffer cache as shown in figure 8. In other words already written log blocks are padded with the sequential buffer cache data. This prevents the unnecessary live page migrations. The SSD log block is looked up on every partial block eviction. The log blocks are identified if they have corresponding page which is yet to be padded for the sequential buffer. The clean pages are read from log block and the erasure block is written along with the dirty pages in the buffer cache. The important difference from the early enlargement is that every partial sequential block eviction, the log block is consulted for padding the clean pages. This merge process reduces the live page migration on the garbage collection process. Moreover this merge process is triggered on every partial sequential buffer eviction, though it is named as late enlargement, the merging happens well ahead of the garbage collection process. This enlargement technique comes in handier when the sequential writes are discrete and slow in nature. At times the sequential write is separated by time; the discrete sequential data will spread across multiple SSD log blocks. Essentially in the garbage collection process, all these log blocks are merged and the final data block is created. However, garbage collection process is either triggered by the host file system or at times the log blocks are near completely utilized. However typically the discrete sequential write happens in a short span of time and eventually the late enlargement reduces the live page migrations and saves quite a few SSD erase/program cycles. Late enlargement prevents the garbage collection for the log block data.

3) **Hybrid Enlargement:** RFLRU introduces a two way enlargement concept where the partially occupied sequential buffer from the buffer cache is enlarged with the clean pages from the log block and data block as shown in figure 9. And the final write block is created and written into SSD. Similar to the previous enlargement techniques, this also reduces live page migration and eliminates SSD erase/program cycle. However this merging technique sets in when the late merge yields partial erasure blocks; otherwise the environmental factors and use cases are same as early merge.

Before the write data enlargement, we should identify the set of associated log blocks, $\mathbb{A}(V_i)$, of a victim block Vi. $\mathbb{A}(V_i)$ is composed of all log and data blocks which has the clean pages to the corresponding empty pages in the victim block. We represent $\mathbb{A}(V_i)$ formally as follows

$$\mathbb{A}(V_i) = \begin{Bmatrix} L_k | \exists p_l \; s.t \; p_l \in V_i^{empty} \wedge p_l \in L_k^{clean}, \\ D_j | \exists p_l \; s.t \; p_l \in V_i^{empty} \wedge p_l \in D_j^{clean} \end{Bmatrix}$$

Where D_j means a data block, L_k means log block and p_l means a page. V_i^{empty} denotes the empty pages in the victim block and L_k^{clean} and D_j^{clean} denotes the clean pages from the log block and data block respectively. For example, $\mathbb{A}(V_i)$ in figure 8 is the log block {L0}. For each empty page in the victim block V_i, the clean pages are read from the associated log and data block and moved into victim block V_i and finally the victim block is written into SSD.

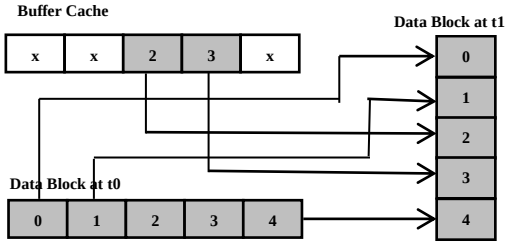

Fig. 7. Early Enlargement-Prefix pages 0&1 and suffix page 4 are padded, this is executed when no preceding and succeeding write sequence is observed on the block and the block is not evicted due to high cache utilization

VI. PROTOTYPE DESIGN AND IMPLEMENTATION

A. Experimental Setup

Trace driven simulation was implemented by extending the FlashSim [21], a simulator for NAND Flash-based SSD's. And FAST [5] was chosen as a base FTL simulation

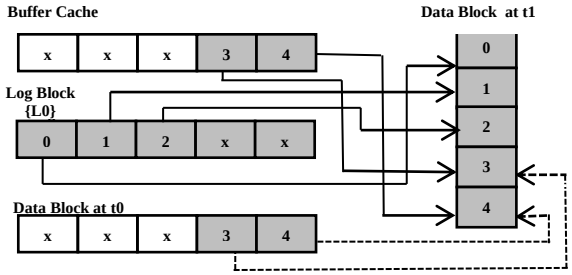

Fig. 8. Late Enlargement-Sequential buffer is enlarged using the clean pages from the SSD log block. Buffer cache and log blocks are merged to create the data block

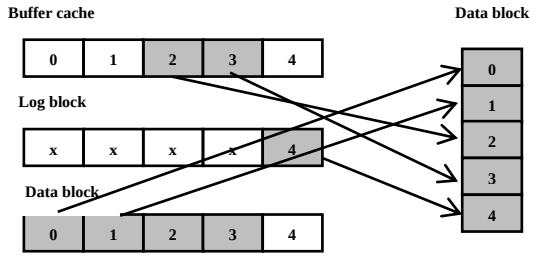

Fig. 9. Hybrid Enlargement -Sequential buffer is enlarged using the clean pages from log and data block

type. In addition to RFLRU implementation, for the benchmark comparison, the page replacement policies namely, LRU, block padding, REF are also implemented. The SSD configuration values are identified in the table I with a reference from [22]. Typically the capacity of SSD's inner DRAM ranges from 16MB to 64 MB, for our simulation runs we have configured from 2 MB to 32MB.

TABLE I
FLASHSIM SPECIFICATIONS

Traces	Read (sectors)
Page read to register	25µs
Page write from register	200µs
Block erase	1.75ms
Die (SSD)size	2GB
Block size	256KB
Page size	4KB
Data register	4KB
Number of Erase cycles	200K
Buffer cache	2MB-32MB

B. Workload Traces

Traces are collected from several different environments and the characteristics of the traces are listed in table 2. The traces represent typical applications. Synthetic trace is acquired using Iometer on Linux Ext4 file system for producing homogeneously distributed random and sequential pattern. It creates large file with complete partition size and then overwrites the sectors randomly, the writes are configured for 4K with 20% random and 100% sequential workloads are created for this test. The Flashsim's Run test script is executed which basically runs the file which has the traces. Upon successful completion of the simulation, the .outv file is analyzed and the results are compared.

TABLE II
CHARACTERISTICS OF TRACES

Traces	Read (sectors)	Write (sectors)	Read/Write
Exchange	27246735	33823876	1/1.24
Financial OLTP(Online transaction processing)	21510794	77982370	1/3.6
Iometer-20% random	10167847	49101834	1/4.8
Iometer-100% sequential	12149099	51100398	1/4.2

VII. EXPERIMENTAL RESULTS

SSDs write behind cache is aimed at improved performance and extending the flash memory lifespan. In this study, the write performance is measured by number of erased blocks, average write response time, total number of pages read and write and number of live page migration. Using these parameters, we characterized the behavior of the proposed buffer management scheme. And the simulation runs are repeated for varying buffer cache size and compared against previously proposed buffer management schemes.

A. DRAM size and performance

We have analysed the buffer cache size impact on write performance; figure 10 shows the execution time for each of the workloads with cache sizes varying in a large range. As RFLRU uses buffer cache to hold the segregated work load, a large cache would help to hold larger portion of workload's working set in the cache until one of the de-stage specified threshold value is reached. When cache size is small, the sequential buffers are evicted before it grows till the erasure block size for the IOR (interleaved or random) workload. This triggers the write enlargement and inner generated write traffic or writes amplification. However the write enlargement is done intelligently by identifying the write attenuation of ongoing sequential writes. When the buffer cache size is increases beyond 16MB, further increase in response time is observed. This demonstrates that RFLRU performs better when the cache size increases especially with IOR workload. Figure 10 shows the execution times for each evaluated schemes by varying the size of the buffer cache between 2MB to 32 MB. In general the execution time declines at times the buffer cache size is more. Overall, RFLRU shows better performance than the rest of the buffer management schemes across all buffer sizes. Particularly, RFLRU demonstrates good I/O performance when the size of the buffer cache is large. For example, BPLRU and REF require 32 MB to achieve the write performance, whereas at 16MB RFLRU attains the same performance. The increased performance is observed when there is more buffers to queue the identified sequence in the write pattern. However at smaller buffer zones, RFLRU performs marginally better than BPLRU and REF as the size of the buffer cache influences the performance. However the performance of LRU FAST is not greatly affected by the buffer cache size. For the 100% sequential load, all the buffer management performs almost similarly.

Table 3 represent the average response time in Iometer 20%random workload with varying buffer cache size. The response time of the request refers to the time period between its arrival at the device and the completion of the

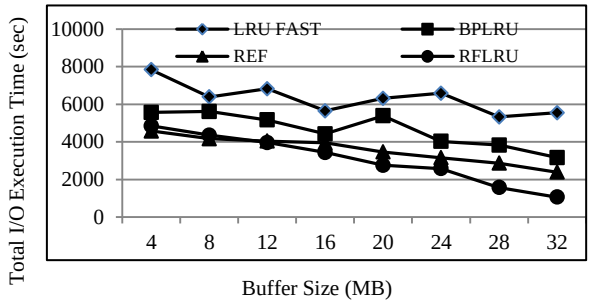

Fig. 10a. Exchange I/O Execution time comparison for different page replacement schemes

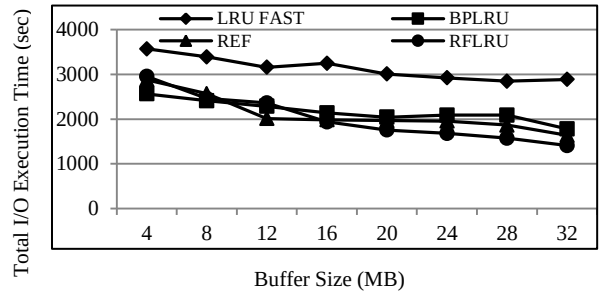

Fig. 10b. Finance I/O Execution time comparison for different page replacement schemes

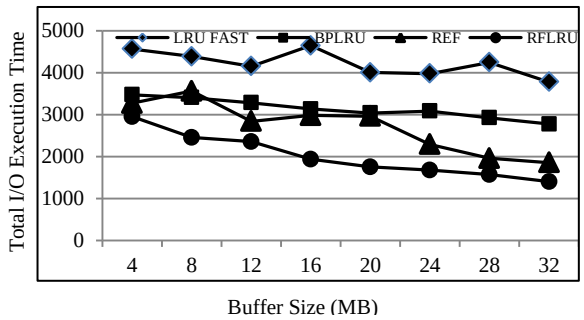

Fig. 10c. Iometer20%random I/O Execution time comparison for different page replacement schemes

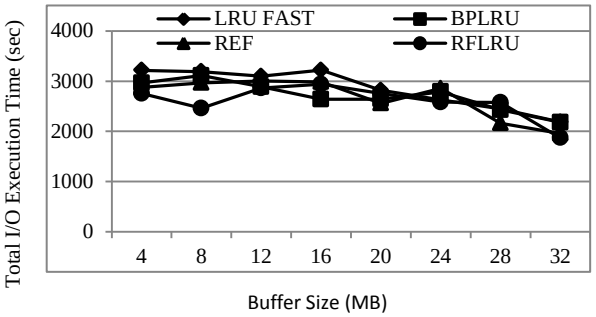

Fig. 10d. Iometer 100% sequential I/O Execution time comparison for different buffer cache management schemes

write operation. This time includes the locality detection, write buffering and garbage collection. As we increase the size of buffer cache, RFLRU surpasses FAST, BPLRU and REF schemes. It may be argued that on the slow writes, RFLRU's response time is poor due to the sequential buffering nature. However this can be improved by increasing the wait time on the LRU queue. The latency of the write workload heavily depends on the full merge triggered by the write sequence. The write response increases on high buffer sizes as the garbage collection over head is reduced due to large amount of switch merge invocation and frequent log block merging is reduced. The faster response time is also attributed by the specified enlargement techniques on the partially occupied data blocks. RFLRU accomplishes marginally better response on lower buffer cache size, whereas BPLRU and REF need more buffer cache to achieve the similar response time. From the response time table 3, it is understood that RFLRU performs better as the buffer cache size goes up.

TABLE III

AVERAGE RESPONSE TIME ACROSS DIFFERENT BUFFER SCHEMES ON ALL WORK LOADS

Average response time (ms)	4MB	8MB	12MB	16MB	20MB	24MB	28MB	32MB
LRU FAST	4.84	4.46	4.2	3.92	3.7	3.2	2.96	3.17
BPLRU	3.4	3.25	3. 18	3.1	2.21	1.75	1.09	1.13
REF	3.5	3.18	3.25	2.98	2.01	1.28	0.78	0.97
RFLRU	3.9	3.01	2.76	2.01	1.48	0.67	0.38	0.21

B. Impact of RFLRU's Random Ahead Technique

Using two different buffers for sequential and random write sequences, the write ahead random technique can recognize more and longer write sequences. In other words, with the given cache size, the technique is expected to hold larger portion of sequential write pattern so this reduces the live page migration cost in FTL garbage collection process. As shown in figure 10, the *financial* workload receives most significant improvements, up to 82% compared to pure sequential writes. As financial workload has many random writes, it can easily break long sequences, leading to relatively low full merges.

C. Write Back Buffer Schemes Comparison

We compare the RFLRU with FAST LRU, BPRLU and REF buffer management policies. BPLRU is a classical write buffer policy for SSD, it has been extensively evaluated. REF is a recently proposed policy which has write ordering along with BPLRU's padding technique. The experimental results are presented in figure 11 and 12. As shown in figure 11, RFLRU erases least physical blocks on most of the workloads. To further explain why RFLRU achieves this lower erase count, we compare the number of live page migration, which is explained in figure 12. Merge operation in the garbage collection process, causes extra read and write generated by the background operations. A superior buffer cache page replacement scheme should help to reduce the number of page read and write, thus alleviate the write amplification. As merging a block indicates migrating pages from one physical block to another physical block, which is a live migration. BPLRU arranges the dirty and clean pages in a logical block group, these groups are the atomic units replaced by the LRU buffer replacement policy. However, though the block associativity is low, BPLRU increases the inner generated read traffic for the clean page padding, and this affects the overall write throughput. In FAST scheme the block associativity of each log block is typically high because the dirty pages are written using the simple LRU policy.

However in REF the dirty pages are reordered to reduce the high log block association. Whereas in our proposed RFLRU buffer management scheme, the log block associativity is naturally reduced by identifying the sequential write pattern and buffering the sequential data stream. In addition to this, RFLRU uses enlargement

different techniques to increase the block utilization and the merging activity is done ahead of the garbage collection

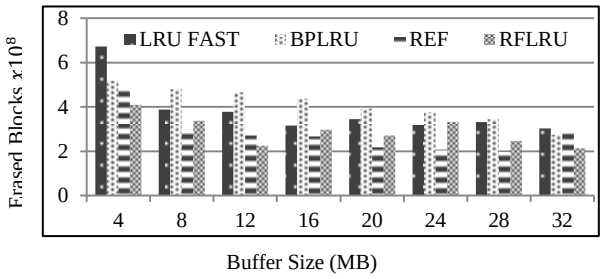

Fig. 11a. Exchange - Erase blocks comparison for different buffer cache management schemes

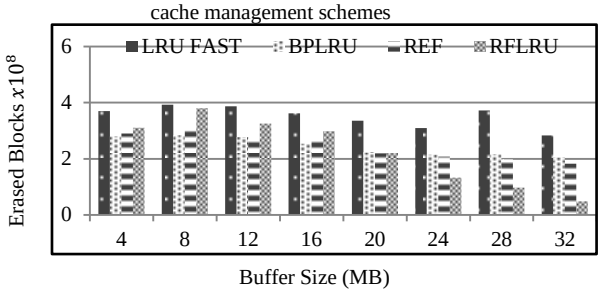

Fig. 11b. Finance - Erase blocks comparison for different buffer cache management schemes

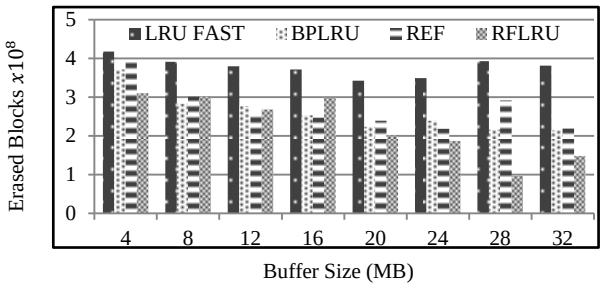

Fig. 11c. Iometer 20%random - Erase blocks comparison for different buffer cache management schemes

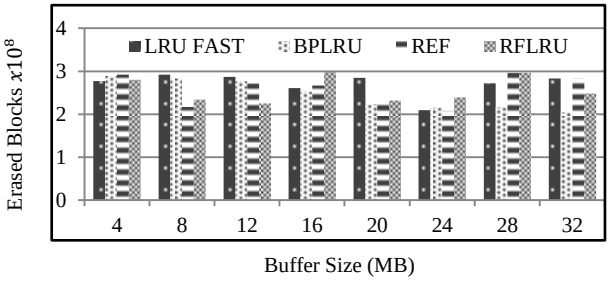

Fig. 11d. Iometer 100% sequential - Erase blocks comparison for different buffer cache management schemes

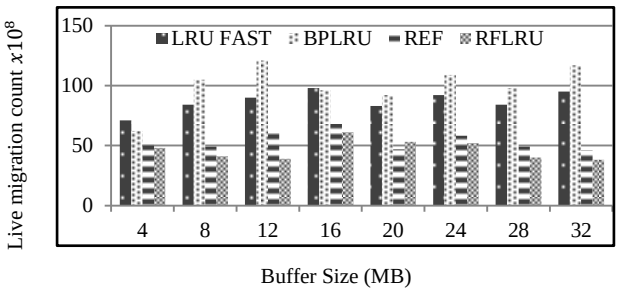

Fig. 12a. Exchange -Live page migration count comparison for different buffer cache management schemes

process. Though the block associativity is one of the crucial parameter for comparing the performance across the buffer management schemes, and block associativity is proportional to the number of page migrations in the merging process triggered by the garbage collection. So we have used the live page migration count for comparing the performance across different schemes. RFLRU exhibits low

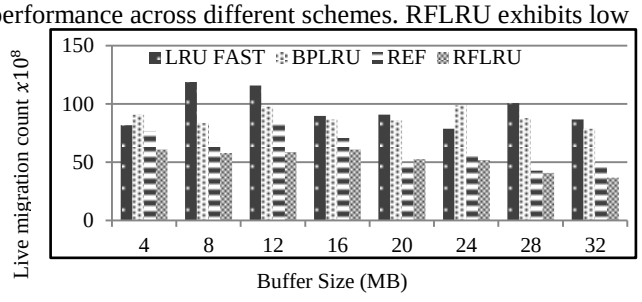

Fig. 12b. Finance - Live page migration count comparison for different buffer cache management schemes

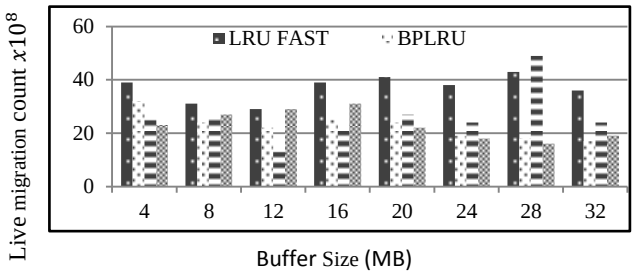

Fig. 12c. Iometer 20%random - Live page migration comparison for different buffer cache management schemes

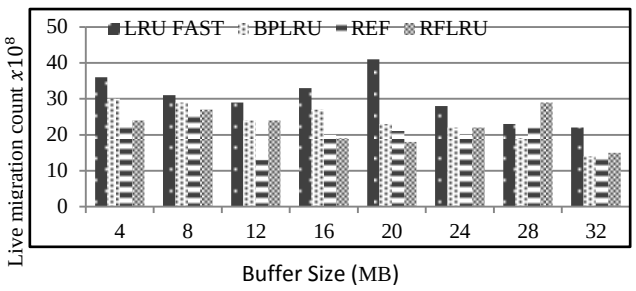

Fig. 12d. Iometer 100% sequential - Live page migration count comparison for different buffer cache management schemes

live page migration on most of the workloads, especially on OLTP work load. Whereas in pure sequential workload the performance improvement is not very significant, refer figure 12d, as the page migration count is almost equal across different buffer management schemes.

D. Assumptions and Discussion

Theoretically the logical sequential pages are located consecutively in SSD and the file system allocates pages sequentially. Again this assumption is also depends on the FTL and the workload characteristics. In continuous sequential workloads, the data placement in SSD is most likely sequential. So the logical sequential write will get translated into physical sequential write, this leads to reduced write amplification. On discontinuous sequential work workloads, the data on the SSD may not be placed adjustment to each other. So identifying the local infrequent sequential pattern in the workload sometimes may not be yield desired write optimization. For example, few file systems maintain an in-memory pre-allocation for every file data. This influences the sequential placement of physical pages in flash memory though the generated sequential writes are discrete in nature. This in-memory pre-allocation helps the slow sequential writes to get written consecutively in SSD, so file system's intervention is required for sequential placement of data in the flash memory.

In LRU write policy, the page references are kept in sorted temporal order. When a page frame is accessed, the page needs to be moved to the most recently used position. This operation requires a global lock to protect the data structures from the concurrent access. Since the page access are common and such a frequent rearrangement of data structure would be expensive. We have used the LRU policy against block level instead of page level, so the LRU approximation algorithms such as Clock [23] may not be required.

Though RFLRU increases the scope for switch merge, the scope is determined by the FTL buffer size, and the buffer cache management is heavily influenced by the host file system through the flush command. If the host file system frequently issues the flush command, the write performance will get heavily affected negatively. This is because the dirty pages have to be written immediately into flash array. Readers may argue that the buffering high temporal locality workload might reduce the number of writes as proposed in the previously proposed algorithms. In RFLRU, sequential buffer eviction is based on multiple parameters listed in de-stage control, however we have not factored the high temporal locality workloads on the random buffers as the study is primarily intended to address the interleaved sequential writes along with little amount of random workload generated from the meta data modification on the file system.

VIII. Conclusion

The proposed RFLRU page replacement scheme is an enhancement of LRU FAST, BPLRU and REF based FTL's for flash memory based SSD. This research makes the following contributions namely, sequence identification, write random ahead, techniques for SSD write enlargement and buffer cache de-stage rules. By analyzing the incoming write sequence, the interleaved sequential write patterns are identified and buffered. And the partial sequential blocks are enlarged with the clean pages through early or late or hybrid enlargement techniques and written into flash memory. More importantly the random data is preferred for write while buffering the sequential data stream; this random ahead technique addresses the slow and interleaved sequential write. The simulation result shows that the proposed scheme reduces the SSD's inner generated write traffic with reduced erase/programs cycles. Additionally, the improved write performance comes up with no increase in SSD log block area with better write response time.

REFERENCES

[1] Eran Gal, Sivan Toledo, School of Computer Science, Tel-Aviv University, "Mapping Structures for Flash Memories: Techniques and Open Problems". *In Proceedings of the IEEE International Conference on Software Science, Technology and Engineering, 2005*.

[2] Understanding the Flash Translation Layer (FTL) Specification, Application Note, Intel Corporation, 1998.

[3] Min, C., Kim, K., Cho, H., Lee, S., and Eom, Y," Sfs: Random write considered harmful in solid state drives*". In FAST'12: Proceedings of the 10th Conference on File and Storage Technologies, 2012.*

[4] F. Chen, D. A. Koufaty and X.D.Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives", *In Proc. of SIGMETRICS'09*, 2009.

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

[6] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," *in Proc. of USENIX Conference on File and Storage Technologies, pp. 239-252, 2008.*

[7] J. Kang, H. Jo, J. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory", *In Proc. of ICES'06*, 2006.

[8] S. Lee, D. Shin, Y. Kim, and J. Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems",In Proc. SPEED'08, Feburary 2008.

[9] Sang-Phil Lim, Sang-Won Lee, Bongki Moon, "FASTer FTL for Enterprise-Class Flash Memory SSDs", *International Workshop on Storage Network Architecture and Parallel I/Os 2010*.

[10] Song Jiang, Lei Zhangy, XinHao Yuany, Hao Huy and Yu Chen, "S-FTL: An efficient address translation for flash memory by exploiting spatial locality",*IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), 2011.*

[11] Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar,"DFTL:a flash translation layer employing demand-based selective caching of page-level address mappings",*In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2009), Washington, DC, USA, March 7-11, 2009.*

[12] Qingsong Wei ; Bozhao Gong ; Pathak, S. ; Veeravalli, B. ;Lingfang Zeng ; Okada, K,"WAFTL: A workload adaptive flash translation layer with data partition", *IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), 2011.*

[13] H. Jo, J. U. Kang, S. Y. Park, J. S. Kim, and J. Lee. "FAB: Flash-aware buffer management policy for portable media players," *IEEE Transactions on Consumer Electronics, vol. 52, no. 2, pp. 485-493, 2006.*

[14] Jian Hu, Hong Jiang, Lei Tian and Lei Xu Department of Computer Science & Engineering University of Nebraska – Lincoln,"PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD", *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE International Symposium, Pages 69 - 78, 2010.*

[15] Hui Zhao, Peiquan Jin, Puyuan Yang, Lihua Yue, "BPCLC: An Efficient Write Buffer Management Scheme for Flash-Based Solid State Disks", *International Journal of Digital Content Technology and its Applications Volume 4, Number 6, September 2010.*

[16] Dongyoung Seo, Samsung Electron, Suwon, Dongkun Shin,"Recently-evicted-first buffer replacement policy for flash storage devices",*IEEE Transactions on Consumer Electronics, Volume:54, Issue:3, Aug 2008*.

[17] Zhi-Guang Chen, Nong Xiao, Fang Liu, Yi-Mo Du,"Reorder Write Sequence by Hetero-Buffer to Extend SSD's Lifespan", *Journal of computer science and technology, Jan 2013.*

[18] Sungjin Lee, Dongkun Shiny and Jihong Kim,"Buffer-Aware Garbage Collection for NAND Flash Memory-Based Storage Systems",*IEEE Transactions on Computers, volume-62, Issue-11 ,September 2012.*

[19] F. Shu and N. Obr. "Data set management commands proposal for ATA8- ACS2", *http://www.t13.org/, 2007.*

[20] http://www.scsita.org/library/white%20papers/SAS_SATA%20Compatibility.pdf

[21] Youngjae Kim, Brendan Tauras, Aayush Gupta, Dragos Mihai Nistor, Bhuvan Urgaonkar, "FlashSim: A Simulator for NAND Flash-Based Solid-State Drives", *Advances in System Simulation, 2009. SIMUL '09. First International Conference on 2009.*

[22] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis,M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In Proc. of USENIX'08, June 2008.

[23] Biplob Debnath, Sunil Subramanya, David Du, David J. Lilja, "Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks", *In 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2009.*

Arul Selvan Ramasamy received the Bachelor of Engineering degree in Electrical and Electronics Engineering from Madras University, Chennai, India in 1996. He received the Master of Engineering degree in Applied Electronics from PSG College of Technology, Coimbatore, India in 2000. At present, he is a doctoral research scholar at Anna University Chennai, India. His research interest includes de-duplication, context aware solutions, file systems and flash memory.

Dr.K.Porkumaran received the Bachelor of Engineering degree in Instrumentation & control Engineering from Madras University, Chennai, India in 1996. He received the Master of Engineering degree in Control Systems from PSG College of Technology, Coimbatore, india in 2000. He received Doctor of Philosophy in Control Systems Engineering from PSG College of Technology, Coimbatore, India in 2006. He is currently a professor and principal at Dr. N.G.P. Institute of Technology Coimbatore, India. His research interests include embedded software, multimedia and real-time systems, storage and file systems.