

Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks

Biplob Debnath, Sunil Subramanya, David Du, David J. Lilja

University of Minnesota, Twin Cities, USA

Email: biplob@umn.edu, {subram, du}@cs.umn.edu, lilja@umn.edu

Abstract—Solid State Disks (SSDs) using NAND flash memory are increasingly being adopted in the high-end servers of datacenters to improve performance of the I/O-intensive applications. Compared to the traditional enterprise class hard disks, SSDs provide faster read performance, lower cooling cost, and higher power efficiency. However, write performance of a flash based SSD can be up to an order of magnitude slower than its read performance. Furthermore, frequent write operations degrade the lifetime of flash memory. A nonvolatile cache can greatly help to solve these problems. Although a RAM cache is relative high in cost, it has successfully eliminated the performance gap between fast CPU and slow magnetic disk. Similarly, a nonvolatile cache in an SSD can alleviate the disparity between the flash memory’s read and write performance. A small write cache that reduces the number of flash block erase operations, can lead to substantial performance gain for write-intensive applications and can extend the overall lifetime of flash based SSDs. This paper presents a novel write caching algorithm, the Large Block CLOCK (LB-CLOCK) algorithm, which considers ‘recency’ and ‘block space utilization’ metrics to make cache management decisions. LB-CLOCK dynamically varies the priority between these two metrics to adapt to changes in workload characteristics. Our simulation based experimental results show that LB-CLOCK outperforms the best known existing flash caching algorithms for a wide range of workloads.

I. INTRODUCTION

Flash memory has rapidly increased in popularity as the primary non-volatile data storage medium for mobile devices, such as cell phones, digital cameras, and sensor devices. Flash memory is popular for these devices due to its small size, low weight, low power consumption, high shock resistance, and fast read performance [1], [2], [3], [4], [5], [6], [7], [8], [9]. The popularity of the flash memory has also extended from embedded devices to laptops, PCs, and enterprise-class server domains [3], [5], [10], [11], [12], [13], [14]. Flash based Solid State Disk (SSD) is regarded as a future replacement for the magnetic disk drive. Market giants like Dell and Samsung have already launched laptops with only SSDs [15], [16]; Samsung, STec, and SimpleTech have launched SSDs with better performance compared to the traditional 15000 RPM enterprise disk drives [17], [18], [19]. Enterprise class SSDs provide unique opportunities to boost up the I/O-intensive applications performance in the datacenters [20], [21]. Compared to hard disks, flash based SSDs are very attractive in the high-end servers of datacenters due to their faster read performance, lower cooling cost, and higher power savings.

Unlike the conventional magnetic disks, where read and write operations exhibit symmetric speed, in flash based SSDs,

the write operation is substantially slower than the read operation. This asymmetry arises as flash memory does not allow overwrite operations and write operations in a flash memory must be preceded by an erase operation. This problem becomes further complicated because write operations are performed at the page granularity, while erase operations are performed at the block granularity. Typically, a block spans 32-64 pages and live pages from a block need to be moved to new pages before the erase is done. Furthermore, a flash memory block can only be erased for a limited number of times, after which it acts like a read-only device. This is known as *wear out* problem. These slow write performance and wear-out issues are the two most important concerns restricting SSDs widespread acceptance in datacenters [20]. Existing approaches to overcome these problems through modified flash translation layer (FTL) are effective for the sequential write access patterns; however, they are inadequate for the random write access patterns [3], [6], [10]. In this paper, our primary focus is to improve the random write performance of the flash based SSDs.

Nonvolatile RAM cache inside an SSD is useful in improving the random write performance [3]. This cache acts as a filter and makes random write stream close to the sequential before forwarding these requests to an FTL. Since overwrite of a page in the flash memory would require an entire block to be erased, to exploit this behavior the caching algorithms working inside the SSDs operate at the logical block granularity rather than the traditional page granularity [3], [22]. In these algorithms, resident pages in the cache are grouped on the basis of their logical block associations. In case of evictions, all pages belonging to the evicted block are removed simultaneously. Overall, this strategy helps to reduce the number of erase operations. In the page-level algorithms, evicting a page makes a free page space available in the cache. However, in case of the block level algorithms, since different block can contain different number of pages, therefore the number of free pages available depends on the number of pages in an evicted block. Due to this behavior, direct application of the existing disk-based well known caching policies like LRU, CLOCK [23], WOW [24], CLOCK-Pro [25], DULO [26], and ARC [27], so on, are inappropriate for flash based SSDs. In addition, existing flash-optimized write caching algorithms are not very effective for the diverse workloads. We have described the drawbacks of these algorithms in Section III. For the flash based SSDs, we need a completely new set of block level write

caching algorithms which will be effective under diverse set of workloads.

In this paper, as our main contribution, we propose a new block-level write cache management algorithm, which we call the Large Block CLOCK (LB-CLOCK) algorithm, for flash based SSDs. This algorithm considers *recency* as well as *block space utilization* (i.e., number of pages currently resident in a logical block) to select a victim block. The LB-CLOCK algorithm is based on the CLOCK [23] algorithm, which has been widely used in operating systems to simulate LRU. However, LB-CLOCK algorithm has two important differences from the traditional CLOCK algorithm. First, it operates at the granularity of logical blocks instead of pages. Second, instead of selecting the first page with recency bit zero as the victim, it uses a greedy criterion to select the first block with the largest *block space utilization* from the *victim candidate set*. The *victim candidate set* contains all the blocks with recency bit set to zeros. The most crucial aspect of the LB-CLOCK algorithm lies in the creation of this *victim candidate set* which dynamically adapts to the workload characteristics. We apply two optimizations, *sequential block detection* and *preemptive victim candidate set selection*, to decide when a specific block becomes part of the *victim candidate set*.

We evaluate the performance of the LB-CLOCK algorithm using typical datacenter application benchmarks, including write intensive online transaction processing (OLTP) type Financial Workload trace and a Web Download trace. Based on these experimental results, the LB-CLOCK algorithm is shown to outperform best known existing flash based write caching schemes, BPLRU [3] and FAB [22]. For the OLTP type Financial workload trace, LB-CLOCK performs 245%-493% fewer block evictions and provides 261%-522% more throughput compared to FAB, while performing 4%-64% fewer block evictions and providing 4%-70% more throughput compared to BPLRU. It is important to mention here that OLTP type applications running in datacenters are particularly stressful workloads for the SSDs. Our experimental results also confirm that LBCLOCK is the most effective algorithm under diverse workloads.

The remainder of the paper is organized as follows: Section II gives an overview of the flash based solid state disk. Section III surveys the existing caching schemes for flash memory. Section IV explains the LB-CLOCK algorithm in detail. Section V describes the experimental setup and workload traces. Section VI explains the simulation results. Finally, Section VII concludes the discussion.

II. OVERVIEW OF THE FLASH BASED SSD

Flash memory can be either NOR or NAND typed. However, due to denser architecture and better performance, NAND based flash memory is more suitable for mass storage, which makes it the preferred choice for the Solid State Disks (SSDs). In this paper, we are focusing on the NAND flash based SSDs.

Figure 1 gives a block-diagram of an SSD. Flash Translation layer (FTL) is an intermediate software layer inside SSD, which makes linear flash memory device act like a virtual disk

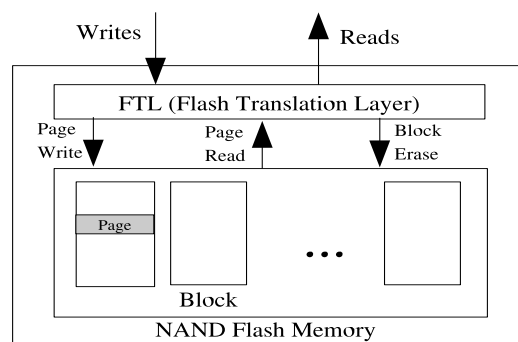


Fig. 1. Solid State Disk (SSD)

drive. FTL receives logical read and write commands from the applications and converts them to the internal flash memory commands. A *page* (also known as *sector*) is the smallest addressable unit in an SSD. A set of pages form a *block*. Typical page size is 2KB and block size is 128KB [28]. Flash memory supports three types of operations: *read*, *write*, and *erase*. Typical access time for read, write, and erase operations are 25 microseconds, 200 microseconds, 1.5 milliseconds, respectively [28]. *Read* is a page level operation and can be executed randomly in the flash memory without incurring any additional cost. *Erase* operation can be executed only at a block level and results in setting of all bits within the block to 1. *Write* is also a page level operation and can be performed only once a page has been previously erased since it selectively sets the required bits to 0. A flash memory block can only be erased for a limited number of times. For example, a single layer cell (SLC) and multi layer cell (MLC) block can be erased only 100K and 10K times, respectively, after which it acts like a read-only device [28]. FTL also uses various wear-leveling techniques to even out the erase counts of different blocks in the flash memory to increase its overall longevity.

In order to mitigate the effect of erase operations, FTL redirects the overwrite of page operations to a small set of spare blocks called *log blocks* [29]. These log blocks are used as a temporary storage for the overwrite operations. If there is no free log block, then FTL selects one of the log blocks as a victim and performs a *merge* operation [29] also termed as *full merge* in [3]. As *full merge* is a very expensive operation, it is desirable to minimize this operation and instead to have a *switch merge* [29] operation. *Switch merge* is a special case of *full merge* operation, wherein the victim log block is found to have the complete sequence of the pages in the right order from the first page to the last. Therefore the log block can be exchanged with the original data block by just updating the mapping table information and the operation can be completed with only one erase of the data block. Log-based FTL schemes are very effective for the workloads with sequential access patterns. However, for the workloads with random access patterns, these schemes show very poor performance [3], [6], [10].

In this paper, our goal is to improve SSDs random write performance. We achieve this goal by placing a nonvolatile cache above FTL. Our proposed write caching policy makes

	CFLRU	FAB	BPLRU	LB-CLOCK
Granularity Level	Page	Block	Block	Block
Target	Host	Host SSD	SSD	SSD
Read / Write	Both	Both	Write	Write
Recency	Yes	No	Yes	Yes
Block Space Utilization	No	Yes	No	Yes

TABLE I
COMPARISON OF DIFFERENT EXISTING CACHING ALGORITHMS
FOR THE FLASH MEMORY

the write requests more FTL friendly, which consequently helps to reduce the total number of erase operations. Reduction in the number of erase operations helps to improve write performance as well as increases lifetime of the SSDs.

III. RELATED WORK

Existing caching algorithms for flash memory can be classified into two main categories. One category operates on the *page-level* and another category operates on the *logical block-level*. The first category includes CFLRU algorithm [30] and the second category includes BPLRU [3] and FAB [22] algorithms. Our proposed algorithm Large Block CLOCK (LB-CLOCK), also operates at the block-level. In the rest of this section, we will discuss each algorithm in detail.

Clean First LRU (CFLRU) [30] buffer management scheme exploits the asymmetric read and write speed of the flash memory. It divides the host buffer space into two regions: *working region* and *eviction region*. Victim buffer pages are selected from the eviction region. The size of these two regions vary based on the workload characteristics. CFLRU reduces number of write operations by performing more read operations. It chooses a clean page as victim instead of a dirty page, as write operation is more expensive than the read operation. When all the pages in the eviction region are clean, the victim is selected in the Least Recently Used (LRU) order. As in this paper we are considering only write operations in the SSD device, CFLRU is not relevant to us.

Block Padding LRU (BPLRU) [3] and Flash Aware Buffer (FAB) [22] cache management schemes group the cached pages that belong to the same erase block in the SSDs into single block. BPLRU uses a variant of the LRU policy named Block Padding LRU (BPLRU) to select a victim in the cache. BPLRU manages these blocks in a LRU list. Whenever any single page inside a block gets hit, the entire block is moved to the Most Recently Used (MRU) end of the list. When there is not enough space in the cache, the victim block is selected from the LRU end of the list. In contrast to BPLRU, FAB considers *block space utilization*, which refers to the number of resident cached pages in a block, as the sole criteria to select a victim block. FAB evicts a block having the largest number of cached pages. In case of tie, it considers the LRU order. Since BPLRU only considers *recency* (i.e., LRU order) to select a victim block, for some write workloads where there are no or marginal recency, for example completely random write workload, FAB outperforms BPLRU. However, in general

BPLRU outperforms FAB for the workloads which have even moderate temporal localities. Here, our goal is to design a caching algorithm that will perform well for diverse set of workloads. Table I gives a comparison among all existing flash caching algorithms and our proposed LB-CLOCK algorithm.

IV. ALGORITHM DESCRIPTION

The Large Block CLOCK (LB-CLOCK) algorithm is inspired from the CLOCK page replacement algorithm [23] used widely in Operating Systems (OS). In this section, at first, we briefly describe the CLOCK algorithm. Then, we explain LB-CLOCK algorithm in detail. Finally, we describe two optimizations in the core LB-CLOCK algorithm.

A. CLOCK Algorithm

CLOCK algorithm is used as an efficient way to approximate the working of Least Recently Used (LRU) algorithm on memory pages [31]. Since it is very expensive to maintain the LRU lists of memory pages on each memory access, CLOCK algorithm maintains one reference bit per page. Whenever a page is accessed (i.e., read or written), the reference bit is set by the hardware. CLOCK algorithm resets this bit periodically to ensure that a page will have reference bit set only if it has been accessed at least once from the duration of the last reset. CLOCK algorithm maintains all the physical memory pages in a circular list with a clock pointer currently pointing to a page. Reference bits of all the pages in the clock pointer's traversal path are serially checked to select the victim page for eviction. If there is a page with reference bit set (i.e., 1), it is reset (i.e., set to 0) and the clock pointer is advanced by one position. This step is repeated until the clock pointer encounters a page with reference bit zero. The corresponding page is chosen as the victim and the clock pointer is moved to the next position.

One of the main reasons why LB-CLOCK algorithm is designed based on the CLOCK algorithm is due to its dynamically created sets of recently used and not used pages. In CLOCK algorithm, at any point in time we can consider pages in the memory to be part of one of these following two sets: (a) *Recently used set*: consisting of pages which have current reference bit values set to 1s. (b) *Not recently used set*: consisting of pages having current reference bit values set to 0s. Therefore, if we need to consider more than one victim selection criteria exploiting recency of page accesses, we can apply other criteria on all the pages which are part of not recently used set and this set gets dynamically updated based on the workload's temporal locality.

B. LB-CLOCK Algorithm

In case of the Large Block CLOCK (LB-CLOCK) algorithm, all pages in the cache are divided into logical groups (*blocks*) based on the physical erase block they belong to in an SSD. A block can be comprised of up to N pages, where N is block size in pages. Every logical block currently in cache is associated with a *reference bit*. This bit is set whenever there is an access to any page of that block. Similar to the

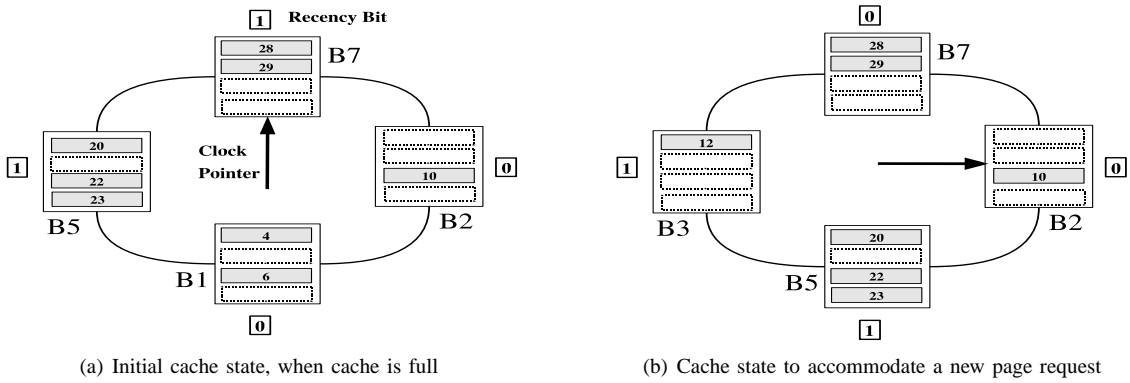


Fig. 2. Working of the LB-CLOCK algorithm

CLOCK algorithm, LB-CLOCK algorithm also uses a clock pointer which traverses in one direction over the circular list of logical blocks currently in cache. Whenever cache is full and an eviction decision needs to be made to accommodate a new write request, LB-CLOCK algorithm selects the victim group of pages at the granularity of a block. Thus, when a block is selected as a victim, all the pages which are part of that block, are simultaneously evicted out from the cache.

In order to select a block for eviction, LB-CLOCK algorithm considers two important metrics: *recency* and *block space utilization*. *Recency* is important for workloads which exhibit temporal locality. On the other hand, *block space utilization*, which refers to the numbers of cached pages in a block, is also an important metric as evicting a block containing many pages results in more cache space available for the future write requests. Thus, evicting larger blocks will consequently help to reduce the total number of block evictions. During the victim selection process, LB-CLOCK checks the recency bit the block pointed by the clock pointer. If currently recency bit is set to 1, it is reset to 0, and the clock pointer is advanced to the next block. This step is repeated until the clock pointer comes across a block with 0 recency bit value. In other words, the clock pointer stops at a block, which had its recency bit set to 0 prior to current round of the victim selection. The set of all blocks, which currently have their recency bit set to 0 is considered to be part of the *victim candidate set*. A *victim block* is the block with the largest *block space utilization* value (i.e., containing the highest number of pages) in this set.

C. An Illustration of LB-CLOCK

Figure 2 illustrates the working of the LB-CLOCK algorithm for a cache size of eight pages and block size of four pages. Block n , B_n , contains at most four pages: $4n$, $4n + 1$, $4n + 2$, and $4n + 3$. The initial cache state with eight pages belonging to four different blocks is shown in Figure 2(a). The shaded pages indicate the currently resident pages in the cache. Pages 4 and 6 belong to block B_1 ; page 10 belongs to block B_2 ; pages 20, 22, and 23 belong to block B_5 ; and page 28 and 29 belong to block B_7 . The clock pointer is currently pointing at B_7 . The pointer moves in the clock-wise direction.

The recency bit of every block is indicated by the value in the adjacent small box. Now, the cache is full. To accommodate any new page request which incurs a page miss, LB-CLOCK needs to evict a block.

Suppose, there is a new write request for page 12 belonging to block B_3 . Since the cache is full the clock pointer is traversed to find a block with recency bit 0. In this traversal, if pointer encounters a block with recency bit 1, it resets the bit to 0 and moves pointer to the next block. In the running example, clock pointer resets recency bit of block B_7 , and moves to block B_2 . As the recency bit of B_2 is 0, clock pointer stops traversing. Now, the *victim candidate set* contains $\{B_1, B_2\}$. LB-CLOCK selects block B_1 as the victim block since B_1 contains the largest number of pages. LB-CLOCK evicts all the pages currently residing in B_1 at the same time to make free space in the cache. Finally, the new write request of page 12 belonging to B_3 is inserted behind B_7 as the CLOCK pointer initially points to B_7 . Finally, the recency bit of B_3 is set to 1. Figure 2(b) shows the new cache state.

D. Optimizations in LB-CLOCK

We apply two optimizations in the basic LB-CLOCK algorithm. In the rest of this section, we discuss these two optimizations.

1) *Preemptive Victim Candidate Set Selection*: Depending on the workload, varying priority should be given to *recency* and *block space utilization*. If a workload has moderate temporal locality, then an algorithm like BPLRU, which gives priority to *recency*, would lead to fewer block evictions. On the other hand, if the workload does not have much temporal locality (random) and have blocks of varying sizes, then prioritizing *block space utilization* would lead to fewer block evictions. In an SSD, each block eviction would incur an block erase operation. Since, erase is a very expensive operation, fewer number of erase operations will lead to better performance derived out of SSDs. Moreover, fewer erase operations will eventually increase the lifetime (i.e., reliability) of SSDs. Hence, a balance between the priority given to *recency* and *block space utilization* is required. However, due to lack of prior information about the workloads, this balance has to be achieved dynamically.

LB-CLOCK achieves this balance by using the following heuristic: every time a block is selected for eviction, LB-CLOCK keeps track of the number of cached pages in that evicted block. Whenever a page is written in a block, LB-CLOCK checks if this is the last page of that block. If the last page of a block is written, then there is a low probability that it is going to be accessed again in the near future. On the other hand, if the current page written on a block is not the last page, then there is still a chance that the block is going to be accessed again in the near future, therefore the corresponding block is not considered as a part of the *victim candidate set*. If this first check succeeds, LB-CLOCK further checks whether that block is completely full (contains all pages) or not. If this second check also succeeds, LB-CLOCK considers the corresponding block as a part of the *victim candidate set*. On the other hand, if the second test fails, LB-CLOCK checks if *block space utilization* value of the block is greater than that of the previously evicted block. If this third test succeeds, LB-CLOCK includes the corresponding block as part of the *victim candidate set* by resetting its reference bit to 0. The second test gives priority to the blocks, which have passed the first heuristic test (i.e., blocks considered as less likely to be written again in the near future) and have larger number of pages than the block evicted in the previous phase of the victim selection. At any point of time, if there are only small sized blocks (i.e., blocks with very less number of pages) in the *victim candidate set* and there is a sudden burst of large sequential writes, then this heuristic will make sure that these recently written large blocks are considered for eviction earlier than the existing relatively small-sized blocks. To accommodate one large write requests, core LB-CLOCK has to evict out many small blocks from cache which results in a large number of block erase operations. The second test would prevent this problem.

2) *Sequential Write Detection*: If a block is being written sequentially, then it is assumed that the same block is unlikely to be accessed in the near future. The idea is similar to the one discussed in BPLRU [3]. When the write page request correspond to the last page of a block and rest of the pages of that block are already in the cache, such a block is considered as *sequentially written block*. For example, if a block can contain 64 pages and first 63 pages are already in the cache, then the cache receives the 64-th page write request, in this case the corresponding block is considered as sequentially written block. BPLRU places such block at the tail of the LRU list, thus making sure that it is being immediately selected as the victim in case of eviction becomes necessary. LB-CLOCK simply resets the corresponding block's reference bit to zero, thus making sure that such block is a part of the *victim candidate set*.

V. EXPERIMENTAL SETUP

This section gives a detail overview of the simulator and traces used to evaluate LB-CLOCK algorithm.

Operation	Time (microseconds)
128-KB Block Erase	1500
2KB-Page Read	25
2KB-Page Write	200
2KB-Data Transfer	13.65
Memory Access Latency	0.0025

TABLE II
TIMING PARAMETERS VALUES USED IN THE SIMULATION. THE FIRST THREE VALUES ARE TAKEN FROM [28], WHILE THE FOURTH VALUE IS CALCULATED BASED ON THE SATA1 BURST RATE OF 150 MB/s [32]

A. Simulator Detail

A cache simulator is implemented as a stand alone tool in C language on the Linux platform. The simulator takes trace file as input with each write request represented by a pair of values $\langle start_page, length \rangle$. Other parameters include the cache size and one of the following algorithms to run: 1) LB-CLOCK, 2) BPLRU [3], and 3) FAB [22]. The simulator executes the entire write requests in the trace file by simulating the working behavior of the designated algorithm.

Our simulation environment and flash translation layer (FTL) settings are similar to the BPLRU [3]. We assume a 80-GB NAND flash memory with block size of 128KB and page size of 2KB. We calculate the write throughput and the total number of block evictions while varying the RAM buffer size from 1MB to 256MB. For generating traces, we create a 80GB NTFS partition in a system running Windows XP Operating System.

To evaluate LB-CLOCK algorithm, we use total number of *block evictions* and *write throughput* as performance metrics. We calculate *write throughput* using the following formula: $total\ amount\ of\ data\ written\ from\ the\ cache / (total\ time\ for\ block\ erases + total\ time\ to\ read\ missing\ pages\ in\ the\ evicted\ blocks + total\ time\ to\ write\ evicted\ number\ of\ full\ data\ blocks + total\ time\ taken\ as\ a\ result\ of\ memory\ accesses\ from\ algorithmic\ overhead)$. The parameter values used for the *write throughput* calculation are listed in Table II. The main reason for choosing total number *blocks evictions* as a metric is that write cache is logically placed above FTL and the blocks evictions from the write cache have direct correlations with the erase operations. The more blocks evictions from the write cache boils down to more tasks for the FTL in terms of wear leveling and garbage collection, which cause frequent erase operations. Therefore, if we can reduce the number of blocks evictions, it will lead to fewer number of erase operations. Since erase is the most expensive operation, therefore reducing the number of block erase operations will help to improve write performance. On the other hand, since a block can only be erased for a limited number of times, reduction in the number of block erase operations will also help to increase lifetime of flash based SSDs.

B. Workload Traces Used

We use the following traces covering different types of applications.

SPC Financial Trace: SPC [33] traces are collected from

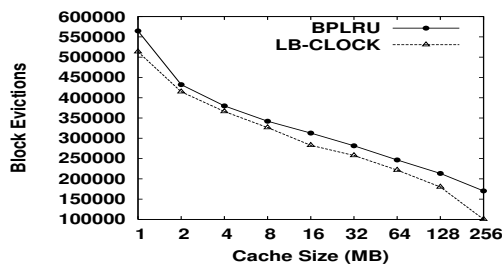


Fig. 9. OLTP type SPC Financial_1 Trace: BPLRU vs. LB-CLOCK algorithm (a closer view)

the UMass Storage Repository [34], which describes the trace data as being from the online transaction processing (OLTP) applications running at two large financial institutions. OLTP type of application running in datacenters is one of the stressful workloads for the SSDs due its write-intensive nature. We have extracted only the writes requests and ignored all the read requests in these traces.

MS OFFICE 2007 Installation Trace: We use Diskmon utility [35] from Microsoft to gather disk writes on NTFS file system while installing MS Office 2007 on an empty 80GB partition running Windows XP operating system.

MP3 Files Copy Trace: We copy over 4GB of MP3 files to an empty 80GB empty NTFS disk partition and collected the disk write requests using Diskmon utility [35].

Web Download Trace: About 7GB of internet data is downloaded from the amazon’s website [36] using the Wget utility [37] to an empty 80GB NTFS partition. The write requests to the disk are collected by using Diskmon utility [35].

Synthetic Trace: To perform a comprehensive study, we also evaluate LB-CLOCK for a synthetic workload which is 100% random and has no temporal locality. We use Iometer utility [38] to generate this workload trace. The Iometer generated disk write requests are collected by using the Diskmon utility [35].

VI. RESULT ANALYSIS

We compare our proposed algorithm Large Block CLOCK (LB-CLOCK) with the current best known algorithms BPLRU [3] and FAB [22] for both real workload traces and synthetic traces. We compare the resultant number of block evictions and write throughput for the cache size varying from 1MB to 256MB.

OLTP type SPC Financial_1 Trace: Figure 3 shows that LB-CLOCK algorithm performs 245%-493% less block evictions and provides 261%-522% more throughput compared to FAB for the cache size upto 16MB. Beyond 16MB FAB does almost same as LB-CLOCK. Although it not clear from Figure 3, but Figure 9 shows that LB-CLOCK performs 4%-70% less block evictions compared to BPLRU. In terms of throughput, it provides 4%-64% more throughput compared to BPLRU. Careful analysis of this workload revealed that it exhibits a very high degree of both spatial and temporal locality. Since FAB is not suited for such a workload, it performs poorly for cache size up to 16MB. Beyond 16 MB,

cache size is large enough to accommodate the temporal locality window, so FAB performs as good as LB-CLOCK. It is also not surprising to find that LB-CLOCK is better than BPLRU for workloads of this nature. BPLRU does not perform well since it considers only *recency*, while by having dynamically varying priority for *recency* as well as *block space utilization* LB-CLOCK performs better than BPLRU.

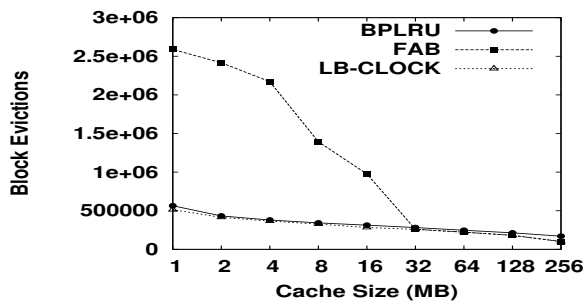
OLTP type SPC Financial_2 Trace: Figure 4 shows that, LB-CLOCK again performs the least number of block evictions. FAB again performs poorly for smaller cache sizes and does almost same as LB-CLOCK for cache size of 64MB and more. LB-CLOCK performs 33%-46% less block evictions and provides 34%-47% more throughput compared to FAB for cache size less than 64MB. On the other hand, LB-CLOCK performs 10%-53% less block evictions compared to BPLRU, while provides 10%-56% more throughput. These results are very similar to the previous ones and further proves that for a workload with a high degree of temporal and spatial locality, LB-CLOCK is the best algorithm compared to the FAB and BPLRU algorithm.

MS OFFICE 2007 Installation Trace: Figure 5(a) show that LB-CLOCK outperforms both FAB and BPLRU for all cache sizes varying from 1MB-256MB. LB-CLOCK performs 2%-4% less block evictions compared to BPLRU. While LB-CLOCK performs 2%-7% less block evictions compared to the FAB for the cache size varying from 1MB-32MB, beyond 32MB cache size FAB performs same as LB-CLOCK. LB-CLOCK also provides 1%-4% throughput improvement compared to BPLRU, while it provides 2%-8% throughput improvement compared to FAB algorithm.

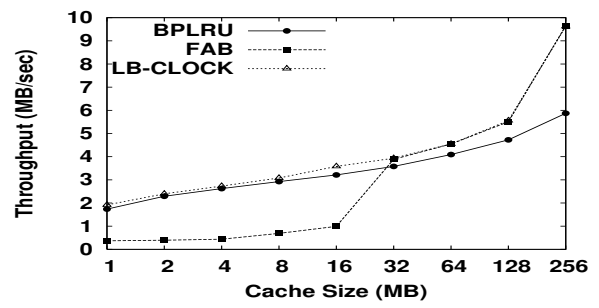
MP3 Files Copy Trace: Figure 6(a) shows that beyond 8MB cache all three algorithms LB-CLOCK, FAB, and BPLRU result in almost same number of block evictions. Even with smaller cache size LB-CLOCK and BPLRU perform very similarly. However, for the cache size less than 8MB, LB-CLOCK and BPLRU outperform FAB. For cache size less than 4MB, LB-CLOCK performs 3% less block evictions and provides 3% more throughput compared to FAB. Since MP3 files copy workload is fairly sequential, moderate cache size is good enough to capture its workload window. LB-CLOCK and BPLRU outperform FAB in the smaller cache sizes, as FAB prematurely evicts some of the larger blocks which are still being to be sequentially written.

Web Download Trace: From the Figure 7(a), it is clear that LB-CLOCK and BPLRU follow a similar trend. However, LB-CLOCK performs lesser number of block evictions. FAB performs poorly for this workload trace for cache sizes less than 128MB. In general, LB-CLOCK performs the best in this case and beats FAB by a big margin. Compared to BPLRU, LB-CLOCK does 2%-6% less block evictions and provides 2%-6% more throughput. Whereas compared to FAB, LB-CLOCK performs 2%-42% less block evictions and provides 2%-45% more throughput.

Synthetic Trace Containing 100% Random Write Requests: This synthetic trace having no or marginal temporal locality, represents one of the worst case write traces for LB-

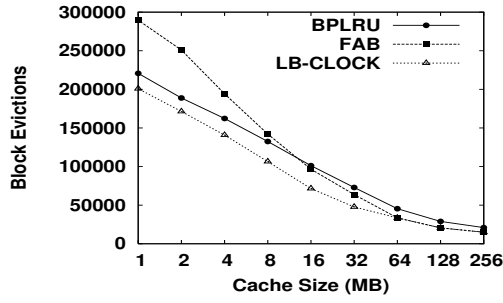


(a)

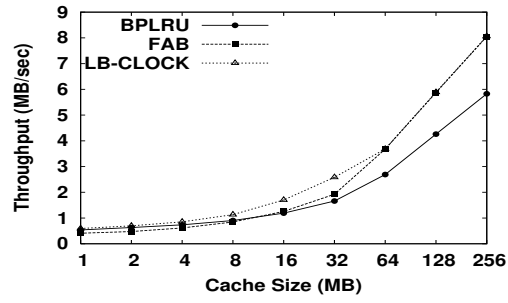


(b)

Fig. 3. OLTP type SPC Financial_1 Trace

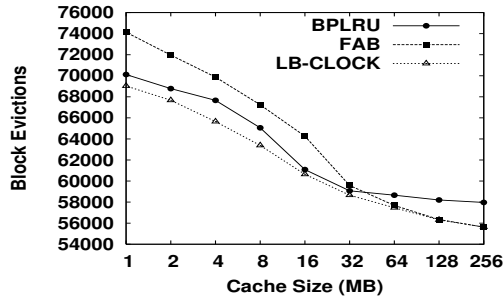


(a)

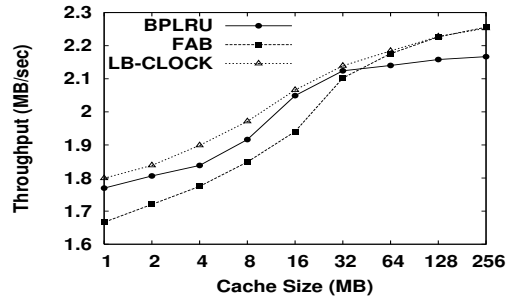


(b)

Fig. 4. OLTP type SPC Financial_2 Trace

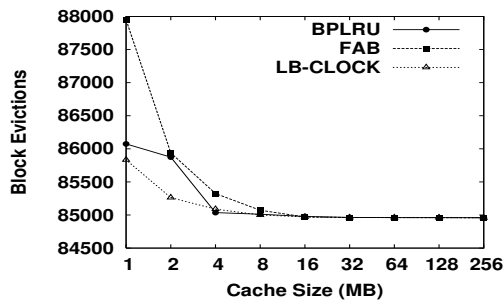


(a)

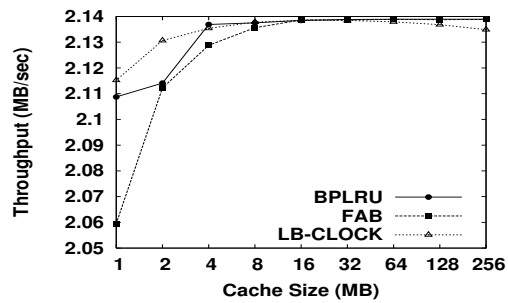


(b)

Fig. 5. MS Office 2007 Installation Trace

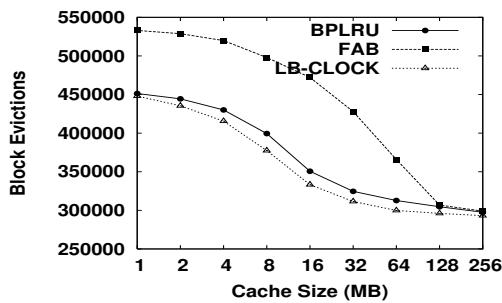


(a)

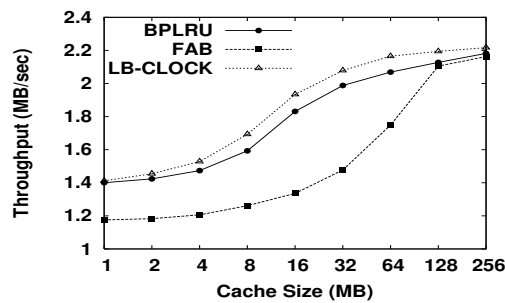


(b)

Fig. 6. MP3 Files Copy Trace

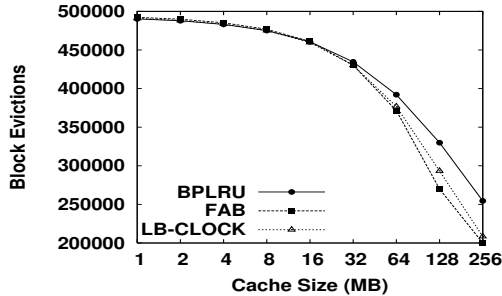


(a)

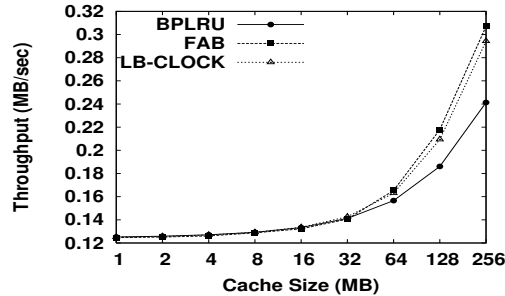


(b)

Fig. 7. Web Download Trace



(a)



(b)

Fig. 8. Synthetic trace containing only 100% random single page write requests

CLOCK and BPLRU, while the best case for FAB. This is because on average the cache will comprise of varying sized blocks and since this is a 100% random workload, none of the pages in the cache are likely to be requested within a very short time. Therefore, the best caching policy in this case is to choose the largest sized (i.e., space utilized) block as the victim ignoring its recency bit. Figure 8 shows that for this type of random workload, all three algorithms perform almost same for smaller cache size range, while for the larger cache size both LB-CLOCK and FAB outperform BPLRU. For example, for the cache size 64MB-256MB range, LB-CLOCK performs 4%-12% less block evictions and provides 4%-12% more throughput compared to BPLRU. While LB-CLOCK performs 1%-8% more block evictions and provides 1%-4% less throughput compared to FAB. Although it is not included in Figure 8 for consistency with other figures, we find that compared to FAB for the 512MB and 1GB cache size, LB-CLOCK provides 7% and 10% more throughput, respectively, which illustrates the dynamic adaptability of the algorithm. This type of workload is not very common. From our observation, all real life workloads always have a fair amount of temporal locality and FAB does not perform well for these workloads. We analyzed this trace to perform a comprehensive analysis of our proposed LB-CLOCK algorithm.

Results Summary: Table III gives a summary of the results for all workload traces. In case of the real workload traces, LB-CLOCK outperforms both BPLRU and FAB. For the synthetic trace, LB-CLOCK outperforms BPLRU, while FAB outperforms LB-CLOCK. However, the synthetic trace result is not important as all the realistic workloads have a fair amount

	LB-CLOCK vs. BPLRU		LB-CLOCK vs. FAB	
	Block Evictions	Throughput	Block Evictions	Throughput
Financial_1	4% to 70%	4% to 64%	245% to 493%	261% to 522%
Financial_2	10% to 53%	10% to 56%	33% to 46%	34% to 47%
MS Office	2% to 4%	1% to 4%	2% to 7%	2% to 8%
MP3	0%	0%	3%	3%
Web	2% to 6%	2% to 6%	2% to 42%	2% to 45%
Synthetic	4% to 12%	4% to 12%	-1% to -8%	-1% to -4%

TABLE III

SUMMARY OF THE IMPROVEMENT IN LB-CLOCK COMPARED TO BPLRU AND FAB FOR THE CACHE SIZE 1MB-256MB

of temporal locality. *We have observed the highest benefit of the LB-CLOCK algorithm in case of the online transaction processing (OLTP) type Financial workloads. This is a very significant result as the write performance of the OLTP type applications on SSDs is one of the major concerns for the datacenters in widely deploying SSDs.* In addition, since block evictions from write cache have direct correlations with physical flash block erases, less block evictions performed by the LB-CLOCK algorithm also helps to mitigate the wear-out problem of the SSDs.

VII. CONCLUSION AND FUTURE WORKS

Flash based Solid State Disks (SSDs) show substantial promise to improve data throughput, power and heat efficiency of the high-end servers in the datacenters. However, relatively slow write performance and wear-out problem are the two big concerns in widely deploying SSDs in the datacenters. In this paper, we are mainly focusing on improving write performance of the SSDs. We have proposed a new write back caching algorithm named as Large Block CLOCK (LB-CLOCK), which

is especially designed to cope with the physical properties of the flash memory. LB-CLOCK algorithm operates on the block level and considers two key metrics: *recency* and *block space utilization* to manage pages in the cache. Depending on the workload behavior, LB-CLOCK dynamically varies the priorities between these two metrics, which helps it to outperform the previously best known algorithms including BPLRU [3] and FAB [22]. In addition, LB-CLOCK helps to deal with the wear-out issue by reducing the number of block erase operations.

Our experimental results confirm that LB-CLOCK algorithm consistently outperforms BPLRU algorithm for real as well as synthetic workload traces with up to 70% better performance for an online transaction processing (OLTP) trace, which is one of the most write-intensive workloads running in the datacenters, and 12% for the synthetic trace in terms of throughput. While LB-CLOCK algorithm outperforms FAB algorithm for the real workload traces including as much as 522% for an OLTP trace and 45% for Web Download trace in terms of throughput, it performs very close to FAB algorithm for the synthetic trace. Therefore, overall our proposed LB-CLOCK algorithm is better than the BPLRU and the FAB algorithm.

There are two issues that we have not addressed in this paper. First, we have designed a caching algorithm which considers only write operations. However, an SSD can also be equipped with a read cache. This read cache can speed up performance by servicing read operations from the read cache. Moreover, it helps to reduce the load in the flash data channel (i.e., bus) which is shared by both read and write operations. By servicing read operations from a cache, the flash data channel's bandwidth can be saved to destage write operations. Clearly, in this case, a read cache will help to improve the performance of write operations. In the future, we are planning to investigate this issue. Second, we have assumed that the write cache is non-volatile. However, if we want to apply our write caching algorithm in the client side, which usually uses volatile DRAM based cache, we have to address the non-volatility issue. In the future, we are also planning to investigate this issue.

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation grant no. CCF-0621462 and the University of Minnesota Digital Technology Center Intelligent Storage Consortium.

REFERENCES

- [1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *USENIX*, 2008.
- [2] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Survey*, vol. 37, no. 2, 2005.
- [3] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *FAST*, 2008.
- [4] G. Kim, S. Baek, H. Lee, H. Lee, and M. Joe, "LGeDBMS: A Small DBMS for Embedded System with Flash Memory," in *VLDB*, 2006.
- [5] S. Lee and B. Moon, "Design of Flash-based DBMS: An In-page Logging Approach," in *SIGMOD*, 2007.

- [6] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, "A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation," *ACM Transaction on Embedded Computing Systems*, vol. 6, no. 3, 2007.
- [7] S. Nath and P. Gibbons, "Online Maintenance of Very Large Random Samples on Flash Storage," in *VLDB*, 2008.
- [8] S. Nath and A. Kansal, "FlashDB: Dynamic Self-tuning Database for NAND Flash," in *ISPN*, 2007.
- [9] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. Najjar, "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices," in *FAST*, 2005.
- [10] I. Koltsidas and S. Vlasos, "Flashing Up the Storage Layer," in *VLDB*, 2008.
- [11] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim, "A case for Flash memory SSD in Enterprise Database Applications," in *SIGMOD*, 2008.
- [12] A. Leventhal, "Flash Storage Today," *ACM Queue*, vol. 6, no. 4, 2008.
- [13] M. Moshayedi and P. Wilkison, "Enterprise SSDs," *ACM Queue*, vol. 6, no. 4, 2008.
- [14] A. Caulfield, L. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications," in *ASPLOS*, 2009.
- [15] "Dell Puts SSD Hard Drives in Latitude D420 and ATG D620," <http://www.notebookreview.com/default.asp?newsID=3658>, April, 2007.
- [16] M. Hachman, "New Samsung Notebook Replaces Hard Disk Drive with Flash," <http://www.extremetech.com>, May, 2006.
- [17] "Zeus-IOPS Solid State Drives Surge to 512GB in Standard 3.5" Form Factor; Offer Unprecedented Performance for Enterprise Computing," <http://www.globenewswire.com/newsroom/news.html?d=117663>, April, 2007.
- [18] "Samsung Enterprise Solid State Drive," http://www.samsung.com/global/business/semiconductor/products/flash/ssd/2008/download/Enterprise_SSD_datasheet_10-08.pdf, Oct, 2008.
- [19] P. Miller, "SimpleTech Announces 512GB and 256GB 3.5-inch SSD Drives," <http://www.techmeme.com/070419/p25#a070419p25>, April 2007.
- [20] D. Reinsel and J. Janukowicz, "Datacenter SSDs : Solid Footing for Growth," <http://www.samsung.com/us/business/semiconductor/news/downloads/210290.pdf>, January, 2008.
- [21] J. Sykes, "SSDs to Boost Data Center Performance," http://download.micron.com/pdf/whitepapers/ssds_to_boost_data_center_performance.pdf, July, 2008.
- [22] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee, "FAB: Flash-aware Buffer Management Policy for Portable Media Players," *IEEE Transactions on Consumer Electronics*, vol. 22, no. 2, 2006.
- [23] F. Corbato, "A Paging Experiment with the Multics System," in *MIT Project MAC Report MAC-M-384*, 1968.
- [24] B. Gill and D. Modha, "WOW: Wise Ordering for Writes - Combining Spatial and Temporal Locality in Non-Volatile Caches," in *FAST*, 2005.
- [25] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," in *USENIX*, 2005.
- [26] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality," in *FAST*, 2005.
- [27] N. Megiddo and D. Modha, "ARC: A Self-tuning, Low Overhead Replacement cache," in *FAST*, 2003.
- [28] W. Hutsell, "Solid State Storage for the Enterprise," in *SNA Tutorial*, 2007.
- [29] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for Compact Flash Systems," in *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, 2002.
- [30] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," in *CASES*, 2006.
- [31] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, Inc., 2004.
- [32] "SATA-IO: Enabling the Future," <http://www.serialata.org/3g.asp>.
- [33] "SPC: Storage Performance Council," <http://www.storageperformance.org/>.
- [34] "University of Massachusetts Amherst Storage Traces," <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- [35] "Diskmon for Windows," <http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx>.
- [36] "Amazon," <http://www.amazon.com>.
- [37] "GNU Wget," <http://www.gnu.org/software/wget/>.
- [38] "Iometer Project," <http://www.iometer.org>.