CFDC—A Flash-aware Replacement Policy for Database Buffer Management

Yi Ou
University of Kaiserslautern
Germany
ou@cs.uni-kl.de

Theo Härder
University of Kaiserslautern
Germany
haerder@cs.uni-kl.de

Peiquan Jin
University of Science and Technology of China
P.R. China
jpq@ustc.edu.cn

ABSTRACT

Flash disks are becoming an important alternative to conventional magnetic disks. Although accessed through the same interface by applications, flash disks have some distinguished characteristics that make it necessary to reconsider the design of the software to leverage their performance potential. This paper addresses this problem at the buffer management layer of database systems and proposes a flash-aware replacement policy that significantly improves and outperforms one of the previous proposals in this area.

1. INTRODUCTION

Flash disks (flash-memory-based solid-state drives) will play an increasingly important role for server-side computing, because they have—compared to magnetic disks—no mechanical parts and, therefore, hardly any perceptible latency. Furthermore, they have a much lower power consumption. Typically, flash disks are managed by the operating system as block devices through the same interface types as those to a magnetic disk. However, the distinguished IO characteristics of flash disks make it necessary to reconsider the design of IO-intensive and performance-critical software, such as a DBMS, to achieve maximized performance.

Traditionally, the goal of buffer replacement policy is the minimization of the buffer fault ratio for a given buffer size. To guarantee data consistency, a dirty buffer page, when selected by the replacement policy as victim, has to be written back to disk before the memory area can be reused. This implies synchronous writes because the process or thread requesting for an empty buffer frame must wait until write completion—potentially a performance bottleneck.

Early in the 80's, Effelsberg and Härder [1] have called attention to the fact that "whether a page is read only or modified" is an important criterion to be considered in the replacement decision. This criterion is now much more important than ever, when flash disks are becoming an important alternative to conventional magnetic disks. The reason for that is the flash's read-write asymmetry—the cost of a page write is an order of magnitude higher than that of a page read. Other criteria important in the context of flash disks are spatial locality and sequentiality of access patterns.

Our contributions are: 1. We propose a novel replacement algorithm for database buffer management taking all the above mentioned criteria into consideration; 2. We propose a generalized two-region scheme which can be applied when designing further flash-aware replacement policies; 3. We implemented and evaluated our method and cross-compared it to competitor algorithms in a real DBMS environment.

The remainder of this paper is organized as follows. Section 2 gives background information on flash disks and introduces the related work. Section 3 introduces our algorithm, while its experimental results are presented in Section 4. We conclude and give an outlook on our future work in Section 5.

2. BACKGROUND

For comprehension, we briefly repeat the most important properties of flash disks and the goal of the flash translation layer (FTL) which is device-related and supplied by the disk manufacturer.

2.1 Flash Memory

The most common types of flash disks are based on NAND flash memory, to which three basic operations can be applied: *read*, *write*, and *erase*. Read and write operations are performed in units of a *page*. The size of a page is typically 2 KB. Erase operations can only be performed in much larger units called *blocks* which contain multiple pages. The size of a block is typically 128 KB. Reading a page from flash memory has very low latency, while writing a page to flash memory is an order of magnitude slower. Another problem with write operations is, once a page is written, the only way of overwrite it is to erase the entire block where the page resides. Furthermore, erase operations can only be performed in units of entire blocks, which cause a slow-down of another order of magnitude compared to single page writes. This implies that, when updating only a single byte in a page, an expensive erase operation and the restoration of a large amount of data are required. Write endurance is measured in erase cycles tolerated by a flash block and is at least 100,000[1] per block [2]. To avoid the premature worn out of blocks caused by highly localized writes, the data should be arranged and managed in a way that erasures are evenly distributed across the entire flash memory. This mechanism is called *wear leveling*.

[1]Recent references report up to 5,000,000 erase cycles for it.

Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN 2009) June 28, 2009, Providence, Rhode-Island

2.2 Flash Translation Layer

To overcome the limitations of flash memory, flash disks employ an intermediate software layer called *Flash Translation Layer* (FTL), which is typically stored in a ROM chip. One of the key roles of FTL is to redirect a page write request to an empty area that has been erased in advance. As a consequence, FTL has to maintain a mapping of the logical page address (the address used by the file system for the write request) to enable a new physical page location on flash memory. For fast look-up, this mapping is maintained in volatile memory. To reconstruct the mapping table at startup or in case of a failure, these logical addresses are stored in a spare area of the flash page.

The mapping can be maintained at the page level or block level. *Page-level mapping* can effectively deal with the limit of erase-before-write, because a write request can be redirected to any empty page in the flash memory. The mapping table keeps track of the *valid* physical (most up-to-date) page locations. If a block contains N pages, one erasure can serve N write requests. However, the mapping table for such an approach becomes prohibitively large. Using *block-level mapping*, the mapping table only maps a logical block address to its physical location, which implies that the offset of a page in the physical block must be identical to its offset in the logical block. For this reason, the address information for block-level mapping is much smaller. However, to update a page, the new content of the page, due to the offset constraint, must be written to the same offset in a newly-allocated empty flash block and the remaining pages of the block have to be copied to the new block, resulting in one block erasure and N pages writes.

2.3 Log-Block-Based FTL

The problems of page-level and block-level mappings are addressed by *hybrid mapping* schemes. Some representatives of them are the so-called *log-block-based approaches* [3, 4]. In such an approach, a dynamic set of flash blocks, called *log blocks*, is maintained to serve the write requests which are always directed to the log blocks. The page addresses in a log block are mapped *at the page level*, thus frequent block erasures can be avoided. The remaining flash blocks, called data blocks, are managed at the block level; these data blocks generally use a much larger flash area than the log blocks.

In the approach of [3], a log block is allowed to serve page writes to only a single data block. If it becomes full, i.e., each page in it has been written once, it is *merged* with the associated data block. If there is no free log block available that can serve the write request to a data block, one of the (not completely filled) log blocks must be freed, i.e., its content must be propagated to a data block. For each page, its valid version—either in the data block or in the log block—is copied to a third, empty block. Then, the third block becomes the new data block. The log block and the old data block are freed and are erased for later use as log block or data block. Thus, a merge operation involves two erasures. An ideal situation happens if a log block contains all valid pages of a data block and their offsets are identical to those of their corresponding pages in the data block, then the log block can be simply marked as the new data block and there is only one erasure necessary to free the old data block. This is called a *switch merge*. But, in general, this approach may suffer from low space utilization in log blocks. Especially for random write patterns with low locality, it is likely that a victim log block being merged is poorly filled with newly modified pages. In the worst case, each page write invokes a merge operation.

In the approach of [4], a log block can serve page writes targeting at multiple data blocks, thus achieving higher space utilization in the log blocks and it is less likely that a page write invokes a merge operation. However, if a log block is associated with multiple, say n, data blocks, a merge operation involves all the associated data blocks. For each of them, the valid pages are copied to an empty block. In this case, $n + 1$ erase operations (n erasures for data blocks and one for the log block) are necessary.

Log-block-based approaches greatly improve write performance of flash disks, which, however, is highly sensitive to *spatial locality* of write patterns. For the same number of write requests, the higher the locality and the lesser data blocks affected, the better is the flash write performance.

2.4 Related Work

The DULO (Dual Locality) replacement policy proposed by Jiang [5] exploits both temporal and spatial locality of access patterns. Although DULO is designed for magnetic disks, its emphasis on spatial locality is very important in the context of flash disks. Spatial locality is also exploited by some operating systems at the block-level IO, for example, by IO schedulers of Linux [6].

FAB (Flash-Aware Buffer) [7] is a buffer management policy designed for personal media players. FAB manages a block-level LRU list. The victim block in the FAB method is the block which contains the largest number of pages. BPLRU [8] also maintains an LRU list at the block level. As a buffer management scheme designed for the write buffer inside flash disks, it employs a page-padding technique which forces switch merges. To compensate the inefficiency of LRU for sequential writes, BPLRU evicts sequentially written blocks prior to randomly written blocks.

REF (Recently-Evicted First) [9] is a log-block-aware replacement policy which maintains a page-level LRU list, but selects victims only from the so-called victim blocks, which are blocks with the largest number of pages in the buffer.

CFLRU (Clean-First LRU) [10] is a flash-aware replacement algorithm for operating systems based on the LRU algorithm. CFLRU addresses the asymmetry of flash IO by allowing dirty pages to stay in the buffer longer than clean pages. Recently, a similar approach was proposed in [11] for special configurations, where magnetic disks and flash disks coexist as external storage. Fig. 1 illustrates region assignment and victim selection. The LRU list is divided into two regions: the *working region* at the MRU end of the list, where most of the buffer hits occur, and the *clean-first region* at the LRU end, where clean pages are always selected as victims over dirty pages. Only when clean pages are not present in the clean-first region, the dirty page at the tail is selected as victim. The size of the clean-first region is determined by a parameter w called the *window size*.

The idea of CFLRU is very important because, by evicting clean pages first, the buffer area for dirty pages is effectively increased—thus, the number of flash writes can be reduced. However, there are several problems with this simple approach. First, because clean pages are not always at the LRU tail, the algorithm has to search backwards for a clean page. Furthermore, a clean page tends to stay in the

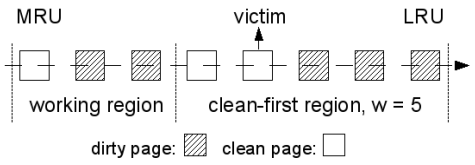

Figure 1: CFLRU replacement policy

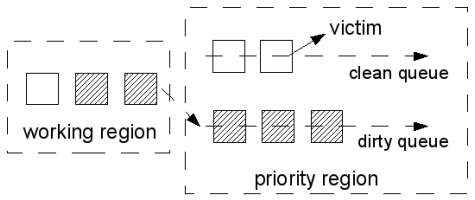

Figure 2: Generalized two-region scheme

clean-first region close to the working region (see Fig. 1), because clean pages are always selected over dirty pages. That means the algorithm often has to walk a potentially very long list in case of a buffer fault. Second, under the LRU assumption, the dirty pages in the clean-first region have a much lower probability of being re-accessed, thus this valuable main memory resource should be better utilized for minimizing the cost of writing back dirty pages. Third, CFLRU has the same problem of LRU: it becomes inefficient when the workload is mixed with long and sequential access patterns, because the hot pages cached so far are pushed away by sequentially accessed pages.

3. THE CFDC ALGORITHM

Our algorithm called *Clean-First Dirty-Clustered* (CFDC) tries to improve CFLRU by tackling these problems.

3.1 The Two-Region Scheme

We address the first problem of CFLRU by introducing *two* queues for the clean-first region: one for the clean pages and one for the dirty pages. A page evicted from the working region goes to the clean queue if it is clean, otherwise to the dirty queue. Upon a buffer fault, if the clean queue is not empty, the tail of the clean queue is returned as the victim, otherwise the tail of the dirty queue. The improved CFLRU behaves the same as the original algorithm in terms of hit ratio and flash write count, but search costs for clean pages are entirely eliminated.

The improved CFLRU can be further generalized as follows: the policy manages two regions: the *working region* for keeping *hot* pages that are frequently revisited, and the *priority region* responsible for optimizing replacement costs by assigning varying priorities to pages. A parameter, *priority window*, determines the size ratio of the priority region to the total buffer. Both regions in our generalized scheme do not have to be bound to a specific replacement policy. For example, different proven replacement policies can be used to maintain high hit ratios in the working region and, therefore, prevent hot pages from entering the priority region.

Fig. 2 shows our generalized two-region scheme of the improved CFLRU method. Here, the working region uses LRU, while the priority region assigns higher priorities to dirty pages. Upon a buffer fault, a victim is selected in the priority region to make room for a page currently in the working region. After this page displacement, the requested page can enter the working region.

3.2 Page Clustering

We address the second and third problem of CFLRU by supporting *page clustering* in the dirty queue: instead of keeping a queue of dirty pages, CFDC maintains a priority queue of page clusters. A *cluster* is a list of pages located in proximity, i.e., whose page numbers are close to each other. Hence, it is similar to a block when block-level LRU in BPLRU and FAB is used. But a cluster has variable size determined by the set of pages currently kept. The page order in a cluster does not correspond to page numbers or offsets, but to the point of time they entered the cluster.

To administrate these clusters, CFDC maintains a hash table with cluster numbers as keys. When a dirty page enters the priority region, we derive its cluster number by dividing its page number by a constant MAX_CLUSTER_SIZE and perform a hash lookup using this cluster number. If the cluster exists, the page is added to the cluster tail and the cluster position in the priority queue is adjusted. Otherwise, a new cluster containing this page is created and inserted to the priority queue. Moreover, the new cluster is registered in the hash table. In case of a page hit in the clusters, the page is simply moved to the working region. Upon a buffer fault, if the clean queue is empty, we select the first page in the lowest-priority cluster as victim. After refilling the priority region with a victim of the working region, the requested page can be loaded.

3.3 Priority Function

For a cluster c with n pages, its priority $P(c)$ is computed according to Formula 1:

$$P(c) = \frac{\sum_{i=1}^{n-1} |p_i - p_{i-1}|}{n^2 \times (globaltime - timestamp(c))} \quad (1)$$

where $p_0, ..., p_{n-1}$ are the page numbers ordered by their time of entering the cluster. The algorithm tends to assign large clusters a lower priority for two reasons: 1. Flash disks are efficient in writing such clustered pages due to their spatial locality; 2. The pages in a large cluster have a higher probability to suffer from sequential accesses.

The sum in the dividend in Formula 1 is used to distinguish between randomly accessed clusters and sequentially accessed clusters (clusters with only one page are set to 1). We prefer to keep a randomly accessed cluster in the buffer for a longer time than a sequentially accessed cluster. For example, a cluster with pages $\{0, 1, 2, 3\}$ has a dividend of 3, while a cluster with pages $\{7, 5, 4, 6\}$ has a dividend of 5. The purpose of the time component in Formula 1 is to prevent randomly, but rarely accessed small clusters from staying in the buffer forever. The cluster timestamp $timestamp(c)$ is the value of *globaltime* at the time of its creation. Each time a dirty page is evicted from the working region, *globaltime* is incremented by 1.

Fig. 3 depicts a priority queue with four clusters, where *globaltime* is 10, *timestamp(c)* is kept at the top right corner of each cluster, and the clustered pages are marked with their page numbers. From left to right, the cluster priorities are obtained using Formula 1: $2/9, 1/8, 1/14, 1/18$.

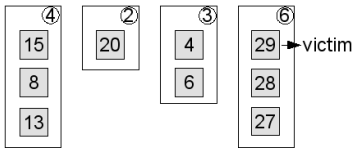

Figure 3: Prioritized clusters

Once a page victim is selected, the cluster priority is set to 0. Hence, subsequent page faults cause this cluster to be emptied, such that it is eventually removed from the priority queue. Because of the cluster property, the removed dirty pages, in turn, are logically close to each other and, because of the space allocation in most DBMSs and file systems, also have a high probability of being physically neighbored. Thus, the write requests received by the flash disk are targeting at a limited number of flash blocks which can be served efficiently as discussed in Section 2.

The time complexity of our algorithm is higher than that of LRU due to the maintenance of the priority queue. However, the queue is maintained in units of clusters and the maintenance is only triggered by a buffer fault and, in case, the page evicted from the working region is a dirty page. As our experiments will show, this overhead pays off.

4. EXPERIMENTS

We have chosen XTC [12] as the database engine used for our experiments, because we could modify its source code and it provides a clean design based on the classical *five-layer reference architecture* [13]. To better explain our results, we have only used its three bottom-most layers in our experiments, i.e., the *file manager* supporting block-oriented access to the data files, the *buffer manager* serving page requests, and the *index manager* providing B-tree and B*-tree implementations. Although designed for XML data management, the processing behavior of these three XTC layers is very close to that of a relational database system.

4.1 Test Environment

We implemented quite a number of replacement polices and integrated them into the buffer manager. Three of them are relevant to this section: LRU, CFLRU, and CFDC. The database engine including all polices and the component for generating the workloads discussed below are completely implemented in Java. The Java platform used in our experiments has version 1.6.0_06.

The test machine has an AMD Athlon Dual Core Processor, 512 MB of main memory, is running Ubuntu Linux with kernel version 2.6.24-19, and is equipped with a magnetic disk and a flash disk, both connected to the SATA interface used by the file system EXT2. Both OS and database engine are installed on the magnetic disk. The test data (as a database file) resides on the flash disk which is a MTRON MSP-SATA7525 based on SLC NAND flash memory and has a capacity of 32 GB.

Compared to simulation-only studies, more accurate and more realistic results may be anticipated by experiments on a real machine. But, it is more difficult to get such results due to several reasons: 1. Flash disks are black boxes from the viewpoint of the user. As some publications have revealed, the IO costs to a flash disk are not constant in time due to the unknown internal state of the flash disk and proprietary algorithms used in the FTL[14]; 2. The operating system and the file system may apply (hidden) data caching which also influences the IO costs of applications.

In our experiments, we deactivated the file-system prefetching and the IO scheduling for the flash disk and emptied the Linux page caches. To ensure a stable initial state, we have sequentially read and written a 512MB file (of irrelevant data) from and onto the flash disk before each experiment.

4.2 Test Data and Workload

The test data was generated by inserting one million equal-length records into a B*-tree with a page size of 4 KB (which is used for all experiments), where half of the space in the leaf pages was unused. Each record is a key-value pair with an integer key of four bytes and a value of 256 bytes. The resulted database file had a size of 641 MB in the file system.

Our baseline workload consists of four sets of transactions: 100,000 point queries, 100,000 point updates, 200 range queries, and 200 range updates, all evaluated via a B*-tree. A point query reads the corresponding value of a single key, whereas a point update changes the corresponding value. A range query sequentially scans 256 records, starting from a given key. Similarly, a range update sequentially reads and updates the value of 256 records. The keys used for the point queries and updates are generated randomly with an 80–20 self-similar distribution, i.e., 80% of the generated keys go to 20% of the key space $[0, 10^6]$. The starting keys for range queries and updates are uniformly distributed in the key space. In a test run, the four sets of transactions are mixed and executed in random order. We believe this setup represents the typical workload for a database system with mixed random and sequential accesses. Therefore, all our experiments use this setup or its variants.

4.3 Buffer Size

To explore how efficiently the buffer is managed by our algorithm, we configured our system with varying buffer sizes, ranging from 1,024 to 8,192 4K pages, i.e., from 4 MB to 32 MB. For each buffer size and each of the three replacement policies LRU, CFLRU, and CFDC, we ran the baseline workload and measured the elapsed time, which was then converted to TPS (transactions per second). The priority region used 50% of the buffer size and the constant MAX_CLUSTER_SIZE was set to 64.

The results are shown in Fig. 4. CFDC clearly outperforms both competing policies, with a performance gain between 14% (at 1,024 pages) and 41% (at 8,192 pages) over CFLRU, which, in turn, is only slightly better than LRU with a maximum performance gain of 6% at 8,192 pages.

We also instrumented the database engine to count the number of page flushes and measured the average time for flushing a single page onto the flash disk. These measurements are shown in Fig. 5 and Fig. 6. Compared to LRU, CFLRU has to perform fewer (about 3–5%) page flushes, which explains its slight performance improvement over LRU. Although CFDC gives higher priority to dirty pages, too, it has to achieve more page flushes than CFLRU (see Fig. 5). This is expected because the recency of dirty pages in the priority window is not the only criterion in CFDC. The decisive advantage of CFDC is explained by Fig. 6: it writes much more efficiently compared to the other two policies due to the page clustering and prioritization. For most of the buffer sizes, the cost of flushing a single page is only a

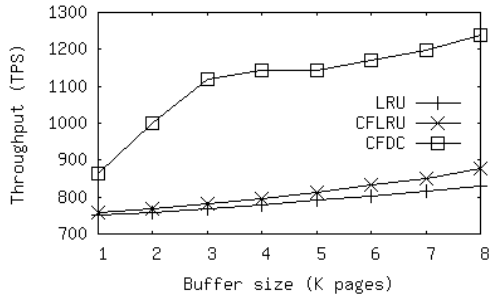

Figure 4: Influence of buffer size

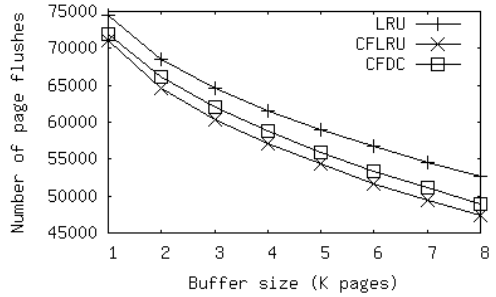

Figure 5: Number of page flushes

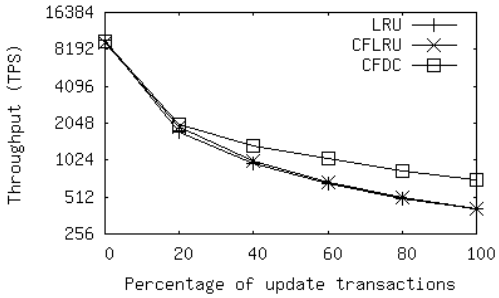

Figure 7: Influence of increasing update ratios

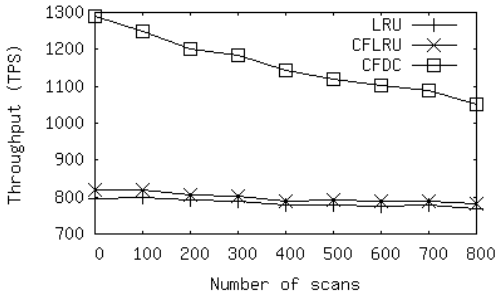

Figure 8: Influence of increasing scan fractions

half of the cost of LRU or CFLRU.

For all experiments explained in the following, we used a fixed buffer size of 4,096 pages and, if not otherwise stated, the priority window was always 50% of the buffer size.

4.4 Update Intensity

CFDC tries to reduce IO cost by clustering updated pages. Therefore, it is very interesting to know how it performs under workloads with high update intensity (percentage of update transactions in the workload). In our baseline workload, 50% are update transactions, i.e., it has an update intensity of 50%. In the following experiment, we varied this parameter from 0% to 100%, while keeping the total number of transactions unchanged. Hence, we obtained for a 20% update intensity 160,000 point queries, 40,000 point updates, 320 range queries, and 80 range updates.

Fig. 7 compares the throughput of CFDC to LRU and CFLRU under these workloads. With 0% update intensity, i.e., no updates at all, CFDC and CFLRU degenerate to LRU and all three algorithms reached a very high throughput level of around 9,500 TPS—in conventional settings, only reachable with arrays of magnetic disks. With increasing update intensity, the performance dropped linearly (note the Y-axis has logarithmic scale of base 2), while CFDC performs best among the three competitors. Under 100% update intensity, the throughput achieved by CFDC (702 TPS) is 1.73 times of that achieved by CFLRU (407 TPS) and LRU (406 TPS). This experiment also confirms that, under heavy-update workloads, CFLRU only yields a marginal performance gain over LRU and it degenerates to LRU, if all buffer pages are dirty.

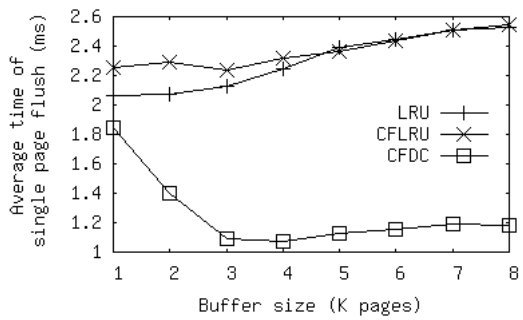

Figure 6: Cost of page flushes

4.5 Sequentiality

The baseline workload consists of 200 range queries and 200 range updates. Each of these transactions accesses a sequence of 256 records, each of length 260 bytes (4B keys and 256B values). Because of the 50% filling, about 32 leaf pages were present.

Because CFDC gives higher priority to clusters of randomly updated pages, its performance gain over LRU and CFLRU should be less noticeable, if the workload consists of a larger part of sequential accesses. If the workload was purely sequential, then no remarkable difference can be observed among the three algorithms. This consideration coincides with the experimental results shown in Fig. 8. In this test, we compared the performance of the three algorithms under workloads with increasing numbers of scans. The other workload parameters were identical to those of the baseline workload. The numbers on the X-axis of Fig. 8 are the sums of 50% range queries and 50% range updates. Obviously, the performance declines with an increasing sequentiality of the workload, because a scan lasts much longer than a single random access.

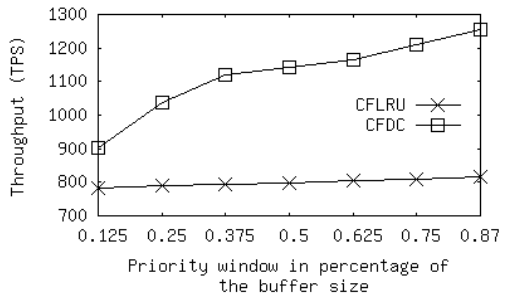

Figure 9: Impact of priority window size

4.6 Priority Window

Another interesting aspect concerns the performance impact of the priority window, provided for both CFLRU and CFDC. We configured both algorithms with an increasing window size ranging from 0.125% to 0.875% of the buffer size (which is 4,096 pages) and ran the baseline workload. As the result in Fig. 9 reveals for a workload with 50% update intensity, both policies benefit from a larger priority window, but CFDC utilizes this resource more efficiently. In our experiments, window sizes were statically configured. However, if severe workload changes are present, it would be necessary to provide dynamic window size adjustment. We leave this as an interesting topic for future research.

5. CONCLUSIONS AND OUTLOOK

Our experiments have shown that the proposed algorithm CFDC significantly outperforms CFLRU. The key for this improvement can be stated as follows: for a flash disk, the number of writes should be minimized but, more important, locality of access patterns, especially spatial locality, should be exploited to the extent possible by the buffer management. Note, this approach implies *NoForce/Steal* provisions for the logging & recovery component which, however, is the standard solution in most DBMSs [15].

A question related to the write optimization of CFDC is whether pages belonging to the same cluster should be all evicted at the same time. We believe this technique has some potential, because, as our raw IO tests on flash disks (not included in this paper) show, writing at larger transfer units to the flash disk generally improves the write performance. However, such an approach requires the pages being flushed to be first sorted by page numbers and then copied together to a larger memory area. A related optimization technique called page padding [8] might be useful in this case.

Since we primarily aimed at improving CFLRU, our experiments focused on three algorithms: LRU, CFLRU, and CFDC. We plan to include other flash-aware replacement policies in our experiments, too. Among the flash-aware replacement policies introduced in Section 2.4, BPLRU is designed for the write cache inside flash devices and, therefore, not directly comparable to our algorithm. We expect that CFDC will outperform the remaining two algorithms FAB and REF. The former, FAB, is designed and optimized for personal media players, its anticipated access patterns (e.g., sequentially reading and writing large audio/video files) largely differ from those of database systems. Although the latter, REF, considers the log-block mechanism inside flash devices to optimize write performance on flash disks, it doesn't give different priorities to clean and dirty pages like CFDC does. Therefore, we expect that CFDC will generate less writes than REF under the same kinds of workloads. But these speculations are subject to examination using empirical experiments in the near future. Moreover, we plan to cross-compare those algorithms under workloads of real database applications.

6. ACKNOWLEDGEMENT

This work is partly supported by the Carl Zeiss Foundation.

7. REFERENCES

[1] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. on Database Sys.*, 9(4):560–595, December 1984.

[2] D. Woodhouse. JFFS: the journalling flash file system. In *Proc. of the Ottawa Linux Symposium*, 2001.

[3] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *Trans. on Consumer Electronics*, 48(2):366–375, 2002.

[4] 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 Trans. on Embedded Computing Systems*, 6(3), July 2007.

[5] S. Jiang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial localities. *In USENIX Conf. on File and Storage Technologies*, pages 101–114, 2005.

[6] J. Axboe. Linux block IO—present and future. In *Proc. of the Ottawa Linux Symposium*, 2004.

[7] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee. FAB: flash-aware buffer management policy for portable media players. *Trans. on Consumer Electronics*, 52(2):485–493, 2006.

[8] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *USENIX Conf. on File and Storage Technologies*, pages 239–252, 2008.

[9] D. Seo and D. Shin. Recently-evicted-first buffer replacement policy for flash storage devices. *Trans. on Consumer Electronics*, 54(3):1228–1235, 2008.

[10] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. *Proc. of Int. Conf. on compilers, architecture, and synthesis for embedded systems*, pages 234–241, 2006.

[11] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow. Arch.*, 1(1):514–525, 2008.

[12] M. P. Haustein and T. Härder. An efficient infrastructure for native transactional XML processing. *Data Knowl. Eng*, 61(3):500–523, 2007.

[13] T. Härder. DBMS architecture - the layer model and its evolution. *Datenbank-Spektrum*, 13:45–57, 2005.

[14] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR*. www.crdrdb.org, 2009.

[15] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.