# CFDC: A Flash-Aware Buffer Management Algorithm for Database Systems

Yi Ou[1], Theo Härder[1], and Peiquan Jin[2]

[1] University of Kaiserslautern, Germany
{ou,haerder}@cs.uni-kl.de
[2] University of Science & Technology of China
jpq@ustc.edu.cn

**Abstract.** Classical buffer replacement policies, e.g., LRU, are suboptimal for database systems having flash disks for persistence, because they are not *aware* of the distinguished characteristics of those storage devices. We present CFDC (Clean-First Dirty-Clustered), a flash-aware buffer management algorithm, which emphasizes that clean buffer pages are first considered for replacement and that modified buffer pages are clustered for better spatial locality of page flushes. Our algorithm is complementary to and can be integrated with conventional replacement policies. Our DBMS-based performance studies using both synthetic and real-life OLTP traces reveal that CFDC significantly outperforms previous proposals with a performance gain up to 53%.

## 1 Introduction

Flash disks will play an increasingly important role for server-side computing, because—compared to magnetic disks—they are much more energy-efficient and they have no mechanical parts and, therefore, hardly any perceptible latency. Typically, flash disks are managed by the operating system as block devices through the same interface types as those to magnetic disks. However, the distinguished performance characteristics of flash disks make it necessary to reconsider the design of DBMSs, for which the I/O performance is critical.

### 1.1 Performance Characteristics

The most important building blocks of flash disks are flash memory and flash translation layer (FTL). Logical block addresses are mapped by the FTL to varying locations on the physical medium. This mapping is required due to the intrinsic limitations of flash memory [1]. The FTL implementation is device-related and supplied by the disk manufacturer. Many efforts are made to systematically benchmark the performance of flash disks [2,3]. The most important conclusions of these benchmarks are:

– For sequential read-or-write workloads, flash disks often achieve a performance comparable to high-end magnetic disks.

B. Catania, M. Ivanović, and B. Thanlheim (Eds.): ADBIS 2010, LNCS 6295, pp. 435–449, 2010.

- For random workloads, the performance asymmetry of flash disks and their difference to magnetic disks is significant: random reads are typically two orders of magnitude faster than those on magnetic disks, while random writes on flash disks are often even slower than those on magnetic disks[1].
- Due to the employment of device caches and other optimizations in the FTL, page-level writes with strong *spatial locality* can be served by flash disks more efficiently than write requests without locality. In our context, spatial locality refers to the property of contiguously accessed DB pages being physically stored close to each other.

Interestingly, many benchmarks show that flash disks can handle random writes with larger *request sizes* more efficiently. For example, the bandwidth of random writes using units of 128 KB is more than an order of magnitude higher than writing at units of 8 KB. In fact, a write request of, say 128 KB, is internally mapped to 64 *sequential* writes of 2-KB flash pages inside a flash block. Note that sequential access is an extreme case of high spatial locality.

### 1.2 The Problem

Flash disks are considered an important alternative to magnetic disks. Therefore, we focus here on the problem of buffer management for DBMSs having flash disks as secondary storage. If we denote the sequence of $n$ logical I/O requests $(x_0, x_1, \ldots, x_{n-1})$ as $X$, a buffer management algorithm $A$ is a function that maps $X$ and a buffer with $b$ pages into a sequence of $m$ physical I/O requests $Y := (y_0, y_1, \ldots, y_{m-1})$, $m \leq n$, i. e., $A(X, b) = Y$.

Let $C(Y)$ denote the accumulated time necessary for a storage device to serve $Y$, we have $C(Y) = C(A(X, b))$. Given a sequence of logical I/O requests $X$, a buffer with $b$ pages, and a buffer management algorithm $A$, we say $A$ is *optimal*, iff for any other algorithm $A'$, $C(A(X, b)) \leq C(A'(X, b))$.

For magnetic disks, $C(Y)$ is often assumed to be linear to $|Y|$. Clearly, this assumption does not hold for flash disks, because $C$ heavily depends on the write/read ratio and the write patterns of $Y$. Therefore, each I/O request, either logical or physical, has to be represented as a tuple of the form $(op, pageNum)$, where *op* is either "R" (for a read request) or "W" (for a write request).

While the above formalization defines our problem, our goal is not to find the optimal algorithm in theory, but a practically applicable one that has acceptable runtime overhead and minimizes I/O cost as far as possible.

An intuitive idea to address the write pattern problem is to increase the DB *page size*, which is the unit of data transfer between the buffer layer and the file system (or the raw device directly) in most database systems. It would be an attractive solution if the overall performance could be improved this way, because only a simple adjustment of a single parameter would be required. However, a naive increase of the page size generally leads to more unnecessary I/O (using

[1] As an example, the MTRON MSP-SATA7525 flash disk achieves 12,000 IOPS for random reads and only 130 IOPS for random writes of 4 KB blocks, while high-end magnetic disks typically have 200 IOPS for random I/O [2].

the same buffer size), especially for OLTP workloads, where random accesses dominate. Furthermore, in multi-user environments, large page sizes favor thread contentions. Hence, a more sophisticated solution is needed.

Even with flash disks, maintaining a high hit ratio—the primary goal of conventional buffer algorithms—is still important, because the bandwidth of main memory is at least an order of magnitude higher than the interface bandwidth provided by flash disks. Based on our discussion so far, we summarize the basic principles of flash-aware buffer management as follows, which are also the design principles of the CFDC algorithm:

**P1** Minimize the number of physical writes.
**P2** Address write patterns to improve the write efficiency.
**P3** Keep a relatively high hit ratio.

### 1.3 Contributions

This paper is an extension of our preliminary work on flash-aware buffer management [4], where the basic ideas of the CFDC algorithm are introduced. The major contributions that distinguish this paper from [4] and improve the CFDC algorithm are:

- We discuss critical issues related to transaction management.
- We demonstrate the flexibility and efficiency of CFDC using a variant of it that integrates LRU-k as a base algorithm. To the best of our knowledge, this is the first work that examines the feasibility of a hybrid algorithm for flash-aware buffer management.
- We introduce metrics to quantitatively study spatial locality of I/O requests.
- We accomplish an extensive empirical performance study covering all relevant algorithms in a DBMS-based environment. So far, such a study is missing in all previous contributions.

The remainder of this paper is organized as follows. Sect. 2 sketches the related work. Sect. 3 introduces our algorithm, while its experimental results are presented in Sect. 4. The concluding remarks are given in Sect. 5.

## 2 Related Work

LRU and CLOCK [5] are among the most widely-used replacement policies. The latter is functionally identical to the Second Chance [6]: both of them often achieve hit ratios close to those of LRU. LRU-k is a classical algorithm specific for DB buffer management [7]. It maintains a history of page references which keeps track of the recent $k$ references to each buffer page. When an eviction is necessary, it selects the page whose *k-th recent* reference has the oldest timestamp. Parameter $k$ is tunable, for which the value 2 is recommended. Both *recency* and *frequency* of references are considered in its victim selection decisions. Thus theoretically, it can achieve higher hit ratios and is (more) resistant to scans than LRU-based algorithms, for which *recency* is the only concern.

CFLRU [8] is a flash-aware replacement policy for operating systems based on LRU. At the LRU end of its list structure, it maintains a *clean-first region*, 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 LRU tail is selected as victim. The size of the clean-first region is determined by a parameter $w$ called the *window size*. By evicting clean pages first, the buffer area for dirty pages is effectively increased—thus, the number of flash writes can be reduced.

LRUWSR [9] is a flash-aware algorithm based on LRU and Second Chance, using only a single list as auxiliary data structure. The idea is to evict clean and cold-dirty pages and keep the hot-dirty pages in buffer as long as possible. When a victim page is needed, it starts search from the LRU end of the list. If a clean page is visited, it will be returned immediately (LRU and clean-first strategy). If a dirty page is visited and is marked "cold", it will be returned; otherwise, it will be marked "cold" (Second Chance) and the search continues.

REF [10] is a flash-aware replacement policy that addresses the pattern of page flushes. It also maintains an LRU list and has a *victim window* at the MRU end of the list, similar to the clean-first region of CFLRU. Victim pages are only selected from the so-called *victim blocks*, which are blocks with the largest numbers of pages in the victim window. From the *set* of victim blocks, pages are evicted in LRU order. When all pages of the victim blocks are evicted, a *linear search* within the victim window is triggered to find a new set of victim blocks. This way, REF ensures that during a certain period of time, the pages evicted are all accommodated by a small number of flash blocks, thus improving the efficiency of FTL.

CFLRU and LRUWSR do not address the problem of write patterns, while REF does not distinguish between the clean and dirty states of pages. To the best of our knowledge, CFDC is the only flash-aware algorithm that applied all the three basic principles P1 to P3 introduced in Sect. 1.

## 3 The CFDC Algorithm

### 3.1 The Two-Region Scheme

CFDC manages the buffer in two regions: the *working region* $W$ for keeping *hot* pages that are frequently and recently revisited, and the *priority region* $P$ responsible for optimizing replacement costs by assigning varying priorities to page clusters. A *cluster* is a set of pages located in proximity, i. e., whose page numbers are close to each other. Though page numbers are logical addresses, because of the space allocation in most DBMSs and file systems, the pages in the same cluster have a high probability of being physically neighbored, too. The size of a cluster should correspond, but does not have to be strictly equal to the size of a flash block, thus information about exact flash block boundaries are not required.

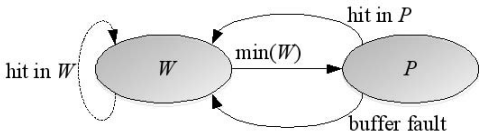



**Fig. 1.** Page movement in the two-region scheme

A parameter $\lambda$, called *priority window*, determines the size ratio of $P$ relative to the total buffer. Therefore, if the buffer has $B$ pages, then $P$ contains $\lambda$ pages and the remaining $(1 - \lambda) \cdot B$ pages are managed in $W$. Note $W$ does not have to be bound to a specific replacement policy. Various conventional replacement policies can be used to maintain high hit ratios in $W$ and, therefore, prevent hot pages from entering $P$.

Fig. 1 illustrates the page movement in our two-region scheme. The code to be executed upon a fix-page request is sketched in Algorithm 1. If a page in $W$ is hit (line 3), the base algorithm of $W$ should adjust its data and structures accordingly. For example, if LRU is the base algorithm, it should move the page that was hit to the MRU end of its list structure. If a page in $P$ is hit (line 5), a page $min(W)$ is determined by $W$'s victim selection policy and moved (demoted) to $P$, and the hit page is moved (promoted) to $W$. In case of a buffer fault, the victim is always first selected from $P$ (line 7). Only when all pages in $P$ are fixed, we select the victim from $W$. Considering recency, the newly fetched page is first promoted to $W$.

### 3.2 Priority Region

Priority region $P$ maintains three structures: an LRU list of clean *pages*, a priority queue of *clusters* where dirty pages are accommodated, and a hash table with cluster numbers as keys for efficient cluster lookup. The cluster number is derived by dividing page numbers by a constant *cluster size*.

The victim selection logic in $P$ is shown in Algorithm 2. Clean pages are always selected over dirty pages (line 1–2). If there is no clean page available, a cluster having the lowest priority is selected from the priority queue of dirty pages and the oldest unfixed page in this cluster is selected as victim (line 3–6). The oldest page in the cluster will be evicted first, if it is not re-referenced there. Otherwise, it would have been already promoted to $W$. Once a victim is selected from a cluster, its priority is set to minimum (line 8) and will not be updated anymore, so that the next victims will still be evicted from this *victim cluster*, resulting in strong spatial locality of page evictions.

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

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

**Algorithm 1.** fixPageTwoRegion

```
data   : buffer B with working region W and priority region P;
         page number of the requested page p
result : fix and return the requested page p
1  if p is already in B then
2      if p is in W then
3          adjust W as per W's policy;
4      else
5          demote min(W), promote p;
6  else
7      page q := selectVictimPr;
8      if q is null then
9          q := select victim from W as per W's policy;
10     if q is dirty then
11         flush q;
12     clear q and read content of p from external storage into q;
13     p := q;
14     if p is in P then
15         demote min(W), promote p;
16 fix p in the buffer and return p;
```

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. 2. The pages in a large cluster have a higher probability of being sequentially accessed.

Both spatial and temporal factors are considered by the priority function. The sum in the dividend in Formula 1, called *inter-page distance* (IPD), 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 an IPD of 3, while a cluster with pages $\{7, 5, 4, 6\}$ has an IPD 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 inserted into the priority queue ($min(W)$ is dirty), *globaltime* is incremented by 1. We derive its cluster number and perform a hash lookup using this cluster number. If the cluster does not exist, a new cluster containing this page is created with the current *globaltime* and inserted to the priority queue. Furthermore, it is registered in the hash table. Otherwise, the page is added to the existing cluster and the priority queue is maintained if necessary. If page $min(W)$ is clean, it simply becomes the new MRU node in the clean list.

**Algorithm 2.** selectVictimPr

```
   data   : priority region P consisting of a list of clean pages L in LRU order and
            a priority queue of dirty-page clusters Q
   result : return a page v as replacement victim
1  if L not empty then
2      v := the first unfixed page starting from the LRU end of L;
3  if v is null then
4      cluster c := lowest-priority cluster in Q with unfixed pages;
5      if c not null then
6          v := the oldest unfixed pages in c;
7          if v not null then
8              c.ipd := 0;
9  return v;
```

After demoting $min(W)$, the page to be promoted, say $p$, will be removed from $P$ and inserted to $W$. If $p$ is to be promoted due to a buffer hit, we update its cluster IPD including the timestamp. This will generally increase the cluster priority according to Formula 1 and cause $c$ to stay in the buffer for a longer time. This is desirable since the remaining pages in the cluster will probably be revisited soon due to locality. In contrast, when adding demoted pages to a cluster, the cluster timestamp is not updated.

### 3.3 Independence of Transaction Management

In principle, recovery demands needed to guarantee ACID behavior for transaction processing [11] may interfere with the optimization objectives P1 – P3. To achieve write avoidance and clustered writes to the maximum possible extent, the buffer manager should not be burdened with conflicting update propagation requirements. Fortunately, our CFDC approach implies a NoForce/Steal policy for the logging&recovery component providing maximum degrees of freedom [11]. NoForce means that pages modified by a transaction do not have to be forced to disk at its commit, but only the redo logs. Steal means that modified pages can be replaced and their contents can be written to disk even when the modifying transaction has not yet committed, provided that the undo logs are written in advance (observing the WAL principle (write ahead log)). Furthermore, log data is buffered and sequentially written—the preferred output operation for flash disks. With these options together, the buffer manager has a great flexibility in its replacement decision, because the latter is decoupled from transaction management. In particular, the replacement of a specific dirty page can be delayed to save physical writes or even advanced, if necessary, to facilitate clustered page flushes and thereby improve the overall write efficiency. Hence, it comes as no surprise that NoForce/Steal is the standard solution for existing DBMSs.

Another aspect of recovery provision is checkpointing to limit redo recovery in case of a system failure, e.g., a crash. To create a checkpoint at a "safe

place", earlier solutions flushed all modified buffer pages thereby achieving a transaction-consistent or action-consistent firewall for redo recovery on disk. Such *direct checkpoints* are impractical anymore, because—given large DB buffer sizes—they would repeatedly imply limited responsiveness of the buffer for quite long periods[2]. Today, the method of choice is *fuzzy checkpointing* [12], where only metadata describing the checkpoint is written to the log, but displacement of modified pages is obtained via asynchronous I/O actions not linked to any specific point in time. Clearly, these actions may be triggered to perfectly match with flash requirements and the CFDC principle of performing clustered writes.

### 3.4 Further Performance Considerations

As outlined, CFDC pe se is not constrained by recovery provisions, in particular, properties such as NoSteal or Force [11]. Such constraints could occur if orthogonality to other components would be violated. An example is the Force policy, with which we could achieve transaction-consistent DB states together with shadowing and page locking. But such an approach would cause low concurrency and overly frequent page propagations—two properties extremely hurting high-performance transaction processing [12].

Prefetching of pages plays an important role for conventional disk-based buffer management: It is not hindered by flash disks. But, because of their random-read performance, prefetching becomes much less important, because pages can be randomly fetched on demand without (hardly) any penalty in the form of access latency. Even better, because prefetching always includes the risk of fetching pages later not needed, CDFC must not use this conventional speed-up technique and can, nevertheless, provide the desired access performance.

As a final remark: The time complexity of our algorithm depends on the complexity of the base algorithm in $W$ and the complexity of the priority queue. The latter is $O(\log m)$, where $m$ is the number of clusters. This should be acceptable since $m \ll \lambda \cdot B$, where $\lambda \cdot B$ is the number of pages in $P$.

## 4 Performance Study

### 4.1 Test Environment

In all experiments, we use a native XML DBMS designed according to the classical *five-layer reference architecture*. For clarity and simplicity, we only focus on its bottom-most two layers, i. e., the *file manager* supporting page-oriented access to the data files, and the *buffer manager* serving page requests. Although designed for XML data management, the processing behavior of these layers is very close to that of a relational DBMS.

[2] While checkpointing often done in intervals of few minutes, systems are restricted to read-only operations. Assume that many GBytes would have to be propagated to multiple disks using random writes (in parallel). Hence, reaction times for update operations could reach a considerable number of seconds or even minutes.

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, 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 DB engine are installed on the magnetic disk. The test data (as a DB file) resides on the flash disk which is a 32 GB MTRON MSP-SATA7525 based on NAND flash memory.

We deactivated the file-system prefetching and used *direct I/O* to access the DB file, so that the influences of file system and OS were minimized. All experiments started with a *cold* DB buffer. Except for the native code responsible for direct I/O, the DB engine and the algorithms are completely implemented in Java. CFDC and competitor algorithms are fully integrated into the XML DBMS and work with other components of the DB engine.

In the following, we use CFDC-k to denote the CFDC instance running LRU-k ($k = 2$) and use CFDC-1 for the instance running LRU in its working region. Both of them are referred to as CFDC if there is no need to distinguish. We cross-compared seven buffer algorithms, which can be classified in two groups: the flash-aware algorithms including CFLRU, LRUWSR, REF, CFDC-k, and CFDC-1; the classical algorithms including LRU and LRU-k ($k = 2$). The *block size* parameter of REF, which should correspond to the size of a flash block, was set to 16 pages (DB page size = 8 KB, flash block size = 128 KB). To be comparable, the *cluster size* of CFDC was set to 16 as well. The $VB$ parameter of REF was set to 4, based on the empirical studies of its authors. Furthermore, we used an improved version of CFLRU which is is much more efficient at runtime yet functionally identical to the original algorithm.

### 4.2 Measuring Spatial Locality

We define the metric *cluster-switch count (CSC)* to quantify the spatial locality of I/O requests. Let $S := (q_0, q_1, \ldots, q_{m-1})$ be a sequence of I/O requests, the metric $CSC(S)$ reflects the spatial locality of $S$:

$$CSC(S) = \sum_{i=0}^{m-1} \begin{cases} 0, \text{ if } q_{i-1} \text{ exists and in the same cluster as } q_i \\ 1, \text{ otherwise} \end{cases} \quad (2)$$

Sequential I/O requests are a special case of high spatial locality, where pages are accessed in a forward or reverse order according to their locations on the storage device. If $d(S)$ is the set of distinct clusters addressed by a sequential access pattern $S$, we have $CSC(S) = |d(S)|$.

Let $R := (p_0, p_1, \ldots, p_{n-1})$ be the sequence of logical I/O requests and $S$ the sequence of physical I/O requests, we further define the metric *cluster-switch factor* ($CSF$) as:

$$CSF(R, S) = CSC(S)/CSC(R) \quad (3)$$

$CSF$ reflects the efficiency to perform clustering for the given input $R$. To compare identical input sequences, it is sufficient to consider the $CSC$ metric alone. For magnetic disks, if we set the cluster size equal to the track size, then $CSC(S)$ approximates the number of disk seeks necessary to serve $S$. For flash disks, we

consider only the $CSC$ and $CSF$ of logical and physical write requests, because flash read performance is independent of spatial locality.

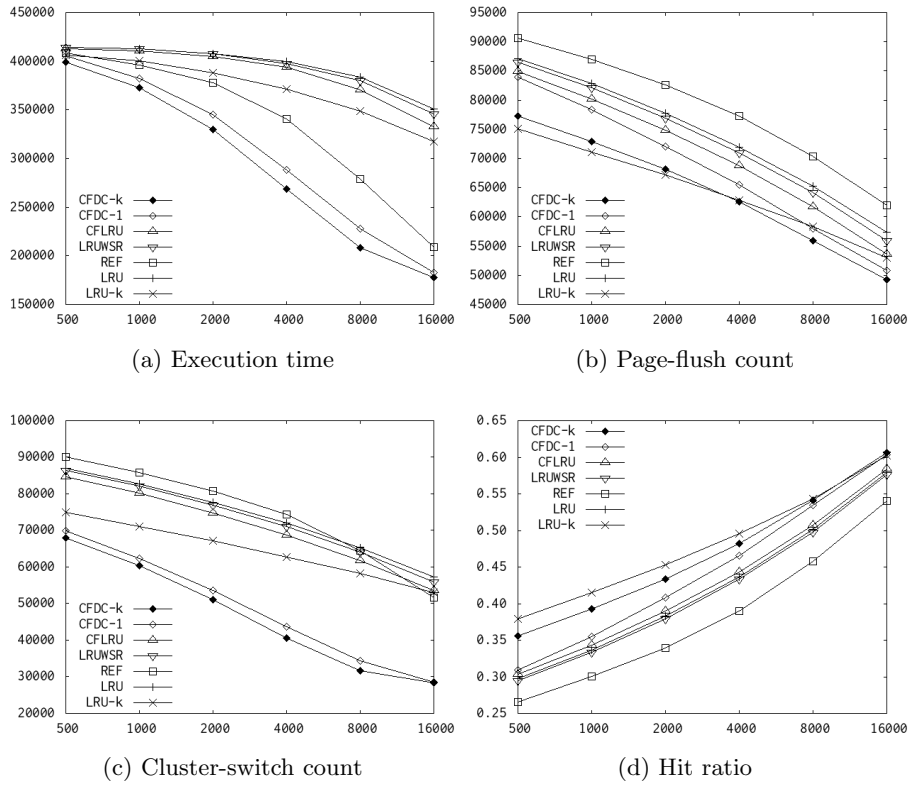
The *clustered writes* of CFDC are write patterns with high spatial locality and thus minimized cluster-switch counts. Compared to CFDC, the sequence of dirty pages evicted by the algorithm REF generally has a much higher $CSC$, because it selects victim pages from a *set* of victim blocks and the victim blocks can be addressed in any order. Because the sequence of dirty pages evicted can be viewed as multiple sequences of clustered writes that are interleaved with one another, we call the approach of REF *semi-clustered writes*.

### 4.3 Synthetic Trace

Our synthetic trace simulates typical DB buffer workloads with mixed random and sequential page requests. Four types of page references are contained in the trace: 100,000 single page reads, 100,000 single page updates, 100 scan reads, and 100 scan updates. A *single page read* requests one page at a time, while a *single page update* further updates the requested page. A *scan read* fetches a contiguous sequence of 200 pages, while a *scan update* further updates the requested sequence of pages. The page number of the single page requests are randomly generated between 1 and 100,000 with an 80–20 self-similar distribution. The starting page numbers of the scans are uniformly distributed in $[1, 10^5]$. All the 100,000 pages are pre-allocated in a DB file with a physical size of 784 MB in the file system. Thus a page fault will not cause an extra allocate operation.

We varied the buffer size from 500 to 16,000 pages (or 4–125 MB) and plotted the results of this trace in Fig. 2a. CFDC-k and CFDC-1 are very close, with CFDC-k being slightly better. Both CFDC variants clearly outperform all other algorithms compared. For example, with a buffer of 4,000 page frames, the performance gain of CFDC-k over REF is 26%. Detailed performance break-downs are presented by Fig. 2b, 2c, and 2d, corresponding to the three metrics of interest: number of page flushes, spatial locality of page flushing, and hit ratio. REF suffers from a low hit ratio and a high write count, but is still the third-best in terms of execution times due to its semi-clustered writes. LRU-k performs surprisingly good on flash disks—even better than the flash-aware algorithms CFLRU and LRUWSR. This emphasizes the importance of principle P3.

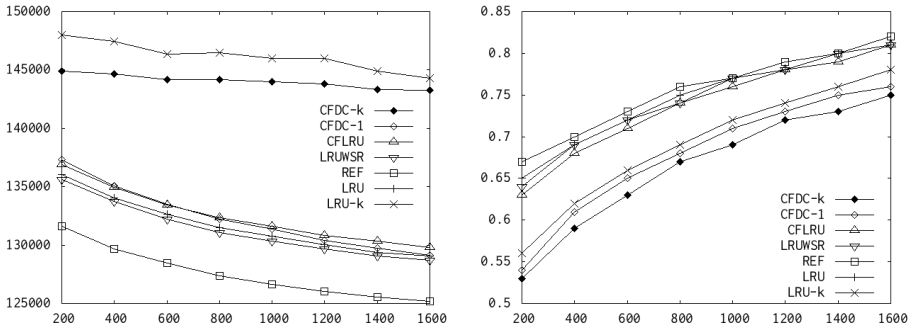
To examine scan resistance, we generated a set of traces by changing the locality of the single page requests of the previous trace to a 90–10 distribution and varying the number of scan reads and scan updates from 200 to 1,600. The starting page numbers of the scans are moved into the interval $[100001, 150000]$. The buffer size configured in this experiment equals the length of a scan (200 pages). Thus, we simulate the situation where sequential page requests push the hot pages out of the buffer. The buffer hits in this experiment are compared in Fig. 3a. While most algorithms suffer from a drop in the number of hits between 5% to 7%, the hits of CFDC-k only decrease by 1% (from 144,926 to 143,285) and those of LRU-k only decrease about 2.5%. This demonstrates that CFDC-k gracefully inherits the merits of LRU-k. Another advantage of CFDC is demonstrated by Fig. 3b: it has always the lowest $CSF$, i. e., its page flushes are efficiently clustered.




(a) Execution time (b) Page-flush count

(c) Cluster-switch count (d) Hit ratio

**Fig. 2.** Performance figures of the synthetic trace




(a) Hit count (b) Cluster-switch factor

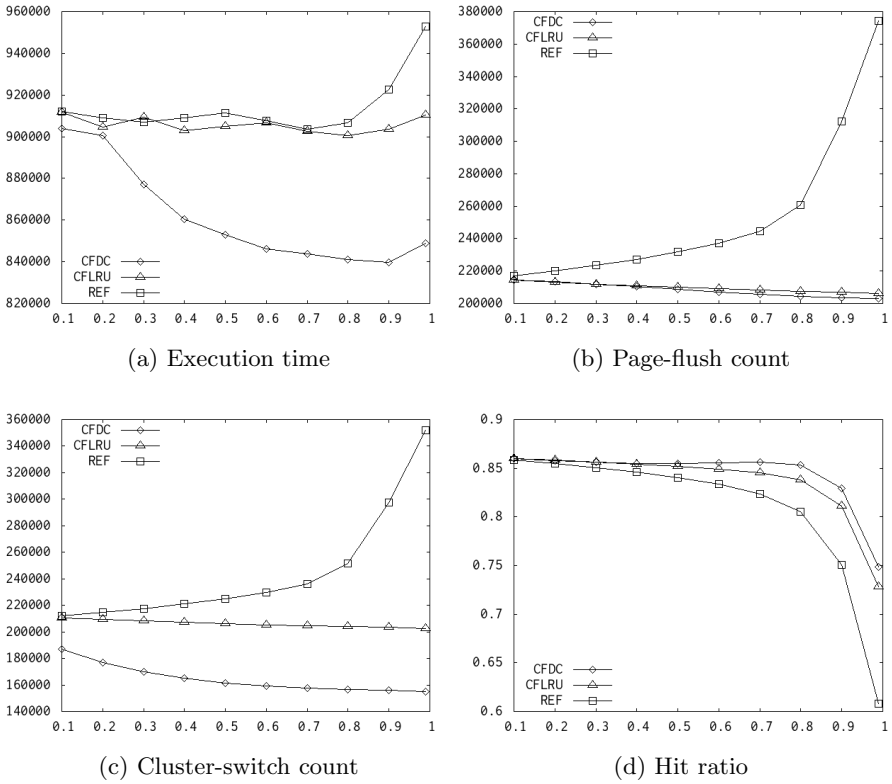
**Fig. 3.** Increasing the number of scans

### 4.4 Real-Life OLTP Traces

In this section we present the experiments performed using two real-life OLTP traces. CFDC-k and LRU-k are of lower practical importance due to their higher complexity ($O(\log n)$), therefore, we keep them out for better clarity.

**The TPC-C Trace.** The first trace was obtained using the PostgreSQL DBMS. Our code integrated into its buffer manager recorded the buffer reference string of a 20-minutes TPC-C workload with a scaling factor of 50 warehouses.

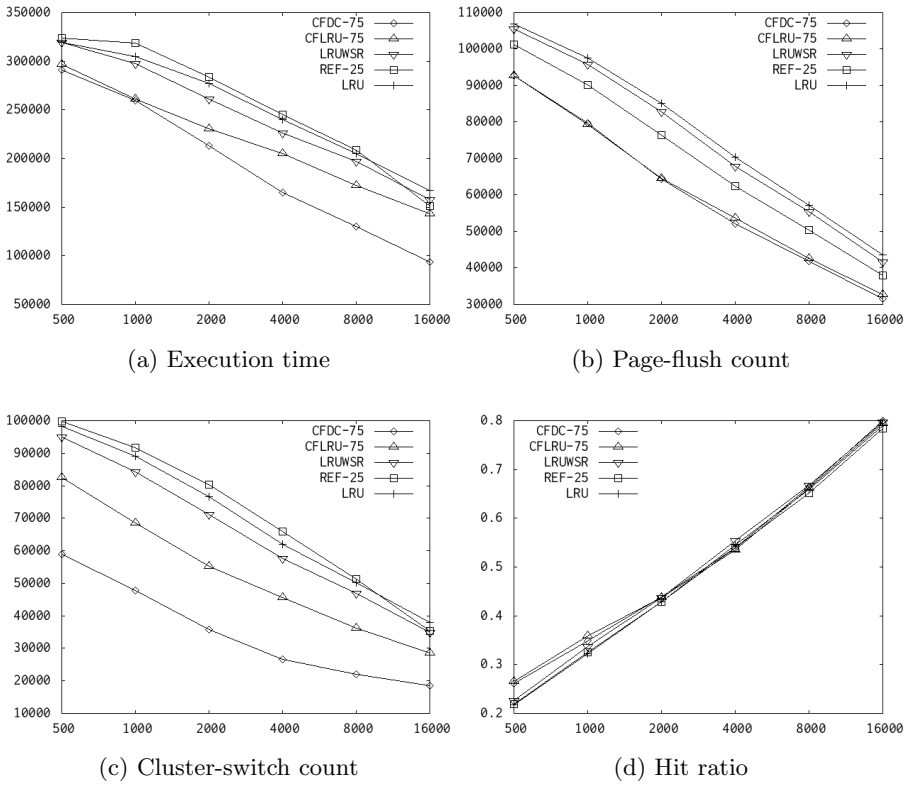
In our two-region scheme, the size of the priority region is configurable with the parameter $\lambda$, similar to the parameter *window size* ($w$) of CFLRU. The algorithm REF has a similar configurable *victim window* as well. For simplicity, we refer to them uniformly with the name "window size". In the experiments discussed so far, this parameter is not tuned—it was set to 0.5 for all related algorithms. To examine its impact under real workload, we ran the TPC-C trace with algorithms CFDC, CFLRU, and REF configured with window size from 0.1 to 0.99 relative to the total buffer size using 1,000 pages in this experiment.




(a) Execution time (b) Page-flush count

(c) Cluster-switch count (d) Hit ratio

**Fig. 4.** Impact of window size on the TPC-C trace




(a) Execution time (b) Page-flush count

(c) Cluster-switch count (d) Hit ratio

**Fig. 5.** Performance figures of the Bank trace

The performance metrics are shown in Fig. 4. The performance of CFDC benefits from an increasing window size. Its runtime goes slightly up after a certain window size is reached (0.9 in this case). This is because, with the size of the working region approaching zero, the loss of the hit ratio is too significant to be covered by the benefit of reducing physical writes and performing clustered writes in the priority region. Similar behavior is also observed for CFLRU at at window size 0.8. For CFDC and CFLRU, a larger window size leads to smaller number of writes. In contrast, the number of physical writes generated by REF grows quickly with an increase of the window size (Fig. 4b), resulting in a sharp runtime increase beginning at window size 0.8. This is due to two reasons: First, in REF's victim window, the sizes of the blocks are the only concern when selecting a victim block, while temporal factors such as recency and frequency of references are ignored. Second, REF does not distinguish between clean and dirty pages such that an increase of the window size does not necessarily lead to more buffer hits of dirty pages.

**The Bank Trace.** The second trace used here is a one-hour page reference trace of the production OLTP system of a Bank. It was also used in experiments of [7] and [13]. This trace contains 607,390 references to 8-KB pages in a DB having a size of 22 GB, addressing 51,870 distinct page numbers. About 23% of the references update the page requested, i. e., the workload is read-intensive. Even for the update references, the pages must be first present in the buffer, thus more reads are required. Moreover, this trace exhibits an extremely high access skew, e.g., 40% of the references access only 3% of the DB pages used in the trace [7].

For each of the algorithms CFDC, CFLRU, and REF, we ran all experiments three times with the window size parameter set to 0.25, 0.50, and 0.75 respectively, denoted as REF-25, REF-50, REF-75, etc., and chose the setting that had the best performance. The results are shown in Fig. 5. Even under this read-intensive and highly skewed workload, CFDC is superior to the other algorithms. The performance gain of CFDC over CFLRU is, e.g., 53% for the 16,000-page setting and 33% for the 8,000-page setting. Under such a workload, most of the hot pages are retained in a large-enough buffer. Therefore, the differences in hit ratios become insignificant as the buffer size is beyond 2000 pages.

The performance study does not focus on the MTRON flash disk. We also ran all the experiments on a low-end flash disk (SuperTalent FSD32GC35M) with similar observations. Furthermore, except for the execution time, other metrics collected are independent of any system and device.

## 5 Conclusions and Outlook

As systematic and empirical study, we extensively evaluated the performance behavior of all competitor algorithms for flash-aware DB buffer management in an identical environment. Therefore, we could accurately and thoroughly cross-compare those algorithms under a variety of parameters and workloads, where the majority of measurement results clearly prove CFDC's superiority among its competitors. The advantages of CFDC can be summarized as follows:

- With the clean-first strategy and the optimization in the priority region, it minimizes the number of physical writes (P1).
- It efficiently writes on flash disks by exploiting spatial locality and performing clustered writes, i. e., it evicts the sequence of writes that can be most efficiently served by flash disks (P2).
- Flash-specific optimizations do not compromise high hit ratios (P3), i. e., a large number of reads and writes can be served without I/O.
- Our two-region scheme makes it easy to integrate CFDC with conventional replacement policies in existing systems. The CFDC-k variant—maybe of lower practical importance due to its higher complexity—served as an example for this flexibility. Modern replacement policies such as ARC [13] and LIRS [14] can be easily integrated into CFDC without modification.

Our experiments did not cover asynchronous page flushing. In practice, page flushes are normally not coupled with the victim replacement process—most

of them are performed by background threads. However, these threads can obviously benefit from CFDC's dirty queue, where the dirty pages are already collected and clustered.

In this paper, we explored flash disks as an exclusive alternative to magnetic disks. However, database systems may employ hybrid storage systems, i. e., flash disks and magnetic disks co-exist in a single system. As another option in DBMS I/O architectures, flash memory could serve as a non-volatile caching layer for magnetic disks. Both I/O architectures posing challenging performance problems deserve a thorough consideration in future research work.

## Acknowledgement

We are grateful to Gerhard Weikum for providing the OLTP trace and to Sebastian Bächle and Volker Hudlet for the team work. We thank anonymous referees for valuable suggestions that improved this paper. The research is partially supported by the Carl Zeiss Foundation.

## References

1. Woodhouse, D.: JFFS: the journalling flash file system. In: The Ottawa Linux Symp. (2001)
2. Gray, J., Fitzgerald, B.: Flash disk opportunity for server applications. ACM Queue 6(4), 18–23 (2008)
3. Bouganim, L., et al.: uFLIP: Understanding flash IO patterns. In: CIDR (2009)
4. Ou, Y., et al.: CFDC: a flash-aware replacement policy for database buffer management. In: DaMoN Workshop, pp. 15–20. ACM, New York (2009)
5. Corbato, F.J.: A paging experiment with the multics system. In: Honor of Philip M. Morse, p. 217. MIT Press, Cambridge (1969)
6. Tanenbaum, A.S.: Operating Systems, Design and Impl. Prentice-Hall, Englewood Cliffs (1987)
7. O'Neil, E.J., et al.: The LRU-K page replacement algorithm for database disk buffering. In: SIGMOD, pp. 297–306 (1993)
8. Park, S., et al.: CFLRU: a replacement algorithm for flash memory. In: CASES, pp. 234–241 (2006)
9. Jung, H., et al.: LRU-WSR: integration of LRU and writes sequence reordering for flash memory. Trans. on Cons. Electr. 54(3), 1215–1223 (2008)
10. Seo, D., Shin, D.: Recently-evicted-first buffer replacement policy for flash storage devices. Trans. on Cons. Electr. 54(3), 1228–1235 (2008)
11. Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Computing Surveys 15(4), 287–317 (1983)
12. Mohan, C., et al.: ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17(1), 94–162 (1992)
13. Megiddo, N., Modha, D.S.: ARC: A self-tuning, low overhead replacement cache. In: FAST. USENIX (2003)
14. Jiang, S., Zhang, X.: LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In: SIGMETRICS, pp. 31–42 (2002)