FBARC: I/O Asymmetry-Aware Buffer Replacement Strategy

Paul Dubs [#1], Ilia Petrov [#2], Robert Gottstein [#1], Alejandro Buchmann [#1]

[#1]*Databases and Distributed Systems Group TU-Darmstadt, Germany,* [#2] *Data Management Lab, Reutlingen University*
[#1]{dubs | gottstein | buchmann}@dvs.tu-darmstadt.de, [#2]ilia.petrov@retulingen-university.de

ABSTRACT

Buffer Management is central to database systems; it minimizes the access gap between memory and disk. Primary criterion of most buffer management strategies is hitrate maximization (based on recency, frequency). New storage technologies exhibit characteristics such as read/write asymmetry and low read latency. These have significant impact on the buffer manager: due to asymmetry the cost of page eviction may be several times higher than the cost of fetching a page. Hence buffer management strategies for modern storage technologies must consider write-awareness and spatial locality besides hitrate.

In this paper we introduce FBARC - a buffer management strategy designed to address I/O asymmetry on Flash devices. FBARC is based on ARC and extends it by a write list utilizing the spatial locality of evicted pages to produce semi-sequential write patterns. FBARC adds an additional list to host dirty pages grouping them into fixed regions called clusters based on their disk location. In comparison to LRU, CFLRU, CFDC, and FOR+, FBARC: (i) addresses *write-efficiency* and *endurance*; (ii) offers comparatively high *hitrate*; (iii) is *computationally-efficient* and uses static grid-based clustering of the page eviction list; (iv) adapts to *workload changes*; (v) is *scan-resistant*. Our experimental evaluation compares FBARC against LRU, CFLRU, CFDC, and FOR+ using trace-driven simulation, based on standard benchmark traces (e.g. TPC-C, TPC-H).

1. INTRODUCTION

As a central component of database systems Buffer Management is not only instrumental to high performance query processing, but also plays a critical role in transaction management. From an architectural perspective, it addresses the task of minimizing the substantial *access gap* in the access latencies of main memory and spinning disks. The access gap is a technological property of traditional memory hierarchies: accessing data in a buffer resident page takes less than hundred nanoseconds whereas accessing disk-based data is in the lower millisecond range (Fig. 1).

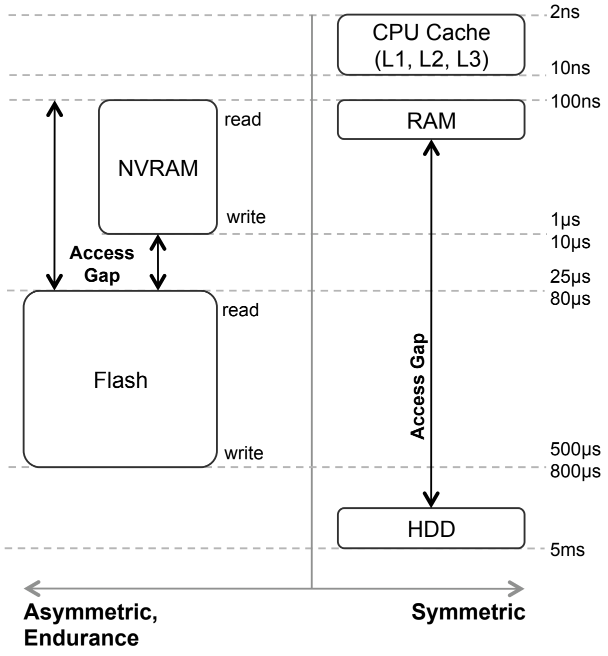

Figure 1: Access Gap in Evolving Memory Hierarchies

Traditional buffer management strategies would utilize temporal locality of page accesses to increase the probability of locating the requested pages in the buffer (hit-rate) and minimize expensive disk accesses. Such strategies utilize two criteria to select victim pages to be evicted from the buffer once new ones are requested: recency and frequency. Since the traditional technologies exhibit symmetric performance (i.e. reads are as fast as writes), maximizing the hit-rate thus minimizing disk accesses of whatever type is a sufficiently general, single design criterion.

Over the last half decade new storage technologies (Flash, Non-Volatile Memories) have emerged. These change the memory hierarchy radically with their properties:

(i) *Low access latencies (operation-specific)* - orders of magnitude lower than HDD latencies.

(ii) *Asymmetry* - Access latencies for read operations vary from approx. 100 ns (PCM) to several microseconds (Flash) [6]. Write operations, are at least an order of magnitude slower, which gives rise to the term *read-write asymmetry.* In addition, write latencies for random and sequential vary significantly.

(iii) *Mixed loads performance* - The mixed-load performance of Flash devices is sub-optimal compared to their pure read or write performance.

(iv) *Endurance* - besides being slower repeated writes (10^5...10^8) to the same memory location can cause the memory cells to malfunction hence the notion of *endurance* or *longevity.*

All these influence the design goals for a buffer management strategy: (a) maximizing the hit-rate may yield sub-optimal performance because page evictions may cause expensive writes; (b) due to the I/O asymmetry a trade off between hit-rate and eviction costs may result in lower overall costs. In the face of the shrinking and operation dependent access gap, we claim that addressing I/O asymmetry should be considered as first-class criterion besides hit-rate. The write-efficiency aspects of I/O asymmetry can be addressed by utilizing spatial locality during the page eviction process. Existing approaches rely on write region clustering, which may result in high computational costs. The design tradeoff, therefore, is to minimize eviction costs and reduce computational intensiveness for the sake of memory consumption and lower hit-rate in order to minimize the overall performance penalty. These are inline with the present hardware trends of: increasing memory volumes and low read latencies.

On Flash devices, in addition, the I/O asymmetry is especially strong in the case of random writes [5]. These are one to two orders of magnitude slower than random and sequential reads respectively [5]. In addition these have long-term effects on performance and endurance. This makes it important to perform sequential or semi-sequential writes. Our initial experiments indicate that when a set of random write I/O requests is concentrated on a relatively small region then they are performed nearly as fast as sequential requests. This effect is also studied in [3]. This observation affects the way spatial locality can be addressed, especially the choice of the clustering algorithm (Section 3). By static clustering the set of dirty pages, the pages for eviction can be subdivided into smaller regions; even though those may not be optimally filled and may result in random writes, they will be executed as sequential ones. Hence the desired tradeoff may be achieved at acceptable computational costs. Append Log Storage is another approach that sequentializes write requests and is studied in [13].

In this paper we introduce FBARC (**F**lash-**B**ased-ARC) - a buffer management strategy designed to address I/O asymmetry on Flash devices. FBARC is based on ARC [22] and extends it by a write list utilizing the spatial locality of evicted pages to produce semi-sequential write patterns. In recent years several similar strategies have been proposed: CFLRU [25], CFDC [24] CASA [23], FOR/FOR+ [21], [18]. In comparison, FBARC:

- addresses *write-efficiency* and *endurance*
- offers comparatively high *hitrate*
- is *computationally-efficient* and uses static grid-based clustering of the page eviction list
- adapts to *workload changes*
- is *scan-resistant* - a property inherited from ARC

The contributions of the present paper are as follows. We introduce the FBARC buffer eviction strategy and discuss its algorithm and design principles. As part of our experimental evaluation we compare FBARC against, LRU, CFLRU, CFDC, and FOR+. Our experiments rely on trace-driven simulation, utilizing TPC-C, TPC-H and pgBench traces. The results indicate that FBARC outperforms them by between 7% and 11%. FBARC achieves similar hitrate and is less computationally-intensive (up to 2.5x). FBARC is scan resistant, exhibits comparatively low I/O times and asymmetry friendly I/O behavior.

The paper is organized as follows: in Section 2 we discuss related approaches. In section 3 we describe the FBARC design principles and algorithm. Section 4 focuses on the experimental design and setup as well as the trace description and simulation. The experimental results are discussed in Section 5.

2. RELATED WORK

Database buffer management is a traditionally well-studied field; a broad overview is provided in [10]. We distinguish between *general purpose* (FIFO, LRU [8], ARC [22]) and *special purpose* algorithms. The special purpose algorithms can be further sub-divided into: (i) *embedded* [20, 16]; (ii) *clustering* [24, 25, 15]; (iii) *page state* [21, 18, 17, 25] and; (iv) *persistent buffer extensions* [9, 4, 19].

General purpose cache replacement algorithms usually can be used at any layer in the memory hierarchy. They, however, do not utilize special system knowledge to speed up accesses. *Special purpose* algorithms on the other hand are designed to use system information and around restrictions of the respective target area. *Embedded algorithms* are designed to use low-level system information for devices with limited computational resources. An elevator algorithm for example needs to know where the disk head is positioned at the moment to decide which block to access next, or an FTL knows if a write will need a full merge or a switch merge. A *clustering algorithm* targets the spatial locality by grouping dirty in-buffer pages into clusters and making the eviction decision based on a composite per-cluster score. An algorithm that uses the *page state* in its decision which page to evict can adapt itself to different kinds of access. That way it can address the read/write asymmetry on Flash media.

CFLRU [25], CFDC [24] and FOR/FOR+ [21] are cache replacement strategies that fall in the class of *page state* dependent replacement algorithms.

CFLRU [25] is a very light weight extension of the traditional LRU algorithm. It prioritizes dirty pages over clean pages by replacing clean pages at the LRU end of the LRU stack first (the so called priority region). This behavior may be suboptimal in respect to the following reasons. Firstly, the priority region tends to fill up with dirty pages, and thus the constant eviction victim selection runtime of LRU degrades to a linear run time based on the size of the priority region. Secondly, it affects the hitrate substantially since clean pages are evicted as soon as they enter the priority region, thus effectively lowering the cache size for clean pages. While the size of the priority region can be dynamic it needs a tuning parameter for the cost of a write or, if the priority region is static, it needs a well chosen size based on the workload and thus replacing one tuning parameter with another.

CFDC [24] improves upon CFLRU in two ways. It still divides the cache into a work region and a priority region, but it further sub-divides the priority region into a clean and a dirty region. This way it overcomes the search-for-clean-pages problem and allows for a constant search time. In addition, CFDC clusters the dirty page priority region based on the spatial locality of the pages. It still evicts clean pages first, but if there are none then the oldest page from the cluster with the least priority will be evicted and the priority for this cluster is set to zero until it is completely evicted. The priority is computed on every change of the priority region for all clusters and is based on the temporal and spatial locality of the pages within the cluster. This causes CFDC to be very computationally intensive.

The FOR [21] algorithm approaches the problem from a different perspective. It classifies pages according to two operations: the last completed and the currently executing. Using the resulting 'interoperation distance' it can recognize a hot page even if it was already evicted, and thus can retain it longer when it is requested again. This approach requires that two data points (last read and last write) are saved for every accessed page over its whole life. Additionally, every operation has to be recorded, i.e. every pin and every dirty flagging has to change the possibly very long operation list. To overcome these issues the authors introduce FOR+ which tries to approximate the interoperation distance based on the currently cached pages. This approximation works by dividing the LRU stack into a hot and cold region and assigning page weights based on region membership. However, as FOR/FOR+ are built on an LRU basis there may be scan resistance issues. Furthermore, both FOR and FOR+ need carefully chosen cost estimation for reads and writes and FOR+ needs another tuning parameter for the size of its cold page index.

The DULO [14] cache replacement algorithm tries to combine both the temporal and the spatial locality on top of a LRU stack. It was designed for rotating disks and thus tries to reduce the amount of random accesses in favor of sequential prefetches. It divides the LRU stack into three parts: the working area at the MRU end which consists of single pages, the fixed size sequencing bank and the sequential area at the LRU end which consists of sequential page sequences. The sequencing bank is used to create sequences of pages that can be evicted at the same time and can be read in later with the help of prefetching. To reduce the sequencing cost DULO uses the CLOCK algorithm, which allows it to do the sequencing only when an eviction is needed instead of doing it on every access that causes the LRU stack to change. At eviction time DULO then evicts a complete sequence of pages from the LRU end. This works well on a system that uses a lot of sequential read accesses, but as prefetching does not work for writes it is difficult to achieve write awareness. On flash based storage DULO is not as efficient, as random reads are not much slower than sequential reads. Although FBARC takes a different approach, it utilizes a CLOCK derived algorithm for cluster selection.

BPLRU [20] is an embedded device cache level replacement algorithm. It works on a block level and targets only the write buffer on a flash device assuming a log-block FTL [7]. As all its pages are dirty and ready to be written at any time it doesn't have to keep any "clean"-hitrates high. As any hit on a page within a block moves the complete block to the MRU position this can lead to a decreased hitrate as older pages are kept longer in the cache. As a protection against cache flushing it uses a sequential write detection algorithm, which puts sequentially accessed blocks directly to the LRU position. When a block has to be evicted BPLRU attempts to perform a switch merge and pads the block if needed, which reduces the amount of the expensive full merges. Another embedded replacement algorithm that relies on spatial locality and focuses on write caches is WOW [12].

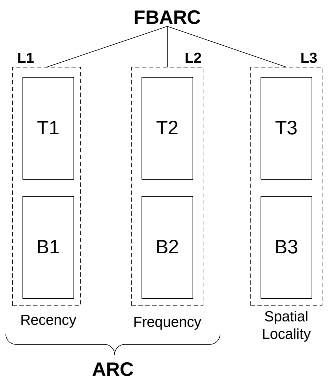

Figure 2: List organization of FBARC

In brief, FBARC is a *special purpose, clustering* buffer management algorithm, which partly uses *page state* information. FBARC relies on ARC as a general purpose algorithm. It features a distinction between recency and frequency and can automatically adapt to workload changes. Next, we introduce the FBARC algorithm.

3. FBARC

FBARC is an extension of ARC. Both strategies employ multiple LRU lists to organize the buffer space (Fig. 2). ARC [22] organizes recently accessed pages in an LRU list L1; and frequently accessed pages in a another list L2. Both L1 and L2 are subdivided into two further lists - L1 into T1 and B1; L2 into T2 and B2. The T-lists (T1 and T2) manage the available buffer frames; the sum of their sizes equals the total number of buffer frames available - *c*. The B-lists (B1 and B2) are virtual lists, containing metadata about pages which were evicted from the respective T-lists. The B-lists contain in total another - *c* page metadata entries; thus the total page metadata maintained by ARC is *2c* entries. In addition, ARC balances the sizes of L1 and L2 to realize the tradeoff between recency and frequency and ultimately adapt to workload changes. We return to this point later in the section with a detailed discussion.

FBARC extends ARC with an additional list (L3) to address spatial locality (Fig. 2). Like L1 and L2 it is subdivided into T3 and B3. Unlike T1 and T2, T3 manages clusters of dirty pages. A cluster contains merely a list of pages and a reference counter. Alternatively, B3 is LRU-organized and contains the metadata for single uncached pages. FBARC uses a static clustering algorithm (called *GRID clustering*). We logically divide the disk space into static equally-sized regions called clusters. As discussed in Section 1: even though the clusters can be 'porously' filled with dirty pages, which will result in random writes requests, they will be performed as if they were sequential since the page addresses are concentrated on a relatively small region (cluster). The cluster size is a tunable parameter, specific to the concrete Flash device. The proposed *GRID clustering* represents a tradeoff between efficiency, simplicity and acceptable computational intensiveness.

In the following we introduce the algorithm in detail. Assume that the cache is completely filled and a page is requested. First, FBARC verifies whether the page has already been processed by locating the metadata for that page (Alg. 1, line 2) in L1, L2 or L3. If it finds it, it checks if the page is cached (Alg. 1, line 3). If so, the page is moved to the MRU position of T2 and the page is returned. If the page was not cached, but the metadata was found, then FBARC uses this information to adjust the balance between T1, T2 and T3 (Alg. 1, lines 6-16) and then invokes the eviction process (Alg. 1, lines 17, 18). The page is then placed at the MRU position of T2 and the page is returned (Alg. 1, line 19).

How the balance between T1, T2 and T3 is changed depends on where the metadata hit occurred. If the metadata was located in B1, then the target size for T1 is increased by one, if it was in B3 then the target size for T3 is increased by one. If it was in B2 then the logical target size for T2 is increased by one. T2 has only a logical target size, as it is the result of $c - ts_1 - ts_3$. So if the target size of T2 has to grow, another target size has to shrink. To decide which target size has to shrink we compare them with each other (Alg. 1, line 9), and then shrink the bigger one. This protects the balance in cases where many hits in B2 would otherwise skew it too fast.

The target size is a soft upper bound under which the lists should stay. If a list is bigger than its target size then the next eviction victim will be chosen from that list.

If the page is unknown (i.e. the look up did not return any metadata) then FBARC might have to evict an additional B-list page. This housekeeping ensures that the complete historic view stays confined within $3c$. First, FBARC checks if L1 (i.e. T1+B1) is the same size as the cache size and whether there is at least one entry in B1 (Alg. 1, line 25). If this is the case then the LRU entry of B1 is removed. Otherwise, it goes on to check whether the total size of all lists (L1, L2 and L3) exceeds the cache size (Alg. 1, line 27). It then checks if L1+L2 is bigger than or equal to $2c$ (Alg. 1, line 28). If so, the LRU entry of B2 is removed, otherwise FBARC checks if the total size of all lists (L1+L2+L3) is equal to $3c$ and in that case removes the LRU entry of B3 (Alg. 1, line 30).

This confines the maximum size of L1 to the cache size, the maximum size of L2 to twice the cache size and the maximum size of L3 to three times the cache size. The reasoning behind this is that L1 represents the recency and an excessively large recency list would contain pages that are not recent any more. L2 on the other hand represents frequency, and the information that a page was frequently accessed in the past is useful even if it was not that recent. Similarly, the information that a page was written to in the past is even more useful, as we try to retain dirty pages longer to reduce the amount of write I/Os that a buffer management strategy produces.

After the housekeeping is done, the eviction process is invoked (Alg. 1, line 34). The new page is read in, placed at the MRU position of T1 and returned (Alg. 1, lines 36-39).

Algorithm 1 FBARC: Get Procedure

```
 1: procedure GET($x$)
 2:     if $x \in buffers$ then
 3:         if $x.origin \in \{t_1, t_2, t_3\}$ then          ▷ Cache Hit
 4:             MOVETOMRU($t_2$, $x$)
 5:         else                                ▷ Cache Miss, Hit in History
 6:             if $x.origin = b_1$ then
 7:                 $ts_1 \leftarrow$ MAX($c$, $ts_1 + 1$)
 8:             else if $x.origin = b_2$ then
 9:                 if $|ts_1| \geq |ts_3|$ then
10:                     $ts_1 \leftarrow$ MIN(0, $ts_1 - 1$)
11:                 else
12:                     $ts_3 \leftarrow$ MIN(0, $ts_3 - 1$)
13:                 end if
14:             else if $x.origin = b_3$ then
15:                 $ts_3 \leftarrow$ MAX($c$, $ts_3 + 1$)
16:             end if
17:             EVICT($x$)
18:             $buffers[x] \leftarrow$ READIN($x$)
19:             MOVETOMRU($t_2$, $x$)
20:         end if
21:     else                                           ▷ Page is unknown
22:         if $|t_1| = c$ then
23:             EVICTLRU($t_1$)                ▷ instead of calling evict
24:         else                                          ▷ Housekeeping
25:             if $|l_1| = c$ then
26:                 REMOVELRU($b_1$)
27:             else if $|l_1| + |l_2| + |l_3| \geq c$ then
28:                 if $|l_1| + |l_2| \geq 2c$ then
29:                     REMOVELRU($b_2$)
30:                 else if $|l_1| + |l_2| + |l_3| = 3c$ then
31:                     REMOVELRU($b_3$)
32:                 end if
33:             end if
34:             EVICT($x$)
35:         end if
36:         $buffers[x] \leftarrow$ READIN($x$)
37:         MOVETOMRU($t_1$, $x$)
38:     end if
39:     return $buffers[x]$
40: end procedure
```

The eviction process first checks if a page has to be evicted at all (Alg. 2, line 2), and if so FBARC continuously evicts as many pages from different source candidates (lists) as needed to free at least one buffer slot. The source selection for eviction candidate is made based on the current and target size of all lists, and on the metadata (if present) of the requested page. First, it checks if T1 is too big (Alg. 2, line 3), then it checks if T3 is too big (Alg. 2, line 5), and if this is also not the case, then it just selects the biggest of all three as the source.

If the eviction source candidate is T1 or T2 then FBARC just tries to evict the page at the LRU position (Alg. 3, line 5). If that page is clean, then the buffer for it is cleaned, and its metadata is moved to the corresponding bottom list (Alg. 3, lines 9, 10). If it is dirty it will be added to T3 (Alg. 4) and the eviction process is restarted (Alg. 2, line 2) as the dirty page was not written out, and thus no free place was created yet. If the source is T3, then a special eviction is started (Alg. 5).

Algorithm 2 FBARC: Eviction

```
1: procedure EVICT($x$)
2:     while $|buffers| = c$ do
3:         if $|t_1| > 0 \land (|t_1| > ts_1 \lor (x \in b_2 \land |t_1| = ts_1))$ then
4:             $source \leftarrow t_1$
5:         else if $|t_3| > 0 \land (|t_3| > ts_3 \lor (x \in b_2 \land |t_3| = ts_3))$ then
6:             $source \leftarrow t_3$
7:         else
8:             $source \leftarrow$ MAX($t_1, t_2, t_3$)
9:         end if
10:        EVICTLRU($source$,$x$)
11:    end while
12: end procedure
```

Algorithm 3 FBARC: Eviction of LRU for l_1 and l_2

```
1: procedure EVICTLRU($source$, $x$)
2:     if $source = t_3$ then
3:         EVICTFROMT3($nil$)
4:     else
5:         $lru \leftarrow$ REMOVELRU($source$)
6:         if $lru.dirty$ then
7:             ADDTOT3($lru$)
8:         else
9:             REMOVE($buffers[lru]$)
10:            MOVETOMRU($source.history$, $lru$)
11:        end if
12:    end if
13: end procedure
```

When a page is added to T3, FBARC calculates its cluster id based on the page number and a given cluster size (Alg. 4, line 2). If a cluster with that cluster id already exists, then the page is added to it and its reference counter is incremented (Alg. 4, lines 10, 5). If there is no pre-existing cluster with that id, then a new one is created and added to the cluster hash map (Alg. 4, lines 7, 8).

Algorithm 4 FBARC: Add Page to t_3

```
1: procedure ADDTOT3($x$)
2:     $clusterId \leftarrow page/clusterSize$
3:     if $clusterId \in clusters$ then
4:         $cluster \leftarrow clusters[clusterId]$
5:         $x.cluster.refCount++$
6:     else
7:         $cluster \leftarrow$ NEW CLUSTER()
8:         $cluster[clusterId] = cluster$
9:     end if
10:    $cluster.pages$.APPEND($x$)
11: end procedure
```

There are two reasons for a page to leave T3. The first is that the page was hit by an access and is moved to T2 (Alg. 1, line 4). In that case FBARC removes the page from its cluster and increases its reference count by one (Alg. 5, lines 2, 3, 7). The reasoning behind this decision is that the page might come back into the cluster if it stays long enough. If the cluster is empty after this operation, then the cluster is removed (Alg. 5, lines 4, 5). The second reason is that T3 was selected as a source in the eviction process. In that case FBARC estimates the eviction cost for each cluster (*clusterCost*) and selects the cluster with the lowest cost as the victim (Alg. 5, lines 10, 11).

The cost is estimated by the reference count and the amount of pages that are in the cluster. Thus a cluster with a low reference count and a lot of pages is more likely to be replaced then a cluster with the same reference count and only a few pages. This allows a cluster with only a few pages to live longer and thus gives it a chance to accumulate more pages. A selection solely based on the cluster size is insufficient, since then very small clusters could block a buffer slot for a long time.

After the victim cluster is chosen, the reference count of all other clusters is decreased by the victims reference count (Alg. 5, line 13). This is mainly to keep the reference count low such that inactive clusters that once had a high activity can be evicted sooner. The pages of the victim cluster are then written in sequential order to the storage and added to the MRU position of B3.

Algorithm 5 FBARC: Evict Page from t_3

```
1: procedure EVICTFROMT3($x$)
2:     if $x \neq nil$ then
3:         $x.cluster.pages$.REMOVE($x$)
4:         if $|x.cluster.pages| = 0$ then
5:             $clusters$.REMOVE($x.cluster$)
6:         else
7:             $x.cluster.refCount++$
8:         end if
9:     else
10:        $clusterCost \leftarrow \lambda x \leftarrow (x.refCount, -|x.pages|)$
11:        $bestCluster \leftarrow$ MIN($clusters$, $key = clusterCost$)
12:        for all $cluster \in clusters$ do
13:            $cluster.refCount -= bestCluster.refCount$
14:        end for
15:        $bestCluster.pages$.SORT()
16:        for all $page \in bestCluster$ do
17:            WRITEOUT($page$)
18:            MOVETOMRU($b_3$,$page$)
19:        end for
20:        $clusters$.REMOVE($bestCluster$)
21:    end if
22: end procedure
```

4. EXPERIMENTAL ANALYSIS

4.1 Experiments

We experimentally evaluated the performance of FBARC, comparing it against a number of different buffer management algorithms (ARC, FOR+, LRU, CFLRU, CFDC) under realistic workloads (TPC-C, TPC-H, pgbench) in different scenarios. We implemented a simulation framework, hosting and executing the set of buffer management strategies. An overview of the trace-driven simulation process is provided in Fig. 3. It comprises three phases, which are described in detail in the following sections: (i) Recording raw traces for different DB workloads (Sect. 4.2.1, 4.2.2); (ii) simulation of the eviction policies and generation of block access traces (Sect. 4.2.3); (iii) execution of the I/O traces on a physical SSD using FIO (Sect. 4.2.4).

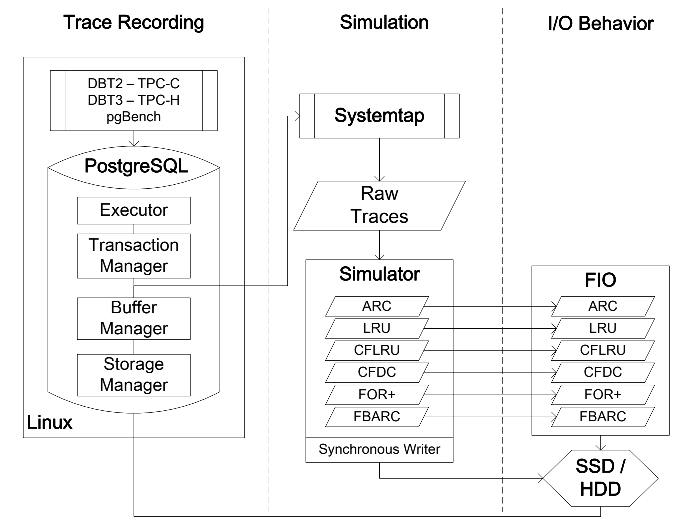

Figure 3: Simulation Process

Blocksize 4 KB	SSD Intel X25-E/64GB	HDD Hitachi HDS72161
	Avg[µs]	Avg[ms]
Sequential Read	53	0.133
Sequential Write	59	0.168
Random Read	167	10.8
Random Write	435	5.6

Figure 4: I/O latencies 4KB

With this setup we are able to study (i) different aspects of spatial and temporal locality, e.g. hitrate and quality of the clustering algorithm; (ii) the computational intensiveness and the I/O behavior; as well as (iii) various additional aspects of the buffer management algorithms, e.g. scan-resistance or workload adaptation, in a controlled environment under repeatable conditions and realistic workloads. For fair performance evaluation we assign equal total amount of memory to all compared algorithms (unless explicitly stated otherwise).

The following system was used to perform the experimental analysis: 4 GB RAM, Intel Core 2 Duo 3 GHz running under Linux (kernel 2.6.41). In addition, we used an Intel X25-E/64GB enterprise SSD and a Hitachi HDS72161 7200RPM SATA2 320GB HDD; the latencies of both drives are listed in Fig. 4.

4.2 Instrumentation

4.2.1 Raw Traces

To ensure repeatable experiments and comparable experimental results under realistic workloads we record traces of PostgreSQL buffer manager method calls. Probing points were set on buffer methods *fetching* a page as well as those *marking a buffer dirty* within PostgreSQL's *bufmgr.c*. We used a Fedora Linux (kernel 2.6.41) with the *Systemtap* extension (translator 1.6; driver 0.152) to record the trace. By setting probing points accordingly ("prior to the buffer manager", Fig. 3), we guarantee that the traces record the real database behavior for the respective OLTP or OLAP workload, and eliminate the influence of PostgreSQL's eviction policy and storage manager. These are simulated for each implemented strategy by the simulation framework (Sections 4.2.3 and 4.2.4)

4.2.2 Workload and Trace Characterization

An out of the box PostgreSQL (version 9.1.1) was instrumented for OLTP and OLAP workloads. Both types of workloads are considered since they stress different aspects of a buffer management strategy, and target asymmetry differently.

As an OLTP benchmark we used *DBT2* [1], an open source implementation of the TPC-C benchmark and *pgbench* [11], which is the PostgreSQL internal benchmark. DBT2 was instrumented with 200 Warehouses (nominal DB size approx. 20GB), 10 database connections and 10 terminals per warehouse. PostgreSQL uses 24 MB shared buffer space. The other OLTP trace was recorded with *pgbench*, which was instrumented with a scale factor of 600. As OLAP load we used *DBT3* [2], an open source implementation of the TPC-H benchmark. PostgreSQL was instrumented with scale factor 3 (nominal DB size approx. 13GB). The default database page size of PostgreSQL (8KB) is used throughout all benchmarks.

For the HDD benchmarks, we used scaled down traces that have the same workload characteristics as the *pgbench* and *DBT2* traces. They are scaled down to one tenth of the original trace size (nominal DB size approx. 2GB). Scaling was performed to reduce the otherwise unacceptably long experimental run-time.

All traces contain index and table accesses. The traces have, however, been purged of PostgreSQL specific "service" accesses to make them more comparable with traces that would be gained by other methods.

Fig. 6, 7, 5, 8 show the access distribution per trace. The dot-dashed blue and dotted green lines show the access frequency per distinct page, while the dashed red and continuous yellow lines show the cumulative part of the total accesses a distinct page receives. We have chosen a logarithmic scale for both the access frequency as well as the distinct page count to emphasize the impact of the most frequently accessed pages.

Trace H: DBT3 (TPC-H) is a typical OLAP benchmark, and that is also reflected in its trace. As Fig. 8 shows the low fraction of write accesses in the TPC-H throughput test (temporary data) and as such we use this trace mostly to investigate the effects of the write awareness extensions on a mostly read only trace. Additionally, as can be seen in the scatter diagram of this trace, it executes complex queries that focus on only a few relations at a single time (see appendix, Fig. 24), which also reflects as the plateaus seen in Fig. 8.

Trace B: The pgbench trace shows a TPC-B like access pattern. Even though some relations are accessed more frequent than others, the access distribution within the relations themselves are very uniformly random (see appendix, Fig. 23). That can also be observed in Fig. 5, as every plateau represents a single relation, and all the pages within it are equally often accessed. The writes are almost equally distributed, but they happen ten times less often.

Trace C: For DBT2 we have recorded two traces. One, as shown in Fig. 6, contains all accesses as they are defined in the TPC-C specification (**Trace C**). The other, as shown in Fig. 7 (**Trace** C_d), contains only accesses from the delivery transaction. While trace C allows us to create a realistic

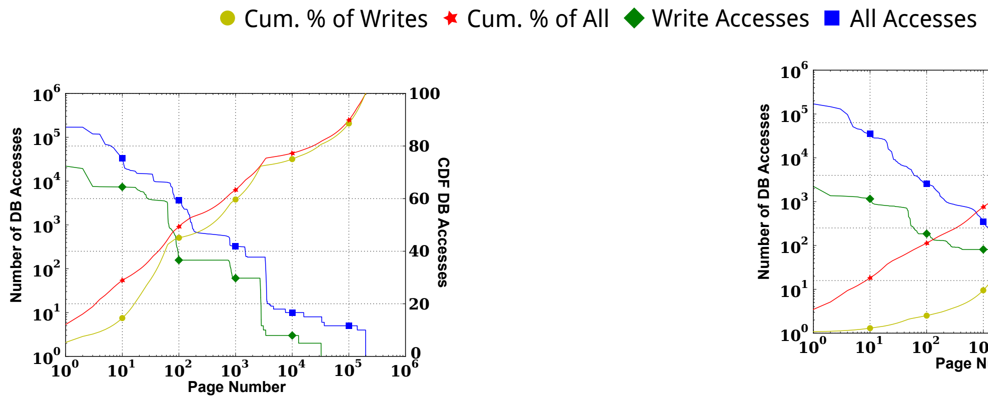

Figure 5: Trace B: pgBench - Scale Factor 600.

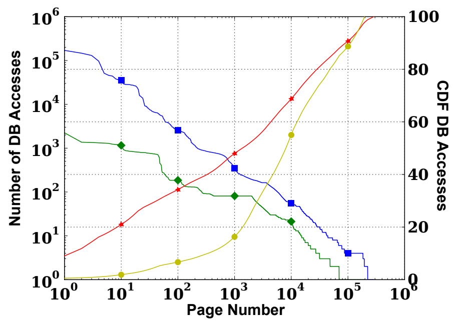

Figure 6: Trace C: DBT2 (TPC-C) - 200 Warehouses.

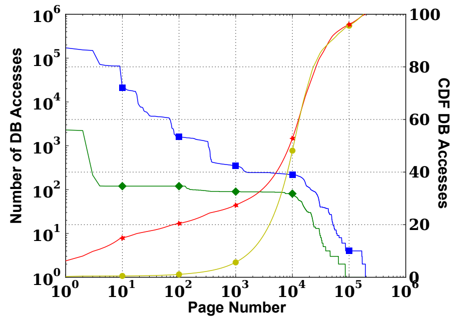

Figure 7: Trace C_d: DBT2 with 200 Warehouses. Delivery transaction accesses.

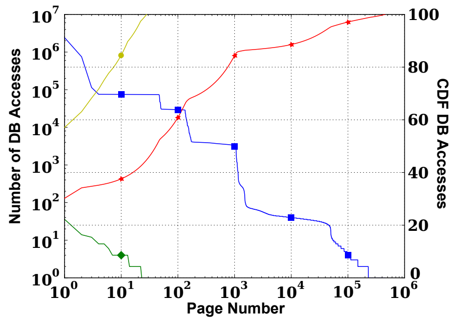

Figure 8: Trace H: DBT3 (TPC-H) - Scale Factor 3.

OLTP load, trace C_d allows us to have a write heavy OLTP-like load.

Fig. 6 and 7 show that for both traces more then half of all accesses are in a very narrow range of pages, as less then 10% of all pages account for more then 60% of all accesses. The write requests though are a lot less concentrated in both traces. Additionally the trace C_d shows a pattern of concurrent semi sequential accesses over most of the involved relations (see appendix, Fig. 22).

Trace SR: In order to stress the scan resistance of the different algorithms the framework simulates large sequential accesses by introjecting "parasites" into the OLTP traces. After an amount of OLTP load a parasite is introjected. The parasite covers (pollutes) the complete cache size and accesses pages which are not covered by the main trace to avoid any interference. After that the OLTP load is resumed for n additional requests, where n is the cache size. The Load is resumed for only n additional requests to highlight the effect of the parasite. Each parasite is either write only or read only (see Section 5.4).

4.2.3 *Simulation and Metrics*

The simulation framework applies the buffer eviction policies to the raw traces. The simulated eviction policies are: (i) ARC; (ii) LRU; (iii) CFLRU with 10% (40%) priority region; (iv) CFDC with 10% (40%) priority region; (v) FOR+; (vi) FBARC (different cluster sizes). The simulation results are reported for the following metrics: (i) *hitrate*, (ii) *CPU time* - measure of computational complexity (Section 5.1); (iii) *I/O time* - measure of I/O performance (Section 5.2); (iv) *overall runtime* - is an interleaving of CPU and I/O time (Section 5.3).

4.2.4 *I/O Trace*

In the final step we benchmarked the traces on a physical SSD and HDD. The simulation framework translates page addresses into block addresses generating an I/O trace, which is used as an input for the FIO benchmark. FIO accesses the SSD/HDD as raw devices using direct I/O, and thus bypasses the I/O cache. We used FIO for the following reasons: a) guaranteed I/O behavior; b) controlled use of I/O parallelism; c) reliable performance metrics (IOPS, runtime). On SSD after each run of a single trace, we executed a series of random write operations (approx. 30GB) using FIO to avoid SSD state dependencies.

5. EXPERIMENTAL EVALUATION

The experiments of our experimental evaluation include: (i) investigation of the hitrate and the computational intensiveness of all algorithms (Section 5.1); (ii) examination of the I/O behavior of all buffer management strategies (Section 5.2); as well as (iii) the investigation of the overall time (Section 5.3). We investigate the influence of sequential operations on the algorithms (scan resistance), which gains importance in database systems optimized for new storage technologies or analytical processing (Section 5.4). Finally, we compare the performance on HDDs to investigate whether the algorithms have desirable features for a general purpose algorithm. Since all those experiments vary the cache size but keep the FBARC specific parameters, i.e. cluster size constant, we additionally analyze the influence of

		Trace B			Trace C			Trace C_d			Trace H		
		1024	2048	4096	1024	2048	4096	1024	2048	4096	1024	2048	4096
Hitrate [%]	LRU	90.26	91.40	92.29	78.68	81.55	83.77	76.53	79.06	79.38	90.02	91.12	92.63
	ARC	89.91	91.29	92.33	78.60	81.06	83.24	76.54	78.98	79.09	89.78	91.07	92.53
	FBARC	88.39	90.43	92.11	77.68	81.17	83.82	73.02	78.40	79.40	89.88	91.06	92.46
	CFLRU 10%	90.13	91.24	92.20	78.48	81.40	83.60	76.19	79.08	79.38	89.92	91.07	92.57
	CFLRU 40%	89.69	90.63	91.73	77.25	80.64	82.88	74.02	78.56	79.42	89.44	90.81	92.40
	CFDC 10%	90.10	91.24	92.20	78.15	81.21	83.50	75.67	79.06	79.39	89.90	91.05	92.57
	CFDC 40%	89.46	90.58	91.72	75.82	79.74	82.34	72.20	78.18	79.66	89.34	90.72	92.38
	FOR+	90.24	91.31	92.25	78.38	81.53	83.49	74.66	78.91	79.62	89.78	90.94	92.52
CPU [s]	LRU	55.94	58.61	63.79	70.74	79.91	94.36	125.83	139.92	145.03	147.07	153.36	179.49
	ARC	63.29	68.60	71.95	78.96	87.70	104.10	138.88	144.62	155.76	167.23	182.80	201.99
	FBARC	81.79	84.18	96.54	152.08	177.51	199.36	293.10	333.55	317.33	188.06	195.14	213.23
	CFLRU 10%	70.43	86.24	113.82	102.13	135.66	195.97	189.31	248.65	371.63	226.18	293.91	414.02
	CFLRU 40%	111.30	160.82	254.13	209.25	315.38	533.11	391.34	593.98	1038.69	223.49	288.71	413.21
	CFDC 10%	80.80	100.34	139.46	113.43	145.13	206.45	190.77	248.10	323.42	163.69	167.02	194.83
	CFDC 40%	130.86	196.65	331.69	198.89	298.83	501.99	328.95	487.48	777.72	161.15	165.83	176.39
	FOR+	174.57	218.34	282.86	257.75	365.33	545.26	352.24	442.31	694.68	392.49	546.64	938.95
I/O [s]	LRU	166.23	156.69	148.07	286.43	250.84	237.59	539.22	484.94	477.87	267.94	257.12	217.49
	ARC	167.55	158.43	148.52	297.54	258.42	232.95	536.95	486.32	485.71	264.70	232.34	215.56
	FBARC	180.44	164.19	148.93	299.94	255.12	224.01	581.28	478.80	441.83	263.56	232.03	216.34
	CFLRU 10%	164.90	157.92	150.86	295.90	254.19	232.62	539.56	485.13	477.74	268.76	256.51	217.32
	CFLRU 40%	171.79	162.42	153.66	297.26	257.29	233.63	566.00	492.66	476.41	269.35	256.42	217.49
	CFDC 10%	167.52	158.64	150.75	301.62	257.59	230.59	554.14	488.28	476.08	269.22	256.74	217.77
	CFDC 40%	173.26	161.02	154.88	329.66	280.03	247.45	613.08	504.01	471.26	268.43	255.33	217.06
	FOR+	164.33	155.48	149.41	292.92	255.91	232.48	559.94	490.47	474.07	268.73	256.54	216.55
Combined[s]	LRU	175.61	166.45	160.18	324.99	288.70	259.57	570.74	516.32	505.07	277.59	265.47	267.89
	ARC	180.16	167.28	159.38	327.83	293.85	264.74	571.20	518.18	513.92	275.08	273.24	284.83
	FBARC	191.48	172.92	160.94	324.19	278.64	245.99	607.10	495.23	456.05	278.14	278.72	292.41
	CFLRU 10%	177.84	167.41	156.95	322.06	285.56	254.51	567.89	509.35	494.43	290.39	321.26	426.94
	CFLRU 40%	179.41	179.68	261.75	326.04	340.06	535.93	583.95	620.08	1048.52	290.96	324.15	423.25
	CFDC 10%	176.43	164.38	161.37	328.39	291.04	254.63	585.46	511.57	497.51	278.87	267.01	280.63
	CFDC 40%	180.71	212.96	341.38	354.38	324.54	512.10	641.94	537.00	834.27	278.66	266.79	262.18
	FOR+	194.31	235.83	294.65	315.24	388.96	565.95	584.18	510.23	723.91	454.84	599.22	971.54

Figure 9: Results for Hitrate, CPU, I/O and Overall Time. Columns indicate amount of pages in the cache (1024 .. 4096). Charts covering specific aspects are presented in Fig. 9, 14, and 20.

the FBARC parameters in terms of the influence of the cluster size on the computational complexity and I/O behavior (Section 5.5). CFLRU and CFDC in special are analyzed with two different priority queue sizes of 10% and 40%. All results presented in this section are the average values of three experimental runs (since the standard deviation was consistently low we do not report it).

We have consolidated the results of all main experiments into Fig. 9. This allows to easily compare the results of different traces and see the influence of different factors on the combined metric. The algorithms are sorted in the given order, to allow for an easier comparison of FBARC with ARC and LRU.

5.1 CPU Time

In this section we examine the computational complexity (CPU time) and hitrate of each eviction policy, for different traces and buffer/cache sizes (Fig. 9, rows 9-16, "CPU"). *CPU time* measures the runtime required for each algorithm to complete the respective trace under full CPU utilization. It is an indication for the algorithm's pure *computational complexity* on the respective trace. I/O time and parallelism are not present.

Here LRU naturally outperforms all other algorithms. For small cache sizes the FBARC performance is comparable to that of other strategies. With increasing cache size FBARC has much better runtime than the other

For small cache sizes FBARC performs mostly comparable to or slower than the other strategies, but as the cache size increases that gap narrows. With increasing cache size FBARC has much better runtime than the other strategies; the highest difference is received on the largest buffer size. While the hitrates for the different algorithms are comparable and grow with the amount of cache used, the CPU times differ significantly.

FOR+ exhibits the highest computational complexity of all algorithms. It increases significantly with growing cache sizes. CFDC exhibits the highest computational intensiveness of all clustering algorithms especially for larger cache sizes. This is due to the employed clustering algorithm. CFLRU and FBARC have similar complexities, with CFLRU being better for smaller buffer sizes and FBARC taking lead for larger buffer sizes. LRU and ARC are both basic algo-

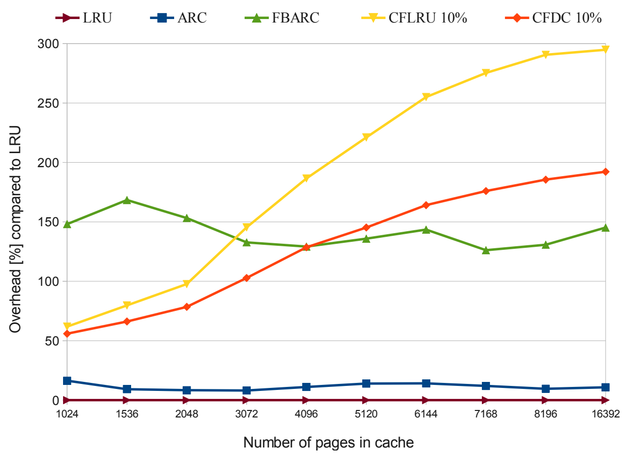

Figure 10: Relative CPU overhead for varying number of pages in cache .

	1024	2048	4096
LRU	6,485,952	5,613,128	4,938,152
ARC	6,511,440	5,761,776	5,098,056
FBARC	6,790,968	5,728,960	4,920,816
CFLRU 10%	6,549,112	5,657,208	4,988,896
CFLRU 40%	6,922,672	5,888,768	5,208,096
CFDC 10%	6,648,344	5,716,696	5,019,144
CFDC 40%	7,358,888	6,163,464	5,373,144
FOR+	6,580,288	5,618,288	5,021,456

Figure 11: Read KB in trace C. Columns indicate amount of pages in the cache (1024 .. 4096).

rithms and thus shine in this metric because they are write-oblivious and do not have the respective complexity.

Upcoming NVM technologies are characterised by higher asymmetry, smaller page sizes and hence yield higher CPU overhead for T3 Maintenance. The relative CPU overhead of FBARC compared to other strategies is (Fig. 10): (i) smaller; (ii) it also grows slower for increasing number of pages. In general, all clustering strategies are affected; to attack the issue the ample computational resources of many core CPUs need to be used.

5.2 I/O Time

In this section we report the I/O times for the investigated replacement strategies for different loads and buffer cache sizes. *I/O time* stands for the I/O time needed to write/read evicted/requested pages for the respective raw (benchmark) trace. The framework creates an I/O trace that corresponds to the simulated eviction policy and is used as input for FIO (Section 4.2.4). It is then executed with synchronous direct I/O without I/O parallelism. The CPU time is minimal and can therefore be neglected.

FBARC reads and writes faster than the other eviction policies. The higher write rate of FBARC is the result of the sequentialisation of random writes. It is significantly faster to write a whole cluster of pages which reside in spatial locality then writing in random. In addition, FBARC makes better use of I/O parallelism of Flash devices.

The results also show that the preferred eviction of clean pages leads to a higher value of read I/O (Fig. 11) and a higher read IOPS, nevertheless the influence of the prioritized eviction of clean data pages does not impact the hitrate negatively, as the results of the CPU time tests show.

	1024	2048	4096
LRU	3,168,016	2,908,216	2,691,712
ARC	3,135,872	2,880,960	2,687,672
FBARC	3,209,080	2,911,360	2,665,472
CFLRU 10%	3,123,896	2,878,440	2,674,600
CFLRU 40%	3,028,568	2,800,088	2,632,928
CFDC 10%	3,210,520	2,933,848	2,703,720
CFDC 40%	3,393,264	3,039,200	2,788,552
FOR+	2,965,504	2,717,352	2,595,992

Figure 12: Written KB in trace C. Columns indicate amount of pages in the cache (1024 .. 4096).

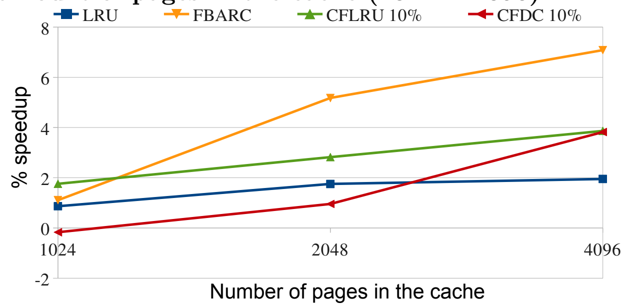

Figure 13: Overall speedup in percent compared to ARC on trace C

FOR+ shows the lowest amount of written bytes (Fig. 12). This is the result of its meta data use. It retains single hot dirty pages longer in cache then any other algorithm, due to its comparatively long write history list. Even though FOR+ writes less than FBARC it is not faster per se since the resulting write patterns lack spatial locality.

5.3 Overall Time

The *overall time* represents a combined metric unifying the *CPU time* and *I/O time*. It interleaves the computational tasks of a buffer strategy with I/O time for page fetch/eviction (Fig. 9, rows 25-32, "Combined"). The I/O requests are executed as synchronous I/O: its blocking nature guarantees that the complete I/O latency is accounted for. Thus I/O parallelism and asynchronous I/O are not present which represents the realistic worst case scenario. (In commercial implementations I/O parallelism will be present to varying degrees, increasing the performance.)

Fig. 9 (rows 25-32, "Combined") depicts the *overall time* of the algorithms under test and Fig. 13 and 14 depict the

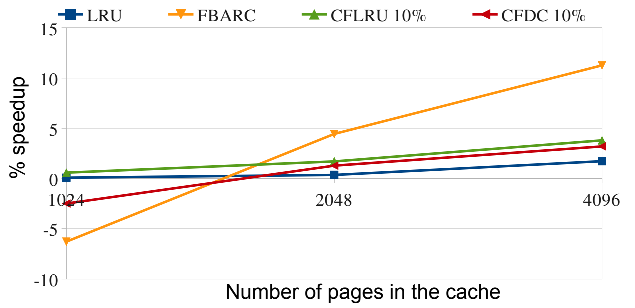

Figure 14: Overall speedup in percent compared to ARC on trace C_d.

relative improvement over ARC on trace C and C_d. We removed CFDC 40%, CFLRU 40% and FOR+ from those charts, because of their significant computational overhead (see also Fig. 9 (rows 13, 15, 16, “CPU”)).

FBARC works very well on trace C and gains up to 7% speedup with a big cache size. On trace C_d FBARC can get a speedup of up to 11% but it seems to struggle with a smaller cache size. The performance lag comes mainly from a comparatively low hitrate for that particular cache size. Even though FBARC has a higher computational intensiveness than some of the other strategies, its resulting I/O behavior does reduce the *overall time*.

In trace H there were so few write requests, that the hitrate and computational overhead are the main source of differences between the different strategies.

In trace B the very uniform random access seems to produce equally mostly unchanged results when compared to LRU and ARC, thus none of the write aware strategies can really improve over the basic strategies there.

5.4 Scan Resistance

Scan resistance is an important characteristic of buffer management algorithms with view of new storage technologies and analytical loads and cache-awareness. At their core both trends employ sequentializing write or read accesses, and become increasingly wide-spread. Not every buffer management algorithm is able to handle such type of accesses efficiently.

In order to stress the scan resistance of the different algorithms the framework simulates large sequential accesses by introjecting *parasites* into the OLTP traces. After an amount of OLTP load a sequential parasite is introjected and the scan resistance of the eviction policies examined. The parasite covers (pollutes) the complete cache size and accesses pages, which are not covered by the main trace to avoid any interference. After that the OLTP load is resumed. We employ both read and write parasite configurations to stress certain parts of the buffer management algorithms.

We compare the hitrate of an eviction policy on a trace without and with (a) a read parasite and (b) a write parasite introjected. Our findings show that the smaller the cache size the more likely the hitrate will drop on all strategies but ARC and FBARC. FBARC is influenced by neither write nor read parasites, while the other policies are significantly influenced. FBARC, like ARC, uses the recency list to filter out pages that are accessed only once, and thus a sequential parasite can only pollute a relatively small part of the cache. Even though a sequential write parasite can flush the T3 list, it does not influence the hitrate in a significant way as these pages are not frequently accessed. Additionally it will buffer the write parasite and write it out sequentially.

CFLRU, CFDC and LRU lose more than 10% of hitrate on smaller cache sizes while FOR+ looses a quarter of that. LRU derivatives such as CFDC and CFLRU react differently to read and write parasites due to the specifuc optimizations. As shown in Fig. 15 the effect of a write parasite on hitrate is twice the effect of a read parasite. LRU does not distinguish between reads and writes hence the identical numbers. The identical hitrate reduction for write parasites for CFDC and CFLRU can be easily explained with the fact that the parasite pollutes the whole buffer cache, regardless of the clustering algorithm and write optimization.

Hitrate [%]	128	256	512	1024	2048	128	256	512	1024	2048
ARC	87.89	90.23	91.70	92.92	93.12					
ARC (Read)	87.89	90.63	91.70	92.92	93.09	0.00	0.39	0.00	0.00	-0.02
ARC (Write)	87.89	90.63	91.70	92.92	93.09	0.00	0.39	0.00	0.00	-0.02
CFDC	88.67	90.63	91.80	92.77	93.09					
CFDC (Read)	80.08	83.20	85.94	88.62	90.14	8.59	7.42	5.86	4.15	2.95
CFDC (Write)	76.17	80.27	83.40	86.47	88.18	-12.50	-10.35	-8.40	-6.30	-4.91
CFLRU	88.67	90.63	91.80	92.77	93.12					
CFLRU (Read)	81.25	84.38	87.89	89.84	90.14	-8.20	-6.25	-3.91	-2.98	-3.00
CFLRU (Write)	76.17	80.27	83.40	86.47	88.18	-13.28	-10.35	-8.40	-6.35	-4.96
FBARC	88.28	90.43	91.70	92.92	93.04					
FBARC (Read)	87.89	90.43	91.60	92.92	92.92	-0.39	0.00	-0.10	0.00	-0.12
FBARC (Write)	88.28	90.43	91.70	92.92	92.94	0.00	0.00	0.00	0.00	-0.10
FOR+	89.45	90.63	91.80	92.82	93.14					
FOR+ (Read)	84.77	87.50	88.67	89.45	89.77	-4.69	-3.13	-3.13	-3.37	-3.37
FOR+ (Write)	85.55	87.50	88.67	89.45	89.79	-3.91	-3.13	-3.13	-3.37	-3.34
LRU	88.67	90.63	91.80	92.97	93.12					
LRU (Read)	76.17	80.27	83.40	86.47	88.18	-12.50	-10.35	-8.40	-6.49	-4.93
LRU (Write)	76.17	80.27	83.40	86.47	88.18	-12.50	-10.35	-8.40	-6.49	-4.93

Figure 15: Influence of Scan Resistance. Absolute Hitrate on the left, difference compared to baseline on the right. Columns indicate amount of pages in the cache (128 .. 2048).

	1024	2048	4096
FBARC	83.82	83.83	83.85

Figure 16: Influence of cluster size on hitrate(trace C). Columns indicate cluster size [pages] (1024 .. 4096)

The effect of read parasites on hitrate reduction of CFDC and CLRU depends on the cache size. For smaller cache sizes CFLRU exhibits smaller hitrate penalties, a fact that is due to the way the balance between read and write regions (Working/Clean-first region in CFLRU vs. Working/Priority region in CFDC) is being handled. FOR+ is not scan resistant, it however shows smaller hitrate reductions than LRU, since it uses additional information to keep hot pages longer than previously unknown pages. As for FOR+ ,regardless of all optimizations read and write parasites affect its hitrate reduction to the same degree.

5.5 Influence of the FBARC Cluster Sizes

Throughout all of the above experiments we kept the cluster size of FBARC (Section 3) constant. Since it accounts for physical parameters of the Flash Device it is a tunable parameter in FBARC. In this experiment we vary it from 1024 pages to 4096 pages, keeping the buffer size constant 4096 pages.

Fig. 16 shows that the cluster size has a marginal influence on the hitrate. For most traces the hitrate is unaffected or changes by less than 1%. Given a skewed workload where dirty buffers in L3 (Fig. 2) are requested will result in a cache hit under FBARC because of the CLOCK based cluster management algorithm.

The cluster size has a significant impact on the overall time (Fig. 17). The pages of the evicted cluster are written out as semi-sequential writes and all I/O operations are synchronous (Section 3, Alg. 5), which only confirms our initial hypothesis. In addition larger cluster sizes reduce the

	1024	2048	4096
FBARC	243.88	238.61	230.79

Figure 17: Influence of cluster size on overall time (trace C). Columns indicate cluster size [pages] (1024 .. 4096).

	1024	2048	4096
FBARC	201.02	182.57	152.56

Figure 18: Influence of cluster size on cpu time (trace C). Columns indicate cluster size [pages] (1024 .. 4096).

	128	256	512	1024	2048
LRU	4720.22	3959.84	3580.07	3219.36	2880.25
ARC	4691.32	4005.01	3625.47	3259.75	2900.58
FBARC	4740.55	3964.56	3568.04	3210.92	2863.26
CFLRU 10%	4762.38	3968.51	3578.98	3231.58	2895.06
CFLRU 40%	5062.04	4183.72	3665.22	3327.70	2981.58
CFDC 10%	4791.82	3975.31	3572.79	3196.70	2829.28
CFDC 40%	5106.15	4114.93	3568.73	3197.88	2815.65
FOR+	4868.25	4034.39	3553.66	3263.65	2977.73

Figure 19: I/O Time [s] on HDD. Columns indicate amount of pages in the cache (128 .. 2048).

amount of clusters managed by L3 and hence the computational complexity of FBARC (Fig. 18).

5.6 Behavior on HDD

In this section we evaluate how the different strategies behave when they are tested on an HDD. It is favorable if a special purpose algorithm performs well even in cases in which it was not intended to be used. For this test we scaled the used traces down to reduce the experimental run-time to an acceptable range (also see section 4.2.2). For this reason we have also included smaller cache sizes in the results.

Fig. 19 shows the overall time for the scaled down trace C to complete. Fig. 20 shows the relative speedup or slowdown compared to ARC. Most of the values are within 5% of ARC and most LRU derived strategies perform very similar.

These results also show that the computational intensiveness does not influence the overall time in any significant way. The I/O time and hitrate mostly dominate the overall performance.

6. CONCLUSIONS

New low-latency storage technologies with asymmetric performance characteristics call for buffer management strategies that treat both temporal and spatial locality as first class citizens. Hence replacement strategies need to optimize for both hitrate and eviction cost to ensure balanced performance.

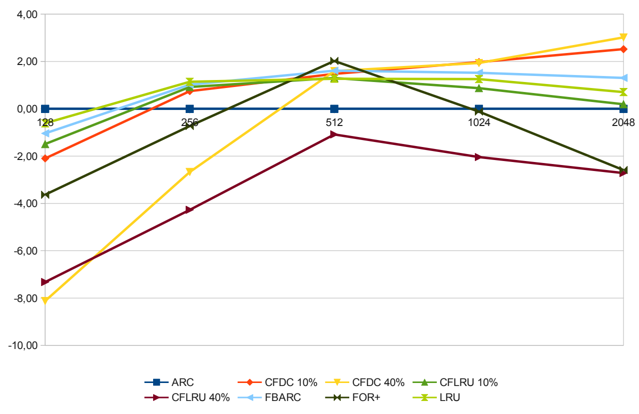

Figure 20: Relative change compared to ARC

In this paper we have presented FBARC, a buffer management strategy addressing I/O asymmetry on flash devices. As part of our experimental evaluation we compare FBARC against, LRU, CFLRU, CFDC, and FOR+. FBARC outperforms other strategies by between 7% and 11% even by achieving similar or slightly lower hitrates. This is a result of FBARC's asymmetry aware I/O behavior, which is also reflected in the up to 7% lower I/O time. Moreover, FBARC also exhibits acceptable I/O behavior on symmetric storage – HDD.

On real world traces FBARC is also less computationally-intensive (up to 2.5x) due to the employed Grid clustering algorithm. In fact, on the traces used, FBARC exhibited the lowest computational-intensiveness of all clustering strategies. FBARC's lightweight clustering algorithm is based on the observation that on Flash devices random write operations concentrated on a small address region are executed with close to sequential performance. This trend is also valid with the newest generation of devices and will persist.

In addition, ARC and FBARC are the only scan-resistant buffer management strategies among those under test. Such property is especially relevant for Data Warehousing systems where large sequential scans may be followed by large sequential update streams. Moreover sequential patterns are even more common for new storage technologies since new algorithms target write sequentialization.

7. ACKNOWLEDGMENTS

This work was supported by the DFG (Deutsche Forschungsgemeinschaft) project 'Flashy-DB'.

8. REFERENCES

[1] Database test suite dbt2.
[2] Database test suite dbt3.
[3] L. Bouganim, B. T. Jónsson, and P. Bonnet. uflip: Understanding flash io patterns. *CoRR*, abs/0909.1780, 2009.
[4] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.
[5] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09*, pages 181–192, 2009.
[6] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proc. CIDR*, pages 21–31, 2011.
[7] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *J. Syst. Archit.*, 55:332–343, May 2009.
[8] P. J. Denning. Working sets past and present. *IEEE Trans. Softw. Eng.*, 6:64–84, Jan. 1980.
[9] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *Proc. SIGMOD*, pages 1113–1124, 2011.
[10] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, Dec. 1984.
[11] E. Geschwinde and H.-J. Schonig. *Postgresql Developer's Handbook*. Sams, 2001.

[12] B. S. Gill and D. S. Modha. Wow: wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 10–10, 2005.

[13] R. Gottstein, I. Petrov, and A. Buchmann. Append storage in multi-version databases on flash. In *Proc. of BNCOD*, 2013.

[14] S. Jiang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *In USENIX Conference on File and Storage Technologies (FAST*, 2005.

[15] P. Jin, Y. Ou, T. Härder, and Z. Li. Ad-lru: An efficient buffer replacement algorithm for flash-based databases. *Data Knowl. Eng.*, 72:83–102, Feb. 2012.

[16] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. Fab: Flash-aware buffer management policy for portable media players. In *IEEE Transactions on Consumer Electronics, 52*, pages 485– 493. IEEE, 2006.

[17] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha. Lru-wsr: integration of lru and writes sequence reordering for flash memory. volume 54, pages 215 – 1223, Aug. 2008.

[18] H. Jung, K. Yoon, H. Shim, S. Park, S. Kang, and J. Cha. Lirs-wsr: integration of lirs and writes sequence reordering for flash memory. In *ICCSA'07*, pages 224–237. Springer-Verlag, 2007.

[19] W.-H. Kang, S.-W. Lee, and B. Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, 2012.

[20] H. Kim and S. Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *Proc. FAST'08*, pages 16:1–16:14, 2008.

[21] Y. Lv, B. Cui, B. He, and X. Chen. Operation-aware buffer management in flash-based systems. In *SIGMOD '11*, pages 13–24, 2011.

[22] N. Megiddo and D. S. Modha. ARC: a Self-Tuning, low overhead replacement cache. In *Proc FAST*, pages 115–130, 2003.

[23] Y. Ou and T. Härder. Clean first or dirty first?: a cost-aware self-adaptive buffer replacement policy. In *IDEAS '10*, pages 7–14, 2010.

[24] Y. Ou, T. Härder, and P. Jin. Cfdc: a flash-aware buffer management algorithm for database systems. In *ADBIS'10*, pages 435–449, 2010.

[25] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee. Cflru: a replacement algorithm for flash memory. In *CASES '06*, pages 234–241, 2006.

APPENDIX

Fig. 21, 22, 23 and 24 show the accesses within each trace. The darker a point, the more often a page at this point was accessed. Please notice that each point represents several thousand pages.

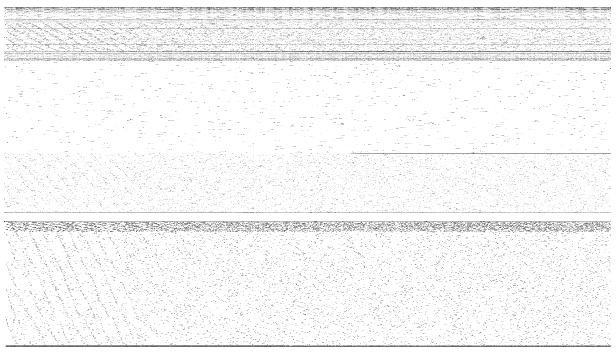
Figure 21: Trace C scatter diagram

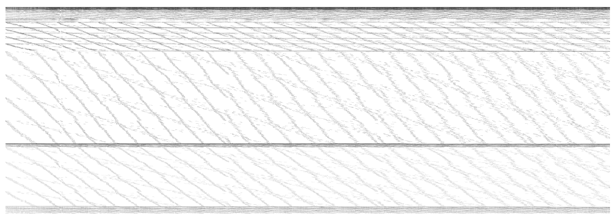
Figure 22: Trace C_d scatter diagram

Figure 23: Trace B scatter diagram

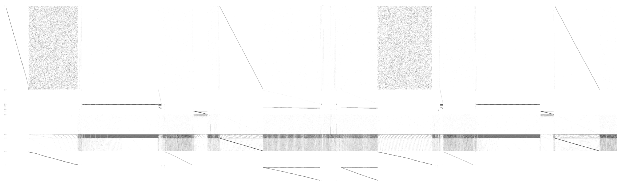
Figure 24: Trace H scatter diagram (some portions left out)