Flash-Aware Buffer Management for Database Systems

Yi Ou, University of Kaiserslautern, Germany
Peiquan Jin, University of Science & Technology of China, China
Theo Härder*, University of Kaiserslautern, Germany

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 flash-based storage devices. We introduce the basic principles of buffer management for such devices and present two efficient buffer algorithms that apply these principles. These algorithms significantly improve the performance of flash-based databases, as confirmed by our trace-driven performance study.

Keywords: buffer; cache; replacement policy; buffer management; database storage; flash SSD; flash memory; flash-based system

1 INTRODUCTION

Flash disks are playing 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 (Graefe, 2007). 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 Device 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 (Woodhouse, 2001; Gal & Toledo, 2005). The FTL implementation is device-related and supplied by the disk manufacturer. Many efforts are made to systematically benchmark the performance of flash disks (Gray & Fitzgerald, 2008; Härder, Schmidt, Ou, & Bächle, 2009; Bouganim, 2009). Using these observations, 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.
- 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. As an example, some flash disks achieve 12,000 IOPS for random reads and only 130 IOPS for random writes of 4 KB blocks (MTRON, 2008), while high-end magnetic disks typically have more than 200 IOPS for random I/O(Gray & Fitzgerald, 2008).

- Due to the employment of device caches and further optimizations in the FTL (Kim, Kim, Noh, Min, & Cho, 2002; Lee, Park, Chung, Lee, Park, & Song, 2007), 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 size*s 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 $Y := (y_0, y_1, \ldots, y_{m-1})$, $m \le n$, 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', $C(A(X, b)) \le C(A'(X, b))$.

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 $(op, pageId)$, 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.

1.3 Basic Principles

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 sufficient. However, a naive increase of the page size generally leads to more unnecessary I/O (using the same buffer size), especially for OLTP workloads, where random accesses dominate. Furthermore, in multi-user environments, large page sizes favor thread contentions. Hence, simply increasing the granularity of I/O is not feasible, therefore, a more sophisticated solution is needed.

Even with flash disks, maintaining a high hit ratio--the primary goal of conventional buffer algorithms--remains important, because memory access is still much faster (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:

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

These principles can not be considered in isolation. For example, to reduce the number of physical writes (P1), one has to delay the replacement of *dirty* pages and choose to first replace *clean* pages having a higher probability of being re-referenced. (*Clean* pages are buffer pages that are not modified since being read from secondary storage, and *dirty* pages are those that are modified at least once in the buffer.) This would negatively impact the hit ratio (P3). A trade-off must be made among these partially conflicting optimization goals.

To improve write efficiency (P2), a technique called *clustered write* is used by our algorithms presented in Section 3. A *cluster* is a set of pages physically located in proximity and having the same cluster number. We derive the cluster number by dividing their page number by a constant *cluster size*. 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 byte 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.

To achieve write avoidance by delaying the replacement of dirty pages, the buffer manager should not be burdened with conflicting write/update propagation requirements. We assume a NoForce/Steal policy for the logging&recovery component providing maximum degrees of freedom (Härder & Reuter, 1983). *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)). 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, replacement of a specific dirty page can be delayed to save physical writes or even advanced, if necessary, to improve the overall I/O 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 not practical anymore, because--given large DB buffer sizes--they would repeatedly imply limited responsiveness of the buffer for quite long periods. While checkpoints are written, which often occurs 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. Today, the method of choice is *fuzzy checkpointing* (Mohan, 1992), 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.

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. Because prefetching always includes the risk of fetching pages later not needed, it is even better for flash-aware buffer algorithms to not use this conventional optimization technique.

1.4 Contributions

This paper extends and improves our previous work (Ou, Härder, & Jin, 2010) and presents two efficient buffer management algorithms for flash-based databases: CFDC (Clean-First Dirty-Clustered) and SAWC (Self Adaptive with Write Clustering). The SAWC algorithm employs a novel and efficient technique: *clustered write with frequency-based filtering*, which distinguishes it from the CASA (Cost-Aware Self-Adaptive) algorithm (Ou & Härder, 2010), from which it is derived.

We study the behavior of our algorithms under different types of workloads and give hints for practical application of these algorithms. The study also confirms the performance advantage of our algorithms over previous proposals.

The remainder of this paper is organized as follows. Section 2 sketches the related work. Section 3 presents our algorithms, while the performance study is reported in Section 4. The concluding remarks are given in Section 5.

2 RELATED WORK

LRU and CLOCK (Corbato, 1969) are among the most widely-used replacement policies. The latter is functionally identical to the Second Chance algorithm (Tanenbaum, 1987): both of them often achieve hit ratios close to those of LRU.

The ARC algorithm (Megiddo, 2003) manages the buffer pool in two lists: one for *recently* referenced pages and the other for *frequently* referenced pages. The sizes of the page lists are dynamically adjusted according to recent workload characteristics. Our SAWC and CASA algorithms take a similar approach to separate the buffer pool in two lists, according to the *clean/dirty* status of buffer pages. However, the list sizes in SAWC and CASA are determined not only by the workload, but also by page-replacement costs.

CFLRU (Park, 2006) 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.

One of the major problems of CFLRU is its tuning parameter w, which is performance critical but difficult to determine. Its optimal value depends on the extent of R/W asymmetry of the storage device and the update intensity of the workload, which may vary over time. Although its authors have mentioned a dynamic version of CFLRU, which automatically adjusts the parameter "based on periodically collected information about flash read and write operations" (Park, 2006), its control logics is not presented. Yoo et al. proposed several variants of CFLRU (Yoo & Lee, 2007), aiming at reducing the number of flash erase operations and improving wear-leveling. These variants have the same limitation as CFLRU and their design goals are different from ours. The CASA algorithm, which will be introduced in Section 3.2, addresses the parameter tuning problem of CFLRU.

LRUWSR (Jung, 2008) 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.

The authors of CCF-LRU (Li & Jin, 2009) further refine the idea of LRUWSR by distinguishing between cold-clean and hot-clean pages. Cold pages are distinguished from hot pages using the Second Chance algorithm. They define four types of eviction costs: cold-clean, cold-dirty, hot-clean, and hot-dirty, with increasing priority, thus cold-clean pages are first considered for eviction, then cold-dirty, and so on. Although LRUWSR and CCF-LRU don't require parameter tuning, their clean-first strategy is carried out only based on the coarse assumption of R/W cost asymmetry and hot-cold detection using the Second Chance algorithm, which, in turn, only approximates LRU. As a consequence, it is difficult for them to reason, when a cold-dirty page should be first considered for eviction over a hot-clean page, and vice versa.

REF (Seo & Shin, 2008) 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.

The CRAW-C algorithm (Park, Lee, Hyun & Bahn 2009), designed for mobile systems which employ compressed file systems, partitions the buffer pool into a read area, write area, and a compressed area considering different costs of read, write, compress, and decompress operations. Each of the three areas is managed in a lightweight fashion similar to the CLOCK algorithm.

Except REF, all previous proposals ignore the spatial locality of page requests and do not address the problem of write patterns (P2). However, REF can not reduce the number of flash writes (P1), because it does not distinguish between clean and dirty states of pages.

3 THE ALGORITHMS

In the following, we present two flash-aware buffer algorithms. Both algorithms try to address the problems ignored by previous proposals, but taking completely different approaches.

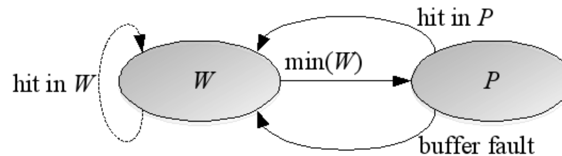

Figure 1 Page movement in the two-region scheme

3.1 The CFDC Algorithm

3.1.1 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 parameter λ, called *priority window*, determines the size ratio of P relative to the total buffer. Therefore, if the buffer has b pages, then P contains $\lambda \cdot b$ 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.

Figure 1 illustrates the page movement in our two-region scheme. The code to be executed upon a 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 no victim page is available from P (e. g., all pages in P are fixed), we select the victim from W (line 9). Considering recency, the newly fetched page is promoted to W (line 15).

```
Algorithm 1: CFDC
   input  : request for page p
   output : the requested page p
   init.  : buffer pool B with b pages, working region W and priority region P,
            |W| = (1 − λ) · b ∧ |P| = λ · b
1  if p ∈ B then
2      if p ∈ 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         physically write q;
12     clear q and read content of p from external storage into q;
13     p ← q;
14     if p ∈ P then
15         demote min(W), promote p;
16 return p;
```

3.1.2 Priority Region

Priority region P maintains three structures: an LRU list L of clean *pages*, a priority queue Q of *clusters* where dirty pages are accommodated, and a hash table with cluster numbers as keys for efficient cluster lookup.

```
Function SelectVictimPr
   input  : priority region P, consisting of a list of clean pages L in LRU order
            and a priority queue of dirty-page clusters Q
   output : return a page v as replacement victim
1  if L ≠ ∅ then
2      v ← the LRU page of L;
3  if v = null then
4      cluster c ← cluster of lowest priority in Q ;
5      if c ≠ null then
6          v ← the LRU page in c;
7          if  v ≠ null then
8              c.ipd ← 0 ;
9  return v;
```

The victim selection logic in P is shown in Function *SelectVictimPr*. Clean pages are always selected over dirty pages (line 1--2). If there is no clean page available, a cluster c having the lowest priority is selected from Q and the LRU page in c is selected as victim (line 3--6). Once a victim is selected from a cluster, its priority is set to minimum (line 8) until all dirty pages in this *victim cluster* are consumed by subsequent page evictions, 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):

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

where $p_0, ..., p_{n-1}$ are the page numbers ordered by their time of entering the cluster. The algorithm tends to assign large clusters a lower priority for two reasons: 1. Flash disks are efficient in writing such clustered pages. 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.

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, because 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.

The time complexity of CFDC 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.

3.2 The SAWC Algorithm

3.2.1 CASA Overview

In the following, we first give an overview of the CASA algorithm, which the SAWC algorithm is based on. The CASA algorithm uses the notion of *cost ratio* to refer to the

extent of R/W asymmetry of the underlying storage device, defined as the ratio of the long-term cost of physical reads to that of physical writes.

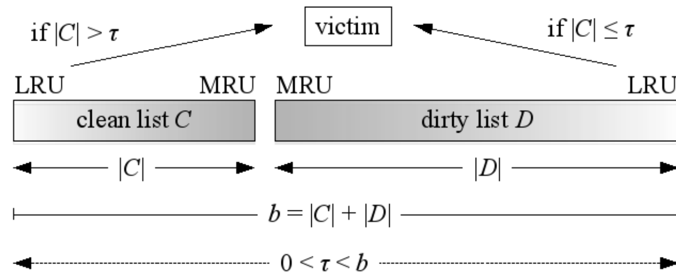

Figure 2 CASA dynamically adjusts the size of the clean list and the dirty list

CASA manages the buffer pool B of b pages using two dynamic lists: the *clean* list C for keeping *clean* pages that are not modified since being read from secondary storage, and the *dirty* list D accommodating *dirty* pages that are modified at least once in the buffer. Pages in either list are ordered by reference recency. Both lists are initially empty while, in the stable state (no empty buffer frames available), we have the following invariants: $|C|+|D|=b, 0\leq|C|\leq b, 0\leq|D|\leq b$, as illustrated in Figure 2.

The algorithm continuously adjusts parameter τ, which is the dynamic target size of C, $0\leq\tau\leq b$. Therefore, the dynamic target size of D is $b-\tau$. The logic of adjusting τ is simple: we invest in the list that is more cost-effective for the current workload. If there is a page hit in C, we heuristically model the current *relative cost effectiveness* of C with $|D|\div|C|$. Similarly, in case of a page hit in D, we model its current relative cost effectiveness with $|C|\div|D|$. The relative cost effectiveness is considered when determining the magnitude of adjustment for τ.

Besides the page request, the algorithm (see Algorithm 2) requires as input also the normalized costs c_R and c_W, which can be derived from the cost ratio such that $(c_R+c_W=1)\wedge(c_R\div c_W=\text{cost ratio})$, i. e., important to the algorithm is the extent of the R/W asymmetry, not the exact costs of physical reads and writes.

The magnitude of the adjustment in τ is determined both by the cost ratio and by the relative cost effectiveness. The adjustment is performed in two cases:

Case 1 A logical R-request is served in C (line 3 of Algorithm 2);

Case 2 A logical W-request is served in D (line 8 of Algorithm 2)

In Case 1, τ is increased by $c_R\times(|D|\div|C|)$. Note $|C|\neq 0$, because it is a buffer hit. The increment combines the "saved cost" of this buffer hit c_R and the relative cost effectiveness $|D|\div|C|$. Similarly, in Case 2, we decrease τ by $c_W\times(|C|\div|D|)$.

```
Algorithm 2: CASA
input  : request for page p in the form (op, pageId), where op = R or op = W;
         normalized costs c_R and c_W, with c_R + c_W = 1 ∧ c_R ÷ c_W = cost ratio
output : the requested page p
init.  : buffer pool B with capacity b; list E of empty pages, |E| = b; lists C
         and D, |C| = 0, |D| = 0; τ ∈ ℛ, τ ← 0
1  if p ∈ B then
2      if p ∈ C ∧ op = R then
3          τ ← min(τ + c_R × (|D| ÷ |C|), b) ;
4          move p to MRU position of C ;
5      else if p ∈ C ∧ op = W then
6          move p to MRU position of D ;
7      else if p ∈ D ∧ op = W then
8          τ ← max(τ − c_W × (|C| ÷ |D|), 0) ;
9          move p to MRU position of D ;
10     else
           // p ∈ D ∧ op = R
11         move p to MRU position of D ;
12 else
13     victim page v ← null ;
14     if |E| > 0 then
15         v ← remove tail of E ;
16     else if |C| > τ then
17         v ← LRU page of C ;
18     else
19         v ← LRU page of D ;
20         physically write v;
21     physically read p into v ;
22     p ← v ;
23     if op = R then
24         move p to MRU position of C ;
25     else
26         move p to MRU position of D ;
27 return p;
```

In case of a buffer fault (line 12--26), if there is no empty page available, τ guides the decision, from which list to select the victim page (line 16 and 19). The actual sizes of both lists are also influenced by the clean/dirty state of requested pages. The clean/dirty state of a requested page p is decided by its previous state in the buffer, i. e., in which list it resides, and the current request type (R or W). If the state of the requested page p is clean (after serving the request), p will be moved to C (line 4 and 24), otherwise to D (line 6, 9, 11, and 26). Therefore, the sizes of C and D are dynamically determined by τ, and the update intensity. Under a workload with mixed R-requests and W-requests, a starvation of one list will never happen, even when $(\tau = 0) \vee (\tau = b)$, while under R-only (or W-only) workloads, a starvation of D (or C) is desired and the starved list recovers as soon as the workload becomes mixed again.

3.2.2 Dynamic Cost-Ratio Detection

Being fundamentally different from algorithms which require parameter tuning, CASA automatically optimizes itself at runtime, given the knowledge concerning cost ratios, which can be provided, e. g., by the device manufacturer or by the administrator. It would be even better if, in the future, devices provide an interface for querying the cost ratio online.

In fact, the elapsed time serving each physical I/O request can be measured online. Therefore, it can be used to derive the cost ratio information. However, these measurements are subject to severe fluctuations. For example, the latency of a physical read on magnetic disks depends on the position of the disk arm. On flash SSDs, a physical write may trigger a much more expensive flash erase operation or even a garbage collection process involving multiple flash erase operations (Bouganim, 2009). Therefore, an n -point moving average of the measured values is used to smooth out short-term fluctuations, because only the long-term average cost is of interest. Hence, the average cost of the last n physical reads (or writes) is used as the basis for the normalized cost c_R (or c_W) required by Algorithm 2. Note, no change to the algorithm is needed to use the dynamically detected costs.

Maintaining the moving average requires the recent n measurements to be remembered. Assuming that two bytes are used to store a measured value, to remember, e. g., 32,768 values, eight pages (page size = 8 KB) are sufficient and, in total, only 16 pages are required for both reads and writes. Two bytes can store timings ranging from 1 to 65,536 μseconds, which can cover the expected physical I/O cost values. Furthermore, burst values out of this range can be safely ignored. The time complexity of maintaining the moving average is independent of n : in our implementation, each new measurement requires only an array element update and a few arithmetic operations.

3.2.3 Integrating Clustered Writes

Having addressed principles P1 and P3, CASA does not optimize its write patterns (P3). This can be improved with the clustered-write technique described as follows. A cluster table T is maintained, which keeps track of the dirty pages, and, for a given page p, returns the set of dirty pages that are in the same cluster of p. To integrate the write clustering technique with CASA, we only have to replace line 20 of Algorithm 2 by Function *ClusteredWrite*. We call the resulting algorithm SAWC.

For a physical-write request with regard to page p, Function *ClusteredWrite* not only writes page p, but also identifies and writes some pages in the same cluster of p. This improves the spatial locality of the flash write requests, but, on the other hand, it is obvious that, for the same workload, SAWC requires a larger number of physical writes than CASA does. For this reason, we filter the candidate pages based on their *update frequency* and skip the dirty pages that are more frequently updated than p. Function *Freq* gives, for page p, its update frequency, which is the number of logical write requests for p while p is in the buffer. This can be simply implemented with a counter associated with each buffer page.

The intuition behind the *frequency-based filtering* is, if page q is more frequently updated than p, flushing q together with p will have no performance gain, because q will likely be updated soon again. In contrast, pages that are less frequently updated will not likely be updated again, flushing them together with p improves the performance due to higher spatial locality.

```
Function ClusteredWrite
  input  : physical-write request for page p
  output : page p physically written, together with the equal or less frequently
           updated pages in its cluster
1 physically write p ;
2 cluster c ← lookup p's cluster in T ;
3 if c exists then
4   foreach q ∈ c ∧ q ≠ p do
5     if Freq(q) ≤ Freq(p) then
6       physically write q ;
7       remove q from c ;
8   if |c| = 0 then
9     remove c from T ;
```

Unlike the priority queue Q of CFDC, table T of SAWC does not maintain an ordering of the clusters, therefore, its maintenance requires only a constant number of operations. The time complexity of SAWC is $O(1)$ and it requires minimal auxiliary data structures. SAWC only uses knowledge local to the cluster to determine the write candidates, as opposed to the cluster selection of CFDC based on the global ordering of clusters. Therefore, CFDC can potentially utilize the buffer more efficiently.

3.3 Implementation Issues

Algorithm 4 requires a request in the form of $(op, pageId)$, i. e., the request type must be present. This may not be the case in some systems, where a page is first requested without explicitly claiming the request type, and it is read or updated some time later. However, most DBMSs use the classical pin-use-unpin (or fix-use-unfix) protocol (Effelsberg & Härder, 1984) (Gray & Reuter, 1993) for pages requests. It is easy to use an update flag, which is cleared upon the pin call and set by the actual page update operation. Upon the unpin call, the buffer manager knows the request type by checking this flag.

In practice, page flushes are normally not coupled with the victim replacement process--most of them are performed by background threads. For better clarity, the presented algorithms do not include the asynchronous page-flushing logic, although they are compatible with this technique. For example, line 2 to line 9 of Function *ClusteredWrite* can be executed in an asynchronous fashion, while in the case of CFDC, the background routines can directly use CFDC's dirty queue, where the dirty pages are already collected and clustered.

4 PERFORMANCE STUDY

Our previous work (Ou, Härder, & Jin, 2010) included an extensive empirical research comparing CFDC with related algorithms including CFLRU, LRUWSR, REF, and LRU. CFDC exhibits significant performance advantage in those experiments. In (Ou & Härder, 2010), we compared the CASA algorithm with CFLRU, LRUWSR, CCFLRU, and LRU, demonstrating its adaptivity. To extend our previous works and avoid repetition, we focus on the comparison among the three algorithms in this paper: CFDC, CASA, and SAWC, using buffer traces of standard benchmarks including TPC-C, TPC-H, and TPC-E. For completeness, we also report our experiments involving a larger set of algorithms using a real-life OLTP trace.

The TPC-C and TPC-H trace was obtained using the PostgreSQL DBMS. Our specific code integrated into its buffer manager recorded the logical page requests received under TPC-C and TPC-H workloads. The TPC-E trace is provided by a leading IT enterprise. The real-life trace is a one-hour page reference trace of an OLTP production system of a Bank. This trace is well-studied and has been used in (O'Neil, 1993; Johnson & Shasha, 1994; Lee & Choi, 2001; Megiddo & Modha, 2003; Jiang & Zhang, 2002). It contains 607,390 references to 8-KB pages in a DB having a size of 22 GB, addressing 51,880 distinct page numbers. About 23% of the requests update the page referenced. Exhibiting substantial locality, 20% (10,376) of the pages are referenced by 72% (434,702) of the requests. The common page size 8 KB is used throughout of this section.

4.1 Test Setup

Our 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. Both OS and the storage engine are installed on the magnetic disk. The data pages being read and updated resides on the flash disk which is a 32 GB MTRON MSP-SATA7525 based on NAND flash memory. The flash disk is accessed by the file manager as RAW devices, therefore, OS and file system influences are eliminated.

The clustering technique improves the spatial locality of page flushes, which can be quantified by the metric *cluster-switch count (CSC)*, defined as follows. Let $S := \{q_0, q_1, \ldots, q_{m-1}\}$ be the sequence of page flushes, the metric $CSC(S)$ reflects the spatial locality of S:

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

The *clustered writes* of CFDC and SAWC are write patterns with high spatial locality and thus minimized cluster-switch counts. Sequential writes are a special case of clustered writes, where pages are updated 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 S and S is a sequential access pattern, we have $CSC(S) = |d(S)|$. For flash devices, a

higher $CSC(S)$ indicates a higher number of expensive flash block erasures required to serve S.

Compared to clustered writes, 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, which 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*.

Let $R := \{p_0, p_1, \ldots, p_{n-1}\}$ be the sequence of logical page requests fed to the buffer manager, we further define the metric *cluster-switch factor* (CSF) to reflect its efficiency in performing clustering:

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

Following the typical layered architecture for database systems (Härder & Reuter, 1983; Härder, 2005), we implemented a database storage engine consisting of two layers: the *file manager* supporting page-oriented access to the data files, and the *buffer manager* serving page requests. The examined algorithms are used by the buffer manager via a common interface. A test program communicates with the buffer manager by sending the logical page requests delivered by the test input: the buffer traces.

We run a trace by sending its page requests one by one to the buffer manager and record the execution time elapsed between two points in time: the start of the first and the end of the last logical request. We also record the number of physical writes (corresponding to the number of flash writes), the CSC (correlated to the number of flash erases), and the hit ratio, for the elapsed period of time.

4.2 TPC-C Workload

Figure 3 shows the results of running the TPC-C trace with buffer size scaled from 500 to 16000 pages logarithmically. To examine the impact of the parameter λ, we did the experiments for CFDC with $\lambda = 0.25$, $\lambda = 0.50$, and $\lambda = 0.75$, denoted as CFDC-25, CFDC-50, and CFDC-75. Compared with CASA, SAWC required a moderately higher number of physical writes (Figure 3b). However, as shown in Figure 3c, it has a much lower CSC, which indicates a much stronger spatial locality of the physical write requests generated by SAWC. The clustered write greatly improved the performance of CASA: with a reduction of execution time up to 25% (for the 16000-page setting in Figure 3a). In terms of hit ratio, CASA and SAWC are very close (Figure 3d).

Both CASA and SAWC had a higher hit ratio than CFDC. However, CFDC required a smaller number of physical writes due to its priority-based optimization, and consequently, a lower number of cluster switches. Generally speaking, SAWC achieved a performance comparable to CFDC (Figure 3a): the former is faster for smaller buffer sizes, while the latter is faster for larger buffer pools. The workload contains a substantial percentage of update requests (18.7%). Therefore, among the three parameter settings of CFDC, the $\lambda = 0.75$ setting achieved the best performance, due to the greediest write optimization.

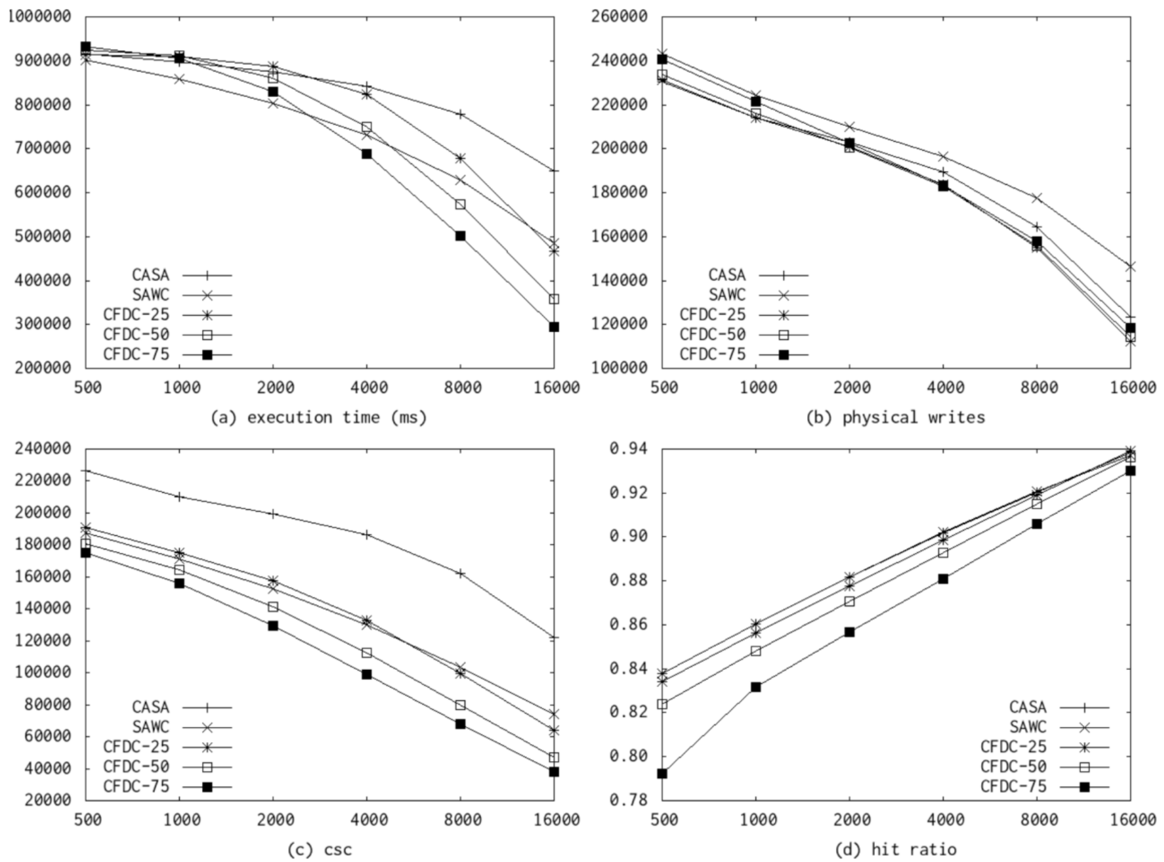

Figure 3 Performance figures of the TPC-C trace

4.3 Read-Intensive Workloads

In contrast to the TPC-C trace, the workload represented by the TPC-E trace is highly read-intensive: it has only 17 k logical writes out of 2.6 million page requests (update percentage is less than one percent). As a consequence, the performance impact of the physical-write count and CSC is ignorable, and the hit ratio (Figure 4b) dominates the execution time (Figure 4a). As opposed to the TPC-C case, where $\lambda = 0.75$ is the best-performing setting of CFDC, for the TPC-E trace, the setting of $\lambda = 0.25$ had achieved the best performance. However, $\lambda = 0.25$ is still suboptimal for this read-intensive workload, because, for such workloads, the optimal window size should be close to zero. The advantage of the adaptivity of CASA and SAWC is clear here: they achieve a higher hit ratio than CFDC, without any parameter tuning.

As an extreme case, we show the results of running a TPC-H trace in Figure 4c and Figure 4d. The TPC-H trace is read-only, therefore, all the tested algorithms behaves like LRU and have the same hit ratio.

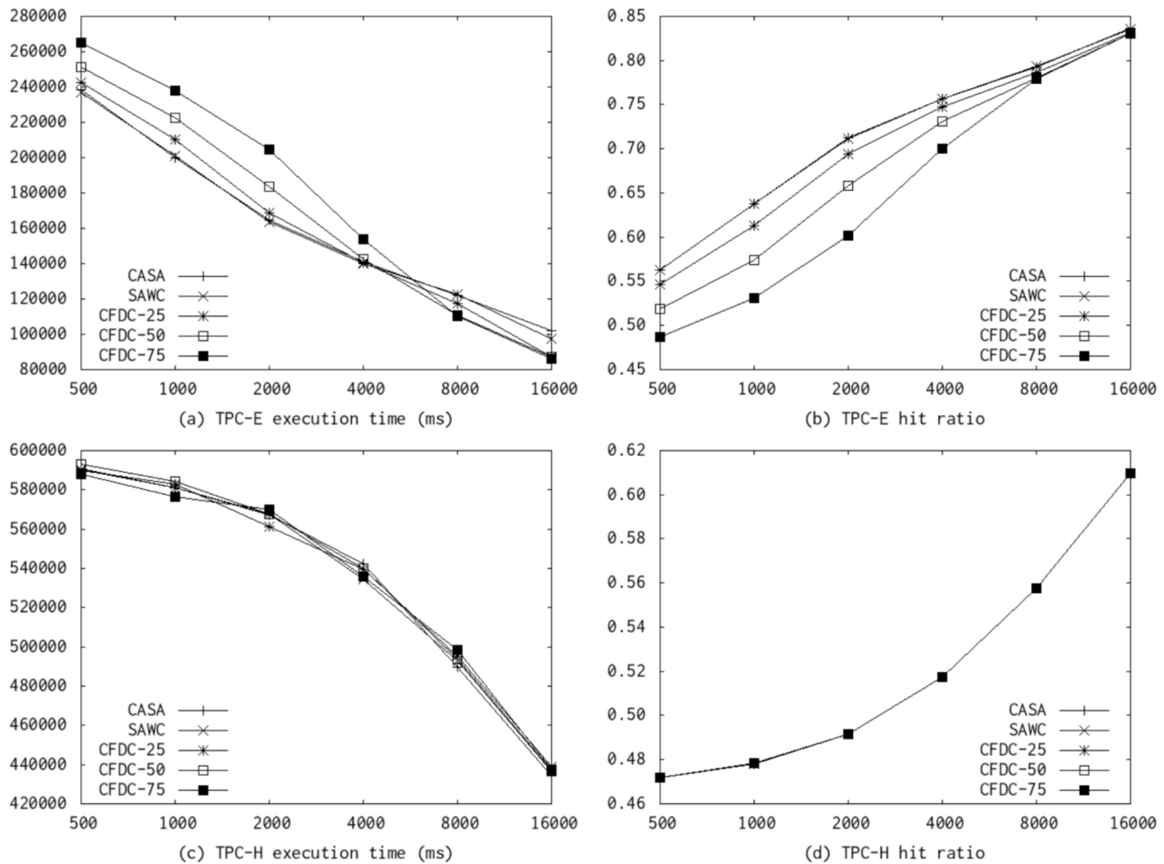

Figure 4 Performance under read-intensive and read-only workloads

4.4 Real-Life Trace

Figure 5 compares various related algorithms running the real-life OLTP trace with a buffer of 1000 pages. All figures are normalized and relative to LRU. The experiment demonstrates the advantages of CFDC: with the clean-first strategy and the optimization in the priority region, it minimizes the number of physical writes (Figure 5b). 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 (Figure 5c). Compared with CASA, the SAWC algorithm has a higher number of physical writes, and a slightly lower hit ratio, but better runtime performance due to the clustered writes.

As a final note, the performance study does not focus on the MTRON flash disk. We also ran all the experiments on a low-end flash disk (SuperTalent, 2008) with similar observations.

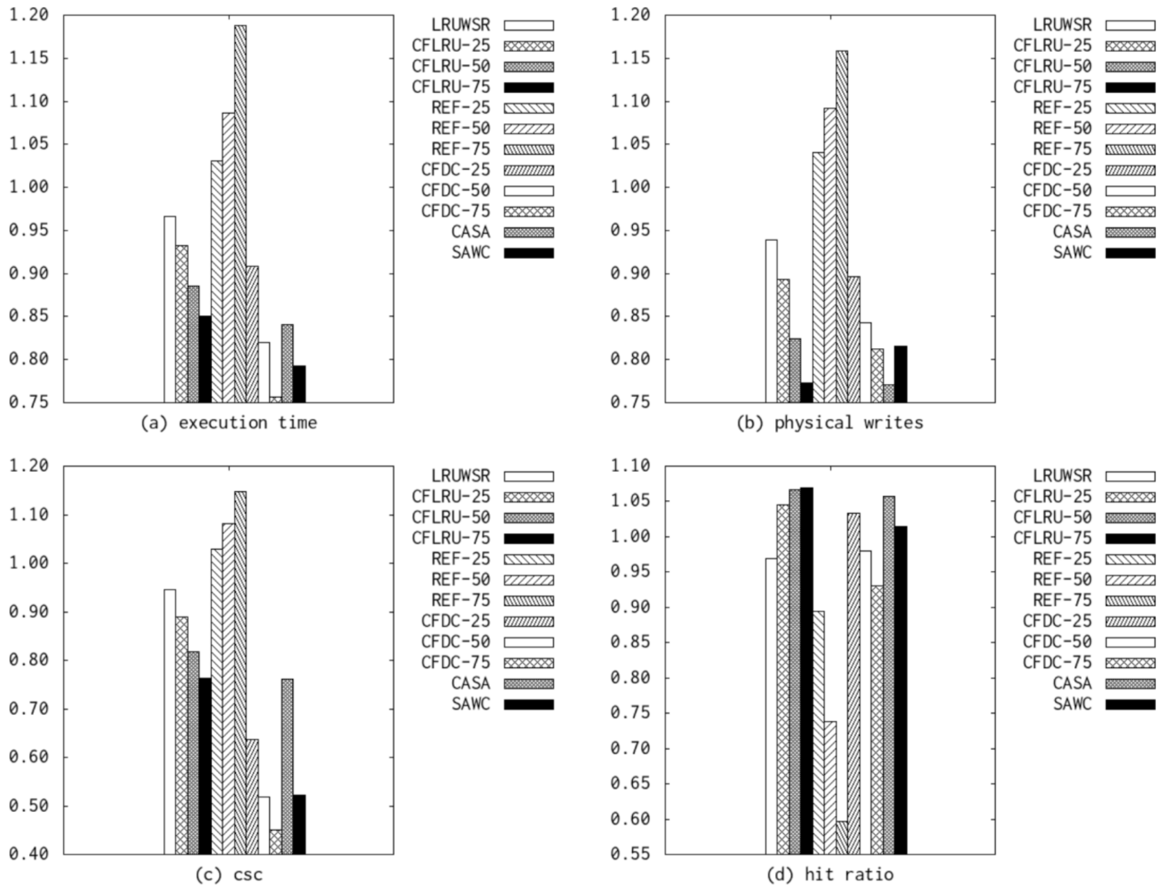

Figure 5 Performance figures relative to LRU under the real-life workload

5 CONCLUSIONS AND OUTLOOK

Due to the increasing popularity of flash-based storage devices, the problem of buffer management for database systems based on flash devices deserve adequate attention. We introduced the basic principles of approaching this problem and presented two flash-aware buffer algorithms, proven to be efficient in our performance study covering a variety of workload types.

The CFDC algorithm is suitable for environments where the workload is relatively write intensive and its characteristics do not change frequently. In such cases, the administrator can better optimize the buffer performance by tuning the parameter λ. For environments with frequently changing workloads, the self-adaptive algorithm SAWC is a better choice. The two-region scheme makes it easy to integrate CFDC with conventional replacement policies in existing systems. Modern replacement policies such as ARC (Megiddo & Modha, 2003) and LIRS (Jiang & Zhang, 2002) can be easily integrated into CFDC without modification.

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.

REFERENCES

Bouganim, L. (2009). uFLIP: understanding flash IO patterns. In *CIDR'09*.

Corbato, F. J. (1969). A paging experiment with the Multics system. In *In Honor of Philip M. Morse* (p. 217). Cambridge, MIT Press.

Effelsberg, W., & Härder, T. (1984). Principles of database buffer management. *ACM TODS , 9*, 560-595.

Gal, E., & Toledo, S. (2005). Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR) , 37*, 138--163.

Graefe, Goetz (2007). The five-minute rule twenty years later and how flash memory changes the rules, *DaMoN'07*, ACM.

Gray, J., & Fitzgerald, B. (2008). Flash disk opportunity for server applications. *ACM Queue , 6*, 18-23,.

Gray, J., & Reuter, A. (1993). *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

Härder, T. (2005). DBMS architecture - the layer model and its evolution. *Datenbank-Spektrum , 13*, 45--57.

Härder, T., & Reuter, A. (1983). *Concepts for Implementing a Centralized Database Management System.* (pp. 28--61). Teubner-Verlag.

Härder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys , 15*, 287-317.

Härder, T., Schmidt, K., Ou, Y., & Bächle, S. (2009). Towards flash disk use in databases - keeping performance while saving energy? In *BTW'09*, pp. 167--186. GI.

Jiang, S., & Zhang, X. (2002). LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance., *Performance Evaluation Review,* 30(1), 31--42.

Johnson, T., & Shasha, D. (1994). 2Q: a low overhead high performance buffer management replacement algorithm. In *VLDB'94* (pp. 439--450).

Jung, H. (2008). LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *Trans. on Cons. Electr. , 54*, 1215--1223.

Kim, J., Kim, J. M., Noh, S. H., Min, S. L., & Cho, Y. (2002). A space-efficient flash translation layer for CompactFlash systems. *Trans. on Cons. Electr. , 48*, 366--375.

Lee, D., & Choi, J. (2001). LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *Trans. on Computers , 50*, 1352--1361.

Lee, S. W., Park, D. J., Chung, T. S., Lee, D. H., Park, S., & Song, H. J. (2007). A log

buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. on Embedded Computing Systems , 6*.

Li, Z., & Jin, P. (2009). CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory. *Trans. on Cons. Electr. , 55*, 1351-1359.

Megiddo, N., & Modha, D. (2003). ARC: A self-tuning, low overhead replacement cache. In *USENIX FAST'03* (pp. 115-130).

Mohan, C. (1992). ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. , 17*, 94--162.

MTRON. (2008). *Solid State Drive MSP-SATA7525 Product Specification*.

O'Neil, E. J. (1993). The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD'93* (pp. 297--306).

Ou, Y., & Härder, T. (2010). Clean first or dirty first? a cost-aware self-adaptive buffer replacement policy. In *IDEAS'10*.

Ou, Y., Härder, T., & Jin, P. (2010). CFDC: a flash-aware buffer management algorithm for database systems. In *ADBIS'10*, *LNCS 6295*, pp. 435-449. Springer.

Park, S. (2006). CFLRU: a replacement algorithm for flash memory. In *CASES'06* (pp. 234--241).

J. Park, H. Lee, S. Hyun, and H. Bahn (2009). A cost-aware page replacement algorithm for NAND flash based mobile embedded systems, In *International Conference on Embedded Software (EMSOFT'09).*

Seo, D., & Shin, D. (2008). Recently-evicted-first buffer replacement policy for flash storage devices. *Trans. on Cons. Electr. , 54*, 1228--1235.

SuperTalent. (2008). *Solid State Drive FSD32GC35M Product Specification*.

Tanenbaum, A. S. (1987). *Operating Systems, Design and Impl.* Prentice-Hall.

Woodhouse, D. (2001). JFFS: the journalling flash file system. In *The Ottawa Linux Symposium.*

Yoo, Y. S., & Lee, H. (2007). Page replacement algorithms for NAND flash memory storages. In *Computational Science and its Applications (ICCSA'07)* (pp. 201--212). Springer.