Clean First or Dirty First? A Cost-Aware Self-Adaptive Buffer Replacement Policy

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

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

ABSTRACT

Flash SSDs originate a disruptive change concerning storage technology and become a competitor for conventional magnetic disks in the area of persistent database stores. Compared to them, they provide a dramatic speed-up for random reads, but exhibit a distinct read/write (R/W) asymmetry, i.e., updates are more expensive than reads. Existing buffer management algorithms for those devices usually trade physical reads for physical writes to some extent. But they ignore the actual R/W cost ratio of the underlying device and the update intensity of the workload. Therefore, their performance advantage is sensitive to device and workload changes. We propose CASA (Cost-Aware Self-Adaptive), a novel buffer replacement policy, which makes the trade-off between physical reads and physical writes in a controlled fashion, depending on the R/W cost ratio, and automatically adapts itself to changing update intensities in workloads. Our experiments show that CASA outperforms previous proposals in a variety of cost settings and under changing workloads.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Main memory, Storage hierarchies; D.4.4 [**Communications Management**]: Buffering, Input/output; H.2 [**Database Management**]: Miscellaneous

General Terms

Algorithms, Experimentation, Performance

Keywords

Replacement Policy, Cache, Database Storage, Flash SSD

1. INTRODUCTION

Flash SSDs (flash-memory-based solid-state drives) 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. Reading a page from a flash SSD is extremely fast, while a page update can be one or two orders of magnitude slower. For this reason, existing buffer management algorithms for flash-based systems, or flash-aware buffer algorithms, usually trade physical reads for physical writes to some extent, in order to improve the overall I/O performance.

Classical buffer algorithms, e.g., LRU, assume that a physical read has about the same cost as a physical write and, therefore, do not apply differing decision policies when read-only or modified pages are replaced in the buffer [1]. This assumption is reasonable for conventional magnetic disks, but not valid for flash-based devices. On the other hand, existing flash-aware algorithms are not applicable for magnetic disks, because their saving in physical writes usually comes at the expense of a higher number of physical reads or lower hit ratios. This observation reveals an important problem, because magnetic disks are expected to co-exist with flash SSDs for a long period of time[1]. In principle, the buffer layer should be aware of the characteristics of the underlying storage devices and adapt itself to changes in such characteristics automatically.

Most of the existing flash-aware buffer algorithms assume that a physical write is *much more* expensive than a physical read. However, in fact, the extent of flash R/W asymmetry is varying from device to device, even among the devices from the same manufacturer. For example, an Intel X25-V SSD achieves 25 K IOPS for reads and 2.5 K IOPS for writes [2], while an Intel X25-M SSD reaches 35 K IOPS for reads and 8.6 K IOPS for writes [3]. We use the term *R/W cost ratio*, or *cost ratio* for short, defined as the ratio between the long-term average time used by physical reads and the same used by physical writes, to express the R/W asymmetry of storage devices, i.e., for a storage device, it tells *how much more* expensive is a write compared with a read, and vice versa. For example, the cost ratios of the above mentioned devices are 1 : 10 and 1 : 4, respectively. Cost ratios deliver important information for making trade-offs between physical reads and physical writes, but they are ignored by existing flash-aware algorithms, i.e., these algorithms are *not aware of the cost.*

As another common problem of existing flash-aware buffer algorithms, they ignore the possibly changing update intensity, i.e., the percentage of write requests, in the workload

[1]Due to the lower $/GB cost of magnetic disks, their dominant market position is not likely to be taken over by flash SSDs in the near future.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IDEAS10 2010, August 16-18, Montreal, QC [Canada]
Editor: Bipin C. DESAI
Copyright ©2010 ACM 978-1-60558-900-8/10/08 $10.00

while making the replacement decision. Some of them leave a parameter for the user to make such an important but difficult performance tuning.

The major contribution of this paper is the design and evaluation of CASA, a cost-aware self-adaptive buffer management algorithm, which addresses the above mentioned problems. The remainder of this paper is organized as follows. Section 2 describes the problem formally. Section 3 sketches the related work. Section 4 introduces our algorithm, while its experimental results are presented in Section 5. The concluding remarks are given in Section 6.

2. THE PROBLEM

We focus on buffer management algorithms for homogeneous storage devices with *possibly* asymmetric R/W costs. Situations where heterogeneous storage devices (e.g., flash SSDs and magnetic disks) co-exist in a *single* storage subsystem, are more related to the data-allocation problem and, thus, are not our focus.

If we denote the sequence of n logical I/O requests $(x_0, x_1, \ldots, x_{n-1})$ as X, a buffer management algorithm A is a function that maps X and a buffer with b pages into a sequence of m physical I/O requests $Y := (y_0, y_1, \ldots, y_{m-1})$, $m \leq n$, i.e., $A(X, b) = Y$.

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

For magnetic disks, $C(Y)$ is often assumed to be linear to $|Y|$. Clearly, this assumption does not hold for flash SSDs, because $C(Y)$ heavily depends on the cost ratio of read and write requests and their percentages in 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). We denote the cost of physically serving an R-request as c_R, and that of physically serving a W-request as c_W. Depending on the characteristics of the underlying storage device, c_R may or may not be equal to c_W. The term *cost* refers to time cost instead of monetary cost, since a cost-optimal algorithm should minimize the accumulated *time* for serving I/O requests.

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 the overall I/O cost. Hit ratios are related to the overall I/O cost, because the number of buffer faults is equal to the number of physical reads, whose cost can not be ignored compared with the cost of main memory access, even with fast flash SSDs. But they are not the only concern according to our model—saving physical writes could be more cost-effective in some cases. We assume that the cost ratio is known, e.g., it may be derived from IOPS figures or average response times. In Section 5.3, we demonstrate an efficient technique that can detect the cost ratios online.

Prefetching of pages plays an important role for conventional disk-based buffer management. However, prefetching always includes the risk of fetching pages later not needed. Furthermore, with flash SSDs, prefetching becomes much less important, because pages can be randomly fetched on demand without (hardly) any penalty in the form of access latency. Therefore, we focus on the core logic of buffer management, i.e., the replacement policy.

3. RELATED WORK

LRU is one of the most widely-used replacement policies. CFLRU [4] 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. The remaining part of its list structure is called *working region*. 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 *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.

The CFDC algorithm [5] also addresses the R/W asymmetry of flash devices by managing the buffer pool in two regions: the *working region* and *priority region*. In the priority region, dirty pages are kept in the buffer longer than clean pages. Furthermore, they are clustered by logical block addresses to enforce higher spatial locality (thus higher efficiency) of page flushing.

LRUWSR [6] is a flash-aware algorithm based on LRU and Second Chance [7], 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 searching from the LRU end of the list. If a clean page is found, it will be returned immediately (LRU and clean-first strategy). If a dirty page marked as "cold" is found, it will also be returned; otherwise, it will be marked "cold" (Second Chance), moved to the MRU (most-recently used) end of the list, and the search continues.

The authors of CCF-LRU [8] 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.

The major problem of CFLRU is its tuning parameter w, which is performance-critical but difficult to determine. Its optimal value depends on the cost ratio 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" [4], its control logics is not presented. Yoo et al. proposed several variants of CFLRU [9], 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.

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 should a cold-dirty page be first considered for eviction over a hot-clean page, and vice versa.

4. THE CASA ALGORITHM

4.1 Overview

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 pages[2] available), we have the following invariants: $|C|+|D| = b, 0 \leq |C| \leq b, 0 \leq |D| \leq b$, as illustrated in Figure 1.

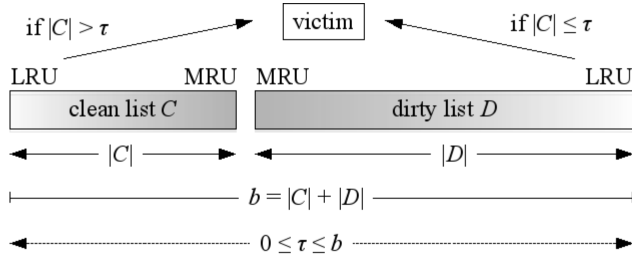


Figure 1: CASA dynamically adjusts the size of the clean list and the dirty list

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 (see Section 4.2). The simple heuristics used here has low overhead and is effective, as shown by our experiments in Section 5.

4.2 The Algorithm

Besides the page request, the algorithm (see Algorithm 1) requires as input also the normalized costs c_R and c_W such that $c_R + c_W = 1 \wedge c_R \div c_W =$ cost ratio. They can be derived from the 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 1);

Case 2 A logical W-request is served in D (line 8 of Algorithm 1)

In Case 1, we increase τ by $c_R \times (|D| \div |C|)$. Note $|C| \neq 0$, since 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 $c_W \times (|C| \div |D|)$.

[2]Empty page refers to the buffer area for a page that has not been used since the start of the buffer manager. Following the convention in the literature, we avoid using the term *buffer frame* (data structure holding a page).

Algorithm 1: CASA

input : request for page p in the form $(op, pageId)$, where $op = R$ or $op = W$; normalized costs c_R and c_W such that $c_R + c_W = 1 \wedge c_R \div c_W =$ 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, $|C| = 0, |D| = 0$; $\tau \in \mathcal{R}$, $\tau \leftarrow 0$

1 **if** $p \in B$ **then**
2   **if** $p \in C \wedge op = R$ **then**
3     $\tau \leftarrow min(\tau + c_R \times (|D| \div |C|), b)$;
4   move p to MRU position of C ;
5   **else if** $p \in C \wedge op = W$ **then**
6   move p to MRU position of D ;
7   **else if** $p \in D \wedge op = W$ **then**
8     $\tau \leftarrow max(\tau - c_W \times (|C| \div |D|), 0)$;
9   move p to MRU position of D ;
10   **else**
    // $p \in D \wedge op = R$
11   move p to MRU position of D ;
12 **else**
13   victim page $v \leftarrow null$;
14   **if** $|E| > 0$ **then**
15     $v \leftarrow$ remove tail of E ;
16   **else if** $|C| > \tau$ **then**
17     $v \leftarrow$ LRU page of C ;
18   **else**
19     $v \leftarrow$ LRU page of D ;
20   physically write v;
21   physically read p into v ;
22   $p \leftarrow v$;
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.

4.3 Implementation Issues

Algorithm 1 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 [10] 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.

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 [11]. NoForce means that pages modified by a transaction do not have to be forced to disk at its commit, but only the redo logs. Steal means that modified pages can be replaced and their contents can be written to disk even when the modifying transaction has not yet committed, provided that the undo logs are written in advance (observing the WAL principle (write ahead log)). 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[3]. Today, the method of choice is *fuzzy checkpointing* [12], where only metadata describing the checkpoint is written to the log, but displacement of modified pages is obtained via asynchronous I/O actions not linked to any specific point in time.

As a final remark: the time complexity of our algorithm is $O(1)$ and it requires minimal auxiliary data structures. As our experiments will show, it is also very efficient.

5. EXPERIMENTS

We implemented the bottom-most two layers of a DBMS: the *file manager* supporting page-oriented access to the data files, and the *buffer manager* serving page requests, which uses the related algorithms via a common interface. Five algorithms were included in our experiments: CASA, LRU (representing conventional buffer algorithms), and the flash-aware algorithms represented by CFLRU, CCFLRU, and LRUWSR. We did not include the CFLRU variants [9] and CFDC [5], due to two reasons: 1. All of them require the tuning of a parameter controlling the sizes of two buffer regions—the same limitations represented by CFLRU; 2. The page-clustering technique of CFDC is complementary to our approach, because page flushing is orthogonal to replacement policies and any clustering technique could be implemented in the page-flushing procedure for all replacement policies.

To explore the behavior of the buffer manager, traces, i.e., page reference strings (as a history recording of a buffer manager's work) are used. All experiments were started with a cold buffer.

5.1 Changing Workload

To examine the behavior of the algorithms under changing workloads, we used a tailor-made trace, called CONCAT. It was built from an OLTP trace and a DSS trace, both of which were gained using specific code integrated into the buffer manager of the PostgreSQL DBMS. The OLTP trace recorded a TPC-C workload with a scaling factor of 50 warehouses, whereas the DSS trace captured a read-only TPC-H query workload. The pages referenced by both traces did not overlap. Table 1 lists specific statistics of these traces recorded. To simulate changing workloads, we concatenated the OLTP and the DSS traces and attached a copy of them at the end, i.e., as final result, the trace CONCAT had the four phases OLTP–DSS–OLTP–DSS and an overall update percentage of 5.6%.

We ran this trace for each algorithm and recorded the number of physical reads and physical writes necessary to serve the logical request stream. Hit ratios can be derived from the number of physical reads and the total number of requests, but, of primary concern is the the overall I/O cost, which, in our simulation, can be presented by the *virtual execution time* T_v:

$$T_v = n_R \times c_R + n_W \times c_W \quad (1)$$

where n_R and n_W are the number of physical reads and physical writes, respectively, and $c_R \div c_W$ = cost ratio.

Figure 2 shows the virtual execution time T_v of CASA and the flash-aware algorithms relative to LRU, for cost ratio 1:1 and 1:64. For CFLRU, we repeated the experiment for window sizes $w = 0.25$, $w = 0.50$, and $w = 0.75$, relative to the buffer size, and denoted as CFLRU-0.25, CFLRU-0.50, and CFLRU-0.75. For improved clarity of the visual presentation, we chose to plot its curve only for the best-performing w-setting.

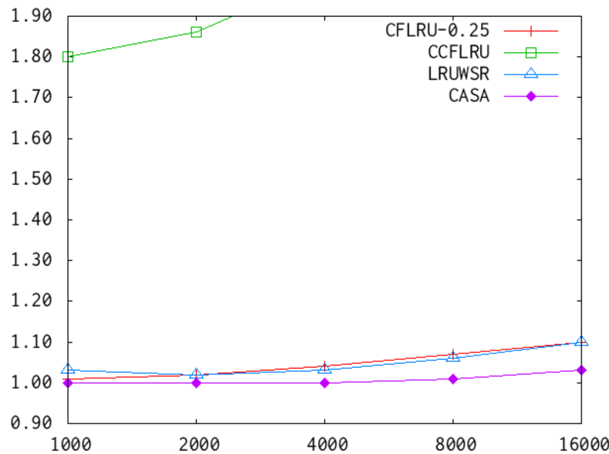
The cost ratio 1:1 (Figure 2a) simulates the case of magnetic disks. Here, LRU exhibited the best performance, because it is immune to variations of update intensities in the workload. While the flash-aware algorithms are clearly outperformed by LRU, T_v of CASA closely approaches that of LRU. Although read and write costs are symmetric, the flash-aware algorithms still discriminate clean pages and try to keep dirty pages as long as possible, resulting in an unjustified high n_R and consequently a higher T_v. For example, with 16,000 buffer pages, n_R of CCFLRU is higher than that of LRU by factor three ($4,444,570$ vs. $1,433,996$), while its n_W savings is only about 12% ($111,891$ vs. $128,456$). As a result, its T_v is two times higher than that of LRU (out of the plot area). For CFLRU, the setting $w = 0.25$ had the best performance, because, with the other two settings, the higher numbers of physical reads were not paid off by the savings in physical writes.

With the cost ratio 1:64 (Figure 2b), the window size $w = 0.75$ of CFLRU achieved a better performance than its other two settings, because it more greedily trades reads

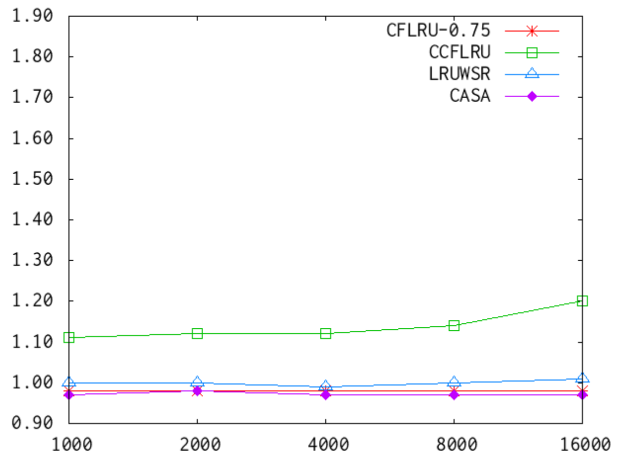
[3]While checkpoints are written, which often occurs in intervals of few minutes, systems are restricted to read-only operations. Assume that many GBytes would have to be propagated to multiple disks using random writes (in parallel). Hence, reaction times for update operations could reach a considerable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

Table 1: Statistics of the OLTP trace and DSS trace

Attribute	OLTP	DSS
number of page requests	1,420,613	3,250,972
number of distinct pages	59,782	104,308
min page number	0	220,000
max page number	219,040	325,639
number of reads	1,156,795	3,250,972
number of updates	263,818	0
update percentage	18.57%	0
locality (number of the hottest pages vs. number of requests for them)	11,957 vs. 1,224,613 (20% vs. 86%)	20,862 vs. 2,875,664 (20% vs. 88%)



(a) T_v for cost ratio 1:1



(b) T_v for cost ratio 1:64

Figure 2: Virtual execution times relative to LRU running the CONCAT trace, for R/W cost ratios 1:1 and 1:64. Buffer size scaled from 1,000 to 16,000 pages.

for writes, and, this behavior paid off here, because physical writes are now much more expensive than physical reads. Having a highly read-intensive workload, the achievable savings in physical writes are rather small. Therefore, the performance advantages of CASA and CFLRU over LRU are not significant and LRUWSR was just comparable to LRU. Nevertheless, CASA outperforms CFLRU, even when the latter used its best setting. Although not significant, its performance advantage is clear without ambiguity. Note, the performance figures are obtained from (discrete-event) simulation, therefore, no measurement error and runtime overhead were introduced.

In Figure 3a, we plot the size of the clean list C of CASA versus the virtual time (request number). The curve clearly reflects the four phases of the workload: it fluctuates around 200 in the OLTP phases and stays at $1,000$ in the DSS (read-only) phases. The violent oscillation in the OLTP phases is only a visual effect. For example, the slowly climbing curve in Figure 3b, reflecting the stage when the clean list gains pages from the empty list during the first 10,000 requests, is squeezed into nearly a vertical line in Figure 3a.

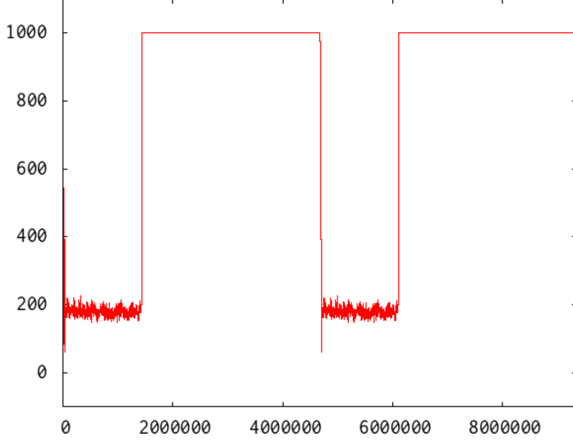
5.2 Cost Awareness

We now study CASA's behavior with various cost ratios, whereas the experiments in Section 5.1 focused on the

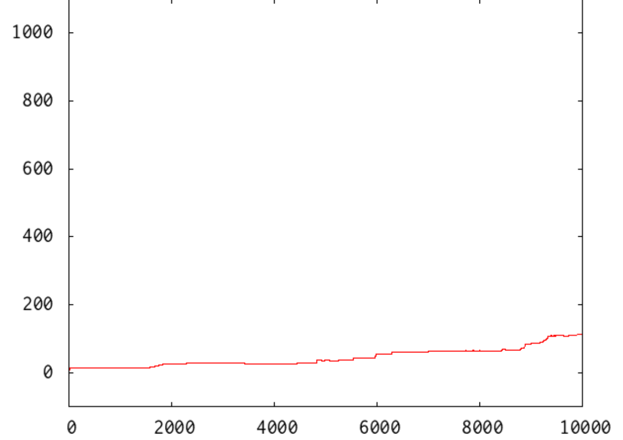
Table 2: Number of physical reads and physical writes running the bank trace using 1,000 and 10,000 buffer pages

	1,000 pages		10,000 pages	
	n_R	n_W	n_R	n_W
CCFLRU	427,012	71,103	237,518	35,642
CFLRU-0.25	393,914	85,818	175,448	43,096
CFLRU-0.50	389,653	79,408	181,223	39,428
CFLRU-0.75	388,917	74,688	195,160	37,188
LRU	403,056	95,849	177,861	51,157
LRUWSR	409,469	90,183	186,306	46,076
CASA 1:1	389,350	77,884	175,985	44,975
CASA 1:4	398,249	72,190	180,558	39,947
CASA 1:16	417,715	71,359	192,149	37,427
CASA 1:64	425,993	71,138	211,015	36,089
CASA 1:128	426,932	71,116	221,344	35,666

changes in the workload. The trace used here 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 [8, 13, 14, 15, 16]. 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



(a) The complete trace



(b) The first 10,000 requests

Figure 3: The size of the clean list changes with the virtual time (request number), reflecting the workload characteristics. The buffer size was 1,000 pages and the R/W cost ratio was 1:64.

referenced. Exhibiting substantial locality, 20% (10,376) of the pages are referenced by 72% (434,702) of the requests.

For CASA, we ran the trace with cost ratios scaling from 1:1 to 1:128. For the remaining algorithms, there is no need to repeat the test for cost ratios 1:4 to 1:128, because only CASA is aware of different cost ratios and can adjust its behavior accordingly. As illustrated by the n_R and n_W figures listed in Table 2, CASA used increasingly more physical reads and, in turn, saved more physical writes, while the relative cost of physical writes was increased.

We calculated the T_v's according to Formula 1 and show their ratios relative to those of LRU in Figure 4. The workload has a relatively high percentage of update requests and, as a result, the flash-aware algorithms could demonstrate their performance advantage over LRU—nearly all of them are below the 1.0 horizontal line in the chart. CASA had clearly the best performance for nearly all settings. For cost ratio 1:128 and buffer size 10,000 pages, it is about 30% faster than LRU. In several cases, it was slightly outperformed by CFLRU with its best w-settings (which were manually optimized). But in real application scenarios, the best w-setting of CFLRU is hard to find: it depends not only on the cost ratio and the update percentage of the workload, but also on the buffer size (Compare Figure 4a and Figure 4b for cost ratios 1:1 or 1:4).

5.3 Dynamic Cost-Ratio Detection

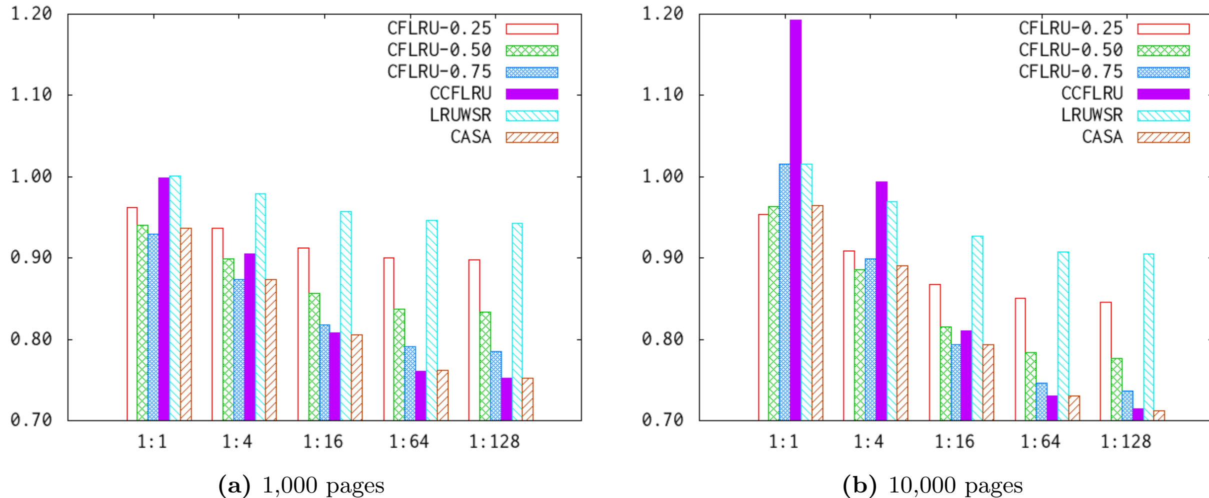
Being fundamentally different from the tuning requirement of CFLRU, our algorithm automatically optimizes itself at runtime, given the knowledge concerning cost ratios. So far, we have assumed that this knowledge is available to the algorithm. It can be provided, e.g., by the device manufacturer or by 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 [17]. Therefore, we use an n-point moving average of the measured values 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 1. Note, no change to the algorithm is needed to use the dynamically detected costs.

Maintaining the moving average requires the last n measurements to be remembered. Assuming that two bytes[4] are used to store a measured value, to remember, e.g., 32,768 values, we need only eight pages (page size = 8 KB) and, in total, 16 pages for both reads and writes. To optimize the moving-average procedure, the measured values can be stored in an array (managed as a FIFO queue). Then, the time complexity of maintaining the moving average is independent of n: for each new measurement, only an array-element update and a few arithmetic operations are involved.

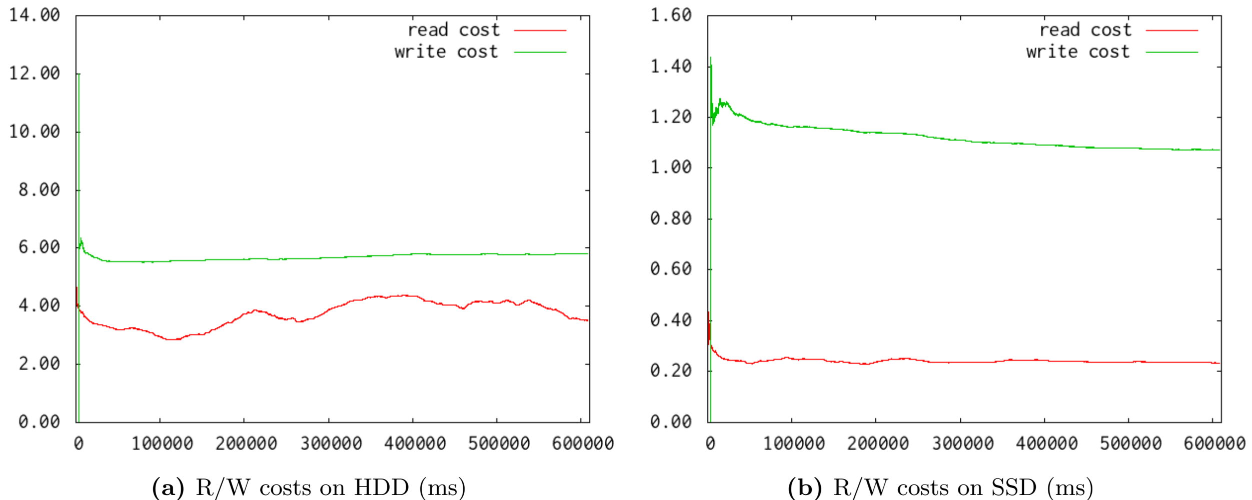
To test this idea, we ran the same trace evaluated in Section 5.2 using real device accesses to a WD WD1500HLFS HDD (magnetic disk) and an Intel SSDSA2MH160G1GN flash SSD. The physical R/W costs were measured and updated online as described above. We chose $n = 32768$ for the n-point moving average, because it is large enough to smooth out the short-term fluctuations and its space overhead is small. Our test machine is equipped with an AMD Athlon Dual Core Processor, 3 GB of main memory, and is running Linux (kernel version 2.6.24) residing on a magnetic disk. To avoid the influence of the down-stream caches along the cache hierarchy, we deactivated the file-system prefetching and the on-device write cache, and set the DIRECT_IO flag while accessing the device under test.

[4]Two bytes can store timings ranging from 1 to 65,536 nanoseconds, which should be enough to cover all the possible physical I/O cost values. Furthermore, burst values out of this range can be safely ignored.



(a) 1,000 pages **(b)** 10,000 pages

Figure 4: Virtual execution times relative to LRU running the bank trace for buffer sizes of 1,000 and 10,000 pages. The R/W cost ratio was scaled from 1:1 to 1:128.



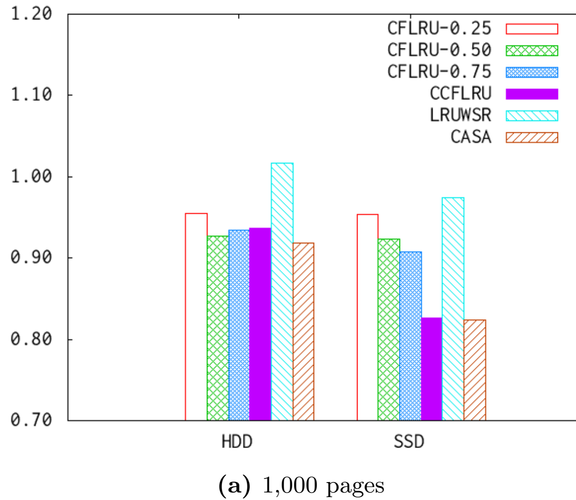
(a) R/W costs on HDD (ms) **(b)** R/W costs on SSD (ms)

Figure 5: Dynamically detected physical R/W costs vs. the virtual time, using an n-point moving average ($n = 32768$), for running the bank trace using 1,000 buffer pages

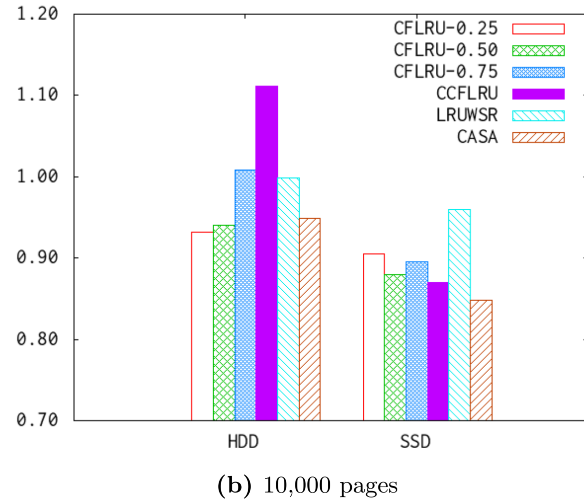
Figure 5 plots the detected R/W costs for the HDD and SSD devices. Our approach effectively hided the bursts in the measure values and amortized them in the cost ratio, which is about 1:1.5 for the HDD and 1:4.5 for the SSD. As an extra advantage, it captures the cost ratio's long-term variations, which are caused by, e.g., the change of read/update patterns (random vs. sequential) in the workload.

We scaled the buffer size from 1,000 to 10,000 pages and measured the real execution times. On the SSD, CASA had the best performance, while on the HDD, it was comparable to CFLRU with the best w-settings, but better than the remaining algorithms. Figure 6 plots the measured execution times relative to that of LRU for buffer sizes of 1,000 and 10,000 pages. The relative *real* performance shown in Figure 6 is roughly comparable with the relative *virtual* performance shown in Figure 4 for cost ratios 1:1 and 1:4.

In summary, our experiments covering varying workload and various cost ratios have demonstrated the problems of existing flash-aware algorithms: their performance advantage over conventional algorithms heavily depends on the update intensity in workloads and the R/W cost ratio of storage devices. A remarkable example is CCFLRU: under the typical update-intensive OLTP workload (Figure 4), it achieved very good performance for highly skewed cost ratios (e.g., 1:64 and 1:128), but suffered from a drastic performance degradation for symmetric R/W costs (1:1). Furthermore, its performance becomes unacceptable under the workload with varying update intensities (Figure 2). In contrast, CASA exhibits a consistently good performance for various configurations, both in the simulation and in the real system.



(a) 1,000 pages



(b) 10,000 pages

Figure 6: Real execution times relative to LRU running the bank trace on the HDD and the SSD, for buffer sizes of 1,000 and 10,000 pages

6. CONCLUSION

The problem of buffer management for asymmetric I/O costs is of great importance with emerging flash SSDs. We proposed to use the R/W cost ratio to capture the R/W asymmetry of those devices and presented CASA—a novel cost-aware self-adaptive algorithm. Our experiments have shown that CASA is efficient and can adapt itself to various cost ratios and to changing workloads, without any tuning parameter. Our solution is not limited to flash-based storage devices, but should be generally applicable to block-oriented storage devices with asymmetric I/O costs.

7. ACKNOWLEDGEMENT

We are grateful to Gerhard Weikum for providing the bank trace, to IBM (Deutschland and USA) for providing further trace data (which could not be included in this paper due to time constraints), and to Peiquan Jin for helpful discussions. We are also grateful to anonymous referees for valuable comments. This research is in part supported by the German Research Foundation and the Carl Zeiss Foundation.

8. REFERENCES

[1] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM TODS*, 9(4):560–595, 12 1984.

[2] Intel Corp. X25-V SATA SSD Datasheet. `http://download.intel.com/design/flash/nand/value/datashts/322736.pdf`, 2010.

[3] Intel Corp. X25-M SATA SSD Datasheet. `http://download.intel.com/design/flash/nand/mainstream/322296.pdf`, 2010.

[4] S. Park, D. Jung, et al. CFLRU: a replacement algorithm for flash memory. In *CASES*, pages 234–241, 2006.

[5] Y. Ou, T. Härder, et al. CFDC: a flash-aware replacement policy for database buffer management. In *SIGMOD Workshop DaMoN (Data Management on New Hardware)*, pages 15–20, Providence, RI, 2009. ACM.

[6] H. Jung, H. Shim, et al. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *Trans. on Cons. Electr.*, 54(3):1215–1223, 2008.

[7] A. S. Tanenbaum. *Operating Systems, Design and Impl.* Prentice-Hall, 1987.

[8] Z. Li, P. Jin, et al. CCF-LRU: A new buffer replacement algorithm for flash memory. *Trans. on Cons. Electr.*, 55:1351–1359, 2009.

[9] Y.S. Yoo, H. Lee, et al. Page replacement algorithms for nand flash memory storages. In *Computational Science and Its Applications (ICCSA 07)*, pages 201–212. Springer, 2007.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 12 1983.

[12] C. Mohan, D. J. Haderle, et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[13] E. J. O'Neil, P. E. O'Neil, et al. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD*, pages 297–306, 1993.

[14] T. Johnson, D. Shasha, et al. 2Q: a low overhead high performance bu er management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[15] D. Lee, J. Choi, et al. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *Trans. on Computers*, 50(12):1352–1361, 2001.

[16] N. Megiddo and D.S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*. USENIX, 2003.

[17] L. Bouganim, B.T. Jónsson, et al. uFLIP: Understanding flash IO patterns. In *CIDR*, 2009.