ACEing the Bufferpool Management Paradigm for Modern Storage Devices

Tarikul Islam Papon
Boston University
papon@bu.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

***Abstract*—Over the past few decades, solid-state drives (SSDs) have been replacing hard disk drives (HDDs) due to their faster reads and writes, as well as their superior random access performance. Further, when compared to HDDs, SSDs have two fundamentally different properties: (i) *read/write asymmetry* (writes are slower than reads) and (ii) *access concurrency* (multiple I/Os can be executed in parallel to saturate the device bandwidth). However, several database operators are designed without considering storage *asymmetry and concurrency* resulting in device underutilization, which is typically addressed opportunistically by *device-specific tuning* during deployment. As a key example and the focus of our work, the bufferpool management of a Database Management System (DBMS) is tightly connected to the underlying storage device, yet, state-of-the-art approaches treat reads and writes equally, and do not expressly exploit the device concurrency, leading to subpar performance.**

In this paper, we propose a new *Asymmetry & Concurrency-aware* bufferpool management (ACE) that batches writes based on device *concurrency* and performs them in parallel to amortize the *asymmetric* write cost. In addition, ACE performs *parallel prefetching* to exploit the device's *read concurrency*. ACE does not modify the existing bufferpool replacement policy, rather, it is a *wrapper* that can be integrated with *any replacement policy*. We implement ACE in PostgreSQL and evaluate its benefits using a synthetic benchmark and TPC-C for several popular eviction policies (Clock Sweep, LRU, CFLRU, LRU-WSR). The ACE counterparts of all four policies lead to significant performance improvements, exhibiting up to 32.1% lower runtime for mixed workloads (33.8% for write-intensive TPC-C transactions) with a negligible increase in total disk writes and buffer misses, which shows that incorporating asymmetry and concurrency in algorithm design leads to more faithful storage modeling and, ultimately, to better device utilization.

***Index Terms*—bufferpool, write asymmetry, concurrency, SSD**

I. Introduction

Modern Devices: Concurrency & Read/Write Asymmetry. The majority of today's secondary storage devices are solid-state disks (SSDs), while traditional hard-disk drives (HDDs) are used primarily as cold or archival storage [20, 62]. SSDs achieve their superior performance by adopting NAND flash memory as their storage medium [2], thus eliminating the mechanical overheads of HDDs (i.e., seek time, rotational delay), and consequently providing benefits like fast random access, low energy consumption, and high chip density [28, 34, 54]. Furthermore, SSDs exhibit a high degree of *internal parallelism* that can be harnessed to increase performance [7, 41]. In other words, an SSD needs to receive multiple **concurrent** I/Os (which can be distributed to different components by the flash controller) to saturate its bandwidth [7]. On the other hand, due to the physics of the flash medium, the cost of reading is considerably lower than the cost of writing which leads to an SSD **read/write asymmetry** where writes can be up to one order of magnitude slower than reads [10]. With these two properties, i.e., *concurrency* (quantified by k) and *read/write asymmetry* (quantified by α), SSD behavior departs from the one of traditional HDDs. These characteristics have two key implications: (i) proper use of concurrency enables better device utilization, and (ii) treating page reads and writes equally in an asymmetric environment is suboptimal [50, 51]. However, many data-intensive systems have not been thoroughly redesigned to account for these characteristics.

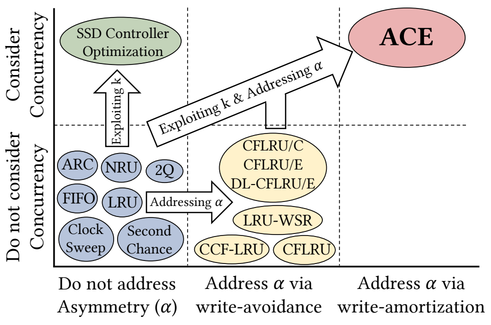

Consider Concurrency
Do not consider Concurrency
SSD Controller Optimization
ACE
Exploiting k
Exploiting k & Addressing α
ARC
NRU
2Q
FIFO
LRU
Addressing α
Clock Sweep
Second Chance
CFLRU/C
CFLRU/E
DL-CFLRU/E
LRU-WSR
CCF-LRU
CFLRU
Do not address Asymmetry (α)
Address α via write-avoidance
Address α via write-amortization

Fig. 1: ACE addresses asymmetry by exploiting concurrency and amortizing writes.

Bufferpool Manager. The part of a database management system (DBMS) that interacts directly with storage devices is the *bufferpool* which works as the interface between memory and the underlying storage device. The bufferpool keeps in memory a set of pages to minimize the number of (slow) disk accesses. If a requested page is already in the bufferpool, it can be served immediately without accessing the disk. In contrast, if the requested page is not available, it has to be fetched from the disk and placed in the bufferpool. If the bufferpool is already full, another page is first written back to disk (if dirty) and evicted, based on a *page replacement policy* [58]. In this work, we show that a bufferpool manager which is carefully tailored to the underlying storage device can significantly improve the system's performance.

The Challenge. In this work, we attempt to address two challenges of state-of-the-art bufferpool managers. (A) First, existing bufferpool managers often assume that the underlying devices have no concurrency ($k = 1$). When writing dirty pages to disk, state-of-the-art bufferpool managers write (evict) one page at a time, hence missing the opportunity to exploit the device concurrency. Although durability mechanisms like logging and checkpointing attempt to ameliorate the effect of random I/Os by converting them to sequential I/Os, the physical page writes from the bufferpool are performed one I/O at a time, which is our primary focus. Furthermore, some systems employ data prefetching [43, 69], however, they primarily rely on sequential I/O instead of concurrent reading. (B) Second, page replacement policies generally do not consider the device asymmetry (α), instead, they treat read and write requests equally (i.e., they consider $\alpha = 1$). Hence, the *to-be-evicted* page's status (dirty or clean) does not depend on whether the requested page is a read or a write. As a result, it is entirely possible that the bufferpool evicts a dirty page (writes to disk) when the incoming page request is a read, essentially **exchanging a read for a write** irrespective of the device asymmetry, leading to suboptimal performance [50].

Figure 1 shows that popular page replacement policies are designed for devices with no asymmetry and concurrency (bottom left, blue). Recently proposed *flash-friendly* policies like CFLRU [53], LRU-WSR [26], and others [38, 73] try to minimize the number of writes by evicting clean pages first, indirectly addressing the asymmetry (bottom middle, yellow). However, these policies also exchange reads and writes interchangeably. There have been some efforts to utilize the device concurrency via modifying the SSD internals [32, 60, 61] (top left, green). However, these solutions lack general applicability because they require extensive redesigning of the SSD controller and they do not target the DBMS bufferpool. Hence, to the best of our knowledge, no bufferpool manager appropriately considers both asymmetry and concurrency.

A New Bufferpool Design Space. Traditionally, the design space of bufferpool management includes primarily a *page replacement policy* and optionally a *read-ahead policy*. The page replacement policy decides the order that pages are *evicted* and *written back*. If the evicted page is dirty, a write-back is issued for that page. Since traditional systems have a single policy for both eviction and write-back, they essentially make one decision for two separate questions: *which page to evict?* and *which page to write-back?* We separate these two questions by introducing a new *write-back policy*, thus, decoupling write-backs from eviction. We maintain one overall *virtual page ordering* of eviction (which is typically outsourced to the existing *replacement algorithm*), however, we have a different *virtual order for writing-back pages*, which depends on (a) the replacement algorithm, (b) whether the page is dirty, and (c) the support write concurrency of the storage device.

A bufferpool management approach can be described by four design decisions: (i) replacement algorithm, (ii) write-back policy, (iii) eviction policy, and (iv) read-ahead policy.

In this augmented design space, the *replacement algorithm* inform both the *write-back* and the *eviction* policies, however, in a different manner. The *write-back policy* uses the virtual order of pages dictated by the *replacement algorithm* and the degree of write concurrency of the device to write-back *only dirty* pages. The *eviction policy* uses the virtual order of pages dictated by the *replacement algorithm* to evict only clean pages (which may have been just written back or were already clean). The decision of how many pages to evict is decided from the application as a decision between prioritizing locality (evict only one page) vs. prefetching (evict multiple pages, but use the *read-ahead policy* to populate the free spots).

Design Goals. With this refactored bufferpool design space, we set the following design goals :

- **Exploit concurrency** to ensure proper utilization of the underlying device parallelism.
- **Bridge asymmetry** via write-amortization to ensure that there is no imbalance when the bufferpool is saturated.
- **Ease of adoption**, so that systems can quickly benefit from our design without extensive engineering effort.

Asymmetry & Concurrency-Aware Bufferpool Manager (ACE). To fulfill these goals, we propose ACE, a new bufferpool manager that utilizes the underlying device concurrency to bridge the device asymmetry (top right of Fig. 1 – colored red). Our approach uses *asymmetry/concurrency-aware* write-back and eviction policies. The write-back policy always writes multiple pages *concurrently* (utilizing the device's write concurrency), hence amortizing the write cost. The eviction policy evicts one or multiple pages at the same time from the bufferpool to enable prefetching. When multiple pages are evicted at once, ACE can *concurrently prefetch* pages to exploit the device's read concurrency. A key advantage of ACE is that it can be integrated with any existing page replacement policy with low engineering effort, while, any prefetching technique can also be integrated, essentially allowing any existing bufferpool manager to be augmented by our approach. Figure 2 shows the *ideal speedup* of such an asymmetry/concurrency-aware bufferpool manager where the baseline system uses LRU. The benefit of ACE is higher for devices with higher asymmetry (up to 2.5×) showing that the asymmetry gap is increasingly important to bridge. We integrate ACE with four page replacement policies and implement them in PostgreSQL to evaluate ACE's efficacy. Our *asymmetry/concurrency-aware* bufferpool manager (i) batches write-back requests (from bufferpool to the disk) in a *storage-aware* manner and (ii) prefetches pages in parallel leading to substantial performance gains with a negligible increase in buffer miss and total writes.

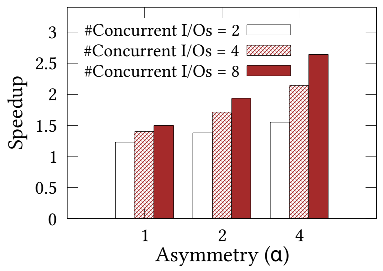


Fig. 2: ACE outperforms state-of-the-art due to better device utilization.

Contributions. Our contributions are as follows:

- We identify the importance of SSD read/write asymmetry and concurrency with respect to bufferpool management.
- We refactor the bufferpool design space by including a write-back policy that consider device-specific properties.

- We propose **ACE**, an *asymmetry & concurrency-aware* bufferpool manager that utilizes the device's concurrency. ACE is flexible enough to be combined with any existing page replacement policy and prefetching technique.
- We implement ACE with PostgreSQL's default replacement algorithm (Clock Sweep) and we add three more replacement algorithms (LRU, CFLRU, LRU-WSR) and their ACE counterparts in PostgreSQL.
- We evaluate the efficacy of ACE against these four algorithms in PostgreSQL. ACE achieves up to 32.1% lower runtime for a mixed workload with negligible increase in total writes and buffer misses. For the TPC-C mix, ACE reduces runtime by up to 24.2% while attaining 33.8% lower runtime for the write-heavy transaction.

II. SSD Properties

We now discuss the two key properties of SSDs – *asymmetry* and *concurrency*; why they exist, and their implications.

A. Read/Write Asymmetry

Read/write asymmetry (α) in SSDs is caused by (i) the *erase-before-write* design, (ii) large erasure granularity, and (iii) garbage collection [4, 10, 46]. In flash-based SSDs, logical page *updates* (at the file system level) are always performed as *out-of-place* updates. The contents of a physical flash page can be updated only after an *erasure* [10, 54], i.e., once a page is written, it cannot be updated until that whole block is erased [1]. Hence, when a page *update* arrives, the controller has to invalidate the old page and write the updated page in a new block. As a result, after a number of writes, the flash medium contains several invalidated pages. To reclaim this invalidated space, the flash controller periodically triggers *garbage collection* which copies the valid pages of a block, writes them in a new block and then erases the previous block. Note that flash cells generally wear out after a certain number of program/erase cycles, which is a measure of the device's *endurance* [2, 45]. Techniques like wear-leveling, overprovisioning, and bad block skipping are commonly used to mitigate the wear-out effect and expand the SSD lifetime [23, 29, 44]. While the read/write granularity is a flash page (typically with size 512B-32KB), the erasure granularity is an erase block (4MB-64MB). The higher erasure granularity, the overhead of maintaining garbage collection, and the extra writes garbage collection incurs, result in *higher amortized write cost*. The asymmetry depends on the specific device and the access granularity. We empirically measure the asymmetry of the devices that we use in our experimental setup (an Optane SSD, a PCIe SSD, a SATA SSD and a Virtual SSD) and summarize them in Table I (in §VI). We observe that all NAND-based SSDs (PCIe, SATA, and Virtual SSD) exhibit significant asymmetry. Optane SSDs generally exhibit low asymmetry because of their 3D XPoint technology [72].

B. Device Concurrency

Modern flash-based SSDs have parallelism in many different levels [6, 41]. Typically, an SSD has multiple channels that are connected to the flash controller. Each channel consists of a shared bus with multiple chips, and each chip contains multiple dies. Each die comprises multiple planes, and each plane constitutes multiple blocks where pages reside. This highly parallelized architecture creates opportunities to efficiently support concurrent storage accesses [7]. If multiple I/Os are issued in parallel, the flash controller tries to parallelize them by distributing them to different parts of the device [41, 55, 63]. As a result, the device's peak bandwidth can only be achieved with multiple concurrent I/Os. The level of *observed concurrency* varies among devices and it also depends on the access type and block size [50]. Most devices have a large number of channels (≥ 8) which is the fundamental form of internal parallelism. Table I (in §VI) reports the empirical read concurrency (k_r) and write concurrency (k_w) of the four SSDs we experiment with, showing that k_r is generally much higher than k_w. Note that for the virtual SSD, k_r and k_w reflect the impact of the cloud provider limiting the device's IOPS allowed. To summarize, both α and k vary across devices depending on the device internals, access granularity, and access pattern, while all NAND-based SSDs exhibit a significant degree of both asymmetry and concurrency.

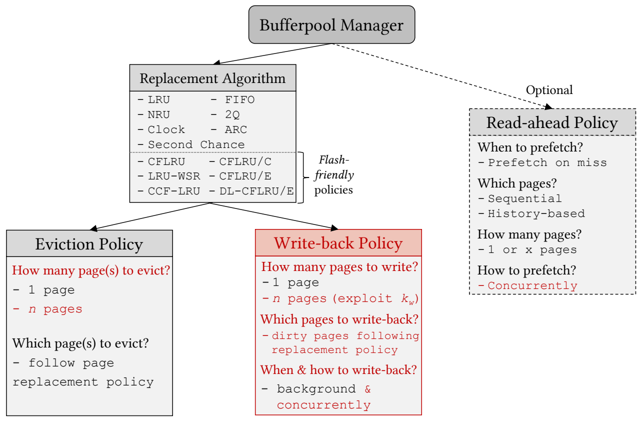


Fig. 3: Bufferpool design space in terms of the design decisions and various options (RED denotes new components)

III. An Augmented Bufferpool Design Space

Traditional bufferpool designs have one main component: the *page replacement policy* (that may lead to a write-back if a dirty page is evicted), which is driven by the replacement algorithm and one optional component: a prefetching module. We depart from this paradigm and separate the write-back decision from the replacement policy. We propose a new bufferpool design paradigm that makes four decisions for every bufferpool design: (i) **a replacement algorithm** that decides the *to-be-evicted page order* and influences the *to-be-written page order*, (ii) **a write-back policy** that decides when, how many, and with which criteria to write back pages, (iii) **an eviction policy** that decides how many and with which criteria to evict pages, and (iv) **a read-ahead policy** that decides when, how many, which pages, and how to prefetch. Figure 3 presents the proposed augmented bufferpool design space along with the various options for each design decision.

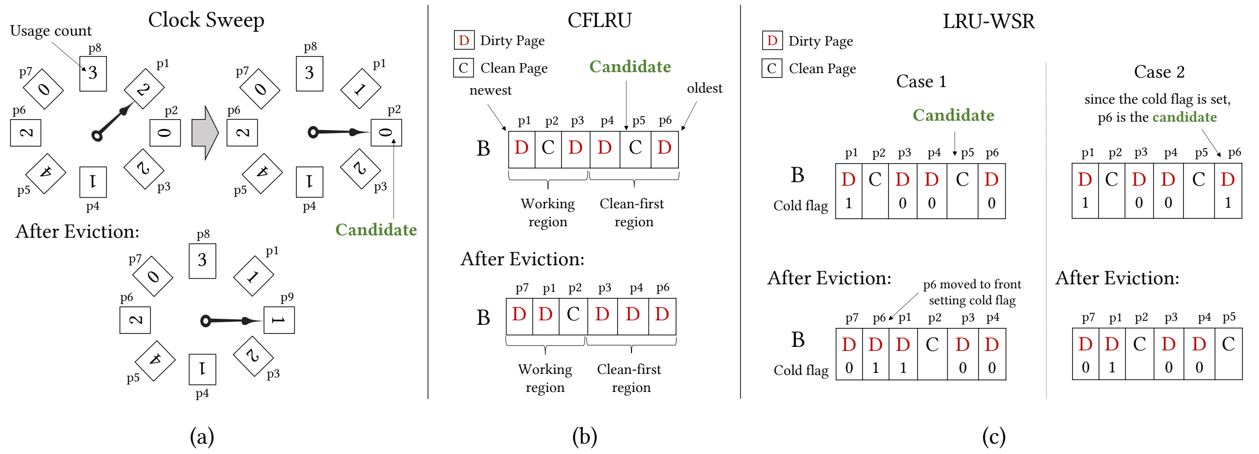


Fig. 4: Popular eviction policies: (a) Clock Sweep evicts pages based on *usage count*; (b) CFLRU tries to evict *clean* pages first [window size $N/2$ used for brevity]; (c) LRU-WSR keeps a *cold* flag to delay dirty page eviction.

A. Background on Page Replacement Algorithm

At the core of every bufferpool design is the page replacement algorithm which decides which page needs to be replaced when the bufferpool runs out of space. Note that this decision essentially creates a *virtual order* of the pages to be evicted. The most popular page replacement algorithm is Least Recently Used (LRU) [49] which tries to keep the most recently accessed pages in the bufferpool. Some other popular algorithms are Clock [31], NFU [66], 2Q [24], NRU [16], FIFO [66], ARC [42], and Second Chance [30]. PostgreSQL adopts the Clock Sweep algorithm [56], a variant of NFU. The algorithm maintains the bufferpool pages as a circular list with each page's *usage count*, while the *candidate* pointer rotates clockwise. If the candidate unpinned page's usage count is 0, the page is selected for eviction. Otherwise, its usage count is reduced by 1, and the pointer moves ahead clockwise (Figure 4a). This algorithm and all the aforementioned algorithms do not differentiate between reads and writes. In recent literature, when optimizing for SSDs, *flash-friendly* policies reduce device wear by absorbing more writes in memory before evicting a dirty page.

Flash-friendly Policies. Clean-First LRU (CFLRU) maintains the LRU order of the pages and divides the LRU list into two regions: *working region* and *clean-first region* [53]. The working region contains the recently accessed pages, while the clean-first region contains the candidate pages for eviction (Figure 4b). To minimize the number of writes, CFLRU evicts clean pages from the clean-first region and when there are no clean pages left in this region, CFLRU evicts dirty pages following LRU. The size of the clean-first region is decided by the *window size* parameter. Although the optimal *window size* depends on the workload, a rule of thumb is that if N is the size of the bufferpool, $\frac{N}{3}$ is a good window size [53].

Another flash-friendly algorithm is LRU with Write Sequence Reordering or LRU-WSR [26]. LRU-WSR delays evicting *cold* dirty pages to reduce the number of writes. Each page in LRU-WSR has a *cold* flag which is cleared every time the page is referenced. If the candidate page for eviction is dirty, it is evicted only if its cold flag is set; otherwise the page is moved to the most recently used position while setting the cold flag and another candidate page is selected based on the LRU order. If the candidate page is clean, it is evicted irrespectively of its cold flag. Figure 4c shows two examples of LRU-WSR. Both CFLRU and LRU-WSR outperform LRU for flash devices, because they prioritize evicting clean pages over dirty pages, indirectly addressing the underlying device asymmetry. Other flash-aware policies [38, 73] adopt similar strategies while considering the access frequency and wear-leveling. All these policies prioritize reads over writes to mitigate device wear caused by writes. While they indirectly address asymmetry up to a point, they do not explicitly consider the specific device asymmetry and concurrency.

B. Write-back Policy

Once a *dirty* page is selected for replacement, the bufferpool has to write the page back to disk. In practice, state-of-the-art systems like PostgreSQL have a separate dirty page buffer to which dirty pages for replacement are first moved, and flushed in the background. Further, page updates are also written in the write-ahead log (WAL) sequentially to make the database crash-resilient, and dirty pages are written-back during checkpointing as well. However, all these write-backs are performed *one page at a time* following the underlying assumption that storage devices can perform one I/O at a time, and rely on the operating system to re-order or batch disk writes. In this way, the read request of a new (not buffered) page is countered by one write-back when the page for replacement is dirty, *exchanging one write for one read*. Since writes are more expensive than reads in modern storage devices, this is an unfair decision that leads to performance degradation, which we also observe experimentally. To address this, **we augment the write-back policy to write multiple dirty pages at once** following the *virtual order* provided by the page replacement algorithm. Note that we can write-back multiple pages concurrently without a penalty due to the underlying write concurrency of the device (k_w). In order to bridge the asymmetry (α), we parallelize at least α writes to amortize their cost for the single read that caused this write-back, or more if the device concurrency permits (if $k_w > \alpha$).

C. Eviction Policy

Following the write-back phase, the eviction policy now decides which and how many pages to evict. Traditionally, systems evict always one page which leads to the one read for one write approach. Since following the new write-back policy we have already written back more than one page, the eviction policy may opt to evict more (now clean) pages to make room for additional pages to be prefetched, assuming the prefetcher has high confidence for its predictions. Hence, we include one additional design choice: **the number of pages to be evicted**. The exact pages to be evicted still follow the virtual order imposed by the replacement algorithm. By splitting the page replacement policy into a write-back and an eviction phase, we exploit the underlying concurrency of the storage device to bridge the asymmetry between reads and writes, using the replacement algorithm as the core decision throughout both phases. This is exactly why our approach benefits any bufferpool page replacement policy.

D. Read-ahead Policy

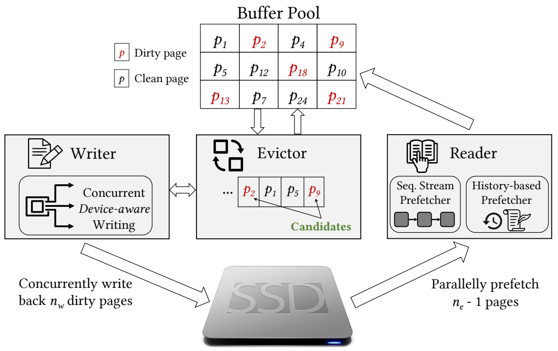
The goal of prefetching is to make data available in memory before it is requested, consequently improving the bufferpool hit rate. The prefetching policy determines when to prefetch, how many and which pages to prefetch. The most common prefetching approach is *sequential prefetching* like One Page Lookahead (OPL) and N-Page Lookahead (NPL), where either a single page (OPL) or multiple pages (NPL) beyond the requested page is prefetched [12, 43, 65]. History-based prefetching uses previous access patterns to predict the next pages to be accessed [18, 25, 36]. This type of prefetching improves performance when accesses are localized. Most commercial systems use simple prefetching policies like sequential prefetching [21, 57], especially when the bufferpool has empty slots. However, systems generally prefetch one page at a time instead of issuing multiple parallel I/Os, hence missing the opportunity to exploit the device's read concurrency. Since prefetching is not as effective without good workload knowledge, it is often treated as an optional choice.

IV. ACE Bufferpool Manager

We now present in detail the proposed *asymmetry & concurrency-aware* bufferpool manager (**ACE**) that addresses the read/write asymmetry via write-amortization. As depicted in Figure 5, ACE is comprised of three components: (i) the Evictor, (ii) the Writer, and (iii) the Reader. The **evictor** determines which page(s) to evict, the **writer** writes-back concurrently dirty pages and the **reader** prefetches pages.

A. Overview of ACE Bufferpool Management

When a request for reading or writing a page P is received, we first search through the bufferpool. If P is not found and the bufferpool is full, then (at least) one page has to be evicted. The page replacement algorithm determines the page to be evicted (termed *top page*). If the top page is *clean*, it is evicted and page P is fetched. Up until this part, ACE is identical to any state-of-the-art bufferpool management. However, if the top page is *dirty*, ACE proceeds as follows:



Buffer Pool
p_1 p_2 p_4 p_9
p_5 p_{12} p_{18} p_{10}
p_{13} p_7 p_{24} p_{21}
p Dirty page
p Clean page
Writer
Concurrent Device-aware Writing
Evictor
... p_2 p_1 p_5 p_9
Candidates
Reader
Seq. Stream Prefetcher
History-based Prefetcher
Concurrently write back n_w dirty pages
SSD
Parallelly prefetch n_e - 1 pages

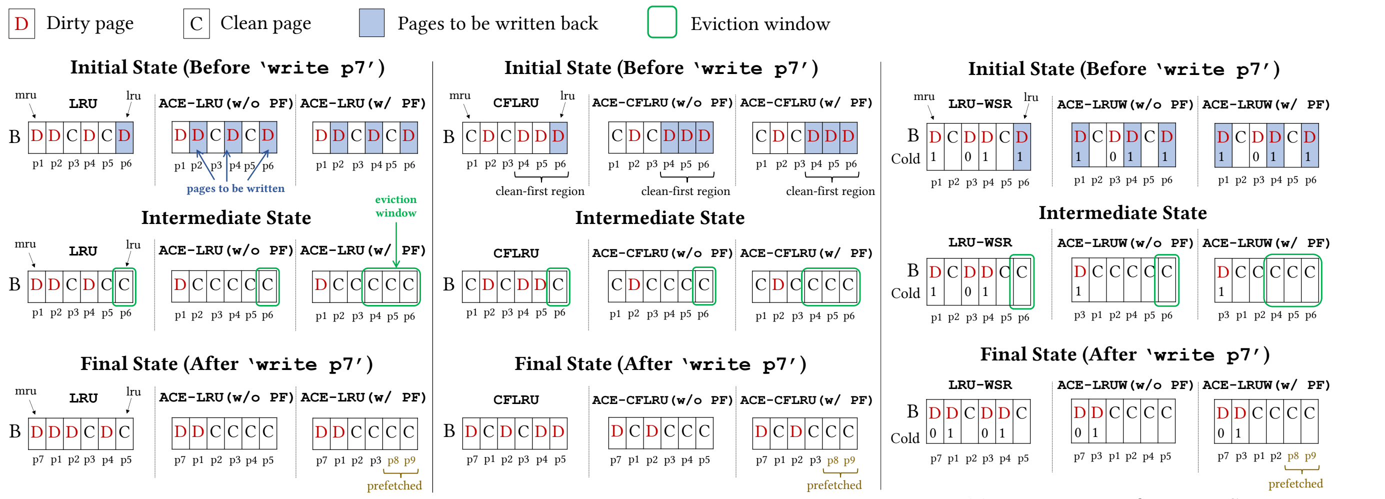
Fig. 5: Abstract overview of **ACE** components

- **ACE without prefetching**: *concurrently write* $\boldsymbol{n_w}$ dirty pages and evict a single page.
- **ACE with prefetching**: *concurrently write* $\boldsymbol{n_w}$ dirty pages, evict $\boldsymbol{n_e}$ pages, and concurrently prefetch $\boldsymbol{n_e - 1}$ pages.

The values n_w and n_e depend on the underlying device concurrency and the potential benefits of prefetching. When prefetching is enabled, ACE evicts n_e pages in order to prefetch $n_e - 1$ pages exploiting the read concurrency of the device. While we anticipate that the prefetching will allow us to have pages that will be accessed by the immediate next requests, the eviction somewhat reduces the locality, so n_e has to carefully balance the read concurrency and the accuracy of the prefetching. We tune ACE to use n_w equal to the optimal write concurrency of the device (k_w). We experimentally tested values for n_e between 1 and k_r, and we empirically set n_e to be also k_w, because evicting k_r pages was hurting locality. Note that for most devices, the read concurrency is significantly higher than the write concurrency ($k_r >> k_w$). Regarding the pages that are selected for write-back and for eviction, both decisions are influenced by the page replacement algorithm. As such, ACE can be combined with any replacement algorithm. In our experiments, we use four popular approaches (LRU, Clock Sweep, CFLRU, and LRU-WSR) that all benefit from the ACE paradigm. Figure 6 shows the effect of incorporating ACE with LRU (Fig. 6a), CFLRU (Fig. 6b), and LRU-WSR (Fig. 6c). Note that ACE always writes n_w dirty pages concurrently irrespectively of prefetching. The full ACE algorithm is listed in Algorithm 1.

B. Writer

The Writer is responsible for concurrently writing-back n_w pages. State-of-the-art systems often write-back pages using a background process, however, these writes are issued one at a time, hence missing out on the opportunity to exploit the parallelism of the underlying storage device. Instead, ACE Writer writes concurrently n_w *dirty* pages. By making sure that $n_w = k_w$, the concurrent writes take place at the same latency as a single write, thus amortizing the cost of k_w writes and fully bridging the read/write asymmetry if $\alpha < k_w$. The pages that are selected for write-back are the next n_w *dirty* pages that the underlying page replacement algorithm would eventually evict. As a result, these carefully batched writes make the subsequent page evictions free (since, with high probability, the following evictions will target clean pages).



(a) ACE on top of LRU (b) ACE on top of CFLRU (c) ACE on top of LRU-WSR

Fig. 6: ACE page selection policies for $n_w = 3$ and $n_e = 3$. (a) ACE writes three dirty pages (p6, p4, p2) following the LRU order; if prefetching is enabled three pages (p6, p5, p4) are evicted, otherwise one page (p6) is evicted. (b) Similarly for CFLRU, ACE writes three dirty pages (p6, p5, p4) from the clean-first region and depending on prefetching either three pages (p6, p5, p4) are evicted or one (p6). (c) For LRU-WSR, ACE finds a dirty page with cold flag not set (p3). This page is moved to the front setting its cold flag. The dirty pages with set cold flag (p6, p4, p1) are selected for concurrent writing.

C. Evictor

Following the completion of the write-back process, the Evictor will evict either one or n_e pages. Since at this point, the *top* page is by definition clean (because the Writer has already written it), it will be evicted to read the requested page P. If prefetching is enabled, the Evictor evicts n_e pages in total to allow for an equal number of pages to be prefetched. Note that after the write-back process, there will be at least n_w contiguous clean pages following the order dictated by the page replacement algorithm. Hence, the Evictor can now evict n_e clean pages to create space for the incoming pages. Earlier, we mentioned that we empirically set $n_e = n_w$, however, even if we allow n_e to be greater than n_w, the Evictor will always be able to evict n_e pages as long as in total the bufferpool has at least n_e clean pages. Essentially, the Writer writes back the n_w first *dirty* pages according to the page replacement algorithm order, and the Evictor evicts the n_e first *clean* pages (after the writing has been performed) according to the page replacement algorithm order. For example, Figure 6a shows that ACE with prefetching will write three dirty pages following the LRU order (p6, p4, p2) and evict the last three (now clean) pages (p6, p5, p4) following the LRU order.

D. Reader

The reader is an optional component whose job is to prefetch pages from disk in case of a buffer miss. Note that for many workloads prefetching does not attain much benefit, hence, commercial systems either do not use any prefetcher, or they use very simple prefetching techniques. The strength of ACE is that any prefetching technique can be employed by the Reader. In fact, we use two prefetchers in our design: a sequential prefetcher and a history-based prefetcher.

Sequential Prefetcher. We use a sequential prefetcher named TaP [36] that uses a table to detect sequential access patterns. The prefetcher uses a *sequential detection module* to determine whether a page miss is part of a sequential stream. Only after detecting a sequential stream, ACE uses the sequential prefetcher, otherwise, ACE uses the history-based prefetcher. When a page miss occurs, the sequential detection module searches the page address in the TaP table. If it is not found in the table, then the address of the next page is inserted in the table, expecting that the current page miss is part of a new sequential stream. In case of a sequential stream, the newly inserted page in the TaP table will be found in the workload soon. Then, the TaP prefetcher starts sequential prefetching, however, ACE does not immediately start prefetching. Instead, if at least 4 sequential page requests are found, then ACE triggers the prefetcher and concurrently reads the next $n_e - 1$ pages along with the page that caused the buffer miss (P). Old page addresses that are not part of any sequential stream are evicted in a FIFO manner.

History-based Prefetcher. This prefetcher also uses a large table-based structure to store the page access history and predicts the next most probable access [18]. The organization of the table is shown in Figure 7. Row i of the table contains the next probable page addresses and their likelihood (in form of a weight) after page i is accessed. For example, given a reference for page 1, the best candidate for prefetching is page 3 since page 3 has a higher weight (9). ACE prefetches only if the weight is more than a certain fetch threshold. The table generation is straightforward: given the current and previous page references, we go to the row in the table indexed

Index	NextPages	Weights
0	{2, 5, 7}	{10, 4, 2}
1	{10, 3, 18}	{3, 9, 1}
2	{6, 9}	{8, 0}
	⋮	⋮

Fig. 7: Table structure of the history-based prefetcher

Algorithm 1: ACE

```
   Input: $P, n_w, n_e$ $is\_pf\_enabled$
1  // $P$ is the accessed page
2  // $n_w$ is the maximum effective write concurrency ($n_w = k_w$)
3  // $n_e$ is the number of concurrent reads during prefetching
4  // $is\_pf\_enabled$ determines if prefetching is enabled or not
5  if $P$ in buferpool then
6      return $P$
7  else
8      // miss! need to bring P from disk
9      if buferpool not full then
10         if $is\_pf\_enabled == true$ then
11             // reads P and prefetches up to $n_e - 1$ pages from
                  disk (depending on available slots)
12             - prefetch_pages ($P, n_e - 1$)
13         else
14             - read $P$ from disk
15         end if
16     else
17         $top\_page$ = replacement_policy.get_one_page_to_evict()
18         if $top\_page$ is clean then
19             // follow classical approach if page is clean
20             - drop $top\_page$ from bufferpool
21             - read $P$ from disk
22         else
23             // $top\_page$ is dirty. concurrently write $n_w$ dirty
                  pages
24             // $P_{wb}$ is a vector containing the candidate dirty
                  pages
25             - $P_{wb}$ = populate_pages_to_writeback()
26             - issue ||length($P_{wb}$)|| concurrent writes, $\forall p \in P_{wb}$
27             - mark ||length($P_{wb}$)|| pages as clean, $\forall p \in P_{wb}$
28             if $is\_pf\_enabled == true$ then
29                 // evict $n_e$ pages
30                 // pages written and to be evicted can be
                      different
31                 // $P_{ev}$ is a vector containing the pages to
                      evict
32                 $P_{ev}$ = replacement_policy.get_n_pages_to_evict()
33                 - drop ||length($P_{ev}$)|| pages from bufferpool, $\forall p \in P_{ev}$
34                 // Now, prefetch
35                 - prefetch_pages ($P, n_e - 1$)
36                 - empty $P_{ev}$
37             else
38                 // evict 1 page
39                 - drop $top\_page$ from bufferpool
40                 - read $P$ from disk
41             end if
42             - empty $P_{wb}$
43         end if
44     end if
45 end if

1  Procedure populate_pages_to_writeback()
2      // follow the underlying page replacement policy to
          generate $P_{wb}$
3      - select next $n_w$ dirty pages based on the underlying page replacement policy
4      - return this vector

1  Procedure prefetch_pages (page $P$, int $x$)
2      if $P$ in Sequential_Table then
3          // start of a sequential stream!
4          // read $P$ and the next $x$ pages concurrently
5          - prefetch_sequential ($P$)
6      else
7          // use the history based prefetcher
8          // read $P$ and $x$ pages (selected by prefetcher)
              concurrently
9          - prefetch_history ($P$)
10     end if
11     /* note that $P$ should be placed in the most recently used
          position in the bufferpool whereas other pages should
          be placed in the least recently used positions        */
12     - place these $x + 1$ pages into bufferpool
```

by the page number of the previous access. If the `NextPages` vector contains the current page, we increase its corresponding weight in the `Weights` vector. Otherwise, if the `NextPages` vector does not point to the current page and its weight is zero, we place a pointer to the current page in the vector with weight 1. If the weight field is nonzero, we decrease the weight. To prevent growing the `NextPages` vector perpetually, `ACE` keeps track of the next 3 most probable pages. This table structure takes approximately 0.6% space of the database size.

E. Putting Everything Together

Traditional bufferpool strategies "exchange" one read for one write when evicting a dirty page from a saturated bufferpool. This approach is not optimal when a write is $\alpha\times$ more expensive than a read. However, it is not possible to exchange α reads for one write, since (i) this would grow the bufferpool perpetually, and (ii) unless we batch – thus delay reads, a system receives one request at a time. Hence, `ACE` tries to *amortize* the cost of writes by concurrently issuing $\boldsymbol{n_w}$ writes, where $n_w = k_w$. The complete `ACE` bufferpool manager's policy is presented in Algorithm 1. If the page to evict is clean, `ACE` follows the classical approach of simply dropping it from the bufferpool (Line 20). Otherwise, the `ACE` Writer identifies the pages to be cleaned and writes them concurrently (Lines 25–27). Depending on whether prefetching is enabled, `ACE` Evictor either evicts multiple (Lines 32–33) or one page (Line 39). The referenced page is placed in the most recently used position while the prefetched pages are placed in the least recently used positions (following the page replacement policy) so that even if the prefetcher's prediction is wrong, the prefetched page can be simply dropped from the bufferpool.

V. Implementation & Integration

We now discuss the implementation effort and the integration with a full-blown relational system, PostgreSQL 11.5. We first discuss the bufferpool manager structure and its operation. We then discuss the implementation effort to make PostgreSQL bufferpool asymmetry and concurrency aware.

PostgreSQL Buffer Manager. The PostgreSQL buffer manager consists of a buffer table, buffer descriptors, and a bufferpool. The buffer table and the buffer descriptors hold page metadata and the mapping information between the data pages and bufferpool frames. The bufferpool layer keeps track of file pages (i.e., data, indexes). The bufferpool is organized as an array, where each slot stores one page of a data file, and is referenced by a `buffer_id`. In PostgreSQL, each page is assigned a unique tag (`buffer_tag`), which contains the file mapping and block location and is used when the buffer manager receives a request. During an eviction, the buffer manager uses the *Clock Sweep* page replacement algorithm. Dirty pages are flushed to storage by two background processes: the **background writer** and the **checkpointer**. Both processes flush dirty pages, however, they have different roles and behaviors. Checkpointer writes a checkpoint record to the WAL file and flushes all the buffered dirty pages during checkpointing. The background writer continues to flush dirty pages in the background to offload the burden of the checkpointer. PostgreSQL uses light-weight locks (of type *content_lock*, *spin_lock* and *io_in_progress_lock*) to protect shared resources and data consistency.

Implementation. To ensure an apples-to-apples comparison we integrate LRU, CFLRU, LRU-WSR, and their `ACE` counterparts (including the default Clock Sweep) in PostgreSQL. Since all three algorithms are LRU variants, we first implement LRU using an LRU *freelist* queue and then build on it for

CFLRU and LRU-WSR. The window size for CFLRU is set to $1/3$ of the bufferpool size as suggested by its authors [53]. For LRU-WSR, we added a *cold* bit with each page descriptor which is checked during eviction. The `buffer_tag` of the pages were used to get metadata information of the corresponding page. We follow the default Clock Sweep implementation's locking mechanism to ensure data consistency.

`ACE` is implemented as a wrapper on top of the underlying page replacement policy. If the candidate page for eviction is dirty, `ACE` first identifies which n_w pages to write following the replacement policy. We implement a method named `FlushNBuffer()` in `bufmgr.c` which takes the candidate pages as input and flush them out to kernel. Further, we modify the background writer and checkpointer's writing mechanism (which is separate from the bufferpool code) to ensure that they always perform n_w writes concurrently. To do this, we augment several PostgreSQL's low-level writing methods (i.e., `pg_pwrite()`) and their corresponding wrapper functions. After the pages are flushed, the eviction (of one or multiple pages depending on prefetching) is performed as described in Section IV-C. Each of the two prefetchers employed by `ACE` uses a hash table: the sequential prefetcher to implement the TaP table structure and the history-based one to keep track of the accesses following each page. Throughout the bufferpool implementation, the `buffer_id` and the `buffer_tag` are frequently used to identify each page, the frame it is stored, and which relation is associated with.

VI. Evaluation

We now show the benefits of the `ACE` paradigm when applied on four state-of-the-art page replacement policies (LRU, CFLRU, LRU-WSR, and Clock Sweep) using both a synthetic benchmark and the standard TPC-C benchmark.

Experimental Setup. We use a machine with two Intel Xeon Gold 6230 2.1GHz processors each having 20 cores with virtualization enabled and with 384GB of main memory. Our experiments involve three storage devices: (i) a 375GB Optane P4800X SSD, (ii) a 1TB PCIe P4510 SSD, and (iii) a 240GB SATA S4610 SSD. We refer to these devices as *Optane SSD*, *PCIe SSD* and *SATA SSD* respectively. In addition, we use a *virtualized* device from Amazon AWS that has 1.2TB SSD capacity and 60000 provisioned IOPS (high-performance SSD). We refer to this device as *Virtual SSD*, and we attach it to a machine from the *t2.micro* family having 2GB main memory with one virtual CPU. For all four devices, we quantify the asymmetry and concurrency through careful benchmarking [50] (summarized in Table I). Unless otherwise mentioned, we use $n_w = k_w$ of the device in use. In most of our experiments, we employ the PCIe SSD, hence we use $n_w = 8$. Before running the experiments, all devices were preconditioned by sequentially writing on the entire device three times to ensure that they have stable performance [13].

Workload. We use four synthetic workloads inspired by prior work [38, 73]. We refer to them as MS (Mixed Skewed), WIS (Write-Intensive Skewed), RIS (Read-Intensive Skewed) and MU (Mixed Uniform). The properties of the workloads are described in Table II. A read/write ratio of 90/10 indicates read (write) operations are 90% (10%) of the total number of operations. A locality 90/10 means that 90% of all the operations are performed on 10% of the pages. We use *pgbench* for our synthetic workloads which is loosely based on TPC-B. We use a scaling factor of 1000 which results in a database size of approximately 15GB. We also show the benefits of our approach with the TPC-C benchmark [68].

TABLE I: Empirical α and k of our SSDs.

Device	α	k_r	k_w
Optane SSD	1.1	6	5
PCIe SSD	2.8	80	8
SATA SSD	1.5	25	9
Virtual SSD	2.0	11	19

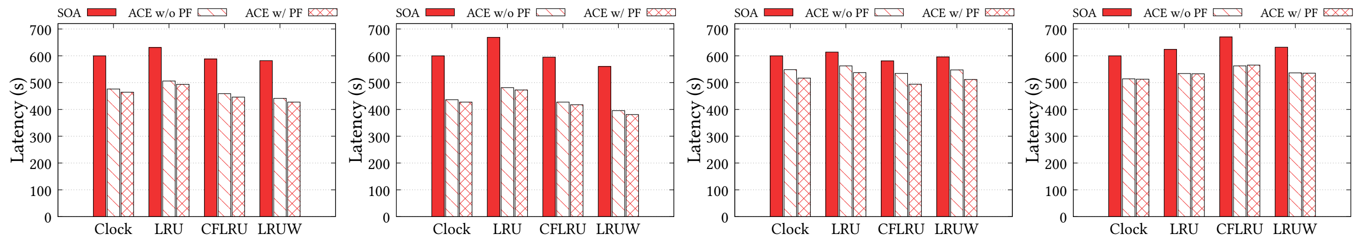
TABLE II: Properties of the synthetic workloads

Workload	Database Size	R/W Ratio	Locality
Mixed Skewed (MS)	15GB	50/50	90/10
Write-Intensive Skewed (WIS)	15GB	10/90	90/10
Read-Intensive Skewed (RIS)	15GB	90/10	90/10
Mixed Uniform (MU)	15GB	50/50	50/50

Experimental Methodology. We run every workload for the default PostgreSQL implementation (Clock Sweep as replacement policy) for 10 minutes and then run the same workload for the other replacement policies and their `ACE` counterparts. For every experiment, we measure (i) workload latency, (ii) transactions per second, (iii) buffer misses/hits, and (iv) total writes. The experiment results are averaged over 5 iterations and the standard deviation was less than 5%. We generally configure PostgreSQL *shared_buffers* (bufferpool) as 1GB ($\sim$6% of the data size). WAL is enabled and the WAL file is written in a separate device following common practice.

A. Experimental Analysis with Synthetic Data

`ACE` Improves Runtime. Our first experiment shows that `ACE` bufferpool management (either with or without prefetching) reduces the total workload latency by up to 32.1%. For this set of experiments we use the PCIe SSD that has $k_w = 8$ and $\alpha = 2.8$. Figures 8a-d show the workload execution time for the baseline Clock Sweep, LRU, CFLRU, and LRU-WSR along with their `ACE` counterparts with and without prefetching for the 4 synthetic workloads in PostgreSQL. The runtime of the `ACE` policies both with and without prefetching is consistently faster than the baseline. Since `ACE` policies utilize the device's write parallelism, it writes back pages more aggressively (but hidden due to the device concurrency), resulting in better performance. `ACE` with prefetching reduces latency by 22.6%, 21.8%, 22.5% and 26.1% for baseline Clock Sweep, LRU, CFLRU and LRU-WSR respectively when running workload MS (Figure 8a), while `ACE` without prefetching reduces latency by 20.6%, 19.7%, 22.0%, and 23.5% respectively. Since the workload is skewed, the prefetching helps avoid some disk access, resulting in slightly better performance. `ACE`'s



(a) ACE achieves high gain in the mixed workload MS.

(b) Gain of ACE is higher in the write-intensive workload WIS.

(c) ACE improves runtime even in read-heavy workload RIS

(d) ACE x has significant gain for mixed uniform workload MU.

Fig. 8: ACE reduces total workload latency for all Clock Sweep, LRU, CFLRU, and LRU-WSR in the PCIe SSD.

TABLE III: Comparison of buffer miss, logical writes and physical writes

WL	ACE - Clock			ACE - LRU			ACE - CFLRU			ACE - LRU-WSR		
	Δmiss	Δl-writes	Δp-writes	Δmiss	Δl-writes	Δp-writes	Δmiss	Δl-writes	Δp-writes	Δmiss	Δl-writes	Δp-writes
MS	-0.001%	0.07%	0.10%	-0.001%	0.06%	0.09%	-0.001%	0.08%	0.12%	-0.001%	0.09%	0.14%
WIS	-0.001%	0.08%	0.12%	-0.001%	0.08%	0.14%	-0.001%	0.08%	0.14%	-0.001%	0.11%	0.17%
RIS	-0.008%	0.06%	0.09%	-0.008%	0.06%	0.09%	-0.008%	0.07%	0.11%	-0.009%	0.12%	0.16%
MU	0.002%	0.09%	0.14%	0.003%	0.10%	0.15%	0.002%	0.09%	0.14%	0.002%	0.10%	0.17%

gain is higher for the write-intensive workload WIS. ACE with prefetching achieves 28.8%, 29.3%, 30.1% and 32.1% lower runtime than baseline Clock Sweep, LRU, CFLRU and LRU-WSR respectively (Figure 8b). This is expected, because for a write-intensive workload, the bufferpool needs to write more pages back to the disk, hence the benefits that come from efficient writing are pronounced. In contrast, for the read-intensive skewed workload RIS, ACE has smaller benefit since ACE does not have enough writes to optimize. However, the gain is still significant (Figure 8c); ACE achieves 8.1% to 13.9% lower runtime. Now, the benefit of prefetching alone is substantial (up to 5%). Finally, the mixed workload MU causes a small increase in total writes ($\leq 0.1\%$), however, this does not affect the overall trends of performance gains, which range from 14.5% to 15.7% (Figure 8d). Since the workload is uniform, the impact of prefetching is insignificant, however, because the prefetched pages are placed last in the virtual order, prefetching does not hurt performance either. In the remainder of our experiments, unless otherwise mentioned, prefetching is enabled.

Performance Gains Do Not Come at a Cost. *We highlight that the overall workload latency improvement observed in these experiments (up to* 32.1%*) does not come at a hidden cost*. Table III compares the buffer misses, application writes and total physical writes for ACE with prefetching enabled for different workloads. The table shows the percentage difference in buffer miss, logical writes and physical writes between ACE[1] and the baseline values. The maximum increase in buffer misses is 0.003% for ACE-LRU on workload MU, and the maximum increase in total writes is 0.12% for LRU-WSR on workload RIS, thus being negligible. Since ACE writes multiple pages at once, it is possible that after writing a page (but not evicting), another write comes in for the same page, consequently, increasing the total number of logical writes slightly. The table also shows that for skewed workloads, ACE with prefetching attains lower buffer misses (up to 0.009%) compared to the baseline algorithms. In contrast, the number of buffer misses increases slightly for the uniform workload because prefetching can not help much in that case.

Impact on SSD Wear Out. We now analyze the impact of ACE on SSD wear out. Since every cell in a NAND flash can sustain a certain number of erases, the total number of writes primarily contributes to an SSD's wear out. We again refer to Table III where we list the percentage difference of both logical and physical writes for different workloads with different page replacement policies. We capture the SMART (Self-Monitoring, Analysis and Reporting Technology) [64] attributes to collect the number of physical writes on the SSD. The table shows that the maximum increase in physical writes is 0.17% while the maximum increase in logical writes is 0.14%. Figure 9 shows the logical and physical writes (LW and PW) on our PCIe SSD as we run our synthetic workloads in PostgreSQL for an extended period (30 mins) with LRU-WSR and ACE-LRU-WSR. Note that, the physical writes are approximately 5-6× higher than logical writes due to flash's garbage collection and wear-leveling [22]. While the total number of writes (logical and physical) for ACE and its baseline replacement policy remains almost the same, the speedup for ACE can be as high as 1.35×, with a negligible increase in physical writes (0.17%). For a read-heavy workload, ACE's performance benefit is small but still comparable to the negligible increase in physical writes. At any rate, the ACE paradigm is always beneficial regarding the overall workload latency, even with a very small fraction of writes. On the other hand, for a purely read-only workload, there will be no performance benefit (but also no increased write amplification or SSD wear out since there will be no writes).

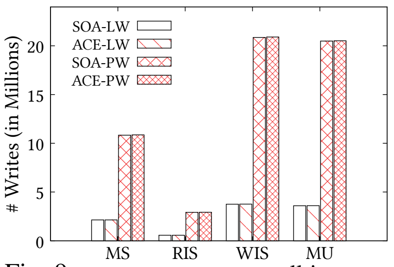
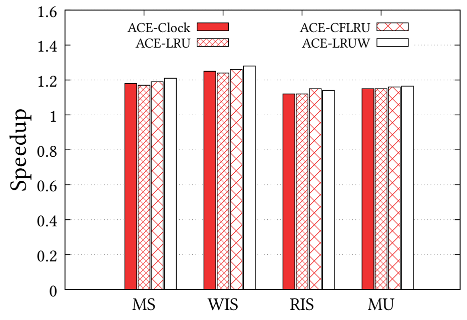
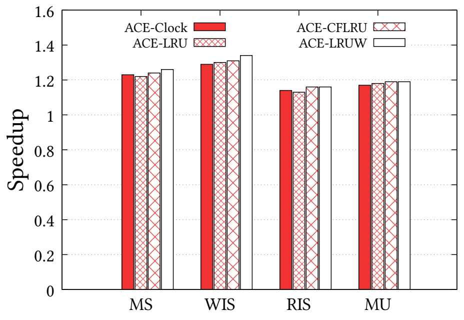


Fig. 9: ACE causes very small increase in total number of writes

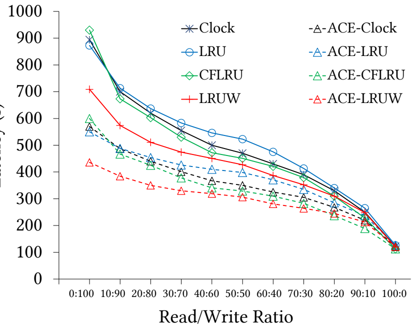
[1]We report the numbers of ACE policies with prefetching enabled as they cause a higher number of writes.



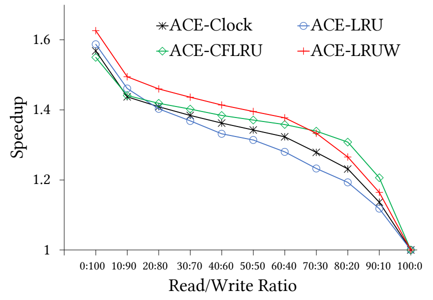
(a) ACE improves latency for all workloads in the SATA SSD



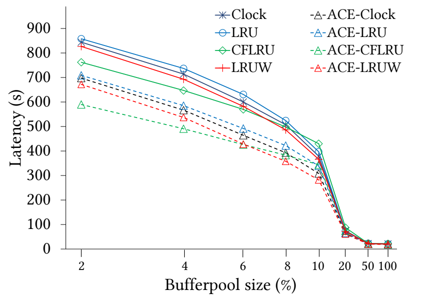
(b) ACE improves latency for all workloads in the Virtual SSD



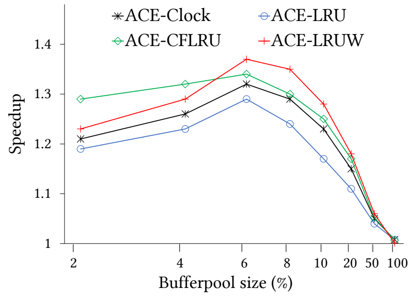
(c) Impact of r/w ratio (PCIe SSD – BP size:**6%**, locality: **90/10**)



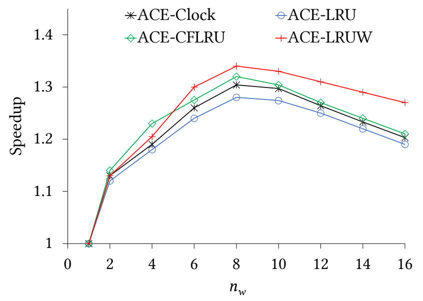
(d) Higher gain for write-heavy workloads (PCIe SSD – BP size:**6%**, locality: **90/10**)



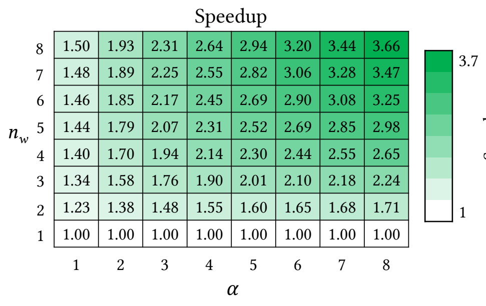
(e) Impact of bufferpool size (PCIe SSD – r/w: **50/50**, locality: **90/10**)



(f) More gain under memory pressure (PCIe SSD – r/w: **50/50**, locality: **90/10**)



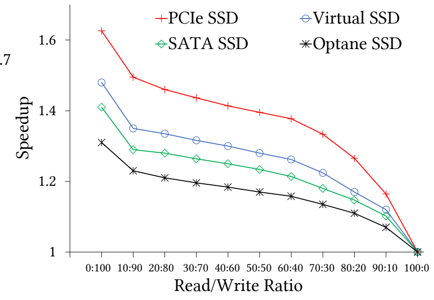
(g) Impact of concurrency (PCIe SSD – r/w: **50/50**, BP size: **6%**, locality: **90/10**)



Speedup

n_w \ α	1	2	3	4	5	6	7	8
8	1.50	1.93	2.31	2.64	2.94	3.20	3.44	3.66
7	1.48	1.89	2.25	2.55	2.82	3.06	3.28	3.47
6	1.46	1.85	2.17	2.45	2.69	2.90	3.08	3.25
5	1.44	1.79	2.07	2.31	2.52	2.69	2.85	2.98
4	1.40	1.70	1.94	2.14	2.30	2.44	2.55	2.65
3	1.34	1.58	1.76	1.90	2.01	2.10	2.18	2.24
2	1.23	1.38	1.48	1.55	1.60	1.65	1.68	1.71
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

(h) Ideal speedup of ACE in the $(\boldsymbol{\alpha}, \boldsymbol{n})$ continuum



(i) Benefit is higher for devices with higher asymmetry (Algo: LRU-WSR, BP size: **6%**)

Fig. 10: (a, b) ACE improves runtime significantly for device with low asymmetry. (c, d) Workloads with higher fraction of writes lead to greater benefits. (e, f) ACE is beneficial across a wide range of bufferpool size w.r.t. data size. (g) Speedup increases as concurrent I/Os are increased until device gets saturated. (h) Spectrum of $(\boldsymbol{\alpha}, \boldsymbol{n})$ – as we move towards higher asymmetry, ACE has higher gain. (i) Empirical evidence showing that devices with higher asymmetry has higher gain for ACE.

Better Performance Even in Low Asymmetry Devices. To analyze the impact of ACE for devices with low asymmetry, we run our synthesized workloads on the SATA SSD ($\alpha = 1.5$) and the Virtual SSD ($\alpha = 2.0$). Figures 10a and 10b show the speedup of ACE (with prefetching) when integrated with the four page replacement algorithms, on the SATA SSD and Virtual SSD, respectively. Both devices are significantly slower than the PCIe SSD, however, the trend of overall performance gain remains. For the regular SSD, the speedup of ACE is $1.12 - 1.28\times$ and for the virtual SSD, it is $1.14 - 1.34\times$. In contrast, the speedup for the PCIe SSD is $1.19 - 1.46\times$ (Figure 8). We observe that for a device with higher asymmetry, the benefit of amortizing the *high* write cost is more than that of a device with lower asymmetry. This supports our thesis that the benefits are larger for higher asymmetry, however, even when there is low/no asymmetry, better utilization of the device's internal parallelism still leads to significant gains.

Write-Intensive Workloads Have Higher Gains. A common observation from the above experiments is that a write-intensive workload benefits more from the ACE bufferpool (Figure 8b). To verify this, we run an experiment varying the read/write ratio from 0/100 (write-intensive) to 100/0 (read-intensive) where the locality is 90/10. Figures 10c and 10d illustrate that the performance of ACE improves drastically as we shift to more write-intensive workloads. For example, as we move away from the most write-intensive workload (0/100) to a balanced workload (50/50), the speedup for ACE (with Clock Sweep) drops from $1.57\times$ to $1.34\times$. As we further move to more read-intensive workloads (towards 100/0), the speedup eventually diminishes. This is because the benefit of ACE stems from efficient concurrent writing, and in a workload with no (fewer) writes, there will be no (less) gain. However, the benefit never fall behind the classical approach since for a read-only workload, ACE behaves exactly the same as a state-of-the-art bufferpool. Figures 8b, 10c and 10d also show that LRU-WSR performs much better than both LRU and CFLRU for a write-intensive workload with high locality. This is because LRU-WSR gives the dirty pages a second chance with the *cold flag*, thus saving a lot of unnecessary disk writes. Figures 10c and 10d also show that CFLRU performs well in a read-intensive setting because it evicts clean pages first, and in a read-heavy workload there will be fewer dirty pages.

Higher Benefits Under Memory Pressure. ACE achieves higher speedup for smaller bufferpool size because a smaller bufferpool causes more evictions, hence more writes. Figures 10e and 10f show the impact of ACE under memory pressure for a mixed skewed workload like MS in the PCIe SSD. Figure 10e shows the actual runtime while Figure 10f shows the speedup of the four ACE policies over the baseline counterparts, as we vary the bufferpool size with respect to the data size. We observe that as the bufferpool size grows beyond 6%, the speedup decreases because larger bufferpool causes fewer evictions (and fewer writes). For a bufferpool size of more than 10%, the latency (and speedup) of all approaches drop drastically because the bufferpool is large enough to hold the working set, resulting in very few disk accesses. On the other side of the spectrum, the speedup for smaller bufferpool sizes (2%, 4%) is slightly less (than that of 6%) because if the bufferpool size is too small, there are too many read I/Os to the disk. Nonetheless, even for a smaller bufferpool, the gain is substantial. For example, for 2% bufferpool size, the speedup for ACE-CFLRU is $1.29\times$, while for 10%, the speedup is $1.25\times$. Figure 10e also shows that CFLRU performs better than both LRU and LRU-WSR for smaller buffer size. As the buffer size increases, the benefit of CFLRU drops because of its lower hit ratio. On the other hand, LRU-WSR always performs better than LRU and Clock Sweep because of its write-optimization policy. *In all cases, the ACE policies outperform their baseline counterparts.*

Device Concurrency Plays A Crucial Role. We run the mixed-skewed workload MS in the PCIe SSD while varying n_w to capture the impact of the write-back concurrency vs. the device concurrency as shown in Figure 10g. Each line shows the speedup of ACE over the corresponding baseline algorithm. As concurrency increases, the *speedup* increases in all cases, which is expected. However, after a certain point ($n_w = 8$), the speedup starts decreasing. There are several factors contributing to this: (i) as the number of concurrent I/O increases, the overhead of thread management increases, (ii) the ideal device write concurrency is $k_w = 8$ in this experiment, and (iii) going over the ideal concurrency does not yield any benefit since the bandwidth gets saturated and attempting to submit more concurrent I/Os does not further increase write throughput. As a result, the overall speedup starts reducing after reaching this threshold. We highlight that even with a small degree of concurrency ($n_w = 4$ or $n_w = 6$), the speed is substantial ($1.2\times$ – $1.3\times$).

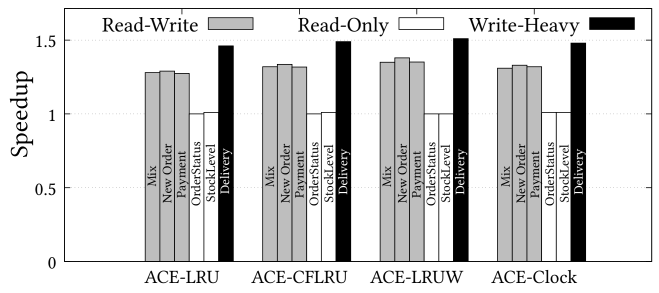


Fig. 11: ACE achieves high speedup for TPC-C mixed transaction, while benefiting the write-intensive transaction the most.

Performance Gain Increases with Asymmetry. Our last experiment highlights that asymmetry impacts the attained gains of the ACE policies. We implement a bufferpool manager with LRU and ACE-LRU without prefetching, and we test them on an emulated device that has ideal asymmetry varied between $\alpha = 1$ (no asymmetry) and $\alpha = 8$, while the device concurrency is $k_w = 8$. We run the mixed-skewed workload MS and normalize the emulated latency of ACE-LRU with respect to that of the baseline LRU for the respective emulated device. Figure 10h shows the ideal speedup of ACE for different values of α and n_w. The continuum of (α, n_w) shows that as we move towards higher asymmetry, the gain increases and it is highest when both asymmetry and concurrency value are maximized. We run an experiment to show this empirically where we vary the read/write ratio with ACE on top of LRU-WSR in all four of our devices (PCIe SSD, SATA SSD, Virtual SSD, Optane SSD). The n_w values were set according to the device k_w. Figure 10i shows that when everything else is the same, the performance gain is higher for devices with higher asymmetry. This is because the benefit of amortizing the asymmetric write cost is higher for a device with a higher write cost. The speedup for the PCIe SSD ($\alpha = 2.8$) is up to $1.63\times$ (write-only workload) while the speedup is limited to $1.48\times$, $1.41\times$ and $1.33\times$ for the Virtual SSD ($\alpha = 2.0$), SATA SSD ($\alpha = 1.5$), Optane SSD ($\alpha = 1.1$), respectively.

B. TPC-C Benchmark

We further use the TPC-C benchmark [68] to demonstrate ACE's efficacy. A TPC-C database consists of nine tables (Warehouse, Stock, Item, District, Customer, History, Order, New-Order, and Order-Line), and the data size depends on the number of *Warehouses*. The benchmark consists of five transactions at different frequencies:

- **NewOrder** (45%): This transaction involves both reads and writes (RW) with 1% failure rate due to invalid inputs.
- **Payment** (43%): This transaction has both reads and writes.
- **OrderStatus** (4%): This is a read-only transaction.
- **StockLevel** (4%): This is a read-only transaction.
- **Delivery** (4%): This is a write-heavy transaction

ACE Accelerates TPC-C by 24%. For the first experiment, we run the standard TPC-C benchmark in PostgreSQL for the four page replacement policies and their ACE counterparts. Each baseline run is 10 minutes long and is configured with 500 warehouses, and 20 users and the resulting database size is approximately 50GB. PostgreSQL *shared_buffers* parameter is configured to be 3GB (6%). Figure 11 shows the performance gain of ACE for the TPC-C mix, and for five workloads each consisting of one of the TPC-C transactions. ACE achieves significant performance gain when integrated with any page replacement policy. For example, the speedup of ACE for the mixed transaction (combination of all five) is $1.29\times$, $1.27\times$, $1.30\times$ and $1.32\times$ when implemented on top of Clock Sweep, LRU, CFLRU, LRU-WSR, respectively. The highest speedup, $1.51\times$, is obtained when running the *Delivery* transaction which is an update-heavy transaction. In addition, as expected, there is no performance gain for the two read-only transactions: *OrderStatus* and *StockLevel*. We observe that the performance results for the TPC-C benchmark corroborate our findings for the synthetic benchmark: (i) ACE offers significant performance benefits when the workload contains even a small fraction of writes, (ii) write-heavy workloads have higher gain, (iii) flash-friendly policies like CFLRU and LRU-WSR outperform other policies. **Overall, ACE reduces the TPC-C mix transaction runtime on modern storage devices by 24% without any significant penalty or other tradeoff.**

ACE Scales with Data Size. Our last experiment shows that the benefits of ACE scale with data size. We again run the TPC-C mix and we increase the number of warehouses, varying the database size from 15GB to 84GB. The bufferpool size is always configured to be 6% of the database size. Figure 12 presents the transactions per minute count (tpmC) of this experiment when running with LRU and ACE-LRU (similar trends were observed for the other policies). The figure shows that ACE's benefits remain as we increase the database size. Specifically, the performance gain of ACE for 125 warehouses is $1.33\times$ while for 1000 warehouses it is $1.24\times$ compared to LRU's tpmC. This small decrease is attributed to the overhead of managing a high volume of data. Overall, this experiment shows that ACE policies scale which makes them ideal for large-scale deployments.

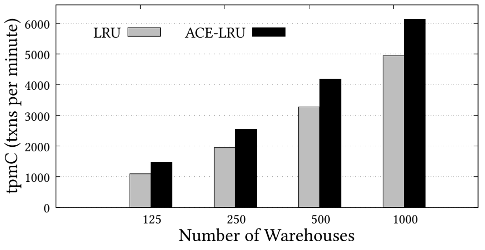


Fig. 12: Gain of ACE scales with data size.

VII. Related Work

Addressing Asymmetry and Concurrency. Read/write asymmetry has been identified as an optimization goal for indexing [3, 8, 9, 37, 39, 70], flash-aware storage engines [5, 27, 47], and other data management operations [14, 15, 19, 52]. Recent research has focused on developing new I/O schedulers for SSDs [41, 55, 59, 63] and on modifying SSD internals to exploit the parallelism [6, 7]. In the same spirit, our work aims to make SSD asymmetry and concurrency a first-class citizen when designing database bufferpool for external memory.

Bufferpool Management. There have been several efforts that focus on developing efficient page replacement policies [16, 24, 30, 31, 42, 49, 66]. However, they are primarily designed for traditional HDDs, hence, they do not address asymmetry or concurrency. Recent work on bufferpool on top of flash devices prioritizes the eviction of clean pages and reduces page writes to minimize device wear-off [26, 38, 53, 73]. Other flash-friendly policies, like FOR and FOR+ [40] use an operation-aware page weight determination for buffer replacement. All these techniques indirectly address asymmetry, however, they do not exploit the device concurrency. Recently proposed works also attempt to exploit the device parallelism by redesigning the SSD controller [32, 60, 61]. In contrast to these approaches, our goal is to develop a bufferpool that expressly utilizes the device concurrency and consequently addresses asymmetry via write-amortization.

In-Memory Database Systems. Systems like LeanStore [35] and Umbra [48] target modern storage devices and depart from the classical bufferpool paradigm using pointer swizzling, variable-size pages, low-overhead replacement strategies, etc. ACE can benefit such systems even further as long as the underlying storage has asymmetry and concurrency.

Prefetching. The most popular prefetching technique is sequential prefetching [12, 36, 67] which is adopted by many commercial systems. Stride-based prefetching is also widely studied primarily for processor caches [17, 33]. History-based prefetching techniques attempt to predict future access patterns based on past access patterns by using history-based table [18], Markov predictor [25], and data compression techniques [11, 71]. In the augmented design space that we propose, any prefetching technique(s) can be integrated.

VIII. Conclusion

Modern solid-state drives are characterized by a *read-write asymmetry* and an access *concurrency*, both of which are essential to fully utilize the device. However, buffer management for DBMS does not explicitly focus on these two properties. In this work, we first refactor the bufferpool design space by separating the eviction policy from the write-back policy. We propose ACE, a novel asymmetry/concurrency-aware bufferpool manager paradigm that batches writes based on device concurrency to amortize the asymmetric write cost. Incorporating concurrency into the write-back policy allows us to custom-tailor any bufferpool manager to the device-at-hand, thus utilizing the device's full potential. ACE can be integrated with *any* existing page replacement and prefetching policy with low engineering effort. We implement ACE in PostgreSQL and measure its benefit when integrated with four state-of-the-art page replacement algorithms. ACE improvements performance by up to 32.1% for synthetic workloads and up to 24.2% for the standard TPC-C mixed transaction (33.8% for write-heavy transactions) without any penalty.

Acknowledgment

We sincerely thank the reviewers for their constructive feedback and the members of DiSC lab for their useful remarks. This work is funded by NSF Grant IIS-2144547, a Facebook Faculty Research Award, and a Meta gift.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008, pp. 147–160.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008, pp. 57–70.

[3] M. Athanassoulis and A. Ailamaki, "BF-Tree: Approximate Tree Indexing," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1881–1892, 2014.

[4] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun, "Efficient Algorithms with Asymmetric Read and Write Costs," in *Proceedings of the Annual European Symposium on Algorithms (ESA)*, 2016, pp. 14:1–14:18.

[5] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD Bufferpool Extensions for Database Systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1435–1446, 2010.

[6] F. Chen, B. Hou, and R. Lee, "Internal Parallelism of Flash Memory-Based Solid-State Drives," *ACM Transactions on Storage (TOS)*, vol. 12, no. 3, pp. 13:1–13:39, 2016.

[7] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 266–277.

[8] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking Database Algorithms for Phase Change Memory," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[9] S. Chen and Q. Jin, "Persistent B+-Trees in Non-Volatile Main Memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[10] M. Cornwell, "Anatomy of a Solid-State Drive," *Communications of the ACM (CACM)*, vol. 55, no. 12, pp. 59–63, 2012.

[11] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993, pp. 257–266.

[12] F. Dahlgren, M. Dubois, and P. Stenström, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," in *Proceedings of the International Conference on Parallel Processing (ICPP), Volume I: Architecture*, 1993, pp. 56–63.

[13] D. Didona, N. Ioannou, R. Stoica, and K. Kourtis, "Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs," *Proceedings of the VLDB Endowment*, vol. 14, no. 3, pp. 364–377, 2020.

[14] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, "Turbocharging DBMS buffer pool using SSDs," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2011, pp. 1113–1124.

[15] P. Dubs, I. Petrov, R. Gottstein, and A. P. Buchmann, "FBARC: I/O Asymmetry Aware Buffer Replacement Strategy," in *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2013, pp. 58–69.

[16] M. Z. Farooqui, M. Shoaib, and M. Z. Khan, "A comprehensive survey of page replacement algorithms," *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume*, vol. 3, 2014.

[17] J. W. C. Fu and J. H. Patel, "Data Prefetching in Multiprocessor Vector Cache Memories," in *Proceedings of the 18th Annual International Symposium on Computer Architecture. Toronto, Canada, May, 27-30 1991*, 1991, pp. 54–63.

[18] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson, "Multiple Prefetch Adaptive Disk Caching," *IEEE Trans. Knowl. Data Eng.*, vol. 5, no. 1, pp. 88–103, 1993.

[19] Y. Gu, Y. Sun, and G. E. Blelloch, "Algorithmic building blocks for asymmetric memories," in *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 112, 2018, pp. 44:1—-44:15.

[20] G. Haas, M. Haubenschild, and V. Leis, "Exploiting Directly-Attached NVMe Arrays in DBMS," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2020.

[21] IBM, "Db2: Prefetching Data into The Bufferpool," *https://www.ibm.com/docs/en/db2/9.7?topic=management-prefetching-data-into-buffer-pool*, 2012.

[22] Z. Jiao, J. Bhimani, and B. S. Kim, "Wear leveling in SSDs considered harmful," in *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*, 2022, pp. 72–78.

[23] Z. Jiao and B. S. Kim, "Generating realistic wear distributions for SSDs," in *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*, 2022, pp. 65–71.

[24] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1994, pp. 439–450.

[25] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *IEEE Trans. Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[26] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: integration of LRU and writes sequence reordering for flash memory," *IEEE Trans. Consumer Electron.*, vol. 54, no. 3, pp. 1215–1223, 2008.

[27] A. Kakaraparthy, J. M. Patel, K. Park, and B. Kroth, "Optimizing Databases by Learning Hidden Parameters of Solid State Drives," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 519–532, 2019.

[28] J.-U. Kang, H. Jo, J. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, 2006, pp. 161–170.

[29] M. Kang, W. Lee, and S. Kim, "Subpage-Aware Solid State Drive for Improving Lifetime and Performance," *IEEE Trans. Computers*, vol. 67, no. 10, pp. 1492–1505, 2018.

[30] K. Kedzierski, M. Moretó, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-LRU replacement policies," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[31] S. Khuri and H.-C. Hsu, "Visualizing the CPU scheduler and page replacement algorithms," in *Proceedings of the SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, 1999, pp. 227–231.

[32] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 239–252.

[33] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Data Prefetching In Shared Memory Multiprocessors," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 1987, pp. 28–31.

[34] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, 2007.

[35] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "LeanStore: In-Memory Data Management beyond Main Memory," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2018, pp. 185–196.

[36] M. Li, E. Varki, S. Bhatia, and A. Merchant, "TaP: Table-based Prefetching for Storage Caches," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 81–96.

[37] Y. Li, B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang, "Tree Indexing on Solid State Drives," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1195–1206, 2010.

[38] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, "CCF-LRU: a new buffer replacement algorithm for flash memory," *IEEE Trans. Consumer Electron.*, vol. 55, no. 3, pp. 1351–1359, 2009.

[39] J. Liu, S. Chen, and L. Wang, "LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.

[40] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-aware buffer management in flash-based systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2011, pp. 13–24.

[41] B. Mao, S. Wu, and L. Duan, "Improving the SSD Performance by Exploiting Request Characteristics and Internal Parallelism," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 472–484, 2018.

[42] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2003, pp. 115–130.

[43] S. Mittal, "A Survey of Recent Prefetching Techniques for Processor Caches," *ACM Computing Surveys*, vol. 49, no. 2, pp. 35:1—-35:35, 2016.

[44] J. Moon, M. Kang, W. Lee, and S. Kim, "Salvaging Runtime Bad Blocks by Skipping Bad Pages for Improving SSD Performance," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 576–579.

[45] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. M. Khessib, and K. Vaid, "SSD Failures in Datacenters: What? When? and Why?" in *Proceedings of the 9th ACM International on Systems and Storage Conference, SYSTOR 2016, Haifa, Israel, June 6-8, 2016*, 2016, pp. 7:1—-7:11.

[46] S. Nath and P. B. Gibbons, "Online Maintenance of Very Large Random Samples on Flash Storage," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 970–983, 2008.

[47] S. Nath and A. Kansal, "FlashDB: dynamic self-tuning database for NAND flash," *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN)*, 2007.

[48] T. Neumann and M. J. Freitag, "Umbra: A Disk-Based System with In-Memory Performance," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2020.

[49] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1993, pp. 297–306.

[50] T. I. Papon and M. Athanassoulis, "A Parametric I/O Model for Modern Storage Devices," in *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, 2021.

[51] T. I. Papon and M. Athanassoulis, "The Need for a New I/O Model," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2021.

[52] H. Park and K. Shim, "FAST: Flash-aware external sorting for mobile database systems," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1298–1312, 2009.

[53] S.-Y. Park, D. Jung, J.-U. Kang, J. Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2006, pp. 234–241.

[54] S. Park and K. Shen, "A performance evaluation of scientific I/O workloads on Flash-based SSDs," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2009, pp. 1–5.

[55] S. Park and K. Shen, "FIOS: a fair, efficient flash I/O scheduler," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012, p. 13.

[56] PostgreSQL, "The Internals of PostgreSQL," *http://www.interdb.jp/pg/pgsql08.html*, 2015.

[57] PostgreSQL, "PostgreSQL: The pg_prewarm module," *https://www.postgresql.org/docs/current/pgprewarm.html*, 2016.

[58] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

[59] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 286–297, 2011.

[60] J. Seol, H. Shim, J. Kim, and S. Maeng, "A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2009, pp. 137–146.

[61] Z. Sha, Z. Cai, F. Trahay, J. Liao, and D. Yin, "Unifying temporal and spatial locality for cache management inside SSDs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE)*. IEEE, 2022, pp. 1–6.

[62] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, "United States Data Center Energy Usage Report," *Ernest Orlando Lawrence Berkeley National Laboratory*, vol. LBNL-10057, 2016.

[63] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013, pp. 67–78.

[64] Smartmontools, "Smart Monitoring Tools," *https://www.smartmontools.org/*.

[65] A. J. Smith, "Sequentiality and Prefetching in Database Systems," *ACM Trans. Database Syst.*, vol. 3, no. 3, pp. 223–247, 1978.

[66] A. S. Tanenbaum, *Modern Operating Systems*. Prentice-Hall, 1992.

[67] M. K. Tcheun, H. Yoon, and S. R. Maeng, "An adaptive sequential prefetching scheme in shared-memory multiprocessors," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 1997, pp. 306–313.

[68] TPC, "Specification of TPC-C benchmark," *http://www.tpc.org/tpcc/*, 2022.

[69] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174–199, 2000.

[70] S. D. Viglas, "Adapting the B +-tree for asymmetric I/O," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7503 LNCS, 2012, pp. 399–412.

[71] J. S. Vitter and P. Krishnan, "Optimal Prefetching via Data Compression," *J. ACM*, vol. 43, no. 5, pp. 771–793, 1996.

[72] K. Wu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Towards an Unwritten Contract of Intel Optane SSD," in *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2019.

[73] Y.-S. Yoo, H. Lee, Y. Ryu, and H. Bahn, "Page Replacement Algorithms for NAND Flash Memory Storages," in *Proceedings of the International Conference on Computational Science and Applications (ICCSA)*, ser. Lecture Notes in Computer Science, 2007, pp. 201–212.