Exploiting SSD Asymmetry and Concurrency for Storage-Intensive Applications

TARIKUL ISLAM PAPON, Boston University, USA
MANOS ATHANASSOULIS, Boston University, USA

Solid-state drives (SSDs) have been replacing the traditional hard disk drives (HDDs) in the past few decades because of their faster read and write speeds, along with superior random access performance. Unlike HDDs, SSDs exhibit two distinct characteristics: (i) *read/write asymmetry*, where writes are slower than reads, and (ii) *access concurrency*, allowing multiple I/O operations to run simultaneously and fully utilize device bandwidth. Despite these, most storage-intensive applications are not optimized for SSD asymmetry and concurrency, often leading to device underutilization. In this thesis, we uncover these crucial SSD properties and outline how we can better exploit these properties from the application perspective.

First, we augment the traditional I/O modeling approaches with the Parametric I/O Model (PIO), a storage model that faithfully represents storage devices by parameterizing read/write asymmetry (α) and access concurrency (k). Using this novel storage modeling, we propose a new ***A**symmetry & **C**oncurrency-awar**e*** bufferpool management (ACE) that batches writes based on device concurrency and performs them in parallel to *amortize* the asymmetric write cost. Further, 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 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. We further present a concurrency-aware graph processing engine CAVE that harnesses the parallelism supported by the underlying SSD device via concurrent I/Os. CAVE traverses multiple paths and processes multiple nodes and edges concurrently without altering the fundamental graph traversal algorithm guarantees. We apply this approach on five popular graph traversal algorithms: the ubiquitous Breadth-First and Depth-First Search algorithms along with three algorithms that use BFS as a building block (Weekly Connected Components, PageRank, and Random Walk). By experimenting with different types of graphs on three SSD devices, we demonstrate that CAVE utilizes the available parallelism, and scales to diverse real-world graph datasets while providing up to three orders of magnitude speedup in runtime compared to the popular out-of-core system GraphChi and up to one order of magnitude speedup compared to GridGraph. Overall, our analysis shows that more faithful storage modeling via incorporating asymmetry and concurrency in algorithm design leads to higher performance and better device utilization.

1 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 [29, 69, 81]. SSDs achieve their superior performance by adopting NAND flash memory as their storage medium [4], 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 [36, 45, 71]. Furthermore, SSDs exhibit a high degree of *internal parallelism* that can be harnessed to increase performance [12, 57]. 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 [12]. The exact level of **concurrency** (k) needed to saturate the device depends on the request type (read or write) and on the specifics of the device. 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 [17].

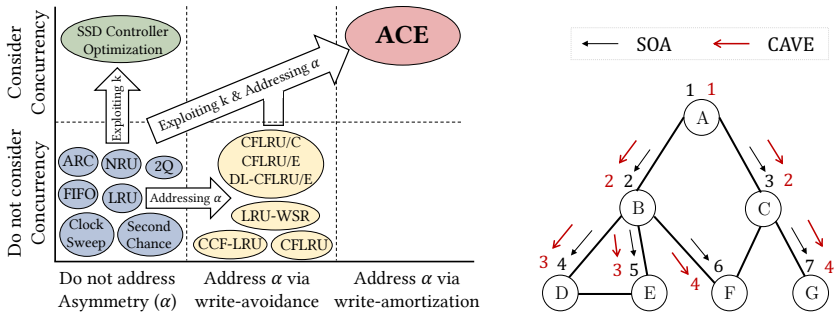


Fig. 1. (A) ACE addresses asymmetry by exploiting concurrency and amortizing writes. (B) The parallellized version of BFS in CAVE takes fewer iterations to converge.

The Parametric I/O Model. 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 [64–66]. This raises the need for a new I/O model [66] that can incorporate these device properties.

How should the I/O model be adapted in light of read/write asymmetry and concurrency?

We propose a simple yet expressive storage model termed Parametric I/O Model (PIO) [65] that considers *asymmetry* (α) and *concurrency* (k) as parameters. Using the device properties, this richer I/O model can accurately capture contemporary devices. We benchmark different types of state-of-the-art storage devices to quantify their α and k. Our abstract analysis reveal that *more informed storage modeling leads to better overall application performance*. However, many data-intensive systems have not been thoroughly redesigned to account for SSD properties. We identify two use cases where better storage modeling can enable better application performance.

Bufferpool Management. 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. We 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. (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 $\alpha = 1$). Figure 1(A) shows that popular page replacement policies are designed for devices with no asymmetry and concurrency (bottom left, blue). Recently proposed *flash-friendly* policies [34, 49, 73, 94] 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 [40, 79, 80] (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.

We propose ACE [67], a new bufferpool manager that utilizes the underlying device concurrency to bridge the device asymmetry (top right of Fig. 1(A) – 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. We integrate ACE with four page replacement policies and implement them in PostgreSQL to evaluate ACE's efficacy where we observe ACE can achieve upto 1.5× speedup.

Graph Management. With the unprecedented growth of interconnected data stemming from various applications like machine learning, recommendation systems, physical sciences, and social networks, analytics over large graphs is becoming increasingly popular in both academia and industry [31, 51, 56**?**]. Graph traversal operations can utilize SSD concurrency by parallelizing node and edge accesses, effectively distributing the workload across the SSD's parallel architecture [6]. This idea takes advantage of the availability of multiple paths that can be explored during graph traversal. While most out-of-core graph processing systems indirectly attempt to utilize the underlying storage parallelism by reducing random (in favor of sequential) I/O, they do not aggressively exploit opportunities for concurrent accesses. Our goal is to parallelize graph traversal algorithms without changing their core properties while utilizing the underlying SSD concurrency.

We build an SSD-aware graph processing system, named CAVE[1] [68] that is able to harness the *concurrency* of the underlying storage devices. Specifically, CAVE provides the necessary infrastructure to parallelize graph traversal algorithms when several independent accesses can be performed in parallel. A prime example is our Parallel Breadth-First Search (PBFS) implementation which is outlined in Figure 1(B). The algorithm accesses the next *wave* of nodes (as we move on a level-by-level fashion) in parallel since we have already identified the nodes of the next wave while processing the current one. This leads to a faster response time of the BFS search simply by carefully exploiting the underlying storage concurrency, resulting in faster convergence within fewer iterations. CAVE uses a block-based file format based on adjacency lists, ensuring that graph metadata, vertex information, and edge information are stored in aligned blocks. Furthermore, CAVE employs a concurrent cache pool mechanism that enhances locality and ensures thread safety. We develop in CAVE the parallelized versions of five popular graph algorithms. In addition to BFS, CAVE offers parallelized, SSD-aware versions of Depth-First Search (DFS), Weakly Connected Components (WCC), PageRank (PR), and Random Walk (RW). We CAVE with two popular out-of-core processing system GraphChi [43] and GridGraph [99], as they are widely recognized for their efficiency in handling large-scale graphs in a single machine. We observe that CAVE can be up to three orders of magnitude faster than GraphChi and up to one order of magnitude faster than GridGraph.

Contributions. Our contributions are as follows:

- We investigate the importance of the key characteristics of modern devices. We seek to answer the question: "*how much are we missing in terms of performance if we do not exploit concurrency and read/write asymmetry?*"
- We introduce the **Parametric I/O Model** (PIO) which considers both α and k. We show the benefits of adding these properties in our model with respect to device utilization and performance.
- 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.

[1] **CAVE**: **C**oncurrency-**A**ware Graph (**V**, **E**) system

- We propose **CAVE**, an SSD-aware graph engine that extracts the most benefit from the underlying SSD via concurrent I/O, its novel file structure, and a concurrent cache pool.
- We develop on CAVE the parallelized version of five popular graph algorithms (BFS, DFS, WCC, PageRank, Random Walk) to showcase that its flexibility to implement diverse graph algorithms.

2 BACKGROUND

We now discuss the necessary backgrounds: SSD properties, bufferpool page replacement algorithms, and graph traversal algorithms.

2.1 SSD Properties

Concurrency. SSDs exhibit a high **internal parallelism** in their architecture [11, 12, 57]. Fig. 2 shows that multiple channels are connected to the flash controller, and each channel consists of a shared bus with multiple chips. Each chip contains multiple dies, each die comprises multiple planes, and finally, each plane constitutes multiple blocks where the pages reside [4]. This highly parallelized architecture creates opportunities to efficiently support concurrent storage accesses [12, 57, 72, 82]. 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 on the access type and block size [65]. Most devices have a large number of channels ($\geq$ 8) which is the fundamental form of internal parallelism.

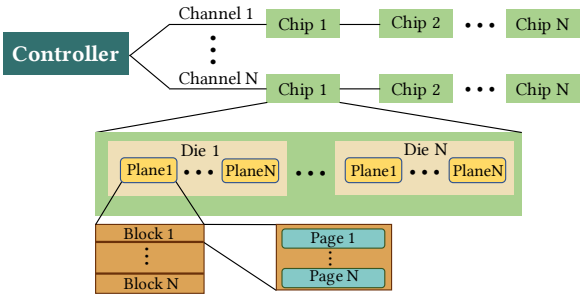


Fig. 2. Internal architecture of an SSD.

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 [8, 17, 61]. 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* [17, 71], i.e., once a page is written, it cannot be updated until that whole block is erased [3]. 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. 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.

2.2 DBMS Bufferpool Essentials

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* [75].

Page Replacement Algorithms. 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) [63] which tries to keep the most recently accessed pages in the bufferpool. Some other popular algorithms are Clock [39], NFU [86], 2Q [32], NRU [23], FIFO [86], ARC [59], and Second Chance [38]. PostgreSQL adopts the Clock Sweep algorithm [74], a variant of NFU. This algorithm and all the aforementioned algorithms do not differentiate between reads and writes. *Flash-friendly policies* like CFLRU, LRU-WSR, and others prioritize reads over writes to mitigate device wear caused by writes [34, 49, 73, 94] to address SSD wear-off. While they indirectly address asymmetry up to a point, they do not explicitly consider the specific device asymmetry and concurrency.

2.3 Graph Traversal Algorithms

Breadth-First Search (BFS). BFS is a graph traversal algorithm that starts from a designated starting vertex and then explores all neighboring vertices in a level-by-level manner [16]. It begins by visiting all the immediate neighbors of the starting vertex, then moves on to their neighbors in subsequent levels. By traversing the graph in a level-wise manner, BFS uncovers the shortest paths and analyzes the structural properties of the graph. Since BFS processes nodes in a level-by-level manner, nodes of the same level can be processed independently (hence concurrently), thus providing an opportunity for parallelizing and, in turn, harnessing the SSD's concurrency.

Depth-First Search (DFS). DFS is a widely-used graph traversal algorithm that starts from a specified vertex and systematically explores as deep as possible along each branch before backtracking [22]. DFS is particularly useful for tasks such as identifying cycles, determining connected components, and finding paths between vertices. While the classical DFS is tricky to parallelize, the pseudo DFS [1] algorithm offers the opportunity to parallelize by running multiple parallel mini-DFSs. A parallel version of pseudo-DFS can dynamically split and distribute the vertex stack among multiple threads, allowing concurrent exploration of different branches of the graph.

Weekly Connected Components (WCC). In an undirected graph, a connected component refers to a subgraph where every vertex is connected to every other vertex through pathways within the graph. WCC aims to identify and group together nodes that are weakly connected [41], meaning they can be reached from each other by traversing the edges regardless of their direction. This algorithm typically involves traversing the graph using techniques like BFS or DFS to identify the connected components. The previous approaches used to exploit SSD concurrency can be used to parallelize WCC. For example, while using BFS to discover WCCs, each subgraph's connected components can be computed concurrently, and the results from different subgraphs can be merged to determine the weakly connected components.

PageRank (PR). PR is a well-known algorithm to estimate the importance of vertices in graphs, originally developed by Google to rank webpages on the Internet [9]. It works by evaluating the importance of a web page based on the number and quality of links pointing to it. The algorithm assigns a numerical value, known as PR score, to each web page on the Internet and measures the importance of a web page based on its backlinks and the quality of those links. PR employs an iterative process. Initially, all pages are assigned an equal PR score. In each iteration, the scores are updated based on the scores of linking pages. This process continues until PR scores converge or after a certain number of iterations. Due to this iterative traversal nature, this algorithm can be parallelized, similar to BFS.

Random Walk (RW). RW is a probabilistic algorithm in which a walker moves through a network (graph), taking steps based on random choices [52]. It is used to analyze the network structure and understand properties such as connectivity and reachability. RW can be viewed as a Markov Chain, where the probability of transitioning to the next state depends only on the current state.

To accelerate RW, we can divide the graph into manageable subgraphs and simultaneously explore multiple nodes within these subgraphs. This approach accelerates the exploration and allows for parallelization of transition probability calculations, making it suitable for estimating node importance through RWs on vast networks. Further, different subgraphs can be processed in parallel while accounting for crossing into a different subgraph.

3 THE PARAMETRIC I/O MODEL

In this section, we present the **Parametric I/O Model** (PIO) [67], a new, simple yet expressive model that takes asymmetry and (read and write) concurrency as parameters. PIO enables better algorithm design and helps to accurately reason about the performance of storage-intensive algorithms and data structures.

> ***PIO*(M, k_r, k_w, α)** *assumes a fast main memory with capacity M, and storage of unbounded capacity that has* ***read/write asymmetry*** α*, and* ***read (write) concurrency*** k_r (k_w)*.*

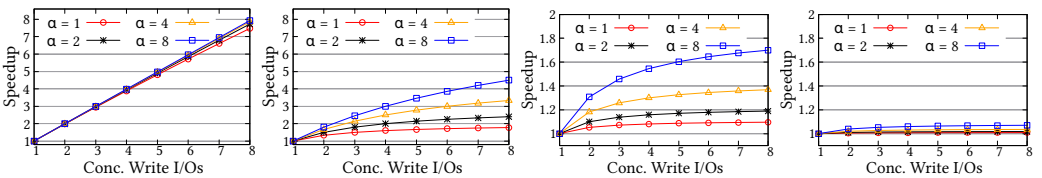
We consider that both memory and storage are divided into fixed-size blocks. Since the model is device-specific, the values of k_r, k_w, and α are either given by the device manufacturer, or by a careful benchmarking [65]. PIO allows us to accurately describe a variety of devices, and reason for their behavior at algorithm-design time. That way, we can make storage-aware optimizations part of the design, as opposed to applying them as an additional *ad hoc* tuning step during deployment. Next, we present how to use PIO to reason about the performance benefits of several fundamental classes of applications.

3.1 Performance Analysis

To analyze the performance of a storage-intensive application, we focus on its interaction with the storage device, that is, on the read and write requests it issues. We classify storage-intensive applications into two classes.

- *Unbatchable Reads, Batchable Writes.* An application that batches writes and utilizes the write concurrency of the device (*example*: concurrent eviction of dirty pages from a bufferpool).
- *Batchable Reads, Unbatchable Writes.* An application that batches reads and utilizes the read concurrency (*example*: concurrent traversal of multiple paths in a graph or in a tree index).

To maintain the generality of the approach, we quantify the performance gain using the frequency of reads (f_r) and writes (f_w), for which $f_r + f_w = 1$. We assume that read requests have unit cost (1) and write requests have cost α, where $\alpha \geq 1$. A device with read concurrency of k_r and write concurrency of k_w can concurrently perform k_r reads and k_w writes.



(A) 1% reads (B) 50% reads (C) 90% reads (D) 99% reads

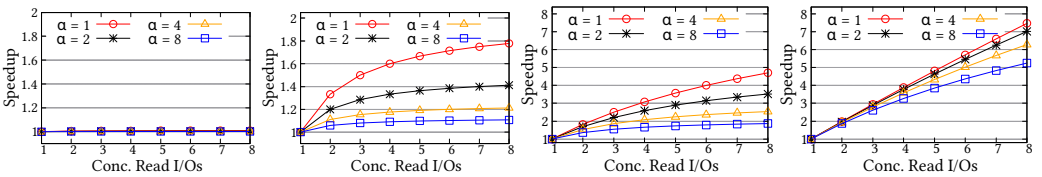
Fig. 3. Speedup of an *Unbatchable Reads, Batchable Writes* application. The speedup is highest for write-intensive workloads and it depends on the asymmetry – *the higher the asymmetry, the higher the speedup.*

Unbatchable Reads, Batchable Writes. This class of applications exploits the write concurrency of the device to batch write requests. As an example, consider a modified DBMS bufferpool manager that selects several dirty pages and writes them concurrently during an eviction. In this scenario,

the application at hand attempts to fully exploit the device's write concurrency via *concurrent flushing* during a page eviction. To achieve this, it submits k_w concurrent writes. Since the device has α asymmetry, the cost of a write following the standard approach of evicting one page at a time, as indicated by the EM model would be $C_W^{EM} = \alpha$. On the other hand, the *amortized* cost per write when we batch k_w writes following PIO is $C_W^{PIO} = \frac{\alpha}{k_w}$. Since reads are not batchable in this application class, each read will have unit cost in both the EM and PIO paradigm, hence, $C_R^{EM} = C_R^{PIO} = 1$. We can now calculate the speedup S_{PIO} of this application based on PIO:

$$S_{PIO} = \frac{f_r \cdot C_R^{EM} + f_w \cdot C_W^{EM}}{f_r \cdot C_R^{PIO} + f_w \cdot C_W^{PIO}} = \frac{f_r + f_w \cdot \alpha}{f_r + f_w \cdot \frac{\alpha}{k_w}} = \frac{k_w \cdot (f_r + f_w \cdot \alpha)}{k_w \cdot f_r + f_w \cdot \alpha} = 1 + \frac{(k_w - 1) \cdot f_w \cdot \alpha}{k_w \cdot f_r + f_w \cdot \alpha}$$

Since $\alpha \geq 1$ and $k_w \geq 1$, then $S_{PIO} \geq 1$ where the maximum value of S_{PIO} can be up to k_w. Fig. 3 shows the speedup when following PIO for different α and k_w values as we change the read/write ratio in the workload. We observe that the speedup increases as more concurrent I/Os are employed, which is expected. Furthermore, we note that the speedup depends on the asymmetry of the device. The *gain is **higher** for a device with **higher** asymmetry*. For example, for $f_r = 0.5$ and when fully exploiting the concurrency of $k_w = 8$ by issuing 8 concurrent I/Os, the speedup for a device with $\alpha = 8$ is 4.5× whereas the speedup for a device with $\alpha = 1$ is 1.78× (Fig. 3B). Since the application batches writes, the gain is maximized for a write-intensive workload (Fig. 3A), when the benefit from efficient writing is more pronounced. For a workload that contains *only* reads or *only* writes, the speedup from the PIO paradigm is the same irrespective of α. The key observation is that for an application with batchable writes, a higher asymmetry between expensive writes and cheap reads leads to a higher speedup.



(A) 1% reads (B) 50% reads (C) 90% reads (D) 99% reads

Fig. 4. Speedup of a *Batchable Reads, Unbatchable Writes* application. The speedup is highest for read-intensive workloads. *The lower the asymmetry, the higher the speedup.*

Batchable Reads, Unbatchable Writes. The second class of applications models scenarios where reads can be issued concurrently to exploit read concurrency, but not writes. As an example, consider a graph store that traverses multiple paths concurrently, thus can accelerate various algorithms including graph search and shortest path. Essentially, the algorithm can visit multiple nodes in parallel instead of one node at a time, and offer faster search time with the same worst-case guarantees. Another example is concurrent traversal of tree indexes [76]. The application performs k_r reads concurrently, thus, $C_R^{EM} = 1$ and $C_R^{PIO} = \frac{1}{k_r}$. The cost of a write request is $C_W^{EM} = C_W^{PIO} = \alpha$ for both paradigms since the writes are not batched. The speedup of this class of applications:

$$S'_{PIO} = \frac{f_r \cdot 1 + f_w \cdot \alpha}{f_r \cdot \frac{1}{k_r} + f_w \cdot \alpha} = \frac{k_r \cdot (f_r + f_w \cdot \alpha)}{f_r + k_r \cdot f_w \cdot \alpha} = 1 + \frac{(k_r - 1) \cdot f_r}{f_r + k_r \cdot f_w \cdot \alpha}$$

Since $\alpha \geq 1$ and $k_r \geq 1$, then $S'_{PIO} \geq 1$, and also $S'_{PIO} \leq k_r$. Fig. 4 presents the speedup of such an application based on PIO. Like before, the speedup increases as more concurrent I/Os are used. However, this time *the gain is **higher** for a device with **lower** asymmetry*. For instance, with $f_r = 0.5$ and $k_r = 8$, the speedup for a device with $\alpha = 1$ is 1.78× and it drops to 1.11× for $\alpha = 8$ (Fig. 4B). Note that, the overall speedup is lower than the previous application class because, while writes

are still more expensive than reads (for $\alpha > 1$), this class of applications can only utilize read concurrency. The speedup is maximized when the workload is read-heavy (Fig. 4A), showing the benefit of batching reads.

The above analysis reveals that the degree of the performance gain/loss depends on the asymmetry and the application type, whereas the gain is achieved through exploiting concurrency.

4 ACE BUFFERPOOL MANAGER

Using this novel storage modeling, we now propose a new Asymmetry & Concurrency-aware bufferpool management (ACE) [67] that batches writes based on device concurrency and performs them in parallel to amortize the asymmetric write cost.

4.1 An Augmented 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* , and it is a topic with significant prior work [23, 32, 34, 38, 39, 49, 59, 63, 73, 86, 94]. 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.

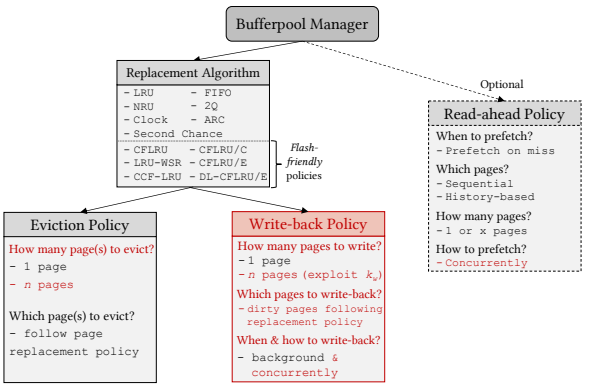


Fig. 5. Bufferpool design space in terms of the design decisions and various options (RED denotes new components)

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 *multiple 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.

4.2 Overview of ACE Bufferpool Management

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 6, 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. 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:

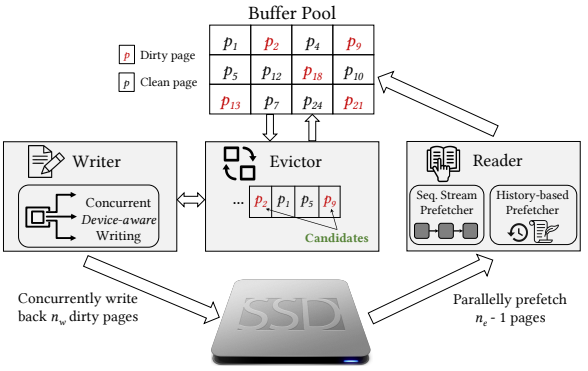


Fig. 6. Abstract overview of **ACE** components

- **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. Figure 7 shows the effect of incorporating ACE with LRU (Fig. 7a), CFLRU (Fig. 7b), and LRU-WSR (Fig. 7c). Note that ACE always writes n_w dirty pages concurrently irrespectively of prefetching. The full ACE algorithm is listed in Algorithm 1.

4.2.1 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).

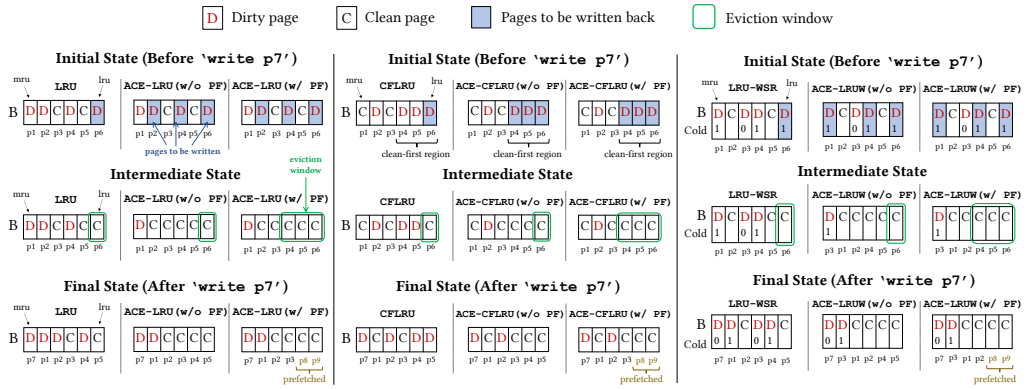


Fig. 7. ACE page selection policies for $n_w = 3$ and $n_e = 3$. (a) ACE writes three dirty pages (p6, p4, p2) following the LRU order; if prefetching is enabled three pages (p6, p5, p4) are evicted, otherwise one page (p6) is evicted. (b) Similarly for CFLRU, ACE writes three dirty pages (p6, p5, p4) 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.

4.2.2 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. 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 7a 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.

4.2.3 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.

4.2.4 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

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, ∀p ∈ P_wb
27             - mark ||length(P_wb)|| pages as clean, ∀p ∈ 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, ∀p ∈ 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
```

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.

4.3 Evaluation

We implement ACE in PostgreSQL 11.5 and now discuss 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 [65] (summarized in Table 1a). 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 $n_w = 8$.

Table 1. Experimental setup for ACE: Devices & Workloads

(a) 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

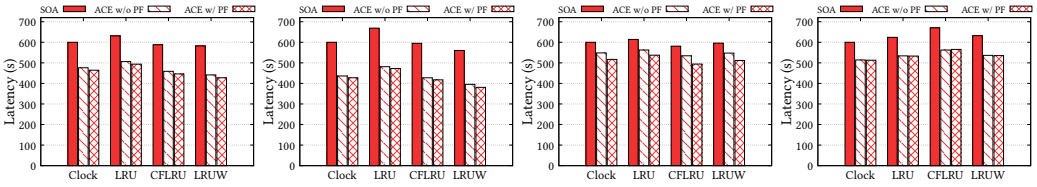
(b) Properties of the synthetic workloads

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

Workload. We use four synthetic workloads as outlined in Table 1b inspired by prior work [49, 94]. We refer to them as MS (Mixed Skewed), WIS (Write-Intensive Skewed), RIS (Read-Intensive Skewed) and MU (Mixed Uniform). 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 [88].

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 (~6% of the data size). WAL is enabled and the WAL file is written in a separate device following common practice.

ACE Improves Runtime without Any Penalty. 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



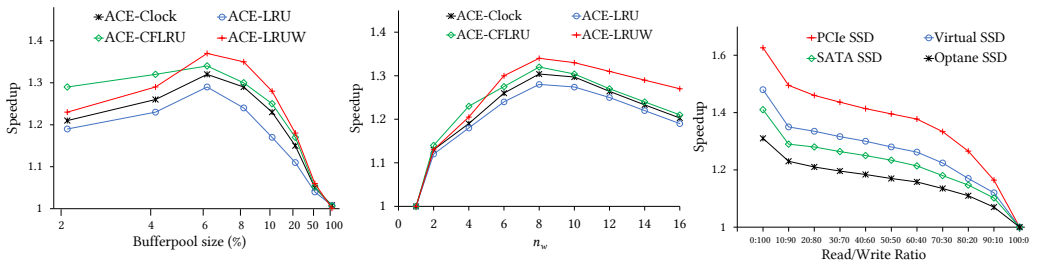
(A) ACE achieves high gain in the mixed workload MS.

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

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

(D) ACE 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.



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

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

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

Fig. 9. (A) ACE is beneficial across a wide range of bufferpool size w.r.t. data size. (B) Speedup increases as concurrent I/Os are increased until device gets saturated. (C) Higher asymmetry has higher gain for ACE.

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. We highlight that the overall workload latency improvement observed in these experiments (up to 32.1%) does not come at a hidden cost. 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.

Write-Intensive Workloads Have Higher Gains. ACE's 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 benefit of *efficient* writing is 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).

Higher Benefits Under Memory Pressure. ACE achieves higher speedup for smaller bufferpool size because a smaller bufferpool causes more evictions, hence more writes. Figure 9A show the impact of ACE under memory pressure for a mixed skewed workload like MS in the PCIe SSD.

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 speedup (and latency) 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×, while for 10%, the speedup is 1.25×.

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 9B. 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 because (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. We highlight that even with a small degree of concurrency ($n_w = 4$ or $n_w = 6$), the speed is substantial (1.2× − 1.3×).

Performance Gain Increases with Asymmetry. Our last experiment highlights that asymmetry impacts the attained gains of the ACE policies. 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 9C 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× (write-only workload) while the speedup is limited to 1.48×, 1.41× and 1.33× for the Virtual SSD ($\alpha = 2.0$), SATA SSD ($\alpha = 1.5$), Optane SSD ($\alpha = 1.1$), respectively.

Experimental Analysis with TPC-C Benchmark. We further use the TPC-C benchmark [88] 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, Payment, OrderStatus, StockLevel, Delivery.

ACE Accelerates TPC-C by 24%. 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 10 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×, 1.27×, 1.30× and 1.32× when implemented on top of Clock Sweep, LRU,

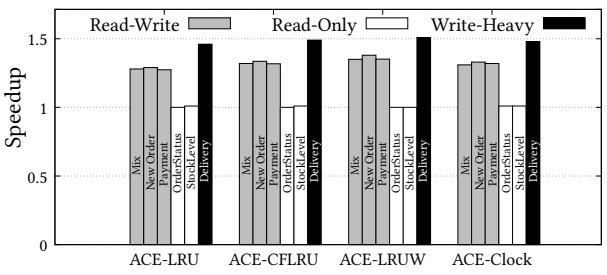


Fig. 10. ACE achieves high speedup for TPC-C mixed transaction, while benefiting the write-intensive transaction the most.

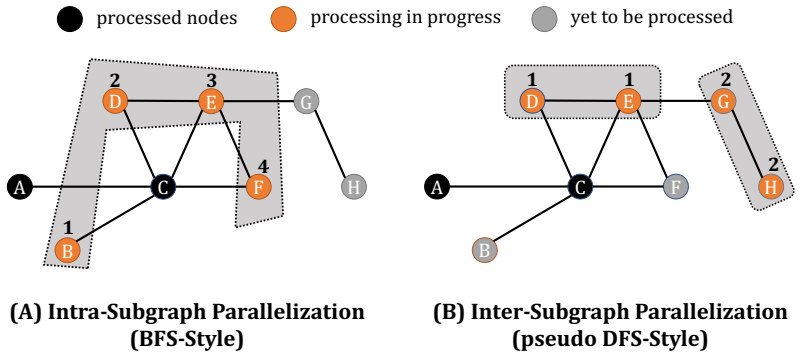


Fig. 11. Example of Intra/Inter-Subgraph Parallelization. (A) { B, D, E, F } are at the same level of BFS and are processed concurrently by 4 threads. (B) As pseudo DFS progresses, the stack is split and two subgraphs ({ D, E } and { G, H }), which are processed in parallel by 2 threads.

CFLRU, LRU-WSR, respectively. The highest speedup, 1.51×, 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.**

5 CAVE GRAPH MANAGER

We now present a concurrency-aware graph processing engine CAVE [68] that utilizes SSD parallelism via concurrent I/Os.

5.1 Parallelizing Graph Traversal Operations

Our objective is to (i) achieve efficient parallelization of graph traversal operations within out-of-core systems, and (ii) ensure that the core properties of the graph algorithms are maintained. In this section, we discuss how to achieve this with **intra-subgraph** and **inter-subgraph** parallelization. We present these two techniques with examples and discuss how they can be seamlessly integrated and leveraged alongside SSD parallelization.

5.1.1 Intra-Subgraph Parallelization. For this approach, we identify subgraphs, the nodes of which can be processed independently so that we can access them in parallel. This means that the processing of one node does not depend on the results or state of other nodes outside the subgraph. Thus, multiple nodes within the subgraph can be processed concurrently by different computing units (threads), allowing for concurrent I/Os, leading to better device utilization. After processing their respective nodes, the results obtained by each thread are aggregated or combined to produce the final result of the algorithm. This ensures efficient efficient exploitation of the underlying device which can speed up the execution of graph processing tasks.

Example. A prime example of this type of parallelization is a *parallel BFS*. BFS explores the graph level by level, where each level represents a set of equidistant vertices from the source vertex. Since vertices of the same level can be accessed independently of each other, all vertices within the same level can be processed concurrently, and thus accessed in parallel using multiple threads. A queue maintains the nodes to be visited next, which are ordered on a per-level basis. Each thread dequeues nodes from the shared queue and processes them independently. The edges of each node

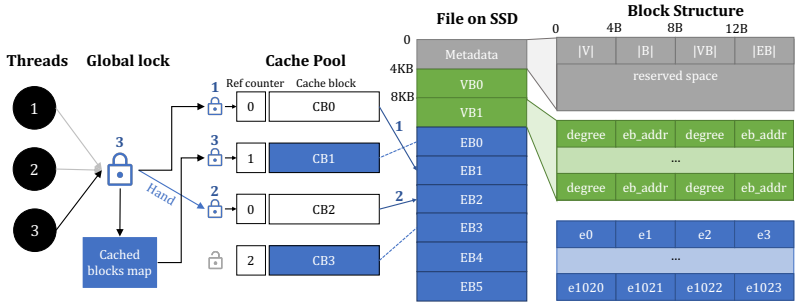


Fig. 12. CAVE's Architecture: block-based file structure (right side) and concurrent cache pool (left side)

are accessed from the underlying SSD concurrently. Figure 11(A) illustrates the application of this technique for parallelizing the BFS algorithm. Once nodes A and C have been traversed, nodes B, D, E, and F are all at the same level (a subgraph where nodes are independent), enabling them to be processed concurrently. Other BFS-based algorithms (e.g., PageRank, WCC) can also be parallelized with this approach as a building block.

5.1.2 Inter-Subgraph Parallelization. The subtle difference between Inter-Subgraph and Intra-Subgraph Parallelization is that it identifies subgraphs that can be independently accessed (like two different branches of DFS) and processes them in parallel. That way, multiple subgraphs (or paths) can be traversed concurrently, thus covering the entire graph faster and allowing for faster convergence. The algorithmic correctness and other properties (like the order of accessing nodes) can be ensured by communication and synchronization between the threads processing independent subgraphs. This approach is particularly useful for large-scale graphs that cannot fit entirely in memory or when distributing the computation across multiple threads.

Example. We now use as an example the *pseudo-DFS* [1]. In the classical DFS algorithm, a stack keeps track of nodes to be explored and maintains the visiting order. In the pseudo-DFS algorithm, the stack can be split into smaller stacks when its size goes beyond a predefined threshold, and the smaller stacks are processed in parallel. This allows for multiple threads to work on different subgraphs (e.g., paths) concurrently. Figure 11(B) shows an example of this approach. In this pseudo-DFS example, after traversing nodes A and C, the stack size grows to four and (assuming this is the threshold) is split in two. The first stack contains nodes D and E, while the second contains G and H. These smaller stacks are processed in parallel, leading to two independent graph traversals with the additional need for communication to avoid crossing from one subgraph (path) to another.

5.2 CAVE Physical Data Layout

CAVE uses a memory-mapped binary file format, with three main parts: the metadata block, the vertex block, and the edge block – right part of Figure 12. They are stored using 4KB aligned blocks to support direct reading and writing from/to the SSDs.

Metadata Block. The metadata block serves as a repository for essential graph information such as the number of vertices, the total number of blocks, edge blocks, and vertex blocks, each of which is stored as a 32-bit integer. The remaining space is reserved for future utilization, allowing for additional usage-specific information to be incorporated when necessary.

Vertex Block. Each vertex block, sized at 4KB, stores information about up to 512 vertices. Within each vertex, 8 bytes are allocated, encompassing two 32-bit unsigned integers: $degree$, eb_addr (edge block index and offset). The low 10-bit of eb_addr represents the offset eb_offset inside of an edge

block, which just fits its capacity of 1024. The high 22-bit states the index of the edge block eb_idx. The reading process can start by calculating the appropriate address $((eb\_idx \cdot 4\text{KB}) + eb\_offset)$.

Edge Block. We utilize a compact representation of edges where each edge is represented by a 4-byte integer denoting the index of the ending vertex. Hence, each edge block can store up to 1024 edges (adding up to 4KB). The edges of vertices with a degree less than 1024 are contained within a single edge block (note that in many datasets, most nodes have indeed a degree of less than 1024). This ensures efficient single read I/O access, while the starting index inside the block (eb_offset) can vary. However, vertices with a degree over 1024, will occupy multiple edge blocks. In this case, the first block always has an $eb\_offset = 0$ to simplify the packing and subsequent reading process. The number of edge blocks per vertex is given by its degree divided by 1024.

5.3 Concurrent Graph Traversal Algorithms

The core idea of our approach is to implement parallel graph algorithms that take advantage of concurrency at the storage level. Our system, CAVE, identifies and parallelizes independent I/Os, similar to how out-of-order processors parallelize load and store commands that are not dependent on each other. This enables parallel graph data processing, allowing multiple nodes to be accessed simultaneously, thus significantly reducing the number of iterations required. We carefully tune CAVE to employ the *optimal concurrency* [65] for the underlying storage devices to guarantee maximum benefit. To demonstrate the benefits of our approach, we parallelize five of the most common graph traversal algorithms: BFS, WCC, PageRank, Random Walk, and DFS. We implement two variants for BFS, WCC, PageRank and Random Walk.

***ProcessQueue* function.** In the context of BFS, WCC, PR, and RW algorithms, the parallelization process is structured as an iterative procedure. Each iteration involves processing a list of vertices (known as the $frontier$), accessing the neighbors of each vertex, updating vertex values, and determining which vertices should be visited in the next iteration, which are stored in the *next* queue. This iterative process can be naturally parallelized by having multiple threads working on individual vertices of the $frontier$ (intra-subgraph parallelization). We achieve this using a *ProcessQueue* function, which takes the $frontier$, a user-defined *process* function and the device's read concurrency k_r as parameters. The *process* function specifies the actions the algorithm should perform for each vertex and its neighbors. The *ProcessQueue* function parallelizes at the vertex level based on the k_r value where each thread is responsible for processing a vertex and executes a *getEdge* operation to retrieve the edge block from the cache pool. Since each edge block stores neighbors of multiple vertices, it is possible that an edge block swapped out from the cache will need to be read again from the disk, especially when the cache size is limited.

***ProcessQueueBlock* function.** To avoid multiple accesses of the same edge blocks, we provide a new variation that processes data at the granularity of edge blocks to benefit from caching. Initially, all edge blocks associated with vertices in the $frontier$ are found. Next, each thread is assigned to work on one of the edge blocks. That block, in turn, may contain (i) the edges of a single vertex where the execution will be the same as before, or (ii) the edges of multiple vertices where the processing of those vertices will now be completed with a single I/O. By simultaneously processing all vertices connected to a specific block, the approach ensures that each edge block is only read once in each iteration. While this strategy involves some overhead in terms of preprocessing the $frontier$, it offers the advantage of being minimally impacted by the size of the cache. Further, the edge block retrieval is performed concurrently, which contributes to its superior runtime performance. The two building-block algorithms are outlined in Algorithm 2.

5.3.1 Parallel Breadth-First Search. We develop a parallel BFS (PBFS for short) algorithm using two queues: the $frontier$ queue that contains the indices of vertices in the current level and the *next*

Algorithm 2: Parallelization Building Blocks

```
1:  function ProcessQueue(frontier, Func Process, k_r)
2:      next ← ∅
3:      // Process vertices in parallel with max k_r threads
4:      for v_1 in frontier do
5:          // Read neighbors from the cache pool
6:          neighbors ← GetEdges(v_1)
7:          // Process v_1 with its neighbors, get next_v queue
8:          next_v ← Process(v_1, neighbors)
9:          mtx.lock()
10:         next.insert(next_v)
11:         mtx.unlock()
12:     end for
13:     return next
14: end function
15: function ProcessQueueBlock(frontier, Func Process, k_r)
16:     block_set ← HashSet()
17:     for v_1 in frontier do
18:         block_idx ← GetBlockIdx(v_1)
19:         block_set.Insert(block_idx)
20:         block[block_idx].Insert(v_1)
21:     end for
22:     next ← ∅
23:     // Process blocks in parallel with max k_r threads
24:     for block_idx in block_set do
25:         // next queue of this block
26:         next_b ← ∅
27:         // Read edge block
28:         block_data ← GetBlock(block_idx)
29:         // For each v_1 associated with this edge block
30:         for v_1 in block[block_idx] do
31:             // Get v_1 neighbors from this block locally and process
32:             neighbors ← ReadFromBlock(block_data, v_1)
33:             next_v ← Process(v_1, neighbors)
34:             next_b.insert(next_v)
35:         end for
36:         // Merge next_b in final next queue
37:         mtx.lock()
38:         next.Insert(next_b)
39:         mtx.unlock()
40:     end for
41:     return next
42: end function
```

queue to store the indices of the neighbors of vertices in the *frontier* queue, which correspond to the vertices in the next level. To leverage parallelism, each vertex in the *frontier* queue is

Algorithm 3: Parallel Breadth-first Search

```
1:  function BFSPROCESS(v_1, neighbors)
2:      next_v ← ∅
3:      for v_2 in neighbors do
4:          if visited[v_2].CAS(False, True) then
5:              // Add v_2 to the next queue of v_1
6:              next_v.INSERT(v_2)
7:          end if
8:      end for
9:      return next_v
10: end function
11:
12: function PBFS(v_s, k_r)
13:     frontier ← {v_s}
14:     vertices_count ← 0
15:     while frontier.SIZE > 0 do
16:         next ← PROCESSQUEUE(frontier, BFSprocess, k_r)
17:         // Or call ProcessQueueBlock()
18:         vertices_count ← vertices_count + frontier.SIZE
19:         frontier ← next
20:     end while
21:     return vertices_count
22: end function
```

assigned to a separate thread so that multiple I/Os can be issued in parallel as shown in Figure 1(A). The complete algorithm is listed in Alg. 3. For each vertex in $frontier$, as *BFSprocess* defines, *ProcessQueue* will assign threads to vertices. Each thread accesses the assigned vertex, retrieves the indices of its neighbors, checks and flags the index of every neighbor as $visited$, inserts it to $next_v$ queue of this vertex, and merges $next_v$ of all vertices to the final *next* protected by a global lock mtx to prevent data races and ensure thread safety. The PBFS level of concurrency is controlled by the number of threads, which we tune according to the physical specification of the SSD (i.e., optimal concurrency). Once all the vertices in the $frontier$ queue have been processed, the contents of the *next* queue are copied back to the $frontier$ queue, and the *next* queue is cleared. This process is repeated until the $frontier$ queue becomes empty, signifying the completion of the BFS traversal. We also developed a *blocked* variant of the $frontier$ processing which uses the *ProcessQueueBlock* function. As discussed in §5.3, this approach discovers the edge blocks of the $frontier$ vertices and allocates threads to edge blocks, parallelizing at the edge block level while ensuring that each edge block is read only once. This results in two benefits: (i) overall runtime improvement since edge blocks are not read multiple times, and (ii) performance does not depend on cache pool size.

5.3.2 Parallel Weakly Connected Components. Computing WCC entails repeatedly searching from each vertex in the graph. Since we utilize the adjacency list format, the most efficient approach to computing WCC involves repeatedly applying the search algorithm starting from each vertex in the graph. During the search process, a visited vertex is marked as *true* and avoided in subsequent iterations. We parallelize WCC by performing multiple concurrent searches using PBFS due to its low overhead and well-established efficiency. Due to space constraints, we do not include the complete algorithm.

Algorithm 4: Parallel PageRank

```
1:  function PRprocess(v_1, neighbors)
2:      pr_next[v_1] ← 0
3:      for v_2 in neighbors do
4:          // Sum up last pr value of neighbors
5:          pr_next[v_1] ← pr_next[v_1] + pr[v_2]
6:      end for
7:      // Add damping factor and divide by its degree
8:      pr_next[v_1] ← (d+(1−d)·pr_next[v_1]) / GetDegree(v_1)
9:      return ∅
10: end function
11:
12: function ParallelPageRank(iterations, k_r)
13:     frontier ← {0, 1, ..., N − 1}
14:     for i ← 0 to N − 1 do
15:         pr[i] ← 1 / GetDegree(i)
16:         pr_next[i] ← pr[i]
17:     end for
18:     // We run it for a number of iterations
19:     while iterations > 0 do
20:         ProcessQueue(frontier, PRprocess, k_r)
21:         pr ← pr_next
22:         iterations ← iterations − 1
23:     end while
24:     return pr
25: end function
```

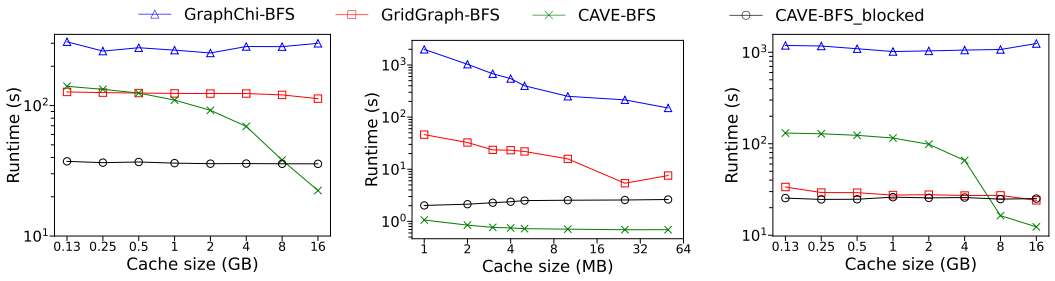
5.3.3 Parallel PageRank. We consider the topology approach for PR, which involves updating the PR values (pr) of all vertices based on the values of their neighbors from the previous iteration (Algorithm 4). Since all vertices need to be processed in each iteration, the $frontier$ queue always contains the entire list of vertices, and there is no need for a *next* queue. Initially, the $frontier$ queue includes all vertices, from vertex 0 to vertex $N-1$. In every iteration, the *ProcessQueue* is called with the desired concurrency to parallelize each step of the algorithm. For the blocked implementation, the *ProcessQueueBlock* function is called. The initial PageRank values, $pr[i]$ and $pr_{next}[i]$, are assigned as the inverses of the degrees of their respective vertices v_i. It is worth noting that in the original PageRank algorithm, the initial PageRank value for each vertex is set to 1, and its neighbors are assigned values of $\frac{pr[i]}{deg[i]}$. To optimize the computation, we perform this division in advance so it does not need to be repeatedly calculated by the neighbors in each iteration.

5.3.4 Parallel Random Walk. A single random walk is inherently a serial process and does not significantly benefit from data concurrency. However, an effective strategy is to run multiple random walks concurrently, which not only improves the precision of the results but also reduces the overall running time. Initially, k vertices are randomly chosen from the whole vertex set and put in the $frontier$ queue. In each iteration, the *RWprocess* function randomly selects one of the neighbors for each vertex in $frontier$ as the successor in the next iteration. Due to space constraints, we do not include the complete algorithm.

Algorithm 5: Parallel Pseudo Depth-first Search

```
1:  function DFSTask(stack, k_r)
2:      max_stack_count ← k_r
3:      while stack.size() > 0 do
4:          // Get and pop vertex at the stack top
5:          v_1 ← stack.top()
6:          stack.pop()
7:          visited_count ← visited_count + 1
8:          neighbors ← GetEdges(v_1)
9:          // Push all unvisited neighbors on stack
10:         for v_2 in neighbors do
11:             if visited[v_2].CAS(False, True) then
12:                 stack.push(v_2)
13:             end if
14:         end for
15:         // Check if stack size is larger than threshold
16:         while stack.size() > max_stack_size do
17:             if stack_count < max_stack_count then
18:                 stack_count ← stack_count + 1
19:                 // Split the stack and generate new task
20:                 new_stack, stack ← stack.split()
21:                 ThreadPool.push(DFStask, new_stack)
22:             end if
23:         end while
24:     end while
25:     stack_count ← stack_count − 1
26: end function
27:
28: function ParallelPseudoDFS(v_s, k_r)
29:     init_stack ← {v_s}
30:     visited[v_s] ← True
31:     stack_count ← 1
32:     visited_count ← 0
33:     ThreadPool.push(DFStask(init_stack, k_r))
34:     ThreadPool.WaitAll()
35:     return visited_count
36: end function
```

5.3.5 Parallel Pseudo Depth-First Search. While DFS is inherently a serialized algorithm, it is possible to enhance its performance by introducing parallelism through a technique known as *unordered* or *pseudo-DFS* [1]. We take inspiration from this idea, and we incorporate a mechanism to monitor the size of the vertex stack for each thread in our implementation (Algorithm 5). In the beginning, only one stack is active with the starting vertex v_s. We create a new *DFStask* with this stack in the thread pool. The *DFStask* continuously pops the stack, reads its neighbors, and pushes them into the stack as a normal DFS does. After visiting the neighbors of a vertex, we check if the size of the stack exceeds a predefined threshold. If it does, the stack is evenly



(A) Dataset: FS (B) Dataset: RN (C) Dataset: SD

Fig. 13. (A) - (C) Performance graph for BFS on our PCIe SSD. CAVE outperforms GraphChi and GridGraph.

divided into two smaller stacks, and one of these stacks is assigned to a new thread for further exploration. This approach allows each thread to independently perform DFS on its allocated stack and split it when necessary. By dynamically splitting the stacks in this manner, we achieve increased concurrency during the DFS traversal. The choice of the threshold value determines the trade-off between concurrency and thread creation overhead. Setting a smaller threshold allows for higher concurrency but may result in a larger number of threads being created. On the other hand, a larger threshold reduces the number of thread creations but may limit the degree of parallelism. The selection of an appropriate threshold is crucial to strike a balance between concurrency and overhead, ensuring efficient execution of the algorithm.

5.4 Evaluation

We now present the experimental evaluation of CAVE for the five algorithms and compare it with two storage-optimized graph processing system GraphChi [43] and GridGraph [99].

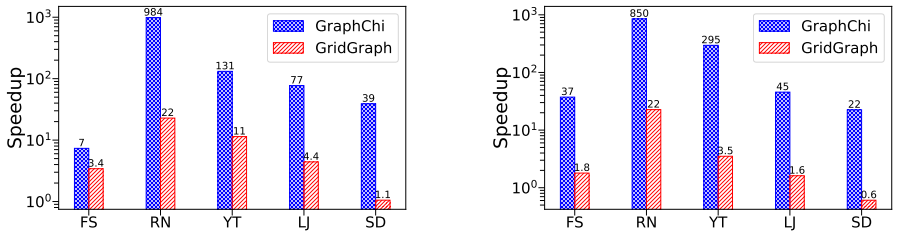
Experimental Setup. We use the same server and storage devices described in Section 4.3. Unless otherwise mentioned, we match the number of concurrent I/Os to k_r of the corresponding device for optimal device utilization [65]. All experimental results are averaged over three iterations, and the standard deviation was less than 1%.

Table 2. Dataset Description

Dataset	Description	#Nodes	#Edges	Diameter	Size
FS	Friendster Social Network	65M	1.8B	32	32 GB
RN	RoadNet Network of PA	1M	1.5M	786	47 MB
LJ	LiveJournal Social Network	5M	69M	16	1 GB
YT	YouTube Social Network	1.1M	3M	20	39 MB
SD	Synthetic data	50M	1.25B	6	42 GB

Dataset. We use four datasets of different sizes and types from the Stanford Large Network Dataset Collection [46]: Friendster Social Network (FS), RoadNet Network of PA (RN), LiveJournal Social Network (LJ) and YouTube Social Network (YT). FS is the largest dataset among these with 65M nodes and 32GB size. We also experiment with a synthetic dataset (SD) generated by us which has 50M nodes and 42GB size. The basic properties of the datasets are presented in Table 2. Due to space limit, while we conduct a comprehensive set of experiments across 5 algorithms, 3 devices, and 5 datasets, we present only the most noteworthy findings mainly focusing BFS and DFS.

CAVE outperforms GraphChi & GridGraph. We evaluate the performance of CAVE, GraphChi and GridGraph as we vary the cache size for three datasets. Figures 13(A) - (C) show the performance of the three systems for BFS when the underlying device is the PCIe SSD. Since the datasets have different sizes, the cache value is set accordingly. The results show that CAVE (both blocked and



(A) CAVE's speedup for PBFS (B) CAVE's speedup for PWCC

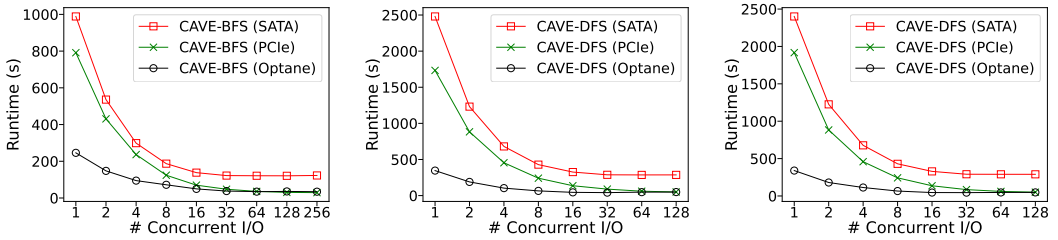
Fig. 14. CAVE performs well across all datasets for both PBFS and PWCC. It performs particularly well for sparse graphs (RN dataset). On the other hand, GridGraph works best for dense graphs (SD dataset)

non-blocked) significantly outperforms GraphChi for any cache ratio and any dataset. For example, Figure 13(A) presents a performance comparison of the BFS implementation of the three systems for the Friendster dataset (65M nodes, 32GB size). The non-blocked implementation benefits from a higher cache size while the blocked implementation remains unaffected by the cache size. This is due to the design techniques of the blocked implementation as it ensures all edge blocks are read only once during an iteration. We observe that the blocked CAVE implementation of BFS provides up to 8.3× speedup compared to GraphChi. GridGraph is faster than GraphChi, however, both CAVE implementations outperform GridGraph. The blocked version provides up to 3.4× speedup across various cache sizes, while the non-blocked variant delivers comparable performance for smaller cache sizes and up to 5.14× speedup for higher cache sizes.

CAVE is Well-suited for Sparse Graphs. In Figure 13(B) we see that in a sparse graph with unusually high diameter (RN dataset), CAVE performs better than both GraphChi and GridGraph. In particular, the non-blocked implementation works well for this graph because, in sparse graphs with lower average degrees where edge blocks can be associated with multiple vertices, all required edge blocks for an iteration can fit in the cache pool without swapping. With a sufficiently large cache, the non-blocked variant benefits from multiple threads working on cache data. Figure 13(B) shows that the blocked implementation (black line) outperforms both GraphChi and GridGraph while the non-blocked implementation (green line) can be up to 3.8× faster than the blocked implementation.

CAVE Provides Good Performance for Dense Graphs. Our synthetic SD dataset is an unusually dense graph (the diameter is only 6 with 50M nodes and 1.25B edges). Figure 13(C) shows that CAVE-blocked provides marginal benefit compared to GridGraph for the SD dataset. The reason behind this is that GridGraph is designed for dense graphs because its data structures and algorithms are optimized to efficiently handle the high connectivity and dense nature of such graphs. GridGraph achieves this by utilizing a grid structure to partition and manage the graph's data.

CAVE Excels Across Different Datasets. A summary of the results is presented in Figure 14(A) which shows CAVE's speedup compared to GraphChi and GridGraph for all five datasets for a specific cache size depending on the dataset on the PCIe SSD device (around 3% for each workload). The speedup of CAVE's BFS compared to GraphChi ranges from $7 - 984\times$ while the speedup compared to GridGraph ranges from $1.1 - 22\times$. The high run time for the RN dataset in Figure 13(B) and the unusually high speedup for this dataset shown in Figure 14(A) is attributed to the high diameter of the graph. For dense graphs like SD, GridGraph performs well because of its grid structure to partition and manage dense graphs, however, CAVE still outperforms GridGraph. Figure 14(B) presents a summary result of WCC for all five datasets on the PCIe SSD (around 3% for each workload) which shows CAVE can achieve $22 - 850\times$ speedup compared to GraphChi and $0.6 - 22\times$ speedup compared to GridGraph. GridGraph performs better than CAVE only for the SD dataset.



(A) Dataset: FS, Algorithm: BFS (B) Dataset: FS, Algorithm: DFS (C) Dataset: SD, Algorithm: DFS

Fig. 15. (A) As number of concurrent I/Os increases, the benefit of CAVE increases until the device gets saturated. (B, C) CAVE's PDFS can attain the maximum benefit of the device by exploiting its *optimal concurrency* value.

CAVE Utilizes Concurrent I/Os. To analyze how the concurrent I/O affects the performance for various devices and datasets for the algorithms, we now vary the number of concurrent I/Os in the blocked PBFS implementation. Since GraphChi or GridGraph does not have the support of varying concurrent I/Os, we do not include them in this experiment. Figure 15(A) shows CAVE's performance graph for the FS dataset for all three of our devices. The figure shows that as we increase the number of concurrent I/Os, the runtime of PBFS decreases until the device gets saturated. For example, the SATA SSD gets saturated when using 16-32 concurrent I/Os (red line), which is consistent with the device's optimal concurrency value (25). However, if we issue more concurrent I/Os, performance starts to degrade because of the thread management overhead while the device is already saturated. The PCIe SSD curve is moved towards higher concurrency, as expected, and Optane SSD has a flatter curve, which is consistent with prior work [65, 67].

CAVE's PDFS Exploits Device Concurrency. We now focus on the impact of I/O concurrency on the PDFS algorithm. We generate a list of target keys at random and run the algorithm to search each key multiple times in a depth-first manner. It is worth mentioning that, GraphChi, GridGraph and most graph processing systems do not support DFS. Figures 15(B) and (C) show the performance of CAVE's PDFS as we vary the number of concurrent I/Os for the FS, and SD datasets across three devices. We observe that for all datasets and devices, as we increase the number of concurrent I/Os, we have improvements in runtime until the device performance plateaus. For example, for the FS dataset, CAVE's PDFS achieves 7.7×, 12.6×, 7.6× speedup on the Optane SSD, SATA SSD, and PCIe SSD, compared to using sequential I/Os (one I/O at a time) as shown in Figure 15(B). We can also reason about each device's concurrency values from the graphs as the runtime flattens out when the device bandwidth is saturated. The figures also show that the fastest device (Optane SSD) has a much lower runtime than the slowest device (SATA SSD). Overall, these experiments show that CAVE can perform parallel pseudo-DFS while leveraging the underlying SSD's concurrency.

6 RELATED WORK

Existing I/O Models. External storage has been traditionally modeled as a simple collection of blocks following the simplicity of the design of a hard disk (EM Model [2]). Blelloch et al. [7, 8] proposed the *Asymmetric RAM* (ARAM) model to analyze algorithms for asymmetric read and write costs, targeting asymmetric non-volatile main memory devices. The main goal of ARAM is to develop write-efficient main memory algorithms. On the contrary, the goal of PIO is to capture the inherent asymmetry and concurrency of *storage devices*, and study how we can use these in the design process of *storage-intensive algorithms*.

Addressing Asymmetry and Concurrency. Read/write asymmetry has been identified as an optimization goal for indexing [5, 14, 15, 48, 50, 89], flash-aware storage engines [10, 35, 62], and

other data management operations [20, 21, 28, 70]. Recent research has focused on developing new I/O schedulers for SSDs [57, 72, 76, 82] and on modifying SSD internals to exploit the parallelism [11, 12]. 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 [23, 32, 38, 39, 59, 63, 86]. 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 [34, 49, 73, 94]. Other flash-friendly policies, like FOR and FOR+ [54] 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 [40, 79, 80]. 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.

Prefetching. The most popular prefetching technique is sequential prefetching [19, 47, 84, 87] which is adopted by many commercial systems. Stride-based prefetching is also widely studied primarily for processor caches [24, 44, 60]. History-based prefetching techniques attempt to predict future access patterns based on past access patterns by using history-based table [27, 47], Markov predictor [33], and data compression techniques [18, 90]. In the augmented design space that we propose, any prefetching technique(s) can be integrated.

Graph Processing Systems. Many scalable graph processing systems like PowerLyra [13], PowerGraph [25], GraphX [26], GBase [37], TurboGraph++ [42], Chaos [77], GraphLab [53], Pregel [56], Gemini [98] can process large graphs in a distributed manner which requires finding optimal partitioning, load-balancing, fault tolerance, and managing the communication overhead. There are some single-node shared-memory systems like Ligra [83], GraphMat [85], GRACE [93], Polymer [95], CGraph [96] that process graphs in memory, and as expected, these systems are highly CPU bound. There are several popular out-of-core processing systems including GraphChi [43], TurboGraph [30], Mosaic [55], X-Stream [78], GridGraph [99], Graspan [91], RStream [92], and FlashGraph [97]. These systems attempt to minimize random disk access while relying on sequential I/O, extensive preprocessing, and optimal data placement. GraphSSD [58] is a graph semantic aware SSD framework where the SSD controller is made aware of the graph data structures stored on the SSD, hence, this approach requires extensive modification of the SSD controller. In contrast, our goal is to develop a general approach for parallelizing graph traversal algorithms and develop the necessary infrastructure to exploit *SSD concurrency*.

7 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. We propose a simple yet expressive parametric I/O model, termed PIO, that considers the asymmetry (α) between reads and writes, and concurrency (k) that different devices may support to enable better algorithm design. By capturing α and k, device-specific decisions can be tuned at both algorithm design time and during deployment and testing. Inspired from PIO, we propose ACE, a novel asymmetry/concurrency-aware bufferpool manager paradigm that batches writes based on device concurrency to amortize the asymmetric write cost. We refactor the bufferpool design space by separating the eviction policy from the write-back policy. 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.

Further, we propose CAVE, a concurrency-aware graph processing system designed to leverage the underlying SSD concurrency. CAVE parallelizes independent I/Os through its concurrent cache pool design, supported by its file structure, enabling the implementation of storage-aware parallel graph algorithms. In both cases, we observe from extensive experimental evaluations that better storage modeling leads to better device utilization and, ultimately, better performance.

REFERENCES

[1] Umut A Acar, Arthur Charguéraud, and Mike Rainey. 2015. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 67:1—-67:12. https://doi.org/10.1145/2807591.2807651

[2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. https://doi.org/10.1145/48529.48535

[3] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 147–160. http://dl.acm.org/citation.cfm?id=1376616.1376634

[4] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 57–70. http://research.microsoft.com/pubs/63596/usenix-08-ssd.pdfhttp://dl.acm.org/citation.cfm?id=1404019

[5] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1881–1892. http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf

[6] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. 2012. Path Processing using Solid State Storage. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 23–32. http://www.adms-conf.org/athanassoulis_adms12.pdf

[7] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. 2015. Sorting with Asymmetric Read and Write Costs. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1–12. https://doi.org/10.1145/2755573.2755604

[8] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. 2016. Efficient Algorithms with Asymmetric Read and Write Costs. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*. 14:1–14:18. https://doi.org/10.4230/LIPIcs.ESA.2016.14

[9] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30, 1-7 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[10] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD Bufferpool Extensions for Database Systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1435–1446. http://dl.acm.org/citation.cfm?id=1920841.1921017

[11] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage (TOS)* 12, 3 (2016), 13:1–13:39. https://doi.org/10.1145/2818376

[12] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. 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)*. 266–277. https://doi.org/10.1109/HPCA.2011.5749735

[13] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2018. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 5, 3 (2018), 13:1—-13:39. https://doi.org/10.1145/3298989

[14] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[15] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797. https://doi.org/10.14778/2752939.2752947

[16] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[17] Michael Cornwell. 2012. Anatomy of a Solid-State Drive. *Communications of the ACM (CACM)* 55, 12 (2012), 59–63. https://doi.org/10.1145/2380656.2380672

[18] Kenneth M Curewitz, P Krishnan, and Jeffrey Scott Vitter. 1993. Practical Prefetching via Data Compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 257–266. https://doi.org/10.1145/170035.170077

[19] Fredrik Dahlgren, Michel Dubois, and Per Stenström. 1993. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP), Volume I: Architecture*. 56–63. https://doi.org/10.1109/ICPP.1993.92

[20] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. 2011. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1113–1124. https://doi.org/10.1145/1989323.1989442

[21] Paul Dubs, Ilia Petrov, Robert Gottstein, and Alejandro P Buchmann. 2013. 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)*. 58–69.

[22] Shimon Even and Guy Even. 2012. *Graph Algorithms (2nd ed.)*. Cambridge University Press. http://www.cambridge.org/us/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/graph-algorithms-2nd-edition

[23] Mohd Zeeshan Farooqui, Mohd Shoaib, and Mohammad Zunnun Khan. 2014. A comprehensive survey of page replacement algorithms. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume* 3 (2014).

[24] John W C Fu and Janak H Patel. 1991. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture. Toronto, Canada, May, 27-30 1991*. 54–63. https://doi.org/10.1145/115952.115959

[25] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[26] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[27] Knuth Stener Grimsrud, James K Archibald, and Brent E Nelson. 1993. Multiple Prefetch Adaptive Disk Caching. *IEEE Trans. Knowl. Data Eng.* 5, 1 (1993), 88–103. https://doi.org/10.1109/69.204094

[28] Yan Gu, Yihan Sun, and Guy E Blelloch. 2018. Algorithmic building blocks for asymmetric memories. In *Leibniz International Proceedings in Informatics, LIPIcs*, Vol. 112. 44:1—-44:15. https://doi.org/10.4230/LIPIcs.ESA.2018.44

[29] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf

[30] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 77–85. https://doi.org/10.1145/2487575.2487581

[31] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

[32] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 439–450. http://dl.acm.org/citation.cfm?id=645920.672996

[33] Doug Joseph and Dirk Grunwald. 1999. Prefetching Using Markov Predictors. *IEEE Trans. Computers* 48, 2 (1999), 121–133. https://doi.org/10.1109/12.752653

[34] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. 2008. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *IEEE Trans. Consumer Electron.* 54, 3 (2008), 1215–1223. https://doi.org/10.1109/TCE.2008.4637609

[35] Aarati Kakaraparthy, Jignesh M Patel, Kwanghyun Park, and Brian Kroth. 2019. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. *Proceedings of the VLDB Endowment* 13, 4 (2019), 519–532. https://doi.org/10.14778/3372716.3372724

[36] Jeong-Uk Kang, Heeseung Jo, Jinsoo Kim, and Joonwon Lee. 2006. 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*. 161–170. https://doi.org/10.1145/1176887.1176911

[37] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2011. GBASE: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*. 1091–1099. https://doi.org/10.1145/2020408.2020580

[38] Kamil Kedzierski, Miquel Moretó, Francisco J Cazorla, and Mateo Valero. 2010. Adapting cache partitioning algorithms to pseudo-LRU replacement policies. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470352

[39] Sami Khuri and Hsiu-Chin Hsu. 1999. Visualizing the CPU scheduler and page replacement algorithms. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*. 227–231. https://doi.org/10.1145/299649.299764

[40] Hyojun Kim and Seongjun Ahn. 2008. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 239–252.

[41] Donald E. Knuth. 2022. *"Weak components", The Art of Computer Programming, Volume IV, Pre-Fascicle 12A: Components and Traversal*. 11–14 pages. https://cs.stanford.edu/$\sim$knuth/fasc12a+.pdf

[42] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 395–410. https://doi.org/10.1145/3183713.3196915

[43] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[44] Roland L Lee, Pen-Chung Yew, and Duncan H Lawrie. 1987. Data Prefetching In Shared Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. 28–31.

[45] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)* 6, 3 (2007). http://dl.acm.org/citation.cfm?id=1275990

[46] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[47] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. 2008. TaP: Table-based Prefetching for Storage Caches. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 81–96. http://www.usenix.org/events/fast08/tech/li.html

[48] Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, Ke Yi, and Robin Jun Yang. 2010. Tree Indexing on Solid State Drives. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1195–1206. http://dl.acm.org/citation.cfm?id=1920841.1920990

[49] Zhi Li, Peiquan Jin, Xuan Su, Kai Cui, and Lihua Yue. 2009. CCF-LRU: a new buffer replacement algorithm for flash memory. *IEEE Trans. Consumer Electron.* 55, 3 (2009), 1351–1359. https://doi.org/10.1109/TCE.2009.5277999

[50] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[51] Ning Liu, Dong-sheng Li, Yiming Zhang, and Xiong-lve Li. 2020. Large-scale graph processing systems: a survey. *Frontiers Inf. Technol. Electron. Eng.* 21, 3 (2020), 384–404. https://doi.org/10.1631/FITEE.1900127

[52] László Lovász. 1993. Random walks on graphs. *Combinatorics, Paul erdos is eighty* 2, 1-46 (1993), 4.

[53] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727. https://doi.org/10.14778/2212351.2212354

[54] Yanfei Lv, Bin Cui, Bingsheng He, and Xuexuan Chen. 2011. Operation-aware buffer management in flash-based systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 13–24. https://doi.org/10.1145/1989323.1989326

[55] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 527–543. https://doi.org/10.1145/3064176.3064191

[56] Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 135–146. https://doi.org/10.1145/1807167.1807184

[57] Bo Mao, Suzhen Wu, and Lide Duan. 2018. Improving the SSD Performance by Exploiting Request Characteristics and Internal Parallelism. *IEEE Trans. on CAD of Integrated Circuits and Systems* 37, 2 (2018), 472–484. https://doi.org/10.1109/TCAD.2017.2697961

[58] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 116–128. https://doi.org/10.1145/3307650.3322275

[59] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 115–130. http://www.usenix.org/event/fast03/tech/full_papers/megiddo/megiddo.pdf

[60] Sparsh Mittal. 2016. A Survey of Recent Prefetching Techniques for Processor Caches. *Comput. Surveys* 49, 2 (2016), 35:1—-35:35. https://doi.org/10.1145/2907071

[61] Suman Nath and Phillip B. Gibbons. 2008. Online Maintenance of Very Large Random Samples on Flash Storage. *Proceedings of the VLDB Endowment* 1, 1 (2008), 970–983. http://www.vldb.org/pvldb/1/1453961.pdf

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

[63] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 297–306. https://doi.org/10.1145/170035.170081

[64] Tarikul Islam Papon. 2024. Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 5644–5648. https://doi.org/10.1109/ICDE60146.2024.00454

[65] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. https://doi.org/10.1145/3465998.3466003

[66] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[67] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

[68] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 125:1—-125:26. https://doi.org/10.1145/3654928

[69] Tarikul Islam Papon, Ju Hyoung Mun, Shahin Roozkhosh, Denis Hoornaert, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Fabric: Transparent Data Transformation. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

[70] Hyoungmin Park and Kyuseok Shim. 2009. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software* 82, 8 (2009), 1298–1312. https://doi.org/10.1016/j.jss.2009.02.028

[71] Stan Park and Kai Shen. 2009. A performance evaluation of scientific I/O workloads on Flash-based SSDs. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 1–5. https://doi.org/10.1109/CLUSTR.2009.5289148

[72] Stan Park and Kai Shen. 2012. FIOS: a fair, efficient flash I/O scheduler. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 13.

[73] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jinsoo Kim, and Joonwon Lee. 2006. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 234–241. https://doi.org/10.1145/1176760.1176789

[74] PostgreSQL. 2015. The Internals of PostgreSQL. *http://www.interdb.jp/pg/pgsql08.html* (2015).

[75] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition.

[76] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *Proceedings of the VLDB Endowment* 5, 4 (2011), 286–297. http://dl.acm.org/citation.cfm?id=2095686.2095688

[77] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: scale-out graph processing from secondary storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 410–424. https://doi.org/10.1145/2815400.2815408

[78] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 472–488. https://doi.org/10.1145/2517349.2522740

[79] Jinho Seol, Hyotaek Shim, Jaegeuk Kim, and Seungryoul Maeng. 2009. 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)*. 137–146. https://doi.org/10.1145/1629395.1629416

[80] Zhibing Sha, Zhigang Cai, François Trahay, Jianwei Liao, and Dong Yin. 2022. 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, 1–6.

[81] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. 2016. United States Data Center Energy Usage Report. *Ernest Orlando Lawrence Berkeley National Laboratory* LBNL-10057 (2016). https://eta.lbl.gov/publications/united-states-data-center-energy

[82] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 67–78.

[83] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27,*

2013. 135–146. https://doi.org/10.1145/2442516.2442530

[84] Alan Jay Smith. 1978. Sequentiality and Prefetching in Database Systems. *ACM Trans. Database Syst.* 3, 3 (1978), 223–247. https://doi.org/10.1145/320263.320276

[85] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[86] Andrew S Tanenbaum. 1992. *Modern Operating Systems*. Prentice-Hall.

[87] Myoung Kwon Tcheun, Hyunsoo Yoon, and Seung Ryoul Maeng. 1997. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. 306–313. https://doi.org/10.1109/ICPP.1997.622660

[88] TPC. 2022. Specification of TPC-C benchmark. *http://www.tpc.org/tpcc/* (2022).

[89] Stratis D. Viglas. 2012. 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. 399–412. https://doi.org/10.1007/978-3-642-33074-2_30

[90] Jeffrey Scott Vitter and P Krishnan. 1996. Optimal Prefetching via Data Compression. *J. ACM* 43, 5 (1996), 771–793. https://doi.org/10.1145/234752.234753

[91] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 389–404. https://doi.org/10.1145/3037697.3037744

[92] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 763–782. https://www.usenix.org/conference/osdi18/presentation/wang

[93] Wenlei Xie, Guozhang Wang, David Bindel, Alan J Demers, and Johannes Gehrke. 2013. Fast Iterative Graph Computation with Block Updates. *Proceedings of the VLDB Endowment* 6, 14 (2013), 2014–2025. https://doi.org/10.14778/2556549.2556581

[94] Yun-Seok Yoo, Hyejeong Lee, Yeonseung Ryu, and Hyokyung Bahn. 2007. Page Replacement Algorithms for NAND Flash Memory Storages. In *Proceedings of the International Conference on Computational Science and Applications (ICCSA) (Lecture Notes in Computer Science)*. 201–212. https://doi.org/10.1007/978-3-540-74472-6_16

[95] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. 183–193. https://doi.org/10.1145/2688500.2688507

[96] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 441–452. https://www.usenix.org/conference/atc18/presentation/zhang-yu

[97] Da Zheng, Disa Mhembere, Randal C Burns, Joshua T Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

[98] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[99] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu