Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry

Tarikul Islam Papon (Expected Graduation Date: June 2024)
Supervised By: Manos Athanassoulis
Boston University, USA

***Abstract*—Solid-state drives (SSDs) have become the dominant storage technology because of their faster read and write speeds and superior random access performance. Unlike their ancestor hard disk drives, 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 model with the Parametric I/O Model (PIO), a new storage model that faithfully represents storage devices by parameterizing read/write asymmetry (α) and access concurrency (k). Second, using this novel storage modeling, we propose a new *Asymmetry & Concurrency-aware* bufferpool management (ACE) that batches writes based on device concurrency and performs them in parallel to *amortize* the asymmetric write cost while performing parallel prefetching to exploit the device's read concurrency. Third, 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. Overall, our analysis shows that more faithful storage modeling leads to higher performance and better device utilization.**

I. Introduction

Modern Devices: Concurrency & Read/Write Asymmetry. Most secondary storage devices today are solid-state disks (SSDs), with traditional hard-disk drives (HDDs) mainly employed for archival storage. SSDs adopt NAND flash memory as the storage medium, eliminating mechanical overheads associated with HDDs, resulting in advantages such as fast random access, low energy consumption, and high chip density [3]. Moreover, because of the hierarchical architecture of SSD internal, SSDs exhibit a high degree of *internal parallelism* that can be utilized to increase performance [2, 9]. In other words, for an SSD to fully utilize its bandwidth, it SSD needs to receive multiple *concurrent* I/Os [2]. The exact level of **concurrency** (k) depends on the request type and on the device specifics. On the other hand, due to flash medium physics, the cost of reading is 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 [3].

The Parametric I/O Model. These SSD characteristics, i.e., *concurrency* (quantified by k) and *read/write asymmetry* (quantified by α) have two key implications: (i) careful exploitation of concurrency allows better device utilization, and (ii) treating reads and writes equally is suboptimal for SSD [9]. This calls for the need for a new I/O model [10] that considers the device properties. We propose a simple yet expressive storage model named the Parametric I/O Model (PIO) [9] that incorporates SSD *asymmetry* (α) and *concurrency* (k), thus, this richer I/O model can accurately capture contemporary devices. We benchmark several types of state-of-the-art storage devices to quantify their α and k. Our abstract analysis reveals 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 enhance performance.

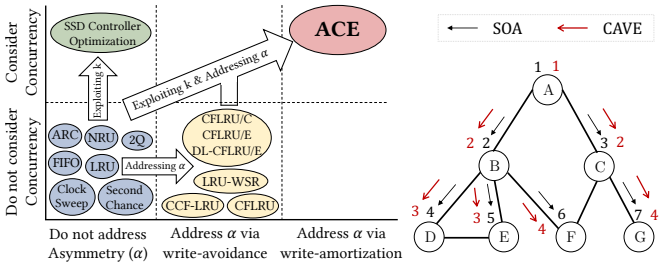


Fig. 1: (A) ACE addresses asymmetry by exploiting concurrency and amortizing writes. (B) The parallellized version of BFS in CAVE takes fewer iterations to converge.

Bufferpool Management. The component of a database management system (DBMS) responsible for direct interaction with storage devices is the *bufferpool* which serves as the interface between memory and the underlying storage device. We address two challenges of state-of-the-art bufferpool managers: (i) existing bufferpool managers often consider that the underlying devices have no concurrency ($k = 1$), hence missing the opportunity to exploit SSD concurrency (Figure 1A - bottom row: blue, yellow) and (ii) page replacement policies generally ignore device asymmetry (α), instead, they treat read and write equally (Figure 1A - left column: blue, green). We propose ACE [11], a new bufferpool manager that utilizes the device concurrency to bridge the device asymmetry (Figure 1A - red). Our approach uses *asymmetry/concurrency-aware* write-back and eviction policies. The write-back policy always writes multiple pages *concurrently* to utilize the *write concurrency* of the device, amortizing the high asymmetric write cost. The eviction policy evicts one or multiple pages

simultaneously from the bufferpool to enable prefetching – 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 and prefetching technique with low engineering effort. This allows any existing bufferpool manager to be augmented by our approach. We integrate ACE with four page replacement policies (Clock Sweep, LRU, LRU-WSR, CFLRU [4, 13]) and implement them in PostgreSQL to evaluate ACE's performance where we observe ACE can achieve upto 1.5× speedup.

Graph Management. Graph traversal operations can leverage SSD concurrency by parallelizing node and edge accesses. While the majority of out-of-core graph processing systems [5, 16] aim to indirectly harness the storage parallelism by minimizing random I/O in favor of sequential operations, they often do not exploit the SSD concurrency. We propose an SSD-aware graph engine, named CAVE that can harness the *concurrency* of the underlying storage devices to traverse multiple paths in parallel (Figure 1B), which results in faster convergence within fewer iterations. CAVE employs a novel block-based file format based on adjacency lists to ensure that graph metadata, vertex and edge information are stored in aligned blocks. Furthermore, CAVE uses a concurrent cache pool to enhance locality and ensure thread safety. We develop in CAVE the parallelized versions of five popular graph traversal algorithms (BFS, DFS, WCC, PageRank, Random Walk). We are working on comparing CAVE against with three popular out-of-core processing systems Mosaic [6], GridGraph [16], and GraphChi [5]. Early experiments show that CAVE can be up to three orders of magnitude faster than GraphChi and up to one order of magnitude faster than GridGraph for the parallel BFS implementation.

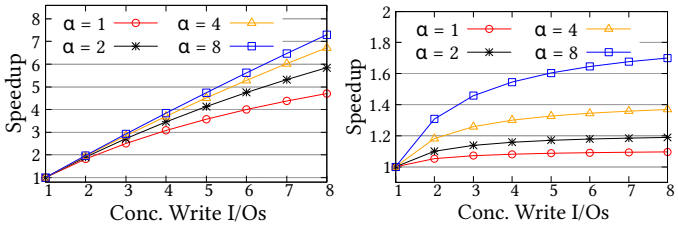
Contributions. Our contributions are as follows:

- We identify SSD read/write asymmetry (α) and concurrency (k) as key characteristics to exploit to improve performance.
- We introduce the **Parametric I/O Model** (PIO) which considers both α and k. We show the benefits of these properties with respect to device utilization and performance.
- We propose **ACE**, an *asymmetry & concurrency-aware* bufferpool manager that utilizes the device's concurrency. We implement ACE in PostgreSQL and evaluate its benefit for four page replacement algorithms.
- We propose **CAVE**, an SSD-aware graph engine that utilizes SSD concurrency via concurrent I/O, its novel file structure, and a concurrent cache pool. We develop parallelized version of five graph algorithms in CAVE.

II. The Parametric I/O Model

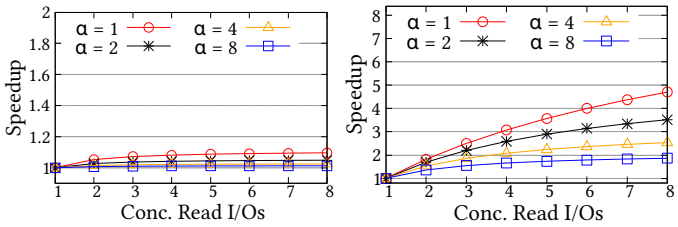
We now present the **Parametric I/O Model** (PIO) [9, 10] that takes read/write asymmetry and concurrency (read and write) as parameters to enable better algorithm design for storage-intensive systems.

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



(A) 10% reads (B) 90% reads

Fig. 2: Speedup is highest for write-intensive workloads and it depends on the asymmetry for *Batchable Writes* applications.



(A) 10% reads (B) 90% reads

Fig. 3: Highest speedup in read-heavy workloads for *Batchable Reads* applications. *Lower asymmetry results in higher gain.*

The values of k_r, k_w, and α are either given by the device manufacturer, or derived by a careful benchmarking process outlined in our full paper [9]. We classify storage-intensive applications as batchable writes and batchable reads to reason about the performance benefits under PIO.

Batchable Writes. This category of applications utilizes the write concurrency of the device by batching write requests. For example, let us consider a modified DBMS bufferpool manager that writes multiple dirty pages concurrently during an eviction. The application at hand attempts to fully exploit the device's write concurrency via *concurrent flushing* k_w dirty pages. Fig. 2 shows the speedup following PIO for different α and k_w values as we change the read/write ratio in the workload. We notice that the speedup increases with more concurrent I/Os, which is expected. We also observe that the gain is highest for a write-intensive workload (Fig. 2A). This is because the application batches writes and so the benefit from efficient writing is more pronounced. Furthermore, the speedup depends on the device asymmetry – the *gain is* ***higher*** *for a device with* ***higher*** *asymmetry*.

Batchable Reads. This category of applications represents situations where concurrent reads can be performed to leverage read concurrency. As an example, let us consider a graph store that traverses multiple paths concurrently, i.e., it can process multiple nodes in parallel and offer faster search time with the same worst-case guarantees. Fig. 3 shows the speedup of such an application based on PIO. Like before, the speedup increases as we increase the number of concurrent I/Os. The speedup is highest for a read-heavy workload (Fig. 3A), which shows the benefit of batching reads. However, now *the gain is* ***higher*** *for a device with* ***lower*** *asymmetry*.

The above analysis reveals that utilizing SSD concurrency yields speedup while the degree of performance improvement depends on the SSD asymmetry and the application type.

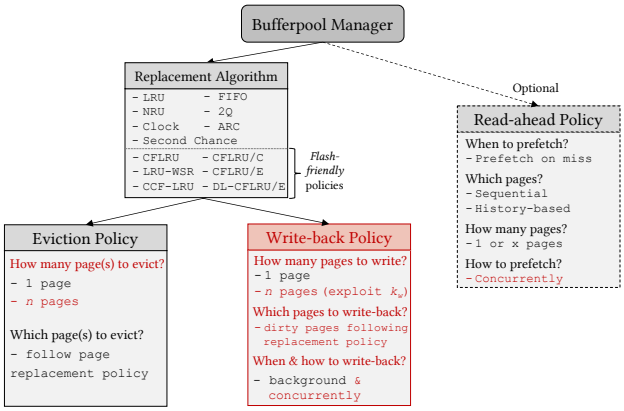


Fig. 4: Bufferpool design space in terms of the design decisions and various options (RED denotes new components)

III. ACE Bufferpool Manager

Following the PIO Model, we now propose a new Asymmetry & Concurrency-aware bufferpool management (ACE) [11].

A. An Augmented Bufferpool Design Space

Traditionally, the design space of bufferpool management primarily includes a *page replacement policy* and, optionally, a *read-ahead policy*. The page replacement policy determines the sequence for *evicting* and *writing back* pages. When the evicted page is dirty, a write-back is initiated for that page. Since traditional systems employ a single policy for both eviction and write-back, they essentially address two distinct questions with one decision: *which page to evict?* and *which page to write back?* To address this, we introduce a new *write-back policy*, decoupling write-backs from eviction (Figure 4). While maintaining an overall *virtual page ordering* for eviction based on the replacement algorithm, we use a slightly different *virtual order* for writing back pages, determined by (a) the replacement algorithm, (b) the page's dirtiness, and (c) the write concurrency of the device.

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

Within this augmented design space, the *replacement algorithm* influences both the *write-back* and the *eviction* policies, albeit in distinct ways. The *write-back policy* uses the virtual order of pages defined by the *replacement algorithm* and the device's write concurrency to write-back *multiple dirty* pages concurrently. The *eviction policy* uses the virtual order of pages defined by the *replacement algorithm* to evict only clean pages. The decision regarding the number of pages to evict is determined by the application, balancing between prioritizing locality and prefetching or read-ahead policy.

B. Overview of ACE

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 accessing a page P is received, ACE first searches 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 decides which page will be evicted (termed *top page*). If the top page is *clean*, it is evicted and page P is fetched. Until this part, ACE behaves like any other state-of-the-art bufferpool management. However, if the top page is *dirty*, ACE proceeds as follows:

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

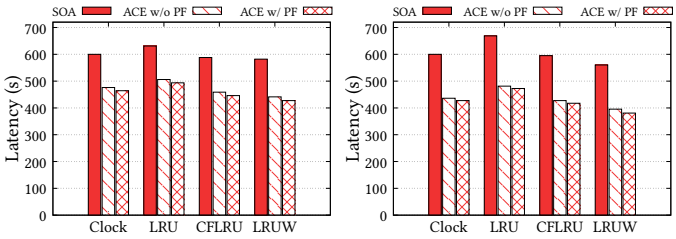
We tune ACE to use n_w equal to the optimal write concurrency of the device (k_w). We experimentally tested values for n_e between 1 and k_r, and we empirically set n_e to be also k_w, because evicting k_r pages hurts locality. This is because, for most devices, the read concurrency is significantly higher than the write concurrency ($k_r >> k_w$). ACE can be combined with any replacement algorithm and prefetching technique.

C. Evaluation

We implement ACE in PostgreSQL 11.5 and evaluate its benefits when applied with four page replacement policies (LRU, CFLRU, LRU-WSR, and Clock Sweep) using both a synthetic benchmark and the standard TPC-C benchmark. Our in-house experimental server involves three storage devices: (i) a 375GB Optane P4800X SSD ($\alpha = 1.1, k_r = 6, k_w = 5$), (ii) a 1TB PCIe P4510 SSD ($\alpha = 2.8, k_r = 80, k_w = 8$), and (iii) a 240GB SATA S4610 SSD ($\alpha = 1.5, k_r = 25, k_w = 9$). Further, we use a *virtualized* device from AWS with 1.2TB capacity and 60000 provisioned IOPS.

ACE Improves Runtime without Any Penalty. This experiment shows that ACE reduces the total workload latency by up to 32.1% for the PCIe SSD that has $k_w = 8$ and $\alpha = 2.8$. Figures 5A and B show the execution time for the baseline algorithms along with their ACE counterparts with and without prefetching for 2 synthetic workloads in PostgreSQL. Since ACE policies utilize the device's write parallelism, it writes back pages more aggressively, resulting in better performance. Because of the skewness in the workload, the prefetching helps to avoid some disk access, resulting in slightly better performance. We highlight that the latency improvement observed in these experiments does not come at any hidden cost. The maximum increase in buffer misses is 0.003%, and the maximum increase in total writes is 0.12%, thus being negligible. ACE's gain is higher for the write-intensive workload (Figure 5B), because for a write-intensive workload, the benefit of *efficient* writing is more pronounced.

ACE Excels for TPC-C. We run the standard TPC-C benchmark in PostgreSQL for the four replacement policies and their ACE counterparts. Figure 6 shows the performance gain of ACE for the TPC-C mix, and for five TPC-C transactions. ACE achieves good performance gain when integrated with any page replacement policy. For instance, the speedup of ACE for the mixed transaction is 1.29×, 1.27×, 1.30× and 1.32× when implemented with Clock Sweep, LRU, CFLRU, LRU-WSR, respectively. The highest speedup (1.51×) is gained



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

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

Fig. 5: ACE reduces total workload latency for all Clock Sweep, LRU, CFLRU, and LRU-WSR in the PCIe SSD.

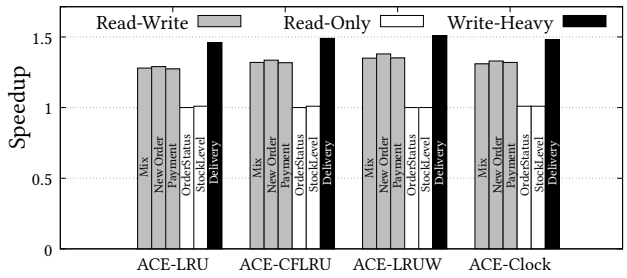


Fig. 6: ACE attains high speedup for TPC-C mixed transaction, while benefiting the write-heavy transaction the most.

for *Delivery*, an update-heavy transaction. The performance results for the TPC-C benchmark validate our observations from the synthetic benchmark: (i) ACE can gain good performance advantages even with a minor proportion of writes in the workload, (ii) write-heavy workloads exhibit higher gains, and (iii) flash-friendly policies such as CFLRU and LRU-WSR outperform other policies. More experiments can be found in our full paper [11].

IV. CAVE Graph Manager

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

A. CAVE Physical Data Layout

CAVE adopts a memory-mapped binary file format, comprising three main sections: the metadata block, the vertex block, and the edge block (depicted on the right side of Figure 7A). These are stored using 4KB aligned blocks to facilitate direct reading and writing from/to SSDs. The metadata block functions as a repository for crucial graph information, including the number of vertices, total blocks, edge blocks, and vertex blocks, each stored as a 32-bit integer. Each vertex block, sized at 4KB, stores information about up to 512 vertices. Within each vertex, 8 bytes are allocated, accommodating two 32-bit unsigned integers: $degree$ and eb_{addr} (edge block index and offset). A compact representation of edges is employed, where each edge is denoted by a 4-byte integer indicating the index of the ending vertex. Consequently, each edge block can store up to 1024 edges, totaling 4KB. The edges of vertices with a degree less than 1024 are consolidated within a single edge block, ensuring efficient single-read I/O access, though the starting index inside the block (eb_{offset}) may vary. However, vertices with a degree exceeding 1024 occupy multiple edge blocks.

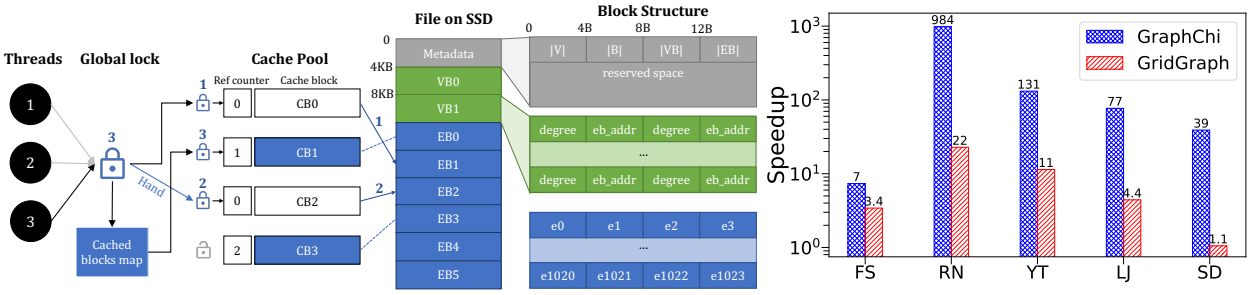
B. Concurrent Graph Traversal Algorithms

The core concept of our approach is the implementation of parallel graph algorithms that leverage concurrency at the storage level. CAVE identifies and parallelizes independent I/Os, akin to how out-of-order processors parallelize load and store commands that are not dependent on each other. This facilitates parallel graph data processing, enabling simultaneous access to multiple nodes (or edges), thereby reducing the number of required iterations. To illustrate the advantages of our approach, we parallelize five common graph traversal algorithms: BFS, WCC, PageRank, Random Walk, and DFS. The parallel BFS (PBFS) algorithm utilizes two queues: the $frontier$ queue containing indices of vertices in the current level and the $next$ queue storing indices of neighbors of vertices in the $frontier$ queue, corresponding to vertices in the next level. To exploit parallelism, each vertex in the $frontier$ queue is assigned to a separate thread, allowing multiple I/Os to be issued in parallel as shown in Figure 1(A)). The level of concurrency in PBFS is controlled by the number of threads, adjusted according to the optimal concurrency of the SSD. Building upon the PBFS algorithm, we develop parallel weakly connected components, parallel PageRank, and parallel Random Walk. Although DFS is inherently a serialized algorithm, its performance can be enhanced by introducing parallelism through a technique known as *unordered* or *pseudo-DFS* [1]. Inspired by this concept, we incorporate a mechanism to monitor the size of the vertex stack for each thread in our PDFS implementation.

C. Evaluation

We are currently performing our experimental evaluation of CAVE for the five algorithms against three storage-optimized graph processing systems Mosaic [6], GridGraph [16], and GraphChi [5]. We use the same server and storage devices described in Section III-C. We use four datasets of different sizes and types from the Stanford Large Network Dataset Collection: 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) which has 50M nodes and 42GB size.

CAVE outperforms GraphChi & GridGraph. Our initial experimental result is presented in Figure 7B which shows CAVE's speedup compared to GraphChi and GridGraph when running the PBFS algorithm for all five datasets for a specific cache size (around 3% for each workload) depending on the dataset on the PCIe SSD device. 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 and the unusually high speedup for this dataset shown in Figure 7B is attributed to the high diameter of the graph where CAVE excels. For dense graphs like SD (the diameter is only 6 with 50M nodes and 1.25B edges), GridGraph performs well because of its grid structure to partition and manage dense graphs, however, CAVE still outperforms GridGraph.



(A) (B)

Fig. 7: (A) CAVE's Architecture: block-based file structure (right side) and concurrent cache pool (left side). (B) CAVE performs well across all datasets for PBFS, especially for sparse graphs (RN), while, GridGraph is optimized for dense graphs (SD).

V. Research Plan

Short-Term. We are currently in the process of completing the experimental evaluation of CAVE by comparing it against three popular out-of-core graph processing systems: Mosaic, GridGraph, and GraphChi. Our plan includes conducting a comprehensive study to highlight CAVE's benefits across various datasets, devices, cache sizes, concurrency, and algorithms. We are also exploring how CAVE can handle graph updates. To manage updates to vertex/edge values, we intend to modify our file architecture to accommodate these changes. Implementing a buffer to batch updates could facilitate concurrent operations, making use of the device's write concurrency. Furthermore, for adding new vertices and edges, we are considering a strategy similar to the log-structured merge (LSM) [8, 15] paradigm.

Long-Term. In the long term, we plan to adopt the Parametric I/O Model for newer SSDs, including computational SSDs (CSSDs), open-channel SSDs (OCSSDs), and zoned namespace SSDs (ZNS SSDs). Our goal is to enable on-the-fly data transformation from stored rows to any column groups at the storage level via near-storage computation in CSSDs and OCSSDs that offer processing power that we can leverage [12]. We plan to implement a near-data vertical partitioner directly in these devices by utilizing in-storage custom logic and reprogrammable logic. We have already worked on this concept for in-memory systems via *Relational Memory* [7, 14] that can provide the optimal layout for any query by exploiting specialized reprogrammable hardware. Furthermore, we plan to redefine the design of Log-Structured Merge (LSM) trees to align with zoned namespace (ZNS) SSDs. ZNS SSDs divide space into equal-sized zones, enabling fast sequential writes and flexibility in data placement and garbage collection. We can leverage these features by (i) treating erase blocks within SSDs as fragments of a sorted immutable run in LSM trees, (ii) incorporating host-side garbage collection, and (iii) optimizing data placement by grouping related data together.

VI. 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) and concurrency (k) that different devices may support. Inspired from PIO, we propose ACE, a novel asymmetry/concurrency-aware bufferpool manager that batches writes based on device concurrency to amortize the high asymmetric write cost. ACE works as a wrapper that can be integrated with *any* page replacement and prefetching policy. Further, we propose CAVE, a concurrency-aware graph processing system designed to leverage the 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. For both ACE and CAVE, we observe from extensive experimental evaluations that better storage modeling leads to better device utilization and, ultimately, better performance.

References

[1] U. A. Acar, A. Charguéraud, and M. Rainey, "A work-efficient algorithm for parallel unordered depth-first search," *SC*, 2015.

[2] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," *HPCA*, 2011.

[3] M. Cornwell, "Anatomy of a Solid-State Drive," *CACM*, 2012.

[4] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: integration of LRU and writes sequence reordering for flash memory," *IEEE Trans. Consumer Electron.*, 2008.

[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," *OSDI*, 2012.

[6] S. Maass, C. Min, S. Kashyap, W.-H. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a Trillion-Edge Graph on a Single Machine," *EuroSys 2017, Belgrade, Serbia*, 2017.

[7] J. H. Mun, K. Karatsenidis, T. I. Papon, S. Roozkhosh, D. Hoornaert, U. Drepper, A. Sanaullah, R. Mancuso, and M. Athanassoulis, "On-the-fly Data Transformation in Action," *PVLDB*, 2023.

[8] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, 1996.

[9] T. I. Papon and M. Athanassoulis, "A Parametric I/O Model for Modern Storage Devices," *DAMON*, 2021.

[10] T. I. Papon and M. Athanassoulis, "The Need for a New I/O Model," *CIDR*, 2021.

[11] T. I. Papon and M. Athanassoulis, "ACEing the Bufferpool Management Paradigm for Modern Storage Devices," *ICDE*, 2023.

[12] T. I. Papon, J. H. Mun, S. Roozkhosh, D. Hoornaert, A. Sanaullah, U. Drepper, R. Mancuso, and M. Athanassoulis, "Relational Fabric: Transparent Data Transformation," *ICDE*, 2023.

[13] S.-Y. Park, D. Jung, J.-U. Kang, J. Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," *CASES*, 2006.

[14] S. Roozkhosh, D. Hoornaert, J. H. Mun, T. I. Papon, A. Sanaullah, U. Drepper, R. Mancuso, and M. Athanassoulis, "Relational Memory: Native In-Memory Accesses on Rows and Columns," *EDBT*, 2023.

[15] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis, "Lethe: A Tunable Delete-Aware LSM Engine," *SIGMOD*, 2020.

[16] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning," *ATC*, 2015.