PulseDB: Fine-Grained DPU-Assisted Compaction Offloading for LSM-Based Key-Value Stores

Lihan Ou*, Yifei Chen*, Yikun Wu*, Yu Zhang*,
Yinchao Ji*, Bo Mao*, and Suzhen Wu*
*School of Informatics, Xiamen University, Xiamen, China
Corresponding Authors: Bo Mao (maobo@xmu.edu.cn) and Suzhen Wu (suzhen@xmu.edu.cn)

***Abstract*—LSM-tree-based key-value stores rely on compaction to maintain read efficiency and reclaim obsolete data, but compaction continuously consumes host CPU cycles and contends with foreground requests, causing write stalls and elevated tail latency. Existing DPU-based approaches offload compaction at whole-task granularity; however, the limited compute capability of DPU ARM cores creates a tradeoff between preserving compaction efficiency and freeing host CPU resources. This paper presents PulseDB, which decomposes compaction into compute-intensive merge-encode sub-tasks and I/O-oriented write-persist sub-tasks that can leverage DPU data-movement and storage-path offload capabilities. PulseDB keeps merge-encode on host x86 cores while asynchronously offloading write-persist to the DPU, forming a heterogeneous pipeline enabled by an `io_uring`-based DPU I/O engine, a pre-allocated SST file pool, and DPU-driven DMA ring queues. On a real NVIDIA BlueField-class DPU platform, PulseDB improves write throughput by up to 82%, reduces host compaction CPU overhead by up to 42%, and lowers P99 tail latency by 29% over vanilla RocksDB; compared with DComp, PulseDB further improves throughput by 11–33%.**

***Index Terms*—Data Processing Units, Key-Value Stores, Log-Structured Merge-Trees, Compaction Offloading**

I. Introduction

Log-Structured Merge-Tree (LSM-tree) based key-value stores have become fundamental building blocks in modern data centers, powering systems such as LevelDB [1], RocksDB [2], Cassandra [3], and X-Engine [4]. Their write-optimized design relies on *compaction*: a background process that reads Sorted String Table (SST) files, merge-sorts key-value pairs, discards obsolete versions, and writes new SST files [5]–[7]. While essential for read efficiency and space reclamation, compaction continuously consumes host CPU cycles, cache capacity, memory bandwidth, and storage I/O bandwidth. It also competes with foreground requests and can trigger write stalls when background processing falls behind incoming writes, causing throughput drops and tail latency spikes [8]–[10]. This pressure is increasing as fast NVMe SSDs shift key-value-store bottlenecks from storage I/O toward CPU processing [11], [12].

Prior work has explored hardware and infrastructure support for reducing compaction overhead. FPGA-based designs build custom dataflow pipelines for merge and encoding stages [13]–[15], GPU approaches parallelize sorting [16], [17], near-data processing moves work closer to storage [18], and CaaS-LSM decouples compaction as a remote service [19]. These systems are effective within their respective deployment models. In parallel, Data Processing Units (DPUs) have become a practical offload substrate in modern cloud infrastructure: they combine ARM cores with DMA engines, RDMA-capable NICs, and storage/network offload capabilities, and are already deployed for networking and storage virtualization tasks [20]–[22]. This makes DPUs a natural candidate for relieving host-side compaction pressure, provided that the offloading boundary matches their heterogeneous hardware strengths.

Prior DPU compaction systems, including DComp [23] and D^{2}Comp [24], offload a full compaction task to the DPU. Yet compaction mixes two kinds of work: compute-heavy merge processing and I/O-oriented persistence. Our measurements show that DPU ARM cores reach only 45.5% of host x86 compaction throughput, about 2.2× slower. This makes whole-task offloading a tradeoff. Aggressive offloading can slow compaction, whereas conservative offloading protects performance but frees only limited host CPU cycles. For example, DComp keeps performance-critical L0→L1 compactions on the host and uses the DPU mostly as an overflow handler for deeper levels, leaving long DPU idle windows and limited CPU relief.

We observe that this limitation stems from a granularity mismatch. A compaction task is not monolithic: it can be decomposed into (1) a *merge-encode* sub-task, which iterates over input SST files, performs merge-sort, and encodes output data blocks, and (2) a *write-persist* sub-task, which writes encoded blocks and synchronizes them to storage. Merge-encode is compute-intensive and benefits from the host's stronger x86 cores and caches. Write-persist is I/O-oriented: it mostly exercises data movement, storage-stack traversal, and persistence synchronization, making it a better fit for our NVIDIA BlueField-3 platform, which provides DMA engines and hardware-offloaded NVMe-oF/RDMA data paths. Our profiling shows that write-persist accounts for 42–53% of compaction wall-clock time and CPU cycles, with duration roughly comparable to merge-encode, exposing an opportunity for heterogeneous pipeline parallelism.

Based on this insight, we present **PulseDB**, a DPU-assisted fine-grained compaction offloading system for LSM-tree key-value stores. PulseDB targets DPU-equipped storage deployments in which the DPU can access the storage device and perform data movement through DMA or storage-path offload engines. Instead of migrating whole compactions, PulseDB

keeps merge-encode on the host and asynchronously offloads write-persist to the DPU, allowing host-side merge work to overlap with DPU-side data movement and persistence. We make the following contributions:

- **A fine-grained sub-task decomposition for host–DPU collaborative compaction.** We identify that compaction contains compute-intensive merge-encode and I/O-oriented write-persist sub-tasks with roughly balanced durations, and place each sub-task on the processor whose strengths match its resource profile.
- **A complete system design addressing the key challenges of fine-grained offloading.** We design three enabling techniques: (1) an `io_uring`-based asynchronous I/O engine with range-sync and user-space dependency tracking to keep the DPU in pace with the host; (2) a statically pre-allocated SST file pool that exploits SST immutability for lightweight cross-node file sharing without distributed file-system overhead; and (3) DPU-driven DMA ring queues with a zero-copy data path that eliminates host-side PCIe involvement.
- **Comprehensive evaluation on a real DPU platform.** We implement PulseDB on RocksDB and evaluate it on an NVIDIA BlueField-3 DPU platform, demonstrating up to 82% write throughput improvement, 42% host compaction CPU reduction, and 29% P99 tail latency reduction over vanilla RocksDB, with 11–33% throughput gains over the state-of-the-art DComp.

The remainder of this paper is organized as follows. Section II presents background and motivation. Section III describes the PulseDB design. Section IV details our experimental evaluation. Section V discusses related work, and Section VI concludes.

II. Background and Motivation

A. LSM-Tree Compaction

In an LSM-tree key-value store, writes first enter a MemTable and are later flushed as Level-0 (L0) Sorted String Table (SST) files. Since independently flushed L0 files may overlap in key range, the engine periodically performs *compaction*: it selects SST files from adjacent levels, merge-sorts their key-value pairs, discards obsolete versions and deletion markers, and writes new SST files while deleting old inputs [5]–[7]. This paper focuses on the widely deployed leveled compaction policy, where levels beyond L0 maintain non-overlapping sorted runs. L0→L1 compactions are especially important because they often overlap many L1 files and have been identified as a primary source of write stalls in production RocksDB deployments [8], [10].

Compaction is necessary for bounded read amplification and space reclamation, but it competes directly with foreground requests. Its merge-sort, data-block encoding, optional compression, and I/O operations consume CPU cycles, cache capacity, memory bandwidth, and storage bandwidth [8], [9]. When background compaction falls behind the foreground write rate, the engine throttles or blocks writes to prevent unbounded LSM-tree growth, causing sharp throughput drops and tail latency spikes [10], [25].

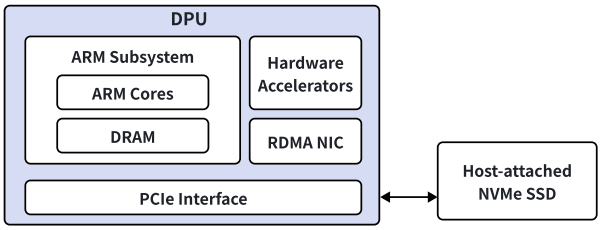


Fig. 1. NVIDIA BlueField-class DPU components relevant to PulseDB.

B. DPU Architecture

A Data Processing Unit (DPU) is a PCIe-attached programmable processor designed to offload infrastructure tasks such as networking, storage virtualization, and security from the host CPU [20], [21]. PulseDB targets NVIDIA BlueField-class DPUs, a representative commercial DPU platform that combines programmable ARM cores with DMA/RDMA and storage-path offload capabilities in the I/O path. As shown in Fig. 1, this DPU class also provides on-board DRAM, an RDMA-capable NIC, hardware accelerators, and a PCIe-attached deployment model [26]. These components make the platform suitable for data movement, protocol processing, and I/O-path coordination rather than arbitrary CPU-intensive computation.

In our target deployment, the SSD is physically attached to the host's PCIe bus and exposed to the DPU through NVMe over Fabrics (NVMe-oF). Its RDMA NIC and NVMe-oF target offload capability handle much of the protocol processing in hardware, allowing the DPU to issue storage I/O with limited ARM core involvement. Prior DPU/SmartNIC storage systems exploit similar data-path advantages for file-system client processing, file-system virtualization, and distributed-file-system pipelining [27]–[29].

C. Limitations of Coarse-Grained Offloading

Existing DPU-based compaction systems offload compaction at whole-task granularity. This is attractive for reducing host CPU contention, but it conflicts with the DPU's weaker general-purpose compute capability. We quantify this gap by running identical compaction workloads on the host x86 cores and on the DPU ARM cores. As Fig. 2 shows, the DPU reaches only 45.5% of the host's normalized compaction throughput, about 2.2× slower. Persistently migrating complete compactions to the DPU can therefore reduce backend processing capacity and eventually trigger more write stalls.

DComp [23] and D^{2}Comp [24] address this limitation through conservative scheduling: performance-critical L0→L1 compactions remain on the host, while deeper-level compactions are dispatched to the DPU only when host-local compaction slots are occupied. We reproduce DComp's core scheduling logic under `db_bench fillrandom` with 20M

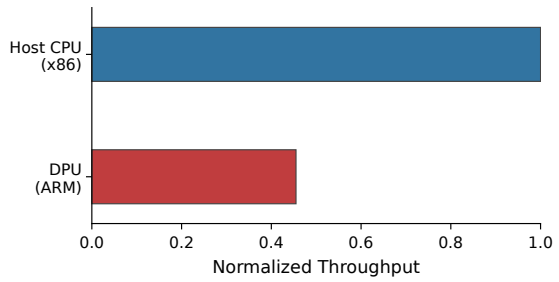


Fig. 2. Normalized compaction throughput: host x86 vs. DPU ARM.

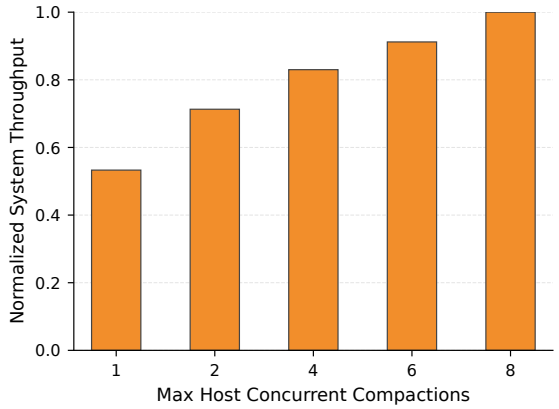


Fig. 3. Normalized write throughput under DComp-style coarse-grained offloading with varying host-local compaction slot counts.

key-value pairs (16 B keys, 512 B values), fix total compaction parallelism at 8 threads, and vary the number of host-local compaction slots.

Fig. 3 shows the resulting tradeoff. As more whole compactions are forced onto the DPU, write throughput steadily declines; with maximum DPU involvement, throughput drops to 53% of the host-only baseline. Thus, aggressive whole-task offloading harms compaction efficiency, while conservative offloading preserves performance but limits host CPU relief.

Resource utilization exhibits the same limitation. Even under the most aggressive configuration, the DPU processes only about half of the compaction data (Fig. 4a). L0→L1 compactions, which account for 34–41% of total compaction data, always execute on the host. Meanwhile, Fig. 4b shows short DPU busy bursts separated by long idle windows. These results show that coarse-grained offloading cannot continuously utilize DPU resources: task-level scheduling leaves L0→L1 compactions on the host and activates the DPU only in intermittent bursts. The key question is therefore not whether to use the DPU, but at what granularity it should collaborate with the host.

D. Opportunities for Fine-Grained Offloading

A compaction is not monolithic. For each output SST file, the engine first iterates over input SST files, merge-sorts keys, filters obsolete entries, and encodes output blocks. It then writes the encoded blocks and synchronizes them to storage through `write`, `sync_file_range`, and `fsync`. We group the first part as the *merge-encode* sub-task and the second as the *write-persist* sub-task. In the conventional path, these two sub-tasks execute serially inside the same host compaction thread.

We instrument RocksDB to measure wall-clock time and CPU cycles for the two sub-tasks under `db_bench fillrandom` while varying value sizes from 128 B to 2048 B. As shown in Fig. 5a and Fig. 5b, write-persist accounts for 45.4% of wall-clock time and 42.4% of CPU cycles at 512 B values; at 2048 B values, both ratios exceed 53%. The two sub-tasks therefore have roughly comparable durations, making the serial compaction time $T_{\text{merge}} + T_{\text{io}}$ a natural target for pipeline overlap.

The two sub-tasks also match different processors. Merge-encode is dominated by key comparisons, block construction, memory operations, and optional compression, all of which benefit from host x86 cores and larger caches. Write-persist mostly exercises data movement, storage-stack traversal, and persistence synchronization. On NVIDIA BlueField-class DPUs, these operations can use DMA and hardware-offloaded NVMe-oF/RDMA data paths, reducing ARM-core work relative to running full merge logic on the DPU [27]–[29]. This motivates PulseDB's fine-grained strategy: retain merge-encode on the host and asynchronously offload write-persist to the DPU so the two phases overlap.

E. Design Challenges

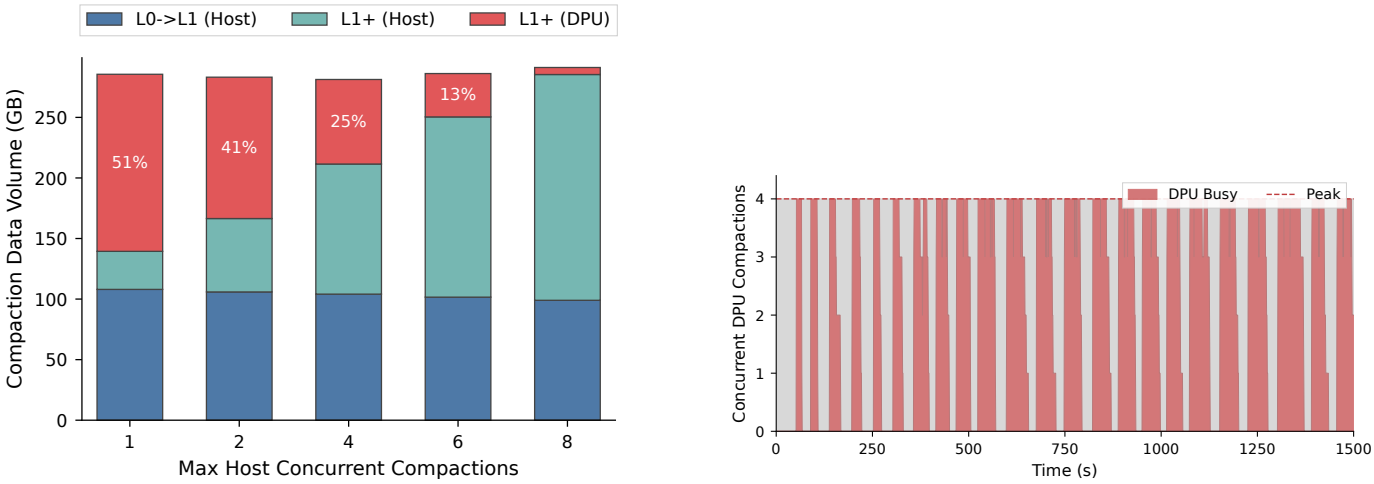
Realizing this strategy requires addressing three challenges.

C1: Heterogeneous pipeline coordination. The host must continue merge-encode while the DPU performs writes and persistence, but the two sides still have strict per-file dependencies: range syncs must follow the writes they cover, and the final close must follow all pending writes and syncs. The DPU-side I/O engine must preserve these dependencies while keeping enough asynchronous parallelism to match the host's data production rate.

C2: Low-cost SST file sharing. The DPU writes output SST files that the host later reads. A general-purpose shared file system would introduce metadata synchronization and cache-coherence overheads that erode offloading benefits. PulseDB instead needs a lightweight mechanism that exploits SST immutability and the near-fixed size of SST files under leveled compaction.

C3: Low-overhead data transfer. Fine-grained offloading creates many host-to-DPU submissions. If each request requires extra copies, per-transfer registration, or synchronous PCIe round-trips, communication overhead can dominate the saved host CPU cycles. Although the DPU DMA engine provides more than 8 GB/s bandwidth [30], PulseDB must expose this bandwidth through batched, zero-copy, DPU-driven transfers.

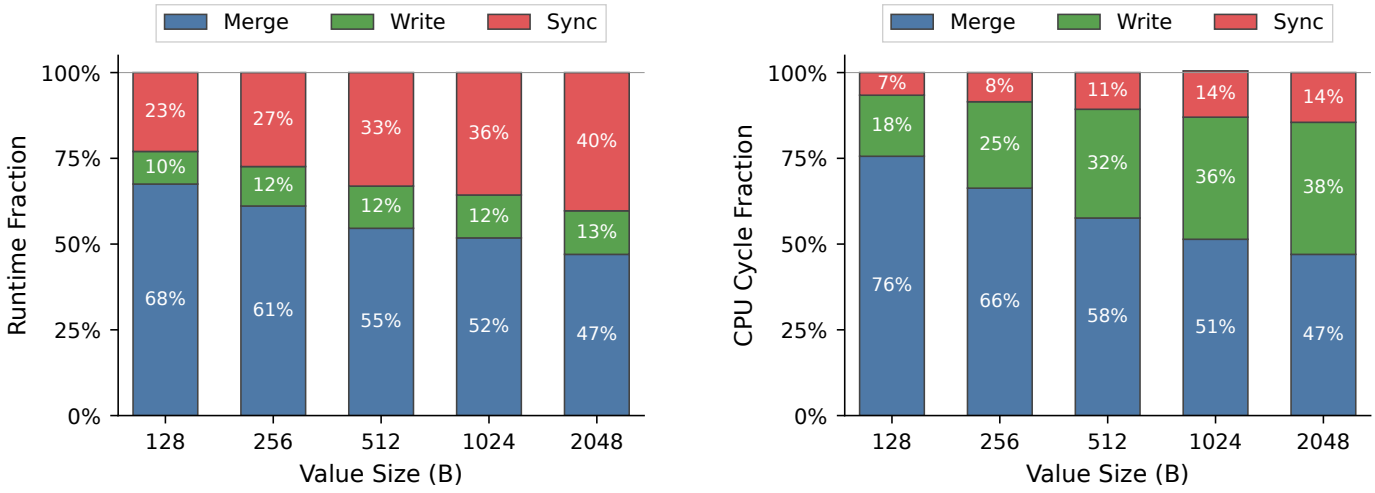
PulseDB addresses these challenges with a range-sync pipeline and `io_uring`-based DPU I/O engine (**C1**), a statically pre-allocated SST file pool with symlink-based naming (**C2**), and bidirectional DMA ring queues with zero-copy data paths (**C3**).



(a) Distribution of compaction data processed by host vs. DPU.

(b) DPU busy/idle timeline during sustained writes.

Fig. 4. DPU participation analysis under DComp-style coarse-grained offloading (`db_bench fillrandom`, 20M KV pairs, key = 16 B, value = 512 B).



(a) Wall-clock time breakdown.

(b) CPU cycle breakdown.

Fig. 5. Compaction sub-task breakdown under varying value sizes (`db_bench fillrandom`, 20M KV pairs, key = 16 B).

III. Design

This section presents PulseDB's design. We first give an overview, then describe the fine-grained compaction pipeline, SST file pool, DPU-driven data transfer, implementation, and limitations.

A. System Overview

PulseDB targets DPU-equipped storage deployments in which the DPU can access the storage device and perform data movement through DMA or storage-path offload engines. It retains merge-encode and metadata management on the host, and asynchronously offloads write-persist work to the DPU. For file sharing, PulseDB leverages a property common to leveled LSM-tree compaction: output SST files are immutable after installation and are generated around a configured target size.

Fig. 6 illustrates the PulseDB architecture. The system comprises three techniques:

1) **Fine-grained compaction pipeline.** PulseDB decouples `write`, `sync_file_range`, and `fsync` from the host compaction thread, processing them with an `io_uring`-based DPU I/O engine and user-space dependency tracking.
2) **Lightweight file sharing via an SST file pool.** A statically pre-allocated pool eliminates runtime metadata mutations on the shared pool partition, while symlinks map logical SST numbers to physical slots.
3) **DPU-driven low-overhead data transfer.** Bidirectional ring queues and DPU-initiated DMA let the host perform only local memory writes and lightweight notifications.

Overall workflow. When the LSM-tree engine triggers a compaction, the host compaction thread constructs a merging iterator over the input SST files and enters the main computation loop. It iterates over key-value pairs, discards expired

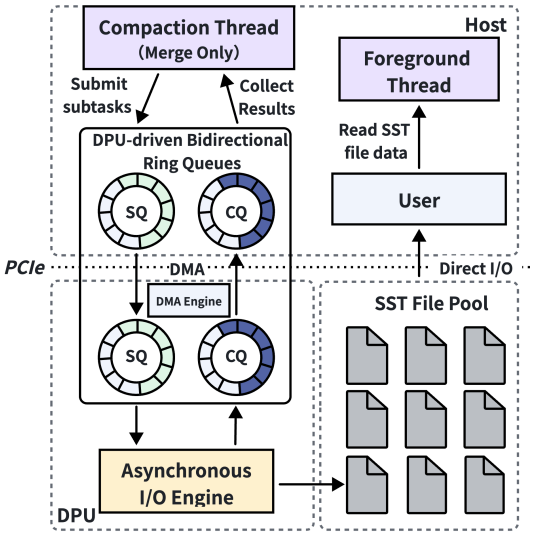


Fig. 6. System architecture of PulseDB. The host retains merge-sort and metadata processing; the DPU handles write and persistence I/O via DMA-backed ring queues and an SST file pool.

or deleted entries, encodes valid entries into data blocks, and appends them to an output buffer allocated directly from pre-registered ring-queue memory. When the buffer accumulates data up to a write threshold, the host asynchronously submits a write request to the DPU and immediately allocates a fresh buffer to continue processing.

Upon receiving a notification, the DPU allocates a slot from the SST file pool for the output file (on its first write request), issues a DMA read to copy the data from the host's request queue into DPU-local memory, and uses `io_uring` to write the data to the pre-allocated SST file. Periodically, the host also submits `range_sync` requests so that the DPU can incrementally flush dirty data via `sync_file_range`, avoiding a large tail-sync penalty. When the output file reaches its size limit, the host submits `sync_close`; the DPU then performs the final `fsync` and closes the file.

After all input key-value pairs have been processed, the host polls the completion queue to collect results from all `sync_close` operations. Each result carries the file number and pool slot identifier, from which the host creates a symlink mapping. Finally, the host installs the new SST files into the LSM-tree version metadata and marks the old input files for deletion.

B. Fine-Grained Compaction Pipeline

In the conventional compaction workflow, merge-sort, `write`, and `fsync` execute serially within a single host thread. PulseDB restructures this flow into a heterogeneous pipeline: the host performs merge-encode while the DPU concurrently executes write and persistence operations, overlapping the two in time so that the end-to-end latency of a single compaction is determined by the longer of the two stages plus a small communication overhead, rather than their serial sum.

1) Pipeline Workflow: Fig. 7 depicts the host-side and DPU-side execution timelines. The host compaction thread submits three types of asynchronous requests to the DPU:

- **`write`**: transfers a buffer of encoded data blocks to be written to the current output SST file.
- **`range_sync`**: triggers `sync_file_range` on the DPU to incrementally flush a specified byte range of already-written data to the storage device.
- **`sync_close`**: performs the final `fsync` and `close` on a completed SST file in a single request, reducing descriptor overhead and queue occupancy.

In the preprocessing phase, the compaction thread prefetches the input SST files and constructs a merging iterator. It then enters the main loop: the iterator locates the smallest key across all input files, discards expired or tombstone-covered entries, and encodes valid entries into an output buffer. This buffer is allocated directly from the pre-registered ring-queue memory (detailed in §III-D), so that the data resides in a DMA-accessible region from the moment it is written, eliminating any subsequent memory copy.

When the output buffer reaches the write threshold, the host posts an asynchronous `write` request to the DPU, allocates a new buffer, and resumes merge-encode without waiting for the DPU. The DPU receives the request, DMA-copies the data to its local memory, and issues an asynchronous `write` via `io_uring`. Meanwhile, when the cumulative unsynced data exceeds a sync interval, the host posts a `range_sync` request so that the DPU can incrementally persist the written data. When the current output SST file reaches its size limit, the host posts a `sync_close` request and begins a new output file by having the DPU allocate a fresh pool slot (§III-C). This loop repeats until all input key-value pairs are consumed.

After the main loop, the host waits for all `sync_close` completions, creates symlinks, installs the new version, and marks old files for deletion.

PulseDB treats DPU offloading as a bounded pipeline stage rather than an unbounded work queue. The host opportunistically uses DPU resources for write-persist work, while the request queue applies backpressure if the DPU temporarily falls behind. Thus, offloaded work cannot accumulate without bound; the host blocks only when the bounded queue lacks reclaimable space.

2) Fine-Grained Sync Mechanism: If the system only performs a single `fsync` when each output SST file is closed, the persistence workload concentrates at the tail of the file's lifetime. During the writing phase, the DPU handles only lightweight `write` requests while the persistence stage remains idle; at close time, a bulk `fsync` creates a long tail-sync delay that surfaces when the host collects completion results, undermining the pipeline overlap.

PulseDB addresses this with the `range_sync` mechanism. During the writing process of each output SST file, whenever the amount of written-but-unsynced data exceeds a configurable sync interval, the host posts a `range_sync`

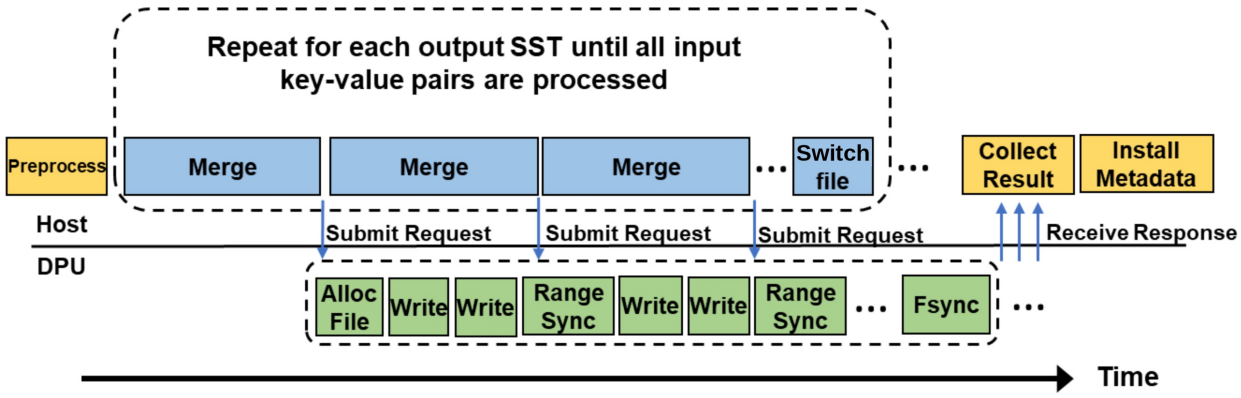


Fig. 7. Timeline of the PulseDB pipelined compaction. Host-side merge-encode overlaps with DPU-side write and persistence, achieving heterogeneous pipeline parallelism.

request. The DPU asynchronously calls `sync_file_range` to flush only the specified byte range to the storage device. By distributing the persistence work across the entire file-writing process, the final `fsync` at close time encounters minimal remaining dirty data, thereby reducing the tail-sync overhead and allowing persistence to overlap more thoroughly with the host's ongoing merge-encode.

3) DPU-Side Asynchronous I/O Engine: The pipeline's effectiveness hinges on the DPU keeping pace with the host's data production rate. A synchronous thread-pool design [11], [12] would spend scarce DPU ARM cycles on per-I/O system calls, blocking waits, and context switching, a cost amplified by the many small write and sync requests generated by fine-grained pipelining.

PulseDB therefore builds a lightweight I/O engine on the DPU around Linux `io_uring`. The engine comprises three components: (1) a *request dispatcher* that parses incoming requests from the DPU-side ring queue and routes them by type (`write`, `range_sync`, or `sync_close`); (2) a *file context manager* that maintains per-active-file state (file descriptor, slot identifier, inflight I/O counters); and (3) an `io_uring` *submission and completion module* that batches I/O requests into SQEs and submits them to the kernel, while a dedicated polling thread harvests CQEs and forwards results to the frontend via a lock-free SPSC queue.

4) User-Space File Dependency Management: Within a single output SST file, strict ordering constraints exist: a `range_sync` must complete only after all `write` operations covering its range have finished, and a `sync_close` must wait for every preceding `write` and `range_sync` on that file. In contrast, I/O operations targeting *different* output files are fully independent and can proceed in parallel.

The native `io_uring` dependency primitives are insufficient for this requirement. `IOSQE_IO_LINK` chains only adjacent SQEs within a single submission batch and cannot span batches. `IOSQE_IO_DRAIN` globally stalls the entire submission queue, serializing I/O across all files and destroying cross-file parallelism.

PulseDB implements a user-space file dependency tracker. Each file context maintains an inflight I/O counter. When a `sync_close` request arrives, it is not immediately submitted; instead, it is deferred in the file context. Each time a `write` or `range_sync` completes for that file, the engine decrements the counter and checks whether the deferred `sync_close` can proceed (i.e., counter reaches zero). If so, it submits an `fsync` SQE linked to a `close` SQE via `IOSQE_IO_LINK` within the same batch, guaranteeing that `close` follows `fsync`. This mechanism preserves intra-file ordering while maximizing inter-file parallelism through `io_uring`'s asynchronous concurrency and the NVMe device's multi-queue capability.

C. Lightweight File Sharing via SST File Pool

In PulseDB, the DPU writes compaction output into SST files while the host subsequently reads them for foreground queries. Since the host and the DPU are separate compute nodes running independent operating systems, they cannot share files through the host's local file system.

1) Why Not a General-Purpose Shared File System?: A natural approach is to use NFS or a distributed file system (e.g., CephFS) so that both endpoints access files through a unified namespace. However, such systems are designed for general-purpose multi-node sharing and impose substantial overhead: NFS requires per-operation network round trips for metadata (create, open, rename) and cache-coherence protocols (invalidation broadcasts, write barriers) to maintain consistency across clients' page caches; distributed file systems additionally involve a metadata service and capability grant/revoke mechanisms. These costs are borne by both the DPU and the host, eroding the CPU savings that offloading is meant to provide [27]–[29].

2) SST File Properties: PulseDB exploits two key properties of SST files to drastically simplify the sharing problem:

- **Immutability.** Once an SST file is fully written and persisted, it becomes read-only for its entire lifetime. There are no concurrent write–write conflicts.

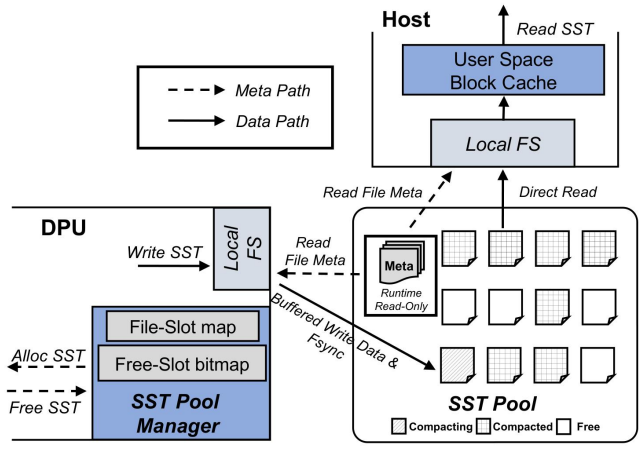


Fig. 8. SST file pool structure and lifecycle. Slots are pre-allocated at initialization. DPU-side allocation and deallocation operate entirely in user-space memory. Symlinks map logical file numbers to physical slot files.

- **Invisibility during writing.** Under LSM-tree MVCC, a new SST file becomes visible to readers only after it is fully written *and* its version metadata is installed. Old SST files continue serving reads during compaction. There are no concurrent read–write conflicts on the same file.

Because of these properties, PulseDB needs no data-block coherence protocol; it only needs to ensure metadata consistency. Leveled compaction also generates output SST files around a configured target size, making static pre-allocation practical.

3) File Pool Design: As shown in Fig. 8, PulseDB stores SST files on a dedicated partition, physically separated from WAL and MANIFEST files. At initialization, the system pre-creates a fixed number of files, each sized to accommodate the target SST size plus padding for index blocks, filter blocks, and footer metadata. Each physical file is identified by a `slot_id`. After initialization, both the host and the DPU mount this partition through their local file-system instances. Since no `create`, `rename`, or `unlink` occurs on the pool partition at runtime, both sides observe a static metadata view.

The DPU maintains a slot allocation table in user-space memory. When a compaction needs a new output file, the DPU allocates a free slot and binds it to the LSM-tree engine's `file_number`. Slot reclamation follows the LSM-tree version lifecycle: after obsolete SST files are removed from the live set, the host deletes their logical links and batches the corresponding file numbers to the DPU, which returns the slots to the free list. Thus, file-pool garbage collection is driven by normal SST deletion and requires no scan or metadata mutation on the pool partition.

The pool is divided into a *resident zone* sized for steady-state load and a *reserve zone* that provides additional slots under peak concurrent compaction pressure. When free slots in the resident zone drop below a low-water-mark threshold, the DPU migrates slots from the reserve zone in bulk—no file-system metadata modification is needed. If even the reserve zone is exhausted, the DPU notifies the host to pause new submissions until in-flight compactions release slots.

Fixed-size slots trade limited internal fragmentation for metadata stability. This tradeoff fits leveled compaction, whose output files are generated around a target size; policies with highly variable SST sizes would require a variable-size or extent-based pool allocator.

4) Symlink-Based Name Mapping: The LSM-tree engine identifies SST files by monotonically increasing file numbers, whereas pool files are named by slot identifiers. PulseDB bridges the two namespaces via symlinks: when the DPU finishes writing and persisting an SST file, it returns the file number and slot identifier to the host through the completion queue. The host then creates a symlink from the logical SST file path to the physical pool file. Unlike `rename`, which would mutate the pool partition's directory entries and inodes—breaking the read-only metadata invariant and potentially causing metadata inconsistencies between the host and the DPU—symlinks are created on the host's own non-pool partition and leave the pool partition untouched.

When old SST files are deleted, the host first removes the corresponding symlink, then batches the file numbers to be reclaimed. Once the batch reaches a threshold, the host sends the numbers to the DPU in a single DMA transfer, amortizing the fixed per-DMA overhead across multiple reclamations.

5) Data Visibility: Although the file pool eliminates metadata conflicts, each node maintains an independent page cache. Even after the DPU has persisted data via `fsync`, the host page cache may still hold stale content for a reused slot. In our implementation, host reads of pooled SST data use aligned Direct I/O on ext4 to avoid serving stale data from the host page cache when a physical slot is reused. PulseDB also relies on the LSM-tree engine's user-space Block Cache, which indexes blocks by logical file number and block offset. When an old SST file leaves the current version, entries for that logical file are evicted. On file systems or configurations without equivalent Direct I/O semantics, the same design would require explicit cache invalidation or delayed slot reuse.

6) Crash Consistency: PulseDB preserves the base LSM-tree engine's commit protocol: new SST files are written and persisted, the version change is appended to and synced in the MANIFEST, and only then are old files removed. The only extra state is the volatile DPU slot allocation table. After a restart, PulseDB recovers it by reading the latest valid MANIFEST version, removing symlinks for non-live SST files, and rebuilding the slot mapping from remaining valid symlinks. Partially written SST files whose symlinks were never created, or whose version changes were never committed, are treated as nonexistent and their slots are reclaimed. If the DPU or PCIe link fails during compaction, PulseDB treats the affected compaction as failed under the same rule; it does not claim online failover.

D. DPU-Driven Low-Overhead Data Transfer

In the fine-grained pipeline, the host submits many data blocks and I/O requests to the DPU. If each submission

incurred synchronous PCIe communication, extra copies, or DMA registration, communication overhead could dominate the saved host CPU cycles. PulseDB therefore pre-registers queue memory once, batches submissions, and lets the DPU drive PCIe data movement through DMA; the host performs only local memory writes and lightweight asynchronous notifications.

1) Bidirectional Ring Queues: PulseDB maintains one queue pair per compaction thread, each consisting of a *request queue* (host→DPU) and a *completion queue* (DPU→host). Mirror copies of each queue reside in both host and DPU memory. Each queue is a single-producer single-consumer (SPSC) ring buffer: the producer atomically advances the write pointer and the consumer advances the read pointer, requiring no locks or CAS operations.

Unlike kernel-space Virtio queues used by prior DPU file-system offloading work [28], PulseDB's queues are entirely in user space, eliminating kernel transitions on every communication. Queue memory is pre-allocated and DMA-registered once at initialization via the DOCA SDK, avoiding per-transfer resource allocation.

The control flow proceeds in four steps: ① The host writes I/O request descriptors and data payloads into its local request queue. When accumulated data reaches a batch threshold (aligned with the write buffer size), the host sends a lightweight asynchronous notification via DOCA Comch, carrying only a few bytes of queue write-pointer offset. ② The DPU receives the notification, and its DMA engine reads the new data from the host's request queue into the DPU's local copy. The host CPU is not involved in the PCIe transfer. ③ After processing the requests and executing I/O, the DPU writes results into its local completion queue, and the DMA engine writes the results back to the host's completion queue. ④ The DPU sends a notification so the host can read the completion results.

The DPU's DMA engine provides approximately 8 GB/s of bandwidth [30], over an order of magnitude above a single LSM-tree instance's typical write bandwidth of 200–300 MB/s. Batch transfer further amortizes the fixed per-DMA overhead.

The control path also handles two auxiliary functions. First, when the request queue runs low on space, the host sends a space-reclaim message to the DPU; the DPU advances the consumer pointer and reports the reclaimed region. Second, when the host enters the completion-wait phase, it sends a progress message so the DPU can prioritize writeback of pending results. A deferred event queue on the DPU side provides backpressure: when local queue space is temporarily insufficient, incoming DMA transfers or result writebacks are queued and retried in FIFO order once space becomes available, preventing the I/O engine's main thread from blocking.

2) Zero-Copy Data Path: Even with DPU-driven DMA, extra memory copies at the endpoints would still consume CPU cycles. Fig. 9 contrasts the conventional and PulseDB data paths.

Host-side zero-copy. PulseDB modifies the LSM-tree engine's compaction output buffer allocator so that buffers are carved directly from the pre-registered request-queue memory. When the compaction thread encodes a data block, the result lands immediately in a DMA-accessible region. Upon posting a `write` request, no copy is needed—the DPU can DMA-read the data in place. The ring-buffer's bounded semantics naturally protect submitted data: a region between the write pointer and the consumer pointer is guaranteed not to be overwritten until the DPU has consumed it, without requiring reference counting or locking.

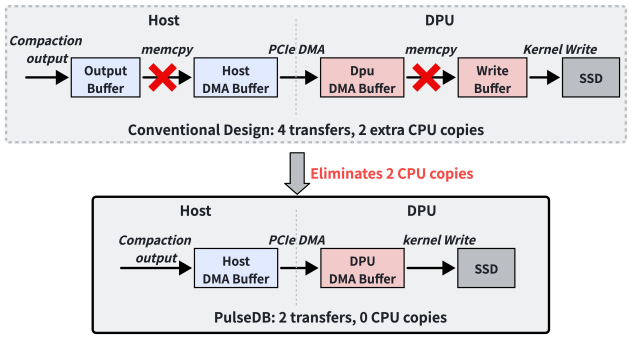


Fig. 9. Comparison of data-copy paths. PulseDB eliminates all intermediate copies by allocating output buffers from pre-registered queue memory and issuing `writev` directly from the DPU receive buffer.

DPU-side zero-copy. After DMA-copying request data into the DPU's local ring buffer, the I/O engine passes the buffer region directly to `writev` as the iovec source. Because data in a ring buffer may wrap around (part at the tail, part at the head), `writev`'s scatter-gather capability avoids the need to linearize the data into a contiguous temporary buffer.

Through this end-to-end zero-copy design, data flows from the host compaction thread's encode output to the DPU file system with no intermediate memory copies, minimizing CPU overhead on both sides of the pipeline.

E. Implementation

Our prototype instantiates this design on RocksDB 10.0.0 and an NVIDIA BlueField-3 DPU with DOCA SDK 2.10.0, adding about 6 KLOC. The host-attached NVMe SSD is exposed to the DPU via NVMe-oF with target offload enabled. On the host, `CompactionJob` allocates one queue pair per compaction, and `WritableFileWriter` exposes three offload calls: `AppendDpu()`, `RangeSyncDpu()`, and `SyncCloseDpu()`. Table-builder data, index, and footer writes are routed through these calls when offloading is enabled. On the DPU, `DmaQpMapping` manages DMA mappings and notifications, `UringEngine` implements the asynchronous I/O engine, and `SstPool` manages slot allocation. PulseDB does not modify core LSM-tree data structures or foreground read/write paths.

F. Limitations and Scope

PulseDB currently targets leveled LSM-tree compaction, where output SST files are immutable after installation and are generated around a configured target size. This covers the

L0→L1 and lower-level compactions evaluated in this paper, but tiered or universal compaction may produce different file lifetimes, overlap patterns, and output-size distributions; supporting them would require adapting the file-pool allocator and scheduling policy. PulseDB also assumes an NVIDIA BlueField-class DPU that can access the storage path and provide efficient DMA or storage-path offload. DPUs without such data-movement support would reduce the benefit of offloading write-persist work. Finally, PulseDB treats DPU or link failure during compaction as a failed compaction and relies on the base LSM-tree recovery protocol; it does not provide online DPU failover.

IV. Evaluation

We evaluate PulseDB on a real NVIDIA BlueField-3 platform. The evaluation asks four questions: How much does PulseDB improve foreground performance? Does it reduce host compaction CPU? Where do the gains come from? How sensitive is the pipeline to DPU-side I/O and queue sizing?

A. Experimental Setup

Hardware. The host server is equipped with two Intel Xeon Gold 5318Y processors (2.10 GHz, 48 physical cores total) and 128 GB DDR4 memory. Storage is provided by an Intel 1.7 TB NVMe SSD. The BlueField-3 DPU connects to the host via PCIe and features 16 ARMv8.2+ A78 cores at 2.0 GHz, 32 GB DDR5 memory, and a ConnectX-7 dual-port 100 GbE RDMA NIC. The DPU accesses the host NVMe SSD remotely through NVMe-oF with Target Offload enabled to minimize protocol processing overhead on the ARM cores.

Software. Both the host (Ubuntu 22.04, kernel 5.15) and DPU run Linux. Our prototype is built on RocksDB 10.0.0 with DOCA SDK 2.10.0, compiled by GCC 11.4.0 at `-O2`, and deployed on ext4.

Baselines. We compare PulseDB against two baselines:

- *RocksDB* [2]: vanilla RocksDB where all compaction runs on the host CPU.
- *DComp* [23]: we reproduce its core scheduling strategy—whole compaction tasks starting from L1 and deeper levels are offloaded to the DPU, while L0→L1 compaction remains on the host. The host retains 4 concurrent compaction threads, the best-performing configuration we measured for DComp.

RocksDB Configuration. To ensure a fair comparison, all three systems share identical RocksDB settings: 4 maximum concurrent compaction threads, 64 MB memtable and SST target file size, 256 MB L1 capacity with a fan-out ratio of 10, 7 levels, Bloom filters at 10 bits/key, 2 GB block cache, no compression, and Direct I/O for reads. Compression is disabled to isolate the merge/write-persist pipeline from compression implementation differences. PulseDB uses a 4 MB write submission threshold, a 1 MB `range_sync` interval, a 32 MB request queue, and a 1 MB completion queue per compaction thread. Before each run we flush the OS page cache via `sync && echo 3 > /proc/sys/vm/drop_caches`. Unless otherwise stated, each data point reports the average over repeated runs after a warm-up phase.

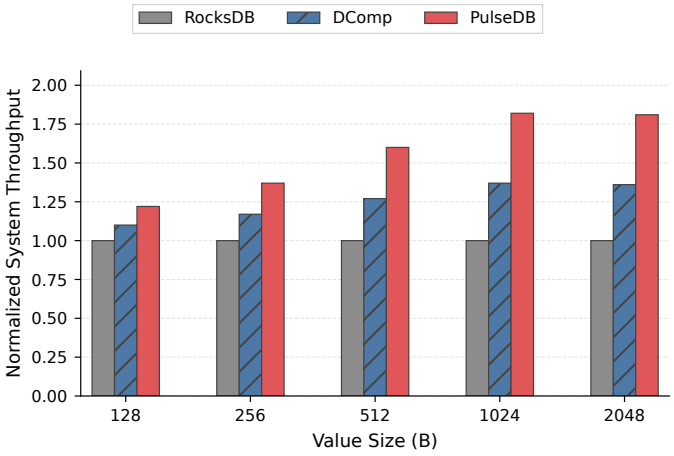


Fig. 10. Normalized write throughput under `fillrandom` (RocksDB = 1).

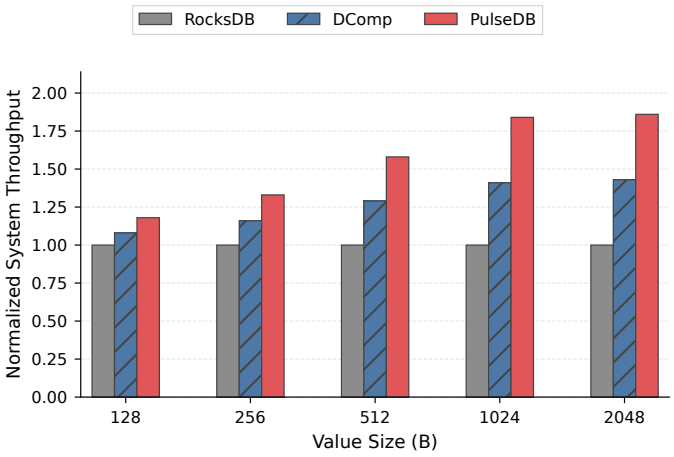


Fig. 11. Normalized write throughput under `overwrite` (RocksDB = 1).

B. db_bench Microbenchmark

We use db_bench with 10 foreground threads writing 40 GB of data (key = 16 B; value = 128–2048 B) under two workloads: `fillrandom` (random inserts into an empty database) and `overwrite` (random updates on an existing 40 GB database).

1) Write Throughput: Fig. 10 shows the normalized write throughput under `fillrandom`. PulseDB achieves a 22–82% improvement over RocksDB across all value sizes, with the gain increasing as values grow larger. Compared with DComp, PulseDB further improves throughput by 11–33%. The trend is consistent with our motivation analysis (§II-D): larger values increase the fraction of compaction time spent in write-persist work, amplifying PulseDB's pipeline benefit.

Fig. 11 shows the `overwrite` results. The trends are similar but the gaps are wider because overwrites trigger more frequent and larger compactions. PulseDB improves throughput by 18–86% over RocksDB and 9–31% over DComp.

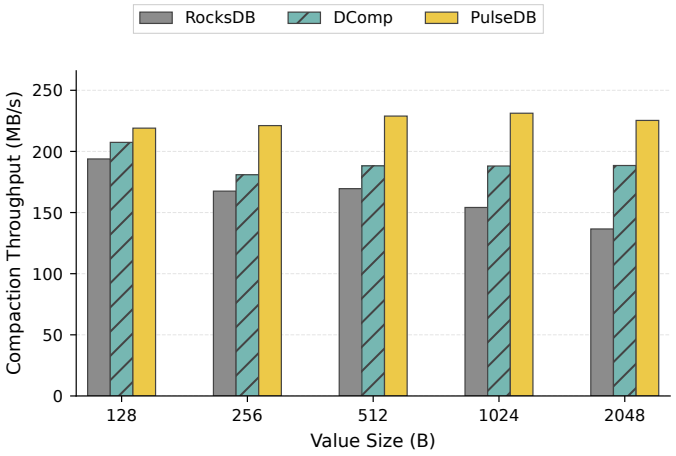


Fig. 12. Compaction throughput under `fillrandom`.

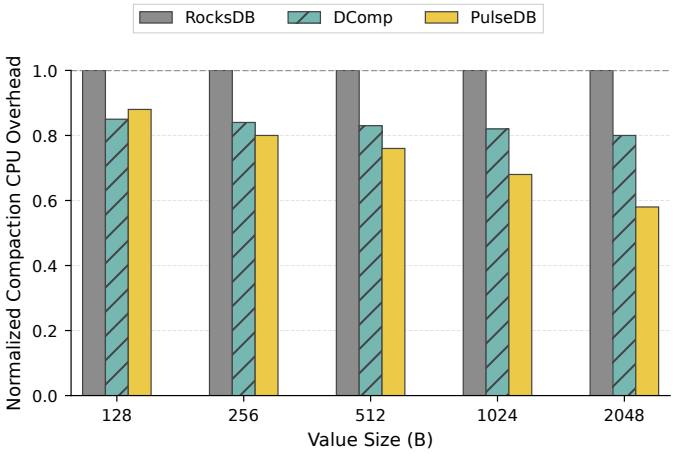


Fig. 13. Normalized host-side compaction CPU overhead under `fillrandom` (RocksDB = 1).

2) Compaction Throughput and Host CPU: We next drill into the compaction process itself. Fig. 12 reports the compaction throughput (MB/s of data produced by compaction). PulseDB outperforms RocksDB by 13–65%, confirming that the pipeline overlap between host-side merge-encode and DPU-side write-persist shortens the compaction critical path. DComp also improves compaction throughput by 7–38% over RocksDB by running some whole compaction tasks on the DPU in parallel with the host, yet still falls short of PulseDB.

Fig. 13 shows the normalized host-side compaction CPU cycles measured with `perf`. PulseDB reduces host compaction CPU overhead by 12–42% relative to RocksDB. The reduction grows with value size because larger values shift a greater share of compaction CPU cycles to write-persist work, which PulseDB offloads to the DPU. By contrast, DComp achieves a 15–20% CPU reduction that plateaus quickly: it offloads whole tasks only for L1+ compactions and still executes L0→L1 compaction—which accounts for 34–41% of total compaction data—entirely on the host. PulseDB's fine-grained approach benefits *all* levels including the critical L0→L1 compaction.

TABLE I
WRITE TAIL LATENCY UNDER `FILLRANDOM` (VALUE = 1024 B).

System	Avg (μs)	P99 (μs)	P99.9 (μs)	P99.99 (μs)
RocksDB	153	3618	6247	22136
DComp	137	3016	4729	18124
PulseDB	113	2567	3644	15108

TABLE II
WRITE STALL RATIO UNDER `FILLRANDOM` (VALUE = 1024 B).

System	Write Stall Ratio (%)
RocksDB	58.2
DComp	50.3
PulseDB	38.9

3) Write Tail Latency: Table I compares write latency at the value size of 1024 B. PulseDB reduces P99 latency by 29% vs. RocksDB and 14.9% vs. DComp; P99.9 latency drops by 41.7% and 22.9%. This improvement comes from accelerating compaction at *all* levels, especially L0→L1, which suppresses L0 file accumulation and the write stalls that dominate the tail.

Table II reports the cumulative write stall ratio, defined as the fraction of total runtime during which the foreground is throttled or blocked by the storage engine. RocksDB stalls for 58.2% of the run, reflecting persistent back-pressure from compaction lag. DComp reduces this to 50.3%; PulseDB further lowers it to 38.9%, a 19.3 percentage-point reduction over RocksDB. By uniformly accelerating compaction across all levels, PulseDB drains stale data faster and sustains a higher effective compaction bandwidth, keeping the foreground path largely unblocked.

C. YCSB Macrobenchmark

To evaluate PulseDB under realistic mixed workloads, we use the Yahoo! Cloud Serving Benchmark (YCSB) [31] with key = 16 B, value = 1024 B. We first load 40 GB of data, then execute 20 M operations with 10 threads under each of the seven standard YCSB phases (Load, A–F). Table III summarizes the workload characteristics.

Fig. 14 shows the throughput of all three systems across the seven YCSB phases.

Write-heavy workloads. PulseDB delivers the largest gains in write-dominated phases: +55% over RocksDB in Load, +35% in Workload A, and +38% in Workload F. High write ratios trigger frequent memtable flushes and cascading compactions; by pipelining write-persist work on the DPU, PulseDB shortens the compaction critical path and relieves back-pressure on the foreground.

Read-heavy workloads. Even under read-dominated Workload B (95% reads), PulseDB still achieves a 13% throughput gain over RocksDB because the remaining 5% writes continuously trigger compactions whose efficiency affects overall performance. Under Workload D (95% reads, Latest distribution), the gain reaches 20%: faster L0→L1 compaction reduces L0 file accumulation and the resulting read amplification for

TABLE III
YCSB WORKLOAD CHARACTERISTICS.

	Load	A	B	C	D	E	F
Write/Update	100%	50%	5%	0%	5%	5%	50%
Read	—	50%	95%	100%	95%	—	50%
Scan	—	—	—	—	—	95%	—
Distribution	—	Zipf	Zipf	Zipf	Latest	Zipf	Zipf

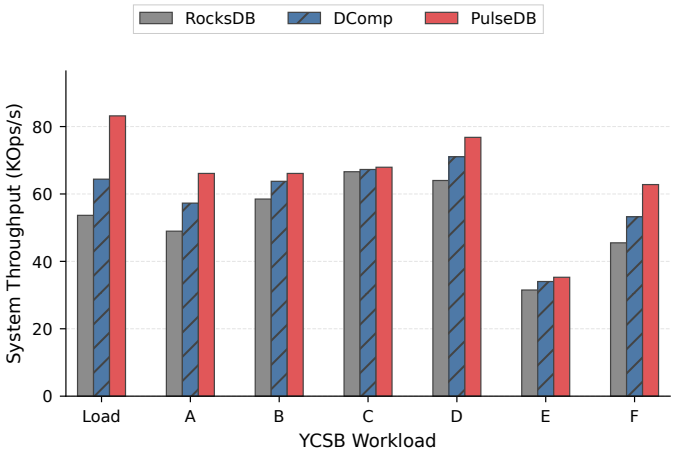


Fig. 14. YCSB throughput across all workload phases.

latest-key lookups. Workload E (95% scans) sees a 12% improvement for a similar reason—fewer L0 files mean fewer merge iterators during range scans.

Pure-read workload. In Workload C (100% reads) PulseDB shows only a marginal 2% improvement, as no compaction occurs during the run phase. The small gain arises from a more compact LSM-tree structure left over from the preceding load phase. Together, the read-heavy phases show that PulseDB preserves throughput under read-dominant workloads; per-cache-state latency breakdowns are complementary to its write-persist offloading focus.

D. Ablation Study

To quantify the contribution of each component, we evaluate four incremental configurations under `fillrandom` with value = 1024 B:

1) **RocksDB**: baseline, all compaction on the host.
2) **+FilePool**: add the statically pre-allocated SST file pool (§III-C); compaction still runs on the host.
3) **+FilePool+AsyncIO**: a CPU-only asynchronous pipeline baseline. It keeps all compaction work on the host but decouples merge-encode from write-persist using host-side `io_uring`.
4) **PulseDB**: the full system with DPU-side pipeline, DMA communication, and `io_uring` I/O engine.

Fig. 15 reports both throughput and normalized host compaction CPU for the four configurations.

Throughput. Adding the file pool alone yields a modest 4% throughput increase by eliminating runtime file creation and deletion overhead. Enabling the CPU-only asynchronous pipeline (**+AsyncIO**) boosts throughput by 86% over RocksDB, demonstrating the substantial benefit of decoupling merge-encode from write-persist execution. The full PulseDB system matches this throughput closely (only 2.1% lower), indicating that DMA transfer and host–DPU coordination do not dominate the critical path.

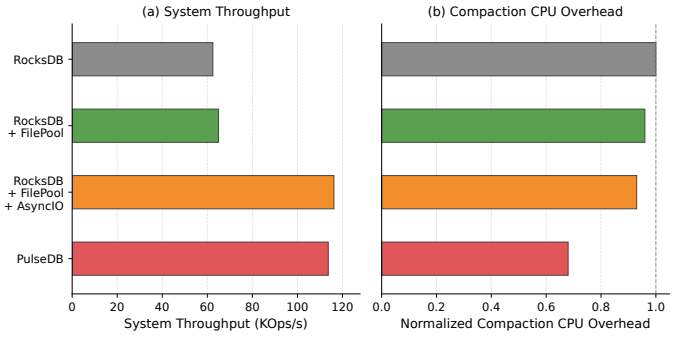


Fig. 15. Ablation: throughput and normalized host compaction CPU.

Host CPU. While **+AsyncIO** reduces host compaction CPU by only 7% because the host still executes all I/O, PulseDB achieves a 32% reduction. The 25-percentage-point gap isolates the value of *offloading* write-persist to the DPU rather than merely making it asynchronous on the host. This confirms PulseDB's core proposition: near-optimal compaction throughput with substantially lower host CPU consumption.

E. DPU-Side I/O Engine Comparison

The DPU-side I/O engine determines how efficiently the DPU executes offloaded write-persist work. We compare four I/O back-ends on the DPU under the same `fillrandom` workload (value = 1024 B):

- **io_uring** (default): interrupt-driven, memory-mapped SQ/CQ.
- **io_uring-poll**: replaces interrupt notification with active polling (`IORING_SETUP_IOPOLL`).
- **SPDK**: user-space NVMe driver, bypasses the kernel I/O stack entirely.
- **Linux AIO**: `libaio`-based asynchronous I/O through `io_submit`/`io_getevents`.

Fig. 16 compares the four back-ends. Relative to the default `io_uring`, `io_uring-poll` improves throughput by 7.6% by eliminating interrupt overhead, while SPDK provides an 11.7% gain by bypassing the kernel entirely. However, SPDK requires exclusive NVMe device access and precludes standard filesystem interfaces, increasing deployment complexity. Linux AIO trails `io_uring` by 13% because its submission and completion paths both require system calls—a cost that is amplified on the DPU's weaker ARM cores. We choose `io_uring` as the default for its strong balance of performance and deployment flexibility: it substantially outperforms Linux AIO, remains within 12% of SPDK, and requires no device exclusivity or dedicated polling cores.

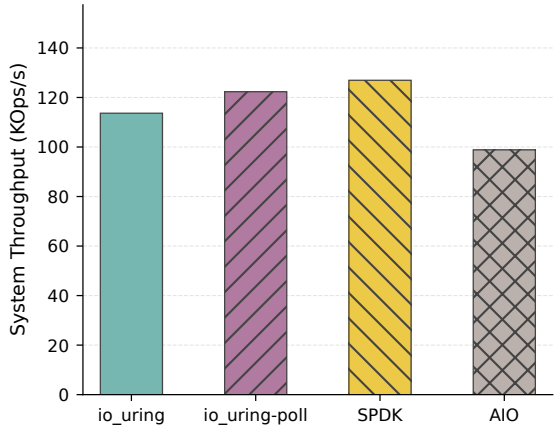


Fig. 16. System throughput with different DPU-side I/O engines.

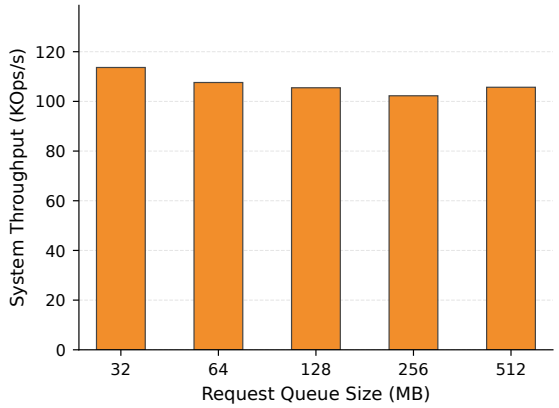


Fig. 17. System throughput vs. request queue size.

F. Request Queue Size Sensitivity

The request queue capacity determines how much data the host can submit to the DPU before blocking for space reclamation. We sweep the per-thread request queue size from 32 MB to 512 MB under `fillrandom` (value = 1024 B), keeping the completion queue fixed at 1 MB.

Fig. 17 shows that throughput varies by roughly 10% across the entire range, indicating moderate sensitivity. The 32 MB configuration achieves the highest throughput: a small queue already sustains the steady-state pipeline, and larger queues incur additional pre-registered memory overhead and DMA management cost without improving host–DPU overlap. PulseDB therefore defaults to 32 MB per thread.

G. Resource Footprint

PulseDB's additional memory footprint is dominated by queue memory. With the default 32 MB request queue and 1 MB completion queue, each active compaction thread uses 33 MB of queue memory per endpoint. With four compaction threads, PulseDB uses 132 MB, less than 0.5% of the BlueField-3 DPU's 32 GB DRAM. The footprint scales linearly with active compaction threads; scheduling multiple database instances on a shared DPU is orthogonal to PulseDB's per-compaction pipeline.

H. Correctness Verification

After each experiment, we verify data integrity using RocksDB's `VerifyChecksum` on all live SST files, run a full-database `CompactRange` followed by a sequential scan to check live keys and values, and test crash recovery by issuing `kill -9` during sustained writes before restarting the database. All checks pass across every experiment configuration, confirming that PulseDB preserves the base engine's correctness guarantees.

V. Related Work

Software-level compaction optimizations. A rich body of work improves compaction through purely software mechanisms. Strategy-oriented approaches tune amplification tradeoffs or coordinate background pacing: Dostoevsky [32] introduces Lazy Leveling, Monkey [33] optimally distributes Bloom filter bits, SILK [9] applies latency-aware I/O scheduling, and ADOC [8] automatically harmonizes compaction across levels. Structural approaches redesign the LSM-tree itself: WiscKey [34] separates keys from values, HashKV [35] adds hash-based value grouping, and PebblesDB [36] introduces fragmented log-structured merge trees. These works reduce compaction overhead or side effects but still run compaction entirely on the host CPU. PulseDB is orthogonal and can be layered on top of any of them.

Hardware-accelerated compaction. FPGA solutions [13]–[15] build custom dataflow pipelines for merge-sort and encoding, while GPU approaches [16], [17] exploit massive parallelism for sorting. Both require purpose-built hardware and impose deployment constraints. Near-data processing [18] and disaggregated compaction services [19] relocate work but depend on specific device capabilities or infrastructure topologies. DComp [23] is the first to offload whole compaction tasks to DPU ARM cores; D^{2}Comp [24] extends this to disaggregated storage with hardware compression; DFlush [30] explores DPU-assisted memtable flushing. All DPU-based schemes adopt coarse-grained, whole-task offloading. PulseDB departs fundamentally by decomposing each compaction into sub-tasks, retaining merge-encode on host x86 cores and offloading only write-persist to the DPU, forming a heterogeneous pipeline that benefits *every* compaction level—including L0→L1 that DComp must leave on the host—without suffering the throughput degradation of running full merge on ARM cores.

DPU-based storage systems. DPFS [28] offloads file-system virtualization, DPC [27] accelerates file-system client operations, LineFS [29] pipelines distributed file-system replication on SmartNICs, and STYX [22] reduces the host memory tax of infrastructure services. PulseDB builds on the same positional advantage of DPUs but tailors the offloading boundary specifically for LSM-tree compaction via a statically preallocated file pool and DPU-driven zero-copy communication.

VI. Conclusion

Compaction is the dominant source of host CPU overhead in LSM-tree key-value stores, yet existing DPU-based offloading schemes migrate whole compaction tasks to the DPU, suffering from either significant performance degradation or limited host CPU relief due to the constrained processing capability of DPU ARM cores.

We presented PulseDB, a fine-grained DPU-assisted compaction offloading system that decomposes each compaction into merge-encode, write, and persistence-synchronization sub-tasks. PulseDB retains merge-encode on the host and asynchronously offloads write and persistence sub-tasks to the DPU, forming a heterogeneous pipeline that benefits every compaction level—including the critical L0→L1. Three key techniques support this design: (1) a DPU-based fine-grained compaction pipeline backed by an `io_uring` asynchronous I/O engine with user-space file dependency tracking, (2) a lightweight SST file-sharing mechanism based on a statically pre-allocated file pool that exploits the immutability and near-fixed size of SST files, and (3) a DPU-driven bidirectional DMA ring-queue communication layer with zero-copy data paths that eliminates host-side PCIe transfer overhead.

Experiments on a real NVIDIA BlueField-class platform show that PulseDB improves write throughput by 22–82% over vanilla RocksDB and 11–33% over the state-of-the-art DComp, while reducing host-side compaction CPU overhead by 12–42% and P99 write tail latency by 29%. Under YCSB write-intensive workloads, PulseDB achieves throughput gains of up to 55%. Ablation results confirm that the full system retains near-optimal throughput while shifting a substantial fraction of compaction CPU work from the host to the DPU. Our results demonstrate that fine-grained sub-task decomposition, rather than whole-task migration, is the key to effective DPU-assisted compaction offloading in heterogeneous datacenter environments. Because PulseDB does not alter the LSM-tree data organization or compaction strategy, it can be combined with complementary software-level optimizations such as alternative compaction policies or key–value separation schemes. We believe the sub-task offloading methodology demonstrated in PulseDB generalizes to other background storage maintenance tasks where compute and I/O phases can be decoupled across heterogeneous processing elements.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Key R&D Program of China No. 2023YFB4502703 and the National Natural Science Foundation of China under Grant No. U22A2027.

References

[1] S. Ghemawat and J. Dean, "LevelDB," https://github.com/google/leveldb, 2011.

[2] Facebook, "RocksDB: A Persistent Key-Value Store for Flash and RAM Storage," https://github.com/facebook/rocksdb, 2024.

[3] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[4] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, 2019, pp. 651–665.

[5] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[6] C. Luo and M. J. Carey, "LSM-based Storage Techniques: A Survey," *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

[7] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis, "Constructing and Analyzing the LSM Compaction Design Space," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2216–2229, 2021.

[8] J. Yu, S. H. Noh, Y.-r. Choi, and C. J. Xue, "ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance," in *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*, 2023, pp. 65–80.

[9] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*, 2019, pp. 753–766.

[10] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020, pp. 209–223.

[11] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "KVell: the Design and Implementation of a Fast Persistent Key-Value Store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019, pp. 447–461.

[12] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021, pp. 17–32.

[13] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun, "FPGA-Accelerated Compactions for LSM-based Key-Value Store," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020, pp. 225–237.

[14] X. Sun, J. Yu, Z. Zhou, and C. J. Xue, "FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores," in *Proceedings of the IEEE 36th International Conference on Data Engineering (ICDE'20)*, 2020, pp. 1261–1272.

[15] D. Tang, W. Wang, Y. Mao, J. Yu, T.-W. Kuo, and C. J. Xue, "STEM: Streaming-based FPGA Acceleration for Large-Scale Compactions in LSM KV," in *Proceedings of the 40th IEEE International Conference on Data Engineering (ICDE'24)*, 2024, pp. 3893–3905.

[16] P. Xu, J. Wan, P. Huang, X. Yang, C. Tang, F. Wu, and C. Xie, "LUDA: Boost LSM Key Value Store Compactions with GPUs," *arXiv preprint arXiv:2004.03054*, 2020.

[17] H. Sun, J. Xu, X. Jiang, G. Chen, Y. Yue, and X. Qin, "gLSM: Using GPGPU to Accelerate Compactions in LSM-tree-based Key-value Stores," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 1, pp. 1–26, 2024.

[18] H. Sun, B. Lou, C. Zhao, D. Kong, C. Zhang, J. Huang, Y. Yue, and X. Qin, "Asynchronous Compaction Acceleration Scheme for Near-data Processing-enabled LSM-tree-based KV Stores," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 6, pp. 1–26, 2024.

[19] Q. Yu, C. Guo, J. Zhuang, V. Thakkar, J. Wang, and Z. Cao, "CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, 2024.

[20] N. Tibbetts, S. Ibtisum, and S. Puri, "A Survey on Heterogeneous Computing Using SmartNICs and Emerging Data Processing Units," *Future Generation Computer Systems*, vol. 176, p. 108207, 2026.

[21] J. Humphries, N. Natu, K. Kaffes, S. Novakovic, P. Turner, H. Levy, D. Culler, and C. Kozyrakis, "Wave: Offloading Resource Management to SmartNIC Cores," in *Proceedings of the Thirtieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*, 2025, pp. 907–923.

[22] H. Ji, M. Mansi, Y. Sun, Y. Yuan, J. Huang, R. Kuper, M. M. Swift, and N. S. Kim, "STYX: Exploiting SmartNIC Capability to Reduce Datacenter Memory Tax," in *Proceedings of the 2023 USENIX Annual Technical Conference (ATC'23)*, 2023, pp. 615–631.

[23] C. Ding, J. Zhou, J. Wan, Y. Xiong, S. Li, S. Chen, H. Liu, L. Tang, L. Zhan, K. Lu, and P. Xu, "DComp: Efficient Offload of LSM-tree Compaction with Data Processing Units," in *Proceedings of the 52nd International Conference on Parallel Processing (ICPP'23)*, 2023, pp. 132–141.

[24] C. Ding, J. Zhou, K. Lu, S. Li, Y. Xiong, J. Wan, and L. Zhan, "D2Comp: Efficient Offload of LSM-tree Compaction with Data Processing Units on Disaggregated Storage," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 3, pp. 1–22, 2024.

[25] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021, pp. 33–49.

[26] J. Liu, C. Maltzahn, C. Ulmer, and M. L. Curry, "Performance Characteristics of the BlueField-2 SmartNIC," *arXiv preprint arXiv:2105.06619*, 2021.

[27] K. Zhong, Z. Yu, Q. Li, X. Luo, L. Long, Y. Tan, A. Ren, and D. Liu, "DPC: DPU-accelerated High-Performance File System Client," in *Proceedings of the 53rd International Conference on Parallel Processing (ICPP'24)*, 2024, pp. 63–72.

[28] P.-J. Gootzen, J. Pfefferle, R. Stoica, and A. Trivedi, "DPFS: DPU-Powered File System Virtualization," in *Proceedings of the 16th ACM International Conference on Systems and Storage (SYSTOR'23)*, 2023, pp. 1–7.

[29] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostic, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*, 2021, pp. 756–771.

[30] C. Ding, K. Lu, Q. Zhang, Z. Ye, T. Yao, D. Wang, H. Wu, and J. Wan, "DFlush: DPU-Offloaded Flush for Disaggregated LSM-based Key-Value Stores," *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, 2025.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, 2010, pp. 143–154.

[32] N. Dayan and S. Idreos, "Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, 2018, pp. 613–628.

[33] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal Navigable Key-Value Store," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, 2017, pp. 79–94.

[34] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating Keys from Values in SSD-Conscious Storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, 2016, pp. 133–148.

[35] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling Efficient Updates in KV Storage via Hashing," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*, 2018, pp. 1–14.

[36] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017, pp. 497–514.