ARTIFACT EVALUATED USENIX AVAILABLE
ARTIFACT EVALUATED USENIX FUNCTIONAL

Flash-Level Asynchronous Execution for In-Storage LSM-Tree Operations

Jaehoon Shim, Hyungjoon Kwon, Hyungsoo Jung, and Jin-Soo Kim
Seoul National University

Abstract

Computational storage devices (CSDs) promise to reduce host–storage data movement by executing computation inside the storage device. This is especially appealing for LSM-tree-based key–value stores, whose reads and background maintenance move large amounts of data through the host. In practice, however, in-storage LSM-tree operations often fail to deliver their full benefit. Existing CSDs largely follow a host-managed, copy-then-execute model that drives in-storage operators through synchronous or weakly overlapped I/O paths. This serializes compute–flash interactions, underutilizes device-side compute resources, and forces host storage software to handle complex coordination logic. As a result, the expected benefits of in-storage processing are often limited or even negated on realistic workloads.

We argue that the missing abstraction is *asynchronous execution with device-resident dependency management*. We present FLAX, a computational storage architecture that makes asynchronous, overlapped in-device I/O execution a first-class design goal for LSM-tree operations. FLAX tracks fine-grained dependencies inside the device, allowing the host to issue large batches of work while the CSD schedules and executes commands asynchronously. To further improve utilization, FLAX combines adaptive data loading with dynamic on-device buffer management, enabling higher concurrency under limited device resources. We integrate FLAX into RocksDB and offload both compaction and reads. FLAX improves throughput by up to 3.1× over host-only execution and 1.8× over a host-managed CSD baseline.

1 Introduction

As data-intensive applications continue to scale [39], host–storage data movement has become a primary bottleneck. This pressure is especially visible in Log-Structured Merge Tree-based key–value stores (LSM-KVSs) [22, 30, 35, 42], which achieve write performance at the cost of background maintenance (flush and compaction) across multiple levels. Since these systems are often provisioned with memory budgets far smaller than the underlying SSD capacity [2, 40, 59, 72], their core operations repeatedly shuttle data and metadata through the host CPU and memory hierarchy, leading to CPU and memory pressure that limits performance metrics.

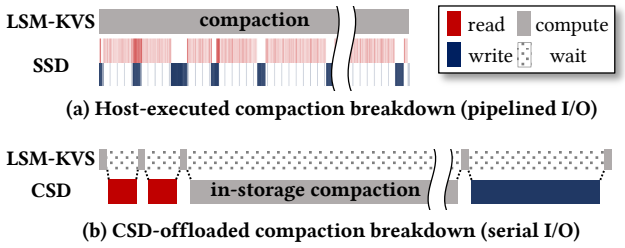


Figure 1: Compaction execution: Host vs. CSD.

A long line of work has sought to reduce this overhead by offloading background work to accelerators such as FPGAs [54, 69], GPUs [53, 74], DPUs [10], and computational storage devices (CSDs) [12]. These designs can improve write throughput and reduce host CPU utilization. In particular, because CSDs sit directly on the I/O path, they can execute computation inside the storage device and avoid the explicit PCIe transfers required by external accelerators [20, 26, 46, 49, 63]. This makes CSDs a compelling substrate for accelerating both large-scale background operations and foreground requests.

In practice, however, offloading heterogeneous LSM-KVS operations to a general-purpose CSD remains difficult. In many cases, CSD-based execution ***fails to outperform carefully tuned host-only processing*** (Figure 1-(a)). A key reason is how current systems orchestrate computation and I/O inside the device. Most existing CSDs, both commercial products and research prototypes, expose an interface in which the host issues commands that trigger a fixed sequence of flash operations and device-side computation, and then waits for completion—a traditional host-managed *copy-then-execute* model [17, 47] (Figure 1-(b)). Even when some overlap is possible, the host-visible semantics remain effectively blocking: ***data dependencies among reads, writes, and compute phases are hidden in firmware***, preventing the host from safely keeping many interdependent operations in flight. For

LSM-KVS workloads with pipelined multi-stage execution, this reduces CSDs to "*smart blocking I/O*" rather than fully asynchronous accelerators. Figure 1 makes this limitation concrete by breaking down compaction executed **(a)** on the host with I/O–compute overlap and **(b)** with CSD assistance, showing that the in-storage I/O–compute–I/O phases remain *serially* coordinated by the host.

In this work, we argue that the missing piece is a principled, device-resident substrate for asynchronous execution in CSDs. Rather than treating near-data computation as an opaque extension of synchronous I/O, CSDs should be driven by a *stream of outstanding operations whose* ***dependencies are managed within the device***. Device-side dependency management is the key enabler: it allows the host to issue large batches of work while the device schedules and executes commands concurrently. To this end, we present FLAX (**F**lash-**L**evel **A**synchronous e**X**ecution), a CSD architecture built for asynchronous, overlapped in-device I/O execution.

FLAX addresses two core obstacles to efficient offloading: tracking inter-operation dependencies and managing limited on-device memory under concurrency. It does so through lightweight *page-granular dependency tracking*, which captures dependencies across both data movement and computation, and three mechanisms built on top of it. ❶ *Asynchronous command execution* shifts dependency handling from the host to the device, allowing FLAX to enforce fine-grained ordering locally for LSM-KVS operations such as flash reads, intermediate-buffer writes, and key lookups. ❷ *Adaptive data loading* schedules internal data transfers based on runtime progress, combining prefetching and demand loading to reduce redundant reads and flash-path contention. ❸ *Dynamic buffer management* lets operations span a logical buffer space larger than physical CSD memory by continuously adjusting the resident working set through eviction and reuse. These mechanisms enable concurrent, partially overlapping offloaded operations without exposing low-level dependency or memory management to the host.

FLAX builds on the NVMe CSD standard [14, 16], which generalizes earlier general-purpose CSDs and provides a flexible command and memory model for diverse offloading scenarios. This foundation lets FLAX realize device-resident, dependency-aware asynchronous execution without proprietary interfaces. To support rapid prototyping and reproducible evaluation, we developed an NVMe CSD-compliant emulator and validated it against physical devices.

For evaluation, we integrated FLAX into RocksDB [38]. Our analysis identifies compaction and reads as the dominant sources of data movement and host overhead, making them the most promising targets for CSD offloading. We therefore offload both. FLAX improves throughput by up to 3.1× over host-only execution and 1.8× over a host-managed CSD baseline. It also reduces write and read tail latencies by up to 85% and 77%, respectively, whereas the host-managed baseline often matches the host-only tail latency. To the best of our knowledge, FLAX is the first general-purpose CSD architecture that efficiently supports concurrent, heterogeneous operations in a real-world storage system. The source code of FLAX is publicly available at https://github.com/snu-csl/FLAX.

Contributions. The contributions of this work are as follows:

- We analyze the NVMe CSD standard and show how its flexible command and memory model generalizes prior general-purpose CSD designs.
- We develop an NVMe CSD-compliant emulator and validate its accuracy against physical devices, a practical platform for designing and evaluating CSD architectures.
- We identify the key limitations of prior general-purpose CSDs for LSM-KVS workloads, including host-managed execution, constrained on-device memory, and interference with foreground host I/O.
- We present FLAX, a general-purpose CSD architecture that overcomes these limitations through page-granular dependency tracking, asynchronous command execution, adaptive data loading, and dynamic buffer management.
- We integrate FLAX into RocksDB and show that offloading both foreground and background operations is essential to deliver substantial performance improvements.

2 Background

2.1 CSD and In-Storage Processing

Computational storage devices [5, 20, 26, 46, 49, 61] integrate storage media with on-device compute resources and local memory, enabling parts of an application's data path to run close to where data resides. For data-intensive workloads, this is appealing: offloading computations such as merging, filtering, and lookup to the device can reduce PCIe traffic, lower host CPU load, relieve host memory pressure, and potentially improve throughput and energy efficiency [11, 21, 58]. Recent work has explored this idea across a broad range of workloads, including analytics, recommendation, and deep learning inference [18, 29, 31, 34, 58, 60, 63]. Yet CSD-based execution often fails to outperform carefully tuned host-only processing. The main reason is not the lack of device-side compute, but how existing CSDs orchestrate computation and I/O and expose that execution model to the host.

Beyond the host-managed, copy-then-execute execution model described earlier, prior CSD designs also suffer from limited generality. Many are tightly coupled to specific compute units or hardware architectures [26, 48, 50]; their APIs often support only a single input and output source per task, which makes it difficult to offload more complex operations spanning multiple data sources [28, 63]; and their specialization to particular access patterns further narrows applicability [43, 75]. As a result, evaluations of these systems have largely been confined to simple synthetic workloads, leaving it unclear how well general-purpose CSDs support heterogeneous, real-world storage workloads.

2.2 NVMe CSD Programming Model

Building on earlier general-purpose CSD architectures and programming models, the NVMe CSD standard [14–16] extends the NVMe storage interface with abstractions for invoking computation near flash. It defines a flexible, hardware-agnostic interface that relaxes several limitations of prior designs while integrating CSD functionality with conventional block I/O on the same device. At a high level, the standard provides (i) a device-memory abstraction for storing inputs, outputs, and intermediate state, and (ii) compute-oriented commands that reference this memory and interact with flash-resident data. These commands are issued over standard NVMe queues and completed through the normal NVMe completion mechanism.

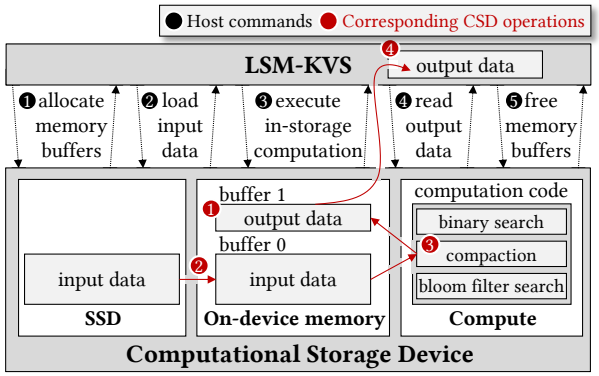


Figure 2: In-storage data processing with a legacy CSD.

Figure 2 shows the standard architecture and offloading procedure. The NVMe CSD model consists of three hardware components: SSD, on-device memory, and compute. The *SSD* component is identical to a conventional NVMe SSD and stores persistent data. The *on-device memory* component provides internal memory for offloaded computations. The *compute* component contains the compute units and their associated code. Each component has its own controller and command set, enabling independent management.

Table 1 summarizes the key NVMe CSD commands. `ALLOC` allocates or frees physically contiguous on-device memory buffers. `READ` and `WRITE` access the SSD, whereas `MEMREAD` and `MEMWRITE` operate on the on-device memory. `MEMCPY` transfers data internally from the SSD to the memory, while `SSDCPY` moves data from the memory back into the SSD. Both copy commands use source range entries [51] to specify multiple sources, enabling bulk data movement. `EXEC` supports multiple input and output buffers and enforces access protection, enabling complex tasks that involve multiple data sources and chained execution via shared buffers.

The standard formalizes a host-managed programming model as a synchronous sequence of NVMe commands, where each command completes before the next is issued. First, the host allocates input and output buffers via `ALLOC` (❶), and the CSD reserves the corresponding on-device memory regions (1). The host then copies input data from the

Table 1: Key NVMe CSD commands.

Component	Command	Description
SSD	Read (`READ`)	Read data from SSD to host
	Write (`WRITE`)	Write data from host to SSD
	Copy (`SSDCPY`)	Copy data from memory/SSD to SSD
On-device memory	Memory read (`MEMREAD`)	Read data from memory to host
	Memory write (`MEMWRITE`)	Write data from host to memory
	Memory copy (`MEMCPY`)	Copy data from SSD/memory to memory
Compute	Load (`LOAD`)	Load/delete computation code
	Execute (`EXEC`)	Execute computation with parameters
	Buffer allocation (`ALLOC`)	Allocate/free on-device memory buffer

SSD into the input buffer using `MEMCPY` (❷), triggering an internal DMA transfer from flash to buffer 0 (2). After the transfer completes, the host issues `EXEC` (❸), and the CSD runs the requested computation—e.g., binary search, compaction—over the loaded data (3). Once execution finishes, the host retrieves the output using `MEMREAD` (❹), and the CSD transfers the results from buffer 1 back to the LSM-KVS (4). Alternatively, the output can be written directly to storage via `SSDCPY`. This step may repeat because of NVMe transfer-size limits [13]. Finally, the host releases the buffers using `ALLOC` (❺).

This programming model offers important benefits over earlier ad hoc CSD interfaces, but still largely assumes a host-managed, *copy-then-execute* execution style. Specifically, inputs are typically copied in bulk, a compute command is issued, and results are copied back only after execution completes. Dependencies among commands, and the relationship between compute kernels and underlying flash I/Os, remain opaque to the host. This limits fine-grained asynchrony and leads to poor device utilization under LSM-KVS workloads.

2.3 Characteristics of LSM-KVS Operations

LSM-KVS operations such as flush, compaction, and read involve substantial data movement and often dominate end-to-end performance [3, 9, 32]. They are inherently *data-driven*: their control flow and data access patterns are determined at runtime by input keys, values, or intermediate results. Compaction is a representative example. It reads multiple sorted SSTables across levels, merges and filters entries (e.g., obsolete versions and tombstones), and writes back new SSTables. The exact sequence of reads and writes depends on key ordering, value metadata, and per-entry decisions made during the merge. As a result, compaction consists of tightly interleaved I/O and computation phases whose dependencies cannot be determined from static metadata alone.

A read operation resembles the index traversal in B$^+$-tree lookups, in which each request follows a sequence of block accesses dynamically determined at runtime by the queried key and the current index structure. Specifically, if the key is absent from the memtable, the system searches SSTables level by level, starting from L_0. For each SSTable, it first reads the index block to locate the relevant data block and then reads that block to check whether the key exists. If the key is not found, the search proceeds to the next level; each subsequent access depends on the result of the previous read. Offloading such traversals to a CSD using bulk staging and

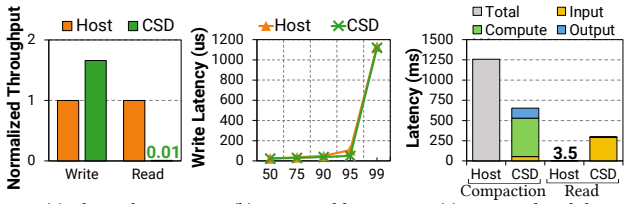
synchronous commands either over-reads data to avoid round-trips or incurs frequent host–device interactions, undermining the benefit of ISP.

These examples highlight two recurring properties of LSM-KVS operations. First, they exhibit fine-grained dependencies between I/O and computation that are hard to express as a small number of coarse-grained offload kernels. Second, their performance is susceptible to how data movement is interleaved with computation, especially when device memory is limited and SSD bandwidth is constrained. Existing general-purpose CSDs, operating under a host-managed copy-then-execute model, can handle simple, partitionable workloads whose access patterns are known in advance. However, they struggle to exploit the characteristics of data-driven LSM-tree operations, where dependencies, access locations, and output sizes are determined at runtime. This either underutilizes device compute units or generates bulk internal traffic that interferes with foreground host I/O. Building on this observation, we present FLAX, which introduces in-device, page-granular dependency management and asynchronous execution to support data-driven LSM-tree operations.

3 Motivation

This section provides an in-depth analysis of the benefits and the limitations of offloading LSM-KVS operations to CSDs. We focus on compaction as the primary background operation and on read as the representative foreground operation for three reasons. First, although flush consumes non-trivial CPU resources, its input data reside entirely in host memory and are not accessible to the CSD. Moreover, because flush simply serializes in-memory data to disk without loading input from storage, it incurs less data movement than compaction. By offloading compaction and read, CPU and memory resources are freed on the host, improving the efficiency of flush. Second, write operations append data to the *memtable* and return immediately. Thus, write latency is influenced by the background compaction, since writes are stalled only when the memtable cannot be flushed due to saturation at lower levels [3, 32]. Third, scans are similar to reads but traverse key ranges by repeatedly invoking the `next()` API. Offloading such calls to the CSD incurs a PCIe round trip per `next()` invocation while offering minimal reduction in data movement.

Figure 3 presents microbenchmark results from RocksDB, comparing host-only execution with CSD offloading. We use 16 B keys and 128 B values (see §6 for more details). To isolate the impact of each offloaded operation, we evaluate two workloads: write-only and read-only. Under the write-only workload, offloading compaction reduces average compaction latency by leveraging in-device data transfers and bypassing the host I/O stack, whereas frequent page cache accesses otherwise degrade host performance [45, 65]. This yields a 66% throughput improvement. However, as shown in Figure 3-(b), CSD offloading fails to improve write tail latency. The limitations are more prominent under the read-only workload:



(a) Throughput (b) Write tail latency (c) Latency breakdown

Figure 3: Microbenchmark results on RocksDB.

CSD offloading increases read latency by 102×. These results demonstrate that LSM-KVS fail to exploit the full potential of CSDs when offloading compaction and read operations.

Compute unit underutilization. Without overlap between data movement and computation, the compute unit in the host-managed programming model remains idle for a substantial fraction of time, e.g., 27% during compaction. The inefficiency is even more noticeable for reads: the full input must be transferred before computation can begin, even though the operation ultimately touches only a small portion of the file (e.g., the index block and a single data block). As a result, compute-unit idle time can reach 99%. Prior CSDs [48, 63] attempt to mitigate this by partitioning inputs and parallelizing computation across multiple compute units. Such optimizations, however, are poorly suited to LSM-KVS computations, whose interdependencies require largely serialized execution. Consequently, the CSD often uses only a single compute unit at a time despite having additional units available.

SSD bandwidth contention. Internal data transfer between SSD and CSD memory is typically performed in large chunks to exploit the SSD's internal parallelism. For example, the standard `MEMCPY` command is designed to copy large objects, such as an entire 64 MB SSTable, in a single operation. However, these bulk transfers can contend with concurrent host I/Os for SSD bandwidth, limiting any benefit for latency-sensitive foreground requests (Figure 3-(b)). Even in the absence of host I/O, internal transfers from multiple offloaded computations can interfere with one another, delaying completion and increasing compute-unit idle time.

Limited concurrency. The host-managed model restricts concurrency for two reasons. First, CSDs operate under tight resource constraints, especially limited on-device memory [19, 61, 63]. Because the host-managed model requires over-provisioned buffers to accommodate unpredictable output sizes and to retain all input and output data until completion, concurrent computations can quickly exhaust available memory and trigger out-of-memory conditions. Second, the strictly serialized execution pipeline limits intra-query parallelism within a single computation, because each computation must complete before the next can begin under the host-managed copy-then-execute model.

Overall, the host-managed copy-then-execute model introduces substantial inefficiencies when offloading LSM-KVS operations. These limitations persist and prevent LSM-KVS workloads from fully exploiting CSD resources, as long as the host remains responsible for dependency management.

4 Design

FLAX aims to enable concurrent offloading of heterogeneous operations through fine-grained in-device dependency management, thus achieving three design goals; (i) FLAX aims to support diverse operations with distinct characteristics, while avoiding reliance on specific hardware features; (ii) FLAX seeks to fully utilize the CSD's compute units, memory, and SSD bandwidth with low contention; (iii) for practical deployment, FLAX's features must be lightweight with minimal overhead in resource-constrained storage devices.

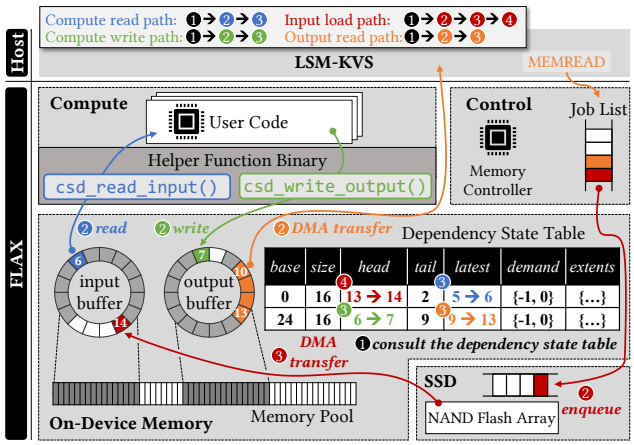


Figure 4: Overall architecture of FLAX.

4.1 Overview

FLAX builds on the NVMe CSD standard (see Figure 4). Its key distinction is how it manages on-device memory to enable page-granular, in-device dependency management. To support this, FLAX introduces the **dependency state table**, a key metadata structure that tracks the page-level progress of each allocated buffer. This table serves as the central coordination point between computation and data movement: it records which pages are ready for access and which pages must be loaded next. Hence, FLAX preserves dependencies and ensures correct execution. To this end, FLAX employs three techniques. **(a)** *Asynchronous command execution* shifts dependency management from the host to the device and improves compute unit utilization by overlapping computation with data movement (§4.3). **(b)** *Adaptive data loading* reduces unnecessary SSD bandwidth consumption by combining prefetching and demand loading based on runtime progress and access patterns (§4.4). **(c)** *Dynamic buffer management* breaks the on-device memory capacity wall by exposing a larger logical buffer space over a smaller physical buffer through page reuse (§4.5).

4.2 Page-Granular Dependency Tracking

FLAX supports page-granular dependency management. One straightforward approach is to maintain a per-page bitmap for each allocated buffer. However, bitmap-based tracking is ill-suited for LSM-KVS operations. Although these operations access data at fine 4 KB granularity, they typically consume and produce data in a monotonic sequential order, making explicit tracking of every page unnecessarily costly. This inefficiency is further amplified by their wide range of access sizes, from 4 KB to 512 KB transfers. To avoid this overhead, FLAX manages buffers at 4 KB granularity and represents buffer validity as contiguous valid region using head and tail pointers. This design preserves fine-grained access while eliminating per-page metadata overhead, and it naturally supports lightweight FIFO-based page reuse for dynamic buffer management (§4.5).

To enable in-device dependency management, FLAX maintains a *dependency state table*, where each entry corresponds to an allocated buffer and contains the buffer's progress metadata. The *base* and *size* fields record the starting page offset and size of the buffer, respectively. The *head* and *tail* fields delimit the valid data region: *head* advances as new data is loaded, while *tail* advances as pages are evicted. Only pages within this region are accessible. The *latest* field records the most recently accessed page offset, and the *demand* field records the offset and size of a request that triggers demand loading. Finally, the *extents* field stores extent-based mapping information—logical start offset, physical start offset, and length—used for demand-loaded data. FLAX maintains eight extent descriptors by default. Together, *latest*, *demand*, and *extents* support adaptive data loading (§4.4).

Access control for computation. To mediate buffer accesses by computation, FLAX provides two helper functions, `csd_read_input()` and `csd_write_output()`. These functions serve three purposes. First, they verify that the requested location lies within the allocated buffer, satisfying the security requirements of the CSD standard. Second, they consult the *dependency state table* to enforce dependency correctness: they check whether the access falls within the valid region and, if not, wait for an in-flight prefetch to update *head* or set *demand* to trigger demand loading. Third, they perform address translation for dynamic buffer management. Because these checks are lightweight, replacing direct buffer accesses with these helper functions adds negligible overhead.

Memory overhead. Each entry in the dependency state table contains five 4 B fields (*base*, *size*, *head*, *tail*, and *latest*), one 8 B field (*demand*), and eight extent descriptors, each occupying 12 B, for a total of 124 B per entry. By default, FLAX maintains 128 entries, enough to support up to 16 concurrent computations with up to eight buffers each. This is sufficient in practice because in-device compute units typically provide up to 16 cores [41], while LSM-KVS typically maintain up to seven levels. Overall, FLAX requires only 16 KB of on-device memory for 4 KB page-granular dependency management and dynamic buffer management.

4.3 Asynchronous Command Execution

FLAX replaces host-managed dependency handling with *asynchronous command execution*. When the host issues input loading (`MEMCPY`) or execution (`EXEC`) commands, FLAX

records the command information internally, immediately returns completion to the host, and continues the operation in the background. This design choice is motivated by the structure of the CSD offloading procedure and the need to preserve compatibility. Under the host-managed programming model, input loading and execution commands return only completion status, while computation results are explicitly retrieved using operations such as `MEMREAD` or `SSDCPY`.

FLAX maintains correctness under asynchronous execution by ensuring that the host observes only valid output data. During output retrieval, FLAX consults the dependency state table and returns only pages within the valid region, while the host continues retrieving output until FLAX indicates that the operation has completed. This iterative retrieval is not specific to FLAX, but is already required by the NVMe SSD transfer size limit. As a result, computation can overlap not only with input loading but also with output retrieval, while remaining largely transparent to the host. In practice, supporting FLAX requires only a single conditional check for output completion and otherwise preserves the existing offloading procedure. Because computation operates on copies in on-device memory, asynchronous execution does not corrupt the original SSD data in the event of a crash.

Using Figure 4, we illustrate how FLAX uses the dependency state table to manage dependencies. In this example, pages 0–15 form the input buffer and pages 24–39 form the output buffer. Before each access, FLAX consults the dependency state table to determine whether the target pages are ready (❶). On the read path, after processing page 5, the compute core accesses page 6 through a helper function. Because page 6 lies within the valid region delimited by *tail* and *head*, the access is allowed (❷), and *latest* is updated to 6 (❸). On the write path, computation produces output at page 7 (❷), causing *head* to advance to 7 (❸). If the host issues a `MEMREAD` for 16 KB during execution, the memory controller checks the valid region of the output buffer. If all requested data is ready, it returns pages 10–13 (❷) and updates *latest* to 13 (❸); otherwise, it returns only the currently valid pages and updates *latest* accordingly.

Handling multiple consumers. FLAX restricts each memory buffer to a single consumer, since supporting multiple consumers would incur additional metadata overhead and synchronization costs. Scenarios with multiple consumers can instead be supported through chained computations, where the produced data is replicated into separate buffers so that each consumer receives an independent copy. Because asynchronous execution overlaps replication with computations, the additional overhead remains small.

4.4 Adaptive Data Loading

Loading the entire input into the memory buffer can exhaust SSD bandwidth and cause unnecessary data transfers. FLAX addresses this issue by employing *adaptive data loading*, which breaks input loading into smaller chunks and schedules them according to each computation's characteristics and progress. For stream access patterns, FLAX prefetches data based on the computation's real-time consumption rate, derived directly from the dependency state table without any complex profiling or sampling. To this end, FLAX tracks the most recently accessed page through the *latest* value and estimates the number of remaining unaccessed pages by comparing it with *head*. When this number falls below a predefined threshold, FLAX fetches the next chunk from the SSD.

To maximize SSD's internal parallelism while balancing bandwidth usage, FLAX sets the chunk size to $(N_c \times N_l \times S_p)$, where N_c and N_l are the number of NAND channels and LUNs, respectively, and S_p is the NAND physical page size. This allows all channels and LUNs to participate in a single data transfer. In our SSD setting, this corresponds to a 512 KB chunk size. To avoid both stalls and premature prefetching, FLAX sets the threshold equal to the chunk size.

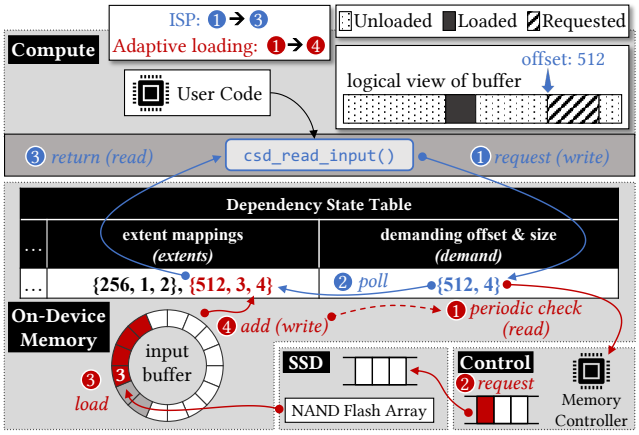


Figure 5: Interaction between ISP and adaptive data loading.

For random access patterns, FLAX enables demand loading to minimize unnecessary data transfer and better support latency-sensitive computations. Figure 5 illustrates how FLAX detects and handles demand loading at runtime. When a computation accesses a page outside the valid region, the helper function determines whether the computation exhibits a random access pattern. If no extent mapping is available, FLAX compares the requested page against *head*; if it is not contiguous with the current valid region, the access is identified as random. Otherwise, FLAX checks the requested page against the extent descriptors to determine whether it falls within a mapped extent. If the request is treated as random access, the offset and size of the requested data are recorded in the *demand* field (❶). The computation then waits until the corresponding extent mapping is added (❷).

The device memory controller periodically checks the *demand* field to determine whether a demand loading request has been issued (❶). If so, it switches from prefetching to demand loading. The controller then issues a request to the SSD's internal job queue (❷), and the SSD copies the requested data to the input buffer (❸). Once the copy is com-

pleted, the *demand* field is cleared and the corresponding extent mapping is added to record the logical start offset, the physical start offset, and the size of the copied data (4). The helper function then stops waiting, allowing the computation to access the data (3).

Deferred prefetching. In FLAX, only the first page is loaded before computation begins, and prefetching is deferred until stream access pattern is confirmed. This avoids initial demand loading overhead for random accesses, which are generally more latency-sensitive than stream access patterns that typically consume the entire dataset and benefit from prefetching. Enabling prefetching and then halting upon random access detection would leave in-flight prefetch I/Os occupying SSD bandwidth, thereby increasing the latency of demand loading requests. By contrast, deferring prefetching until stream access is confirmed has minimal impact on stream-based computations, as their longer computation time and prefetching outweigh the small delay introduced.

4.5 Dynamic Buffer Management

Dynamic buffer management allows computations to perceive the allocated, physically contiguous memory buffer as larger than its physical capacity. This overcomes the memory capacity wall and improves compute unit utilization by supporting large datasets and multiple concurrent computations. However, realizing this larger-than-physical buffer abstraction within the device introduces several challenges. First, buffer reuse and eviction must be managed entirely in software, as the storage device lacks specialized hardware such as an MMU. Second, unaligned access, in which a single data item spans multiple logical pages, must be handled seamlessly. Third, the metadata and translation mechanism used to map logical pages onto physical pages must remain lightweight to minimize memory and computation overhead.

We considered several buffer management policies that preserve a low-overhead design. LRU and Clock require tracking page access history, which adds metadata and complexity. More importantly, they do not preserve physical contiguity for logically contiguous pages, forcing extra handling for unaligned accesses on both the compute side and within FLAX. FIFO avoids these issues. It naturally supports unaligned accesses through sequential buffer reuse and requires only head and tail pointers, which FLAX already maintains for dependency management. As a result, FIFO adds no metadata overhead. It also removes the need for an explicit logical-to-physical mapping table for stream access patterns: the physical location of a logical page is computed by taking the logical page number modulo the buffer size (*size*) and adding the buffer base offset (*base*). For these reasons, FLAX adopts FIFO as its buffer management policy.

Handling eviction for input and output buffers. Since the input buffer holds copied, read-only data, evicting input pages requires no write-back. When adaptive data loading needs space, FLAX reclaims it by advancing *tail*. If extent mapping is in use, it also reclaims the corresponding extent and removes its mapping. New data is then loaded by advancing *head*. The output buffer, in contrast, holds generated data that cannot be discarded and re-fetched from the SSD. To preserve correct output reads, FLAX evicts only pages between *tail* and *latest*, which are guaranteed to have already been consumed. Because LSM-KVS operations do not revisit output after consumption, this policy safely reclaims space without affecting correctness.

Handling wrap-around for misaligned access. When *head* reaches the end of the buffer, FLAX wraps around and reuses pages in circular order. To support misaligned accesses across the boundary, it reserves a *ghost page* at the end of the buffer that mirrors the first page. On wrap-around, data loaded into the first page is copied into the ghost page, allowing computations to access boundary-spanning items without special handling. For output buffers, writes to the ghost page are copied back to the first page to preserve correctness. We use a single 4 KB ghost page because typical LSM-KVS items are smaller than 4 KB [6, 62]. The overhead is minimal because the ghost page is used only on wrap-around and requires just one extra memory copy.

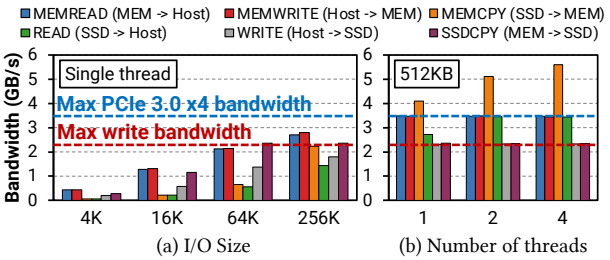
5 Implementation

We implement FLAX by extending NVMeVirt [27], a state-of-the-art SSD emulator, to model a general-purpose NVMe-compliant CSD. Although commercial CSDs [61], research prototypes [48, 63], and a CSD emulator [68] exist, we choose to extend NVMeVirt for three reasons. First, it presents itself as a native NVMe device, allowing FLAX to run unmodified applications under realistic system conditions. Second, it models modern SSD internals in detail, including NAND latencies, die-, channel-, and PCIe-level contention, and write buffering, all of which are critical for accurately evaluating CSD benefits. Third, it exposes architectural parameters such as memory capacity, core frequency, and internal data paths, enabling broader exploration of the CSD design space.

5.1 Emulating General-Purpose CSD

Configuration details. We use NVMeVirt's Samsung 970 PRO model for the SSD, configured with a capacity of 350 GB. We reserve 4 GB of system memory as on-device memory, matching the compute memory capacity of Samsung SmartSSD [61]. For compute, we provision 16 host CPU cores to emulate the capability of recent ARM-based platforms, e.g., the ARM Cortex-A78 in BlueField-3 [41]. Following prior emulation methodology [43, 66, 67], we pin these cores and throttle them to 1 GHz to approximate the wimpy cores commonly used in prototype CSDs [28, 63] and DPUs [41].

Validation. As of this writing, no standard-compliant CSD is commercially available. We therefore validate our emulator by comparing the performance trends of each component against those of corresponding physical devices. First, we measure data-transfer commands and compare them with



(a) I/O Size (b) Number of threads

Figure 6: Performance of data-transfer commands.

Samsung 970 PRO and Samsung SmartSSD [61], a prototype CSD that integrates an SSD and an FPGA via a PCIe bridge. As shown in Figure 6-(a), `MEMREAD` and `MEMWRITE` are much faster than `READ` and `WRITE` because they access DRAM rather than NAND flash. This trend is consistent with SmartSSD, where reading 4 KB from FPGA memory (167 MB/s) outperformed SSD reads (52 MB/s), highlighting the effect of backend media on I/O performance.

Figure 6-(b) shows that `MEMREAD`, `MEMWRITE`, and `READ` saturate at 3.5 GB/s, matching the PCIe 3.0 ×4 limit. In contrast, `MEMCPY` reaches 5.6 GB/s, demonstrating the benefit of in-storage data movement without PCIe involvement. For large transfers (e.g., 64 MB), `MEMCPY` reaches up to 6.4 GB/s, matching the SSD's maximum internal bandwidth from eight NAND channels at 800 MB/s each [56]. Although SmartSSD's PCIe bridge prevents exact replication of this behavior, our results are consistent with prior CSD studies that exploit higher internal SSD bandwidth [11, 19]. This advantage is not unique to PCIe 3.0, as internal NAND bandwidth continues to outpace host-interface bandwidth even on recent PCIe generations [55, 57].

For compute, our 1 GHz core scores 2.18 on SPEC CPU2017 [1], compared to 3.15 for BlueField-3's 2 GHz Cortex-A78, indicating that it reasonably emulates a low-performance ARM core. Overall, these results confirm that our emulator captures the key performance characteristics expected of a standard-compliant CSD.

Toward real hardware. Deploying FLAX on real hardware requires firmware and runtime support for the mechanisms described in §4, but does not require changes to the SSD component's FTL or NAND layout. FLAX can therefore be implemented on future standard-compliant devices as a firmware-level extension.

5.2 Application Changes for CSD Offloading

User-level library. We provide a user-level library that serves as a wrapper to issue CSD standard commands via the `ioctl` system call. For `MEMCPY` and `SSDCPY` commands, the library uses the `FIEMAP` ioctl to retrieve file extent metadata and construct the source range entry. We also provide a function that invokes the `fallocate` system call and clears the *unwritten* flag, as done in prior work [48, 63], to ensure that data written to the SSD by `SSDCPY` is readable.

`COMPOUND` command. The current standard requires at least seven commands for a single offload, which is a substantial overhead for latency-sensitive operations. To reduce this cost, we introduce a new NVMe CSD command, `COMPOUND`, that merges two `ALLOC` commands for input and output buffer allocation, `MEMCPY` for input loading, and `EXEC` for computation into a single command. This reduces the command sequence to three steps: `COMPOUND`, output retrieval, and buffer release. Although `COMPOUND` is not required to use FLAX, it can further amplify its performance benefits.

Offloading compaction. Offloading compaction can speed up background maintenance, reduce write stalls, and prevent host cache pollution [58, 69]. We target higher-level compactions (level L_1 and above), where SSTables are more likely to contain cold data and thus benefit most from CSD offloading. In contrast, L_0–L_1 compaction remains on the host to fully exploit cached files. To further improve offloading efficiency, we decompose compaction into a merge-sort task followed by a checksum computation task. We then split the checksum into subtasks and pipeline them by chaining each subtask's output to the next subtask's input. This pipelined execution is enabled by FLAX, which supports multiple concurrent computations via dynamic buffer management and overlaps their execution. Overall, compaction offloading requires 272 lines of code (LoC), and integrating FLAX adds only 7 LoC for completion validation (§4.3).

Offloading reads. We offload reads similarly to compaction: a read is delegated to the CSD only when the requested data is not cached and resides beyond levels L_0 or L_1. RocksDB first searches the memtables, then files in L_0 and L_1, which are typically cached in host memory. If the key is still not found, it proceeds to higher levels. To offload reads efficiently, we split the search into two phases. In the first phase, RocksDB identifies the candidate file at each level and checks whether its index block is cached. If the index block is cached, RocksDB performs a binary search on it and records the location of the corresponding data block for offloading. Otherwise, it records the location of the index block itself. This process continues through the last level, collecting the metadata needed for offloading (i.e., index- or data-block locations). In the second phase, RocksDB offloads the read to the CSD using the collected metadata. If the metadata points to an index block, the CSD reads that block, performs a binary search to locate the target data block, reads the data block, and searches within it. If the key is not found, the CSD continues to the next level using the corresponding metadata. Overall, implementing read offloading requires 205 LoC.

Crash consistency. Offloaded compactions preserve RocksDB's MANIFEST-based consistency. Output SSTables produced by the CSD become visible only after RocksDB durably persists them and commits the corresponding MANIFEST update; input SSTables are deleted only after this commit. Thus, a crash during offloaded compaction may

leave obsolete output files, which will be discarded during recovery.

6 Evaluation

This section evaluates RocksDB under three configurations: host-only execution, a host-managed CSD baseline, and FLAX. We answer three questions:

Q1. *How much does* FLAX *improve compaction and how does this affect application throughput and tail latency?*

Q2. *Can* FLAX *overcome the limits of the host-managed model and improve latency-sensitive foreground reads?*

Q3. *Can* FLAX *efficiently sustain concurrent offloading of heterogeneous operations under mixed workloads?*

6.1 Experimental Setup

Evaluation environment. We use a 256-core machine with two AMD EPYC 9754 processors and 768 GB of DRAM. To minimize interference, we place RocksDB and FLAX on separate NUMA nodes. Unless otherwise noted, we cap RocksDB memory at 4 GB using Linux `cgroups`, reflecting a realistic memory-to-dataset ratio [7, 59]. All experiments use 12 client threads.

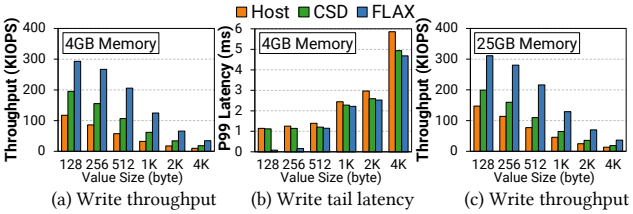
RocksDB configuration. We use RocksDB v9.3.1. Unless otherwise noted, we keep most parameters at their default values. We set *subcompactions* to 2 to accelerate L_0–L_1 compaction and disable both compression and *level_compaction_dynamic_level_bytes*. We evaluate unmodified RocksDB rather than its KV-separated variant [36] to isolate the impact of offloading from that of layout-level optimizations. FLAX is nevertheless compatible with KV-separated stores, which still perform compaction and multi-level reads while introducing additional offloadable operations such as value-log garbage collection [12, 33].

On-device memory allocation. For offloading, the host-managed CSD baseline requires input and output buffers sized to the dataset: the input buffer must hold the full dataset, while the output buffer must be overprovisioned to accommodate unpredictable output sizes. In contrast, FLAX uses fixed-size buffers—128 MB input and output buffers for compaction, and 16 MB buffers for reads—eliminating the need to provision memory proportional to dataset size. Unless otherwise noted, FLAX uses `COMPOUND` commands for offloading.

6.2 Write Performance with FLAX

To evaluate the impact of *compaction offloading*, we use a write-only workload with 16 B randomly generated keys while varying value sizes from 128 B to 4 KB. We insert 800 M KV pairs for the 128 B value size and proportionally reduce the number of KV pairs as the value size increases to maintain a comparable dataset size.

As shown in Figure 7-(a), `FLAX` outperforms both host-only execution (`Host`) and host-managed CSD (`CSD`) across all value sizes, primarily because it offloads compaction and overlaps computation with data transfer. For example, with



(a) Write throughput (b) Write tail latency (c) Write throughput

Figure 7: RocksDB under a write-only workload.

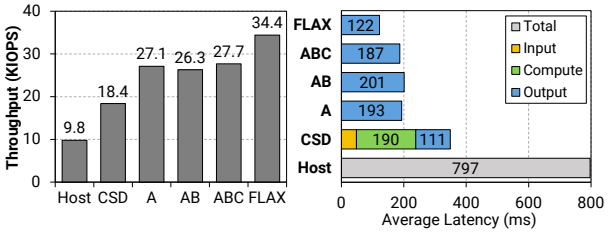
4 KB values, `Host` reads approximately 700 GB and writes 900 GB of data through the host I/O stack, whereas `FLAX` reduces host-side traffic to only 20 GB of reads and 300 GB of writes by handling the rest internally. Larger values fill levels more quickly and thus trigger compaction more frequently, allowing `FLAX` to achieve greater gains over `Host`. Although the gains are smaller than those of `FLAX`, `CSD` also improves performance by exploiting in-storage data access and bypassing frequent trips through the host I/O stack.

In Figure 7-(b), however, `CSD` fails to improve write tail latency at small value sizes. This is because flush operations occur much more frequently than compaction in such cases, and the host-managed model's bulk output transfer consumes excessive SSD write bandwidth, stalling flush operations. In contrast, `FLAX` asynchronously transfers output as it becomes ready, reducing bandwidth contention and alleviating write tail latency. This effect is most pronounced at the 128 B value size, where `CSD` lowers tail latency by only 2% compared to `Host`, while `FLAX` reduces it by 93%.

Under larger memory. Although memory capacity is typically limited relative to large-capacity storage, we also evaluate performance with 25 GB of memory allocated via `cgroups` to examine compaction performance under more generous memory provisioning (approximately 1:4 memory-to-dataset ratio). Figure 7-(c) presents the results. Although `Host` improves by 35% on average compared to 4 GB memory, `FLAX` still achieves 2.6× higher throughput by avoiding frequent traversal of the host I/O stack.

Ablation study. To gain deeper insight into the performance improvements achieved by FLAX, we analyze the contribution of each technique by incrementally applying them to the host-managed CSD baseline. **A** enables only asynchronous command execution; **AB** adds dynamic buffer management; **ABC** decomposes compaction into merge-sort and checksum tasks and pipelines them; and **FLAX** pipelines the merge-sort with two interleaved checksum tasks. Adaptive data loading is enabled in all configurations, as it has a negligible impact on the write-only workload.

Figure 8 presents the throughput and compaction latency breakdown across different configurations for the 4 KB value size. With asynchronous command execution enabled, input loading and computation commands return immediately, so the output handling latency reflects the entire offloading process. Host-only execution (**Host**) overlaps data transfers and computation but still pays the cost of the host I/O stack, includ-



(a) Write throughput (b) Compaction breakdown

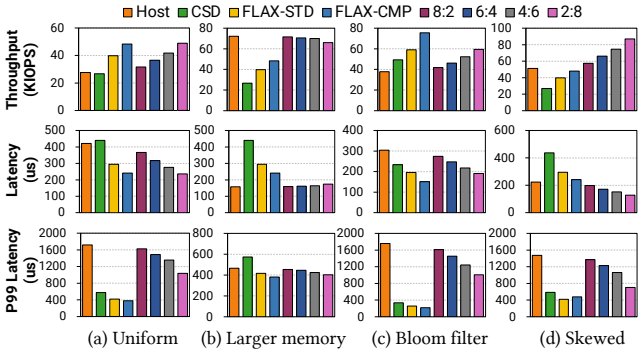
Figure 8: Impact of each FLAX feature.

ing page-cache accesses and system-call overhead [65, 70]. The host-managed CSD (**CSD**) avoids this cost through offloading, yet its copy-then-execute model serializes the data transfers and computation that **Host** pipelines. **A** restores this overlap inside the device, reducing compaction latency by 45% and improving throughput by 1.47× over the host-managed CSD. The latency reduction matches the compute-unit idle fraction observed in host-managed CSD, confirming that **A** successfully hides data transfer latencies through overlapping. This also indicates that dependency tracking and access control via helper functions incur negligible overhead. The `COMPOUND` command provides no additional benefit here, as compaction is a long-running operation where system call overhead is negligible.

AB increases compaction latency by only 4%, indicating that dynamic buffer management incurs only modest overhead even when handled entirely in software, owing to the lightweight FIFO policy and simple address translation. **ABC** increases throughput by 5%. Notably, decomposing compaction into pipelined tasks is only feasible with dynamic buffer management in place, which enables multiple concurrent tasks to execute within limited on-device memory. **FLAX** interleaves checksum computation across two cores, reducing compaction latency by 35% and achieving 1.24× higher throughput. This demonstrates that FLAX can efficiently handle multiple consumer scenarios through pipelined execution while hiding the replication overhead via overlapping. Interleaving checksum computation across additional cores yields marginal gains because merge-sort becomes the pipeline bottleneck. In summary, all techniques remain lightweight and collectively essential to fully exploit the potential of CSD.

6.3 Read Performance with FLAX

To evaluate the impact of *read offloading*, we use a read-only workload under various conditions. The database is preloaded with 800 M randomly generated KV pairs with 16 B keys and 128 B values. We use 128 B values because they are representative of real-world scenarios [6]. After loading, we clear the page cache and issue reads for 300 seconds over approximately 108 GB of key-value data. We evaluate eight read configurations: four dedicated and four hybrid. `Host` and `CSD` denote host-only execution and the host-managed CSD, respectively. `FLAX-STD` denotes FLAX using standard commands, and `FLAX-CMP` denotes FLAX using the `COMPOUND` command. In the hybrid configurations, requests are split between the host and FLAX using the `COMPOUND` command; e.g., `2:8` handles 20% of requests on the host and offloads 80% to FLAX.



(a) Uniform (b) Larger memory (c) Bloom filter (d) Skewed

Figure 9: Read performance under various configurations.

Uniform random workload. Figure 9-(a) presents the throughput, average latency, and tail latency of read operations under a uniform distribution. By default, RocksDB pins the index block of every SSTable, so only data blocks require storage access in this experiment. `CSD` increases average read latency by 4% compared to `Host`, due to the synchronous command sequence imposed by the host-managed programming model. `FLAX-STD`, on the other hand, overlaps data transfers with computation, reducing read latency by 33% compared to `CSD`. `FLAX-CMP` further reduces read latency by 18% compared to `FLAX-STD` by removing the system call overhead through the `COMPOUND` command.

Tail latency. The high read tail latency of `Host` stems from frequent accesses to storage through the host I/O stack, and is further exacerbated by contention caused by frequent eviction and reloading within the RocksDB block cache. Over the course of the experiment, `Host` reads 118 GB of data through the I/O stack, whereas the offloading-based configurations read only 4 GB. Therefore, `CSD`, `FLAX-STD`, and `FLAX-CMP` all reduce read tail latency, with `FLAX-CMP` achieving the largest reduction of 78%.

Hybrid configurations. Among the hybrid configurations, throughput improves as the offloading ratio increases, with `2:8` outperforming `8:2` by 1.54× in throughput. The improvement stems from two factors. First, the host execution path in hybrid configurations exhibits lower latency than `Host` because contention on the block cache is reduced. Second, although the host execution path can populate the block cache and allow read offloading to skip already cached blocks, its benefit from better host-memory utilization is partly offset by host I/O stack overhead. As a result, `2:8`, which has the lowest host execution ratio, achieves the throughput closest to `FLAX-CMP` among the hybrid configurations.

Under larger memory. We repeat the same experiment with 8, 16, and 25 GB of memory, and Figure 9-(b) presents the

Table 2: Read performance with demand loading.

	Host	CSD	FLAX
Throughput (IOPS)	3388.73	34.82	3548.15
Latency (ms)	3.52	343	3.36
99th Latency (ms)	6.41	461.78	4.51

25 GB results. With sufficient memory, `Host` reduces storage access via caching and block cache contention is alleviated by less frequent eviction, outperforming all offloading-based configurations. The intermediate memory sizes reveal where this transition occurs: at 8 GB, `Host` outperforms `FLAX-CMP` by 5% but still trails the hybrid configurations; at 16 GB, `Host` leads all offloading-based configurations. These results are also influenced by the characteristics of hardware and the CSD offloading procedure. Unlike prior work [43, 73] that targets ultra-low-latency devices such as Optane SSDs [24], FLAX operates on NAND flash SSDs with relatively higher read latency, requiring multiple storage accesses to fully realize the benefits of in-storage processing. Additionally, the cost of allocating and releasing memory buffers and retrieving output through explicit commands adds a constant overhead that can be critical to latency-sensitive operations. Overall, this experiment shows that read offloading can be less favorable under generous memory provisioning. Nevertheless, FLAX still outperforms the host-managed CSD by 81% and remains beneficial under low memory-to-dataset ratios, which are more common in production deployments.

With bloom filters. Figure 9-(c) presents read performance under uniform random workload with bloom filters [4] enabled at the commonly adopted setting of 10 bits per key. By allowing files to be skipped, bloom filters reduce `Host` read latency by 27%, while `FLAX-STD` and `FLAX-CMP` benefit more, with reductions of 33% and 37%, respectively. This gap stems from the memory pressure caused by large filter blocks, a behavior also observed in prior work [40, 72]. With small values, each SSTable contains many KV pairs, causing the filter block to grow to approximately 400 KB. Offloading-based configurations alleviate this pressure by offloading reads, allowing them to fully exploit bloom filters. This is further confirmed by the hybrid configurations, which fall short of `FLAX-CMP` because their host-side execution path reintroduces memory pressure. In terms of tail latency, bloom filters provide no improvement for `Host`, as they are more effective at reducing average latency [23]; in the worst case, every level must still be searched while contention remains.

The benefit of the `COMPOUND` command becomes more pronounced with bloom filters, reducing read latency by 23% compared to `FLAX-STD`. This is because skipping files lowers contention at the NAND flash, which reduces NAND read latency and increases the relative fraction of system call overhead. Consequently, `FLAX-CMP` achieves larger gains than in the uniform workload without bloom filters, reducing latency by 50% and improving throughput by 2× compared to `Host`.

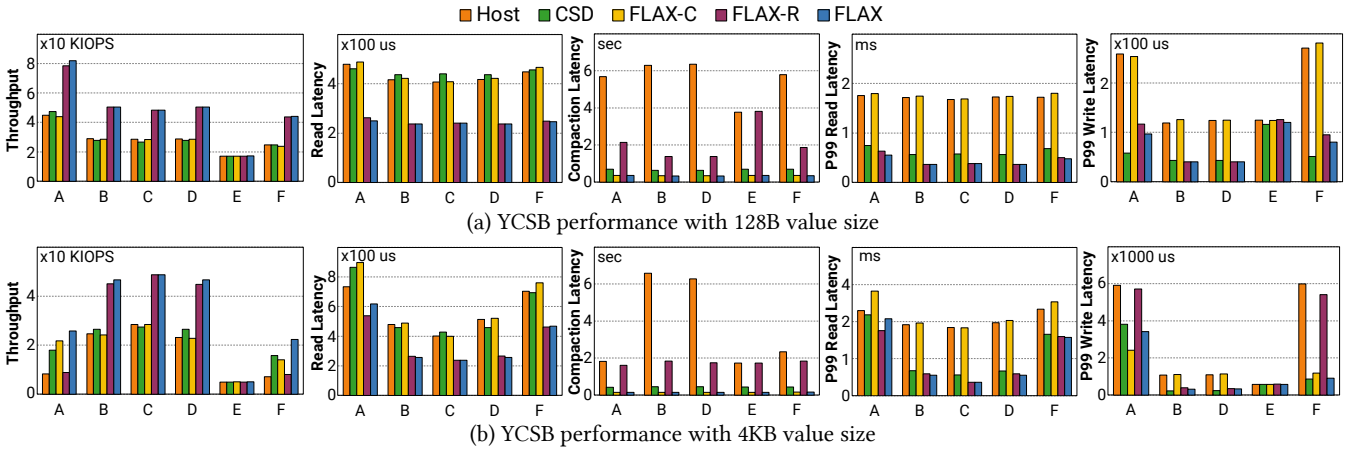
Skewed workload. Next, we evaluate reads under a skewed query distribution following the YCSB standard Zipfian distribution with $\theta = 0.99$. As shown in Figure 9-(d), the performance gain of `FLAX-CMP` over `CSD` and `FLAX-STD` remains similar to the uniform case, as the distribution does not affect the offloading procedure. Due to the skewed access pattern, `Host` outperforms `FLAX-CMP` by 7% in throughput. However, `Host` still exhibits high tail latency because even under extreme skew, the block cache cannot hold all data blocks associated with hot keys. As a result, storage accesses still occur, and block cache contention persists.

Hybrid configurations. With the skewed access pattern, hybrid configurations that can leverage both caching and offloading achieve up to 1.7× higher throughput over `Host` and 1.81× over `FLAX-CMP`. Under Zipfian skew, the block cache captures much of the hot working set, and offloading occurs only when the data block is not cached. The offloading ratio therefore effectively governs only cache-miss reads targeting cold data. As a result, `2:8`, with the highest offloading ratio, achieves the highest throughput and lowest latency by offloading the most cold reads to FLAX while retaining hot data in the block cache. This is also reflected in tail latency, where `2:8` achieves the lowest among hybrid configurations by reducing eviction and insertion in the block cache.

Index block caching. To better control memory usage, many deployed systems [37, 59] and prior work [71] cache index blocks rather than pinning them, limit the number of open files, and support direct reads. This leaves many index blocks uncached, allowing us to evaluate FLAX when demand loading becomes critical. Table 2 presents the overall results. `CSD` increases latency by 97× over `Host`, as the host-managed model requires entire SSTables to be loaded before computation can begin, even though the computation itself is lightweight. A stronger host-managed alternative is to split read offloading into separate index-block and data-block searches, avoiding bulk loading. This improves performance, but remains slower than FLAX because each read still incurs multiple host–device round trips. In contrast, FLAX reduces latency by 5% over `Host`, demonstrating that adaptive data loading can detect random accesses and perform demand loading without host coordination. The relatively modest gain is because index blocks are large; the benefits of bypassing the host I/O stack grow with smaller sizes [43, 73]. Nonetheless, the 30% reduction in tail latency demonstrates the benefit of in-storage processing even with large value sizes.

6.4 YCSB Performance with FLAX

We evaluate FLAX using the YCSB benchmark [8], which emulates real-world production workloads with various read–write ratios. We introduce two additional configurations: `FLAX-C`, which offloads only compaction, and `FLAX-R`, which offloads only reads. `CSD` and `FLAX` offload both. We use RocksDB's default configuration in which every SSTable's index block is pinned, as adopting index block caching config-



(a) YCSB performance with 128B value size

(b) YCSB performance with 4KB value size

Figure 10: YCSB performance across different value sizes.

uration triggers an out-of-memory error when both operations are offloaded to the host-managed CSD. We use value sizes of 128 B and 4 KB. For the 4 KB configuration, we load 40 M KV pairs; workloads A, B, C, D, and F each issue 10 M operations, and E issues 3 M. For 128 B values, we load 800 M KV pairs; workloads A, B, C, D, and F each issue 50 M operations, and E issues 30 M to trigger sufficient compactions. Overall, we use a uniform distribution to mitigate hot-key skew and more clearly isolate the impact of offloading.

Figure 10 presents the throughput, average read latency, average compaction latency, and read and write tail latencies across the YCSB workloads. Compared to `Host` and `CSD`, `FLAX` improves throughput by an average of 1.63× and 1.66× for 128 B values, and 2.16× and 1.53× for 4 KB values, respectively. Workload E shows similar performance across configurations because scans are not offloaded and its low write ratio rarely triggers compaction. As discussed in §3, we do not offload scans because RocksDB implements range scans as repeated `next()` calls; offloading each call would introduce a host–device round trip with little reduction in data movement.

Write-intensive workloads. The gains are more pronounced in write-intensive workloads (A and F) with larger values, where frequent compactions have a greater impact on overall performance. Under these workloads, `FLAX` achieves 3.15× higher throughput than `Host`. This also explains why `CSD` shows only marginal improvement for 128 B values but larger gains for 4 KB. With 128 B, throughput is largely dominated by read performance, and comparable or worse read latency under the host-managed model outweighs the benefits of compaction offloading. With 4 KB, however, compaction becomes the dominant bottleneck, allowing `CSD` to improve throughput by reducing compaction latency by 84%, despite its higher read latency. In contrast, `FLAX` reduces both read and compaction latency by 46% and 94%, respectively, compared to `Host`, improving throughput by 1.8× even with 128 B values. `FLAX` also reduces write tail latency by 68% on average across both value sizes. For read tail latency at 128 B, FLAX achieves a 70% reduction over `Host`. At 4 KB, even with more frequent compactions, FLAX still achieves the lowest read tail latency among `Host` and `CSD`. This confirms that adaptive data loading effectively mitigates resource contention when both compaction and reads are concurrently offloaded.

Read-intensive workloads. In 128 B read-intensive workloads (B, C, and D), `CSD` reduces throughput by 5% on average relative to `Host`, consistent with our read-only results. `FLAX`, however, improves throughput by 1.8× over `Host` and 1.77× over `CSD`, while reducing read tail latency by 76% and 27%, respectively, across value sizes.

Impact of offloading both compaction and reads. In contrast to prior work that offloads only background operations [12, 54, 69], FLAX enables LSM-KVS to offload both compaction and reads. Comparing `FLAX` with `FLAX-C` and `FLAX-R` shows why both are needed. With 128 B values, `FLAX-R` performs comparably to `FLAX`, whereas `FLAX-C` delivers lower average throughput than `Host`. Offloading only compaction increases write throughput, which intensifies contention for host resources and interferes with host-side reads. For example, in YCSB-A, offloading compaction alone reduces compaction latency by 94%, but the resulting increase in write throughput raises host read latency by 2%.

The situation differs at 4 KB. Since compaction is the dominant bottleneck, `FLAX-C` improves throughput for write-intensive workloads but performs similarly to `Host` under read-intensive ones. Conversely, `FLAX-R` benefits read-intensive workloads but offers little gain for write-intensive ones. Only `FLAX` consistently surpasses `Host` across workloads, regardless of value size and read-write ratio, while simultaneously reducing both read and write tail latency. These results confirm that a general-purpose design that supports concurrent offloading of both compaction and read operations is essential for delivering robust performance gains.

7 Related Work

Near-Data Processing. Background LSM-KVS operations have been widely offloaded to FPGAs [54, 69], GPUs [53, 74],

and DPUs [9, 10]. While these reduce host CPU usage, they operate outside storage, requiring PCIe transfers that offer much lower bandwidth than internal NAND channels [11, 19] and can bottleneck under heavy data movement [25]. They also rely on host coordination and use host memory as an intermediate buffer. BPF-oF [64] extends XRP [73] to offload LSM-KVS reads at the NVMe layer. Although the approach is similar, it still depends on PCIe transfers and host memory. Additionally, eBPF's strict constraints make it difficult to also offload compaction. In contrast, FLAX supports both background and foreground operations inside storage, avoiding PCIe transfers and host coordination while achieving lower latency than NVMe-layer approaches.

In-Storage Processing. PStore [52] focuses on task-level asynchronous execution, where the host and device independently perform compaction under scheduling policies, whereas FLAX enables page-granular in-device asynchrony within and across offloaded operations, overlapping computation with data transfers. SODE [43] supports B^+-tree lookup offloading through in-device demand loading. FLAX generalizes this idea to concurrent, heterogeneous LSM-tree operations, including both foreground reads and background compaction. AegonKV [12] targets value-log GC offloading in KV-separated systems, improving performance for write-intensive workloads through a specialized SmartSSD design; FLAX is complementary and could support such data-driven GC on its general-purpose runtime. KV-CSD [44] and PinK [23] redesign the KV store inside the device behind a KV interface, whereas FLAX preserves the conventional LSM-KVS and its block-oriented interface while selectively offloading operations. Overall, unlike prior designs bound to specific operations or access patterns, FLAX supports concurrent heterogeneous operations and improves performance across a wide range of workloads.

8 Conclusion

Host–storage data movement increasingly constrains LSM-tree-based key–value stores, yet existing CSD designs fall short of delivering their full benefits because their host-managed, copy-then-execute model serializes execution and limits concurrency. To address these challenges, we propose FLAX, a novel CSD architecture built around device-resident dependency management. By tracking fine-grained dependencies within the device, FLAX enables asynchronous, overlapped execution while enforcing correct ordering without host-side coordination. FLAX further maximizes concurrency and improves device utilization by combining adaptive data loading with dynamic buffer management. We integrated FLAX into RocksDB and offloaded both compaction and reads. Our evaluation demonstrates that in-device dependency management and support for concurrent execution through a general-purpose design are essential to realizing the full benefits of CSD offloading for LSM-tree operations.

Acknowledgments

We would like to thank our shepherd, Zhichao Cao, and the anonymous reviewers for their invaluable feedback. We also thank Ilkueon Kang for his help during the early stages of this work. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No.RS-2021-II211363, No.RS-2024-00460762, and No.RS-2024-00349594) and the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No.2022R1A2C2008427). This work was also supported in part by a research grant from Samsung Electronics. Jin-Soo Kim is the corresponding author (jinsoo.kim@snu.ac.kr).

References

[1] Spec cpu2017. http://www.spec.org/cpu2017.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[4] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. Polardb meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*, pages 29–41, 2020.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[7] Ceph. Bluestore configuration reference. https://docs.ceph.com/en/reef/rados/configuration/bluestore-config-ref/#confval-osd_memory_target.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud

serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[9] Chen Ding, Kai Lu, Quanyi Zhang, Zekun Ye, Ting Yao, Daohui Wang, Huatao Wu, and Jiguang Wan. Dflush: Dpu-offloaded flush for disaggregated lsm-based key-value stores. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.

[10] Chen Ding, Jian Zhou, Kai Lu, Sicen Li, Yiqin Xiong, Jiguang Wan, and Ling Zhan. D2comp: Efficient offload of lsm-tree compaction with data processing units on disaggregated storage. *ACM Transactions on Architecture and Code Optimization*, 21(3):1–22, 2024.

[11] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230, 2013.

[12] Zhuohui Duan, Hao Feng, Haikun Liu, Xiaofei Liao, Hai Jin, and Bangyu Li. Aegonkv: A high bandwidth, low tail latency, and low storage cost kv-separatedlsm store with smartssd-based gc offloading. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 321–335, 2025.

[13] NVM Express. NVM Express. https://nvmexpress.org/.

[14] NVM Express. Nvme computational programs command set. https://nvmexpress.org/specification/computational-programs-command-set.

[15] NVM Express. Nvme express state of the union. https://nvmexpress.org/wp-content/uploads/NVM-Express-State-of-the-Union.pdf.

[16] NVM Express. Nvme subsystem local memory command set. https://nvmexpress.org/specification/subsystem-local-memory-command-set.

[17] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 224–235, 2019.

[18] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, et al. Genstore: A high-performance and energy-efficient in-storage computing system for genome sequence analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 635–654, 2022.

[19] Nika Mansouri Ghiasi, Mohammad Sadrosadati, Harun Mustafa, Arvid Gollwitzer, Can Firtina, Julien Eudine, Haiyu Mao, Joël Lindegger, Meryem Banu Cavlak, Mohammed Alser, et al. Megis: High-performance, energy-efficient, and low-cost metagenomic analysis with in-storage processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 660–677. IEEE, 2024.

[20] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: a framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 153–165, 2016.

[21] Ali HeydariGorji, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H Chou. Leveraging computational storage for power-efficient distributed data analytics. *ACM Transactions on Embedded Computing Systems*, 21(6):1–36, 2022.

[22] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[23] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. Pink: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187, 2020.

[24] Intel. Intel® optane™ ssd dc p5800x series. https://www.intel.com/content/www/us/en/products/sku/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[25] Hongsun Jang, Jaeyong Song, Jaewon Jung, Jaeyoung Park, Youngsok Kim, and Jinho Lee. Smart-infinity: Fast large language model training using near-storage processing on a real system. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 345–360. IEEE, 2024.

[26] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[27] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. Nvmevirt: A versatile software-defined virtual nvme device. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 379–394, 2023.

[28] Gunjae Koo, Kiran Kumar Matam, Te I, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–231, 2017.

[29] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/softwareco-programmable framework for computational ssds to accelerate deep learning service on large-scale graphs. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 147–164, 2022.

[30] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[31] Kitaek Lee, Insoon Jo, Jaechan Ahn, Hyuk Lee, Hwang Lee, Woong Sul, and Hyungsoo Jung. Deploying computational storage for htap dbmss takes more than just computation offloading. *Proceedings of the VLDB Endowment*, 16(6):1480–1493, 2023.

[32] Junkai Liang and Yunpeng Chai. Cruisedb: An lsm-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1032–1043. IEEE, 2021.

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in {SSD-conscious} storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, 2016.

[34] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. Deepstore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 224–238, 2019.

[35] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[36] Meta. Blobdb. https://github.com/facebook/rocksdb/wiki/BlobDB.

[37] Meta. Myrocks. http://myrocks.io.

[38] Meta. Rocksdb. https://rocksdb.org.

[39] Steve Morgan. The 2020 data attack surface report. https://cybersecurityventures.com/wp-content/uploads/2020/12/ArcserveDataReport2020.pdf, 2021.

[40] Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, and Manos Athanassoulis. Lsm-trees under (memory) pressure. In *International Workshop on Accelerating Analytics and Data Management Systems*, 2022.

[41] NVIDIA. Nvidia bluefield-3 spec. https://docs.nvidia.com/networking/display/bf3dpu/specifications#src-2449222537_Specifications-B3240DPUsSpecifications.

[42] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[43] Chanyoung Park, Minu Chung, and Hyungon Moon. Selective on-device execution of data-dependent read i/os. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 373–390, 2025.

[44] Inhyuk Park, Qing Zheng, Dominic Manno, Soonyeal Yang, Jason Lee, David Bonnie, Bradley Settlemyer, Youngjae Kim, Woosuk Chung, and Gary Grider. Kv-csd: A hardware-accelerated key-value store for data-intensive applications. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 132–144, 2023.

[45] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. Scalecache: A scalable page cache for multiple solid-state drives. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 641–656, 2024.

[46] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. Closing the b+-tree vs.lsm-tree write amplification gap on modern storage hardware with built-in transparent compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 69–82, 2022.

[47] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 325–339, 2023.

[48] Zhenyuan Ruan, Tong He, and Jason Cong. Insider: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019.

[49] ScaleFlux. CSD 3000. https://scaleflux.com/products/csd-3000/.

[50] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmablessd. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, 2014.

[51] SNIA. Towards copy-offload in linux nvme. https://www.snia.org/educational-library/towards-copy-offload-linux-nvme-2021.

[52] Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. Asynchronous compaction acceleration scheme for near-data processing-enabled lsm-tree-based kv stores. *ACM Transactions on Embedded Computing Systems*, 23(6):1–33, 2024.

[53] Hui Sun, Jinfeng Xu, Xiangxiang Jiang, Guanzhong Chen, Yinliang Yue, and Xiao Qin. glsm: Using gpgpu to accelerate compactions in lsm-tree-based key-value stores. *ACM Transactions on Storage*, 20(1):1–41, 2024.

[54] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. Fpga-based compaction engine for accelerating lsm-tree key-value stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1261–1272. IEEE, 2020.

[55] Tech Power Up. Samsung 9100 Pro 1 TB. https://www.techpowerup.com/ssd-specs/samsung-9100-pro-1-tb.d2318.

[56] Tech Power Up. Samsung 970 Pro 512 GB. https://www.techpowerup.com/ssd-specs/samsung-970-pro-512-gb.d54.

[57] Tech Power Up. Samsung 990 Pro 1 TB. https://www.techpowerup.com/ssd-specs/samsung-990-pro-1-tb.d861.

[58] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. Near-data processing in database systems on native computational storage under htap workloads. *Proceedings of the VLDB Endowment*, 15(10):1991–2004, 2022.

[59] Hao Wang, Jiaxin Ou, Ming Zhao, Sheng Qiu, Yizheng Jiao, Yi Wang, Qizhong Mao, Zhengyu Yang, Yang Liu, Jianshun Zhang, et al. Lavastore: Bytedance's purpose-built, high-performance, cost-effective local storage engine for cloud services. *Proceedings of the VLDB Endowment*, 17(12):3799–3812, 2024.

[60] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 717–729, 2021.

[61] Xilinx. Smartssd computational storage drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[62] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[63] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. λ-io: A unified io stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, 2023.

[64] Ioannis Zarkadas, Tal Zussman, Jeremy Carin, Sheng Jiang, Yuhong Zhong, Jonas Pfefferle, Hubertus Franke, Junfeng Yang, Kostis Kaffes, Ryan Stutsman, et al. Bpf-of: Storage function pushdown over the network. *arXiv preprint arXiv:2312.06808*, 2023.

[65] Yekang Zhan, Tianze Wang, Zheng Peng, Haichuan Hu, Jiahao Wu, Xiangrui Yang, Qiang Cao, Hong Jiang, and Jie Yao. Rearchitecting buffered i/o in the era of high-bandwidth ssds. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*, pages 579–595, 2026.

[66] Jian Zhang, Yujie Ren, and Sudarsun Kannan. Fusionfs: Fusing i/o operations using ciscops in firmware file systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 297–312, 2022.

[67] Jian Zhang, Yujie Ren, Marie Nguyen, Changwoo Min, and Sudarsun Kannan. Omnicache: Collaborative

caching for near-storage accelerators. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 35–50, 2024.

[68] Qiuyang Zhang, Jiapin Wang, You Zhou, Peng Xu, Kai Lu, Jiguang Wan, Fei Wu, and Tao Lu. Cemu: Enabling full-system emulation of computational storage beyond hardware limits. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 323–341, 2026.

[69] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. Fpga-accelerated compactions for lsm-based key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, 2020.

[70] Yinan Zhang, Shun Yang, Huiqi Hu, Chengcheng Yang, Peng Cai, and Xuan Zhou. Succinctkv: a cpu-efficient lsm-tree based kv store with scan-based compaction. *ACM Transactions on Architecture and Code Optimization*, 21(4):1–26, 2024.

[71] Fuheng Zhao, Zach Miller, Leron Reznikov, Divyakant Agrawal, and Amr El Abbadi. Autumn: A scalable read optimized lsm-tree based key-value stores with fast point and range reads. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 2824–2837. IEEE, 2025.

[72] Wenshao Zhong, Chen Chen, Xingbo Wu, and Jakob Eriksson. Disco: A compact index for lsm-trees. *Proceedings of the ACM on Management of Data*, 3(1):1–27, 2025.

[73] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. Xrp:in-kernel storage functions with ebpf. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[74] Hao Zhou, Yuanhui Chen, Wu Zeng, Lixiao Cui, Gang Wang, and Xiaoguang Liu. Gpcomp: Using gpu and ssd-gpu peer to peer dma to accelerate lsm-tree compaction for key-value store. *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[75] Chen Zou and Andrew A Chien. Assasin: Architecture support for stream computing to accelerate computational storage. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 354–368. IEEE, 2022.

A Artifact Appendix

Abstract

FLAX is a computational storage architecture that makes asynchronous, overlapped in-device I/O execution a first-class design goal. By tracking fine-grained data dependencies inside the device, FLAX allows the host to issue large batches of commands that the CSD schedules and executes asynchronously, while adaptive data loading and dynamic on-device buffer management sustain high concurrency under limited device resources. This artifact includes the source code and scripts to reproduce the experimental results presented in the paper.

Scope

The artifact includes the source code and experimental scripts to validate the following claims presented in our paper. For detailed descriptions, please refer to CLAIMS.md.

- **Write-only workloads (§6.2).** FLAX outperforms both Host and the host-managed CSD in throughput and tail latency, and its lightweight techniques are collectively essential to fully exploit the CSD's potential.
- **Read-only workloads (§6.3).** Under tight host memory budgets, FLAX achieves the highest read throughput, whereas with sufficient memory or skewed accesses, Host prevails. The `COMPOUND` command further improves performance over standard commands, and FLAX outperforms the host-managed CSD in all configurations.
- **Mixed workloads (§6.4).** FLAX outperforms Host, the host-managed CSD, FLAX-C, and FLAX-R in throughput, read and compaction latency, and tail latency, across all value sizes and all workloads except E. This confirms that offloading both foreground reads and background compaction is essential.

Contents

The artifact includes the source code of FLAX and a modified RocksDB that offloads operations to the CSD, along with scripts for running the benchmarks. It consists of four components:

1. **CSD-Virt**: A standard-compliant NVMe CSD emulator that supports FLAX.
2. **Linux kernel**: A modified Linux kernel (v6.0.10).
3. **RocksDB**: A modified version of RocksDB that supports compaction and read offloading.
4. **ForestDB-Bench**: A benchmark suite for running the evaluations.

Hosting

The artifact is available at https://github.com/snu-csl/FLAX.

Requirements

We evaluated FLAX on a machine equipped with two AMD EPYC 9754 128-core processors and 768 GB of DRAM, running Ubuntu 24.04.3 LTS with our modified Linux kernel (v6.0.10). We assigned 35 cores in total to the FLAX emulator, reserved 350 GB of memory for the emulated backend storage, and throttled the frequency of the FLAX compute cores to 1 GHz. At a minimum, FLAX requires six CPU cores to function, and the machine must have sufficient DRAM to hold both the emulated backend storage and the on-device memory.