KEIGO: Co-designing Log-Structured Merge Key-Value Stores with a Non-Volatile, Concurrency-aware Storage Hierarchy

Rúben Adão
INESC TEC & University of Minho
ruben.d.adao@inesctec.pt

Zhongjie Wu
Yale University
zhongjie.wu@yale.edu

Changjun Zhou
McGill University
changjun.zhou2@mail.mcgill.ca

Oana Balmau
McGill University
oana.balmau@mcgill.ca

João Paulo
INESC TEC & University of Minho
joao.t.paulo@inesctec.pt

Ricardo Macedo
INESC TEC & University of Minho
ricardo.g.macedo@inesctec.pt

ABSTRACT

We present KEIGO, a concurrency- and workload-aware storage middleware that enhances the performance of log-structured merge key-value stores (LSM KVS) when they are deployed on a hierarchy of storage devices. The key observation behind KEIGO is that there is no *one-size-fits-all* placement of data across the storage hierarchy that optimizes for all workloads. Hence, to leverage the benefits of combining different storage devices, KEIGO places files across different devices based on their parallelism, I/O bandwidth, and capacity. We introduce three techniques – *concurrency-aware data placement*, *persistent read-only caching*, and *context-based I/O differentiation*. KEIGO is portable across different LSMs, is adaptable to dynamic workloads, and does not require extensive profiling. Our system enables established production KVS such as RocksDB, LevelDB, and Speedb to benefit from heterogeneous storage setups.

We evaluate KEIGO using synthetic and realistic workloads, showing that it improves the throughput of production-grade LSMs up to 4× for write- and 18× for read-heavy workloads when compared to general-purpose storage systems and specialized LSM KVS.

PVLDB Reference Format:
Rúben Adão, Zhongjie Wu, Changjun Zhou, Oana Balmau, João Paulo, and Ricardo Macedo. KEIGO: Co-designing Log-Structured Merge Key-Value Stores with a Non-Volatile, Concurrency-aware Storage Hierarchy. PVLDB, 18(9): 2872 - 2885, 2025.
doi:10.14778/3746405.3746414

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dsrhaslab/keigo.

1 INTRODUCTION

Log-structured merge-tree (LSM) key-values stores (KVS), such as RocksDB [36], LevelDB [18], and Speedb [43], have become a fundamental storage building block for a variety of data-intensive applications, including databases [3, 12, 33, 37], caching systems [6, 34], file systems [1, 2], and analytics engines [8, 16, 30]. Their wide adoption is driven by their natural fit for write-heavy workloads. They buffer writes in main memory, flush them as sorted files to storage, and merge (compact) files across multiple levels of increasing capacities [30, 38]. However, the amount of data stored in these KVS is growing exponentially (in the order of hundreds of GiBs to TiBs), raising performance concerns as many applications require high throughput and low tail latency [5, 28].

To overcome this challenge, prior work proposes heterogeneous storage hierarchies combining the *performance* of emerging non-volatile byte-addressable technologies (*e.g.,* 3D XPoint [20], PCM [24]) with the *storage capacity* of traditional block-addressable devices (*e.g.,* NVMe SSD, SATA SSD). These solutions are provided as general-purpose hierarchical storage systems [26, 29, 39, 45, 53], serving as back-end to different applications (including LSM KVS), or as specialized KVS purposely built for heterogeneous storage [9, 40, 44, 50]. In this paper, we classify storage devices into *performance devices* and *capacity devices*. *Performance devices* offer high performance but reduced storage space (*e.g.,* NVMM), while *capacity devices* offer larger and cheaper storage alternative (*e.g.,* NVMe SSD, SATA SSD). Existing systems typically prioritize placing as much of the LSM structure as possible in the *performance device*, starting with the performance-sensitive components of the LSM, *i.e.,* the commit log (C_{log}) and top levels of the LSM tree (L_0 and L_1), followed by filling any leftover space with data from other levels. The remaining levels of the LSM are placed on the *capacity device*.

However, we show that *placing LSM components across complex and heterogeneous storage hierarchies solely based on the I/O bandwidth and capacity of devices leads to degraded performance*. To understand how the use of different storage devices impacts the performance of LSM-based KVS, and as our first contribution, we conduct an experimental study combining NVMM, NVMe SSD, and SATA SSD devices, where we report the following key findings (§3). First, under write-heavy workloads, placing as many LSM components as possible on the *performance device* is actually detrimental, degrading throughput up to 35% when compared to placing a smaller subset of data items. This phenomenon is caused by the increased write concurrency placed on the *performance device*, stemming from foreground writes to C_{log}, flushing in-memory data to L_0, and multiple parallel compactions occurring over different levels of the LSM tree. Second, since storage devices scale better for concurrent reads, placing as many components in the *performance device* is beneficial for read-intensive workloads. This means that there is no winning placement strategy that simultaneously fits the requirements of read and write workloads.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746414

Motivated by these findings, as our second contribution, we propose Keigo, a concurrency- and workload-aware storage middleware that accelerates the performance of LSM-based KVS for both read- and write-intensive workloads. The novelty of Keigo lies in how it carefully places each KVS component across the hierarchy of storage devices by consolidating the inherent properties of LSMs with the characteristics of each device, including *device-level parallelism*, *I/O bandwidth*, and *storage capacity*. Keigo can be used with multiple LSM KVS to provide a better placement of LSM components over the storage hierarchy. It is adaptable to changing read:write ratios and skew, and only needs minimal device profiling to create an initial data placement scheme.

Keigo introduces three techniques. To optimize write workflows, the LSM levels placed in the *performance device* are not only determined by the device I/O bandwidth and capacity but also by the number of concurrent writers that can work in parallel (§4.2). This means that under write-heavy workloads, Keigo stores just a small subset of files of the KVS (*i.e.,* C_{log} and lower levels of the LSM tree) on the faster device, reducing write contention. Second, for read-heavy workloads, Keigo introduces a per-device persistent read-only cache that continuously tracks and copies hot KVS files from slower to faster storage devices, maximizing the hit ratio on the upper levels of the hierarchy (§4.3). Third, for easy compatibility with different LSM KVS, we use context propagation techniques [31, 32]. We expose a POSIX-like interface where system calls propagate the internal KVS operation that started a given POSIX request (*e.g.,* C_{log}, flush, compactions) to Keigo, which uses this information to make efficient placement decisions (§4.4).

As our third contribution, we validate the performance of Keigo through a comprehensive experimental evaluation, using both synthetic (*i.e.,* YCSB benchmark [13]) and production workloads from Meta [7] and Nutanix [28]. We demonstrate that Keigo can enhance a range of LSM KVS by adding it to three popular systems with less than 100 LoC: RocksDB, LevelDB, and Speedb. Further, we show that Keigo improves LSM KVS performance in two storage hierarchies, composed of DRAM-NVMM-NVMe and DRAM-NVMe-SATA SSDs. We compare Keigo against general-purpose storage systems, namely ext4 and OpenCAS [39], and state-of-the-art LSMs designed for heterogeneous storage devices, namely BushStore [44] and PrismDB [40]. Results show that Keigo outperforms all systems across all testing scenarios. Compared to general-purpose storage systems, Keigo improves RocksDB throughput up to 4× and 12× under write-heavy and read-heavy workloads, respectively. As for the specialized LSM-based KVS, Keigo improves RocksDB throughput up to 1.6× for write-heavy workloads and up to 18× for read-heavy workloads. Moreover, for Meta and Nutanix production workloads, Keigo improves the next best system by 1.15× and 1.4×, respectively. Finally, Keigo enhances the performance of Speedb and LevelDB's baselines by up to 8.85× and 5.76×.

2 BACKGROUND

This section provides background on devices that make up current storage hierarchies and discusses classic strategies to manage them. Further, it provides background on the organization of LSM KVS.

2.1 Heterogeneous storage management

Data centers use heterogeneous storage hierarchies composed of different storage devices, each offering trade-offs of performance, capacity, and cost. Emerging NVMM devices (*e.g.,* 3D XPoint [20], PCM [24]) enable byte-addressable persistent storage with performance closer to that of DRAM. Ultra-low latency NVMe SSDs (*e.g.,* Z-NAND [11]) deliver μs-scale latencies with larger capacity than NVMM, while traditional block-addressable devices (*e.g.,* SATA SSD, HDD) provide a denser and cheaper alternative.

To manage a heterogeneous storage hierarchy, systems mainly follow two strategies: *caching* and *tiering*. For simplicity, we assume a two-tier hierarchy, made of a *performance device* that is faster, smaller, and expensive (NVMM), and a *capacity device* that provides cheaper and large capacity storage (NVMe SSD). Our design, however, can be used for multi-tiered storage hierarchies (§4).

Caching. In caching, the *performance device* is used as a persistent cache to accelerate the *capacity device* [29, 39, 45]. Hot data resides in the *performance device*, ensuring low latency and high throughput under skewed read-heavy workloads. On cache misses, the *capacity device* serves the requests. The item placement is determined by an eviction policy (*e.g.,* LRU, LFU). Depending on the writing policy (*e.g.,* write-back), the *performance device* can absorb write operations, flushing dirty data upon explicit synchronization.

Tiering. In tiering, data items are partitioned across devices according to a specific placement scheme, which is often driven by the items' popularity, size, consistency guarantees, and more [26, 40, 53]. Contrary to caching, items only reside in one of the devices and are not constantly promoted/evicted to/from the *performance device*.

2.2 Heterogeneous storage in LSM

LSM overview. Log-structured merge tree (LSM) key-value stores (KVS), such as RocksDB [36], LevelDB [18], and Cassandra [27], are widely adopted storage systems that are optimized for write-intensive workloads [30, 38]. Write operations are absorbed by a memory component (C_m or *memtable*) that when is full, it is *flushed* to persistent storage in one large sequential I/O operation. The flushed C_m is then merged by background threads in a tree-like structure maintained in persistent storage (C_{disk}). C_{disk} contains multiple levels of increasing sizes (L_0, L_1, ..., L_N), where each level contains multiple *immutable* sorted files, called SSTs. This merging operation is called *compaction*. LSM KVS can run several concurrent compactions using dedicated background threads, in addition to the foreground load. To avoid losing data held in C_m, writes are backed up in a *write-ahead log* (C_{log}) that also resides in persistent storage. To improve read performance, data items may be temporarily held in an in-memory region called *block cache*. Foreground reads first access C_m, followed by the block cache, followed by the SST files.

With this multi-level structure, LSMs are a natural fit for leveraging storage hierarchies. So far, heterogeneous storage in LSMs has been tackled through two main approaches.

Using general-purpose tiered storage systems. A commonly used approach is to use general-purpose hierarchical storage systems (*e.g.,* OpenCAS [39], Ziggurat [53], P2Cache [29]), which usually reside at the kernel and do not require POSIX-compliant applications to be modified to obtain performance gains. The storage

system determines which device services each I/O request based on the data access pattern (*e.g., write* followed by *fsync* [29, 45], small *vs.* large writes [53, 54]). However, while these systems are designed to handle a wide range of applications, they are agnostic of LSM I/O logic, treating all requests in the same way regardless of their origin or priority. For example, while compactions at different levels exhibit similar access patterns — sequentially read SST files from disk, merge sort them in memory, and sequentially write new SST files to disk — they incur different performance costs [5, 52].

Building new LSMs from the ground-up. Alternatively, prior work proposes building new LSM KVS for heterogeneous storage. These new designs consider the trade-offs of each storage device with careful placement of the LSM components [9, 10, 44], employ new compaction schemes compliant with the storage hierarchy [40, 50], and redesign the write operation flow [9, 23]. However, such systems are difficult to adopt in practice, as LSMs are typically core components in large pipelines. A significant implementation effort is required to replace existing LSMs with such alternatives. This problem becomes further accentuated when new storage devices are released, deprecating the previous optimizations.

Keigo provides a middle-ground between these approaches. By offering a middleware tailored for LSM KVS, Keigo is aware of the I/O flows and priorities unique to LSMs, leveraging this information when deciding the placement of different components. Moreover, its design is flexible: it can be used by various LSM KVS and can be adapted to heterogeneous storage hierarchies of different depths.

3 ISSUES WITH HETEROGENEOUS STORAGE FOR LSM

We conduct an experimental study to understand how hierarchies of different storage devices impact the performance of LSM KVS. We consider hierarchies combining byte-addressable and block-based storage devices. Our observations *complement* previous studies that explore idiosyncrasies of individual storage devices [17, 19, 21, 46, 48], by pinpointing their impact in LSM-based KVS.

Hardware and OS configurations. We run the experiments in a server with two 18-core Intel Xeon processors, 192 GiB of memory, a 128 GiB Intel Optane DCPMM (AppDirect mode), a 1.6 TiB Dell PM1725b NVMe SSD, and a 480 GiB Intel S4610 SATA SSD, using Ubuntu Server 20.04 with kernel 5.4.0. We restrict the main memory to 16 GiB using Linux control groups to ensure *1)* the storage and memory configurations of the server form a hierarchy and *2)* that most of the requests are submitted to persistent storage.

LSM KVS configuration. The experiments were conducted over RocksDB configured with two C_m of 128 MiB, a thread pool of four threads for internal operations (including a flushing thread), and a 1 GiB block cache. These configurations correspond to production settings at Nutanix [5] and follow RocksDB tuning guidelines [35]. We evaluate RocksDB with 8 client threads using write-intensive (YCSB A, 1:1 r:w ratio) and read-only (YCSB C) workloads with a uniform key distribution. Before each experiment, RocksDB was pre-loaded with 50M key-value pairs (1KB value).

Storage setups. The experiments were conducted over the following storage configurations: *1) ext4* corresponds to an ext4 file system mounted over the NVMe or SATA SSDs; *2) ext4-DAX* corresponds to NVMM backed by an ext4 file system with DAX enabled to perform direct I/O; *3)* we implemented a custom backend that combines NVMM (backed by a PMDK driver [22]) and NVMe SSD (ext4 file system). The first two setups show storage systems commonly used in production, while the latter demonstrates a storage hierarchy adopted by prior work [29, 40, 44, 44, 45, 50].

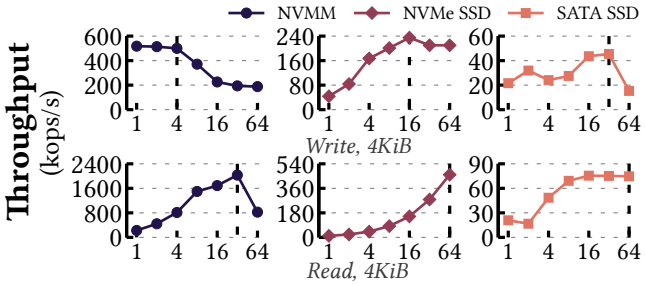


Figure 1: Performance of NVMM, NVMe SSD, and SATA SSD storage devices for FIO write and read workloads with increasing number of threads (1 to 64, x-axis). Vertical lines mark the maximum performance achieved by each device.

3.1 Device-level parallelism

Before exploring the performance of LSM under heterogeneous storage, we first show how each storage device performs when exposed to different workloads and concurrency levels. We conducted experiments using the `fio` benchmark with sequential read and write workloads (`sync` I/O engine) under 4 KiB blocks with increasing worker threads (1 to 64). Each worker operates over a distinct device region to avoid overlapping working sets. We report the mean result over 5 runs, with less than 5% standard deviation.

Figure 1 depicts the throughput of the NVMM, NVMe SSD, and SATA SSD devices. Storage devices have different degrees of parallelism. For *write workloads*, NVMM reaches its peak performance under 4 threads (500 kops/s), while NVMe SSD scales up to 16 threads. NVMe SSD devices are highly parallel, containing multiple I/O channels connected to independent flash dies. While the SATA SSD achieves the highest parallelism level, it experiences the lowest performance due to its limited interface bandwidth and architecture [21]. Beyond 4 workers, NVMM's performance decreases up to 2.6× (188 kops/s under 64 threads). As observed in [19, 48], the reason behind NVMM's poor scalability is caused by contention in the XPBuffer (*i.e.,* increased evictions and write backs to the memory media) and the integrated memory controller (*i.e.,* limited queue capacity when multiple cores target a single DIMM). Consequently, *under severe write contention, the NVMe SSD achieves similar or better performance than NVMM* — under 16, 32, and 64 threads, the NVMM achieves 226 kops/s, 194 kops/s, and 188 kops/s, while the NVMe SSD achieves 235 kops/s, 210 kops/s, and 210 kops/s, respectively. For *read workloads*, all storage devices showcase a higher parallelism level. Specifically, NVMM achieves its maximum performance under 32 threads, while block-based storage devices can scale up to 64 threads. Contrary to write workloads, NVMM read performance is limited by the number of physical NUMA cores [48].

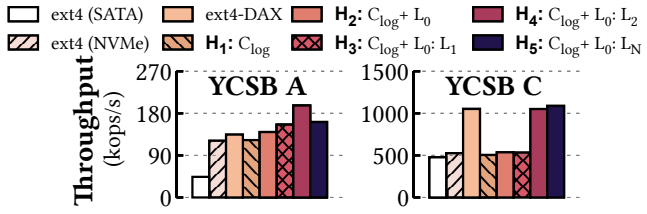


Figure 2: RocksDB performance when placing LSM components across devices under mixed and read-only workloads.

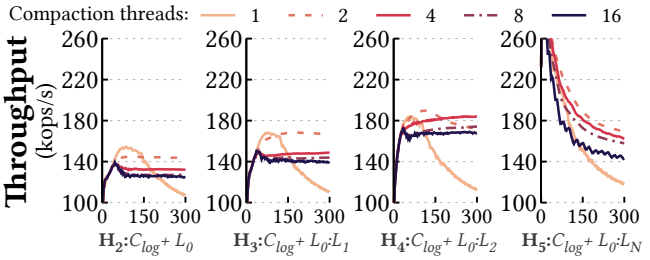


Figure 3: RocksDB performance over time (in seconds) under different placement schemes with increasing number of concurrent compactions. The caption of each plot depicts the components placed on NVMM.

> **Takeaway 1.** *Due to the characteristics of each storage device, device I/O bandwidth should not serve as the delimiting factor for managing a heterogeneous storage hierarchy. Devices must be carefully combined to realize the full potential of the storage hierarchy in terms of performance, parallelism, and capacity.*

3.2 Placement of LSM components

We now explore the impact of placing different LSM components (excluding C_m) across the storage hierarchy. To this end, we compare RocksDB throughput under ext4 with NVMe and SATA SSDs, ext4-DAX, and the custom storage backend configured with different placement schemes. In the latter, we start by placing C_{log} on NVMM and keep the remainder components on the NVMe SSD (H_1). We then incrementally place more components on NVMM—C_{log}+L_0 (H_2), C_{log}+L_0:L_1 (H_3), ... — until the full dataset is serviced from this device. Figure 2 depicts the performance of RocksDB under write- and read-intensive workloads for all storage setups.

Write-intensive workloads. Placing all LSM components in the faster storage device (ext4-DAX and H_5 setups) increases RocksDB throughput up to 1.4× and 3.7× when compared to ext4 with NVMe and SATA SSDs, respectively. Also, the custom storage backend improves throughput up to 20% over ext4-DAX due to its PMDK driver, which stores and accesses data items (both I/O and metadata) directly from user-space. However, maybe surprisingly, we observe that *placing the entire dataset on NVMM is actually detrimental*, degrading throughput up to 35% when compared to just placing a small subset of data items (H_4). The reason behind this performance decrease is a direct result of the increased write concurrency placed over NVMM, which exceeds the maximum parallelism supported by the device (§3.1). Placing more components on NVMM results in more workers concurrently writing to it, which are originated from foreground writes to C_{log}, flushing C_m to L_0, and multiple parallel compactions occurring over different levels of the LSM tree.

Read-intensive workloads. Contrary to write-intensive workloads, RocksDB performance improves with the number of components placed on NVMM. Specifically, ext4-DAX and H_5 surpass ext4 throughput by up to 2.3×. This is because while NVMM's writes do not scale well, reads can scale up to the number of physical cores (§3.1). As such, *placing the entire dataset in the faster tier of the storage hierarchy is beneficial for read-intensive workloads.*

> **Takeaway 2.** *Tiering is the recommended strategy to improve write performance in LSM-based KVS. LSM components closer to C_m must be placed on faster storage tiers. Not only because these handle the critical write data path, but also because placing more LSM levels degrades overall performance due to the poor write concurrency of emerging storage devices.*

> **Takeaway 3.** *Caching is the recommended strategy to improve read performance in LSM-based KVS. Leveraging the fact that emerging storage devices offer excellent read bandwidth and scalability, one should strive to maximize their hit ratio.*

3.3 Concurrent compactions

We now analyze the performance impact of increasing the number of writers and LSM components stored across the storage hierarchy. We fixed the number of client (8) and flushing (1) threads but varied the number of compaction threads (1 to 16). Figure 3 shows the throughput over time across all configurations under YCSB A.

When a small amount of data is persisted in NVMM (namely, H_2 and H_3), RocksDB's performance is suboptimal since a large amount of requests are serviced by the NVMe SSD. Increasing the number of threads does not improve performance since the problem lies in the placement scheme. Similarly to §3.2, RocksDB achieves the best throughput performance under the H_4 configuration, when using 4 compaction threads. With a low number of threads (*i.e.,* up to 2), RocksDB experiences write stalls, which occur when flushes and low-level compactions are slow or on hold [5, 52]. On the other hand, too many compaction threads increases the number of writers contending the NVMM. These effect becomes further accentuated when all components reside on the faster tier (H_5).

> **Takeaway 4.** *The KVS performance is directly impacted by the placement of components across the hierarchy and the number of writers. This means that there is no single placement scheme that fits all workloads and system configurations.*

3.4 Popularity of LSM levels

We now investigate how different data distributions impact the storage hierarchy. We analyze the accesses over time across the LSM (levels and block cache) under different distributions, including *zipfian 0.99* (high skew), *zipfian 0.80* (medium skew), and *uniform*.

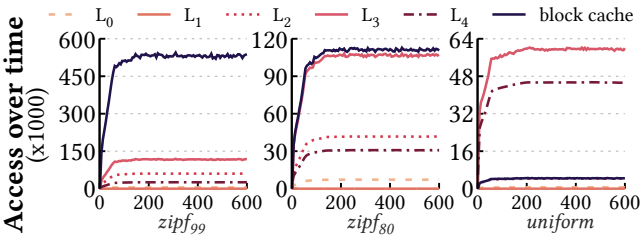


Figure 4: Read operations done over time (in seconds) across LSM levels under different data distributions.

Read workflows. Figure 4 depicts the number of foreground reads observed at each LSM level for a read-only workload. We observe that L_3 is the most accessed level. While L_0 to L_2 have a combined size of approximately 3 GiB, L_3 holds 25 GiB of the dataset. On the other hand, even though L_4 can accommodate 10× more data than L_3, it holds the older and colder portion of the dataset [30, 51]. We also observe that the block cache places an important role in highly skewed workloads, being able to service a large amount of reads, but its impact fades as the distribution becomes less skewed. This means that for low skewed read-intensive workloads, levels higher than L_2 should be stored in the faster tier as well; otherwise, a significant portion of requests will be made over the NVMe SSD.

Write workflows. Due to the design of LSM KVS, the write data path follows the same workflow regardless of the data distribution. As such, components that are on the critical data path (*i.e.*, C_{log}, L_0, L_1) should be placed on the faster tier, while the placement of other levels must be chosen to prevent write contention.

> **Takeaway 5.** *When considering different data distributions, there is no winning placement strategy that simultaneously fits the requirements of read and write workflows.*

4 KEIGO STORAGE MIDDLEWARE

We propose Keigo, a concurrency- and workload-aware storage middleware that accelerates the performance of existing LSM KVS running on a hierarchy of storage devices. Through its design, Keigo automatically capitalizes on the strengths of the devices and compensates for their weaknesses, all while requiring minimal changes to the KVS. Following the takeaways discussed in §3, Keigo's design is built following five core principles.

Parallelism, bandwidth, and capacity-aware LSM placement. Keigo realizes the full potential of the storage hierarchy by placing LSM components according to the *parallelism*, *bandwidth*, and *capacity* of each storage medium.

Maximize hit ratio on faster devices. Keigo exploits the high read bandwidth, scalability, and low latency of faster storage devices (*e.g.*, NVMM), maximizing the number of requests served from them and minimizing read latency across the hierarchy.

Automatic partitioning and tuning. To adapt to the performance characteristics of the different storage mediums, Keigo automatically manages the LSM partitioning across the hierarchy and the concurrency of background data movements between devices.

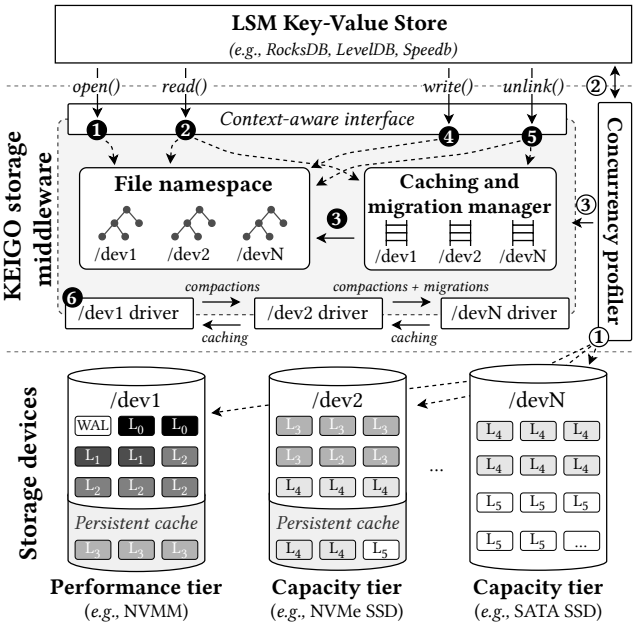


Figure 5: Keigo architecture. It follows a multi-tier storage hierarchy, co-designed with the properties of LSM KVS and the inherent characteristics of different storage devices.

Flexibility and extensibility. Keigo supports different combinations of storage devices and enables system designers to implement custom storage drivers and placement policies to comply with their workloads and system requirements.

Low intrusiveness. Keigo requires minimal changes to LSM KVS, minimizing the work needed to maintain and port it to new systems.

4.1 Keigo design overview

Figure 5 provides a high-level view of Keigo. The system is a user-level middleware that sits between the LSM KVS (*e.g.*, RocksDB, LevelDB) and a hierarchy of heterogeneous storage devices. Building on the observation that placing as many KVS files as possible on the fastest storage tier can simultaneously improve read performance but degrade write performance, Keigo makes placement decisions that account for *device-level parallelism*. To optimize writes, Keigo restricts placement on the faster device to only the LSM components in the critical data path, ensuring that performance-sensitive operations benefit from high throughput and low latency while keeping the number of active concurrent writers within the device's supported parallelism. To optimize reads, Keigo asynchronously caches hot SST files on faster tiers of the hierarchy, lowering read latency without interfering with write performance.

Keigo is organized as follows. At its core, a *file namespace* component provides a logical-to-physical mapping of KVS files (*e.g.*, SST, C_{log}) to their corresponding location, transparently abstracting a hierarchy of storage devices into a single logical one. The hotness of each file is continuously tracked through a *caching and migration manager*, which decides when to move data across devices either to improve the KVS performance (*caching*) or for space management

(*migrations*). To enable combining different types of storage devices, Keigo exposes a *driver interface* that allows implementing custom I/O logic for each device, while supporting specialized interfaces and I/O protocols (*e.g.,* PMDK [22], io_uring). In this paper, we assume a storage hierarchy made of byte-addressable (NVMM) and block-addressable storage devices (NVMe and SATA SSDs), acting as *performance* and *capacity devices*, respectively. DRAM plays a minimal role in Keigo's design, as most LSM KVS already perform internal memory management (*e.g.,* block cache, memtable) to optimize operations in the critical data path. To automate the partitioning of files across the hierarchy, Keigo integrates an *offline concurrency profiler* that generates placement schemes based on the performance and parallelism of the storage devices in the hierarchy, as well as the KVS workload. In sum, Keigo is responsible for managing how files are efficiently stored across a hierarchy of devices, while other responsibilities, such as thread management and compaction scheduling, remain under the control of the KVS. Keigo is driven by three main techniques:

1) Concurrency-aware data placement (§4.2). On-disk LSM components are split across the storage hierarchy based on two placement policies, driven simultaneously by the properties of the LSM and by the parallelism, I/O bandwidth, and capacity of each storage device. With *performance-aware placement*, LSM components whose operations are in the critical data path (C_{log} and levels closer to C_m) and whose number of concurrent writers does not exceed the device's supported parallelism are placed on the faster storage device (`/dev1`). With *capacity-aware placement*, the bottom LSM-tree levels, which accommodate the colder majority of data and is where non-critical work is conducted, are placed on the remainder devices (`/dev2`, ..., `/devN`).

2) Persistent, read-only caching (§4.3). Read-heavy workloads can experience low performance if a substantial amount of requests are handled by slower devices in the hierarchy. Keigo integrates a per-device persistent read-only cache that tracks and copies hot SST files from slower to faster storage devices, maximizing the hit ratio on the upper levels of the hierarchy and improving the KVS performance under low skewed, read-intensive workloads.[1]

3) Context-based I/O differentiation (§4.4). To minimize the intrusiveness of porting existing KVS, Keigo exposes a POSIX-like interface where system calls are passed with an additional *context* field. The *context* defines the internal KVS operation that originated a given POSIX request (*e.g.,* flush, C_{log}, L_N compaction), and is used to determine the device that will handle it. This is achieved by using *context propagation* [31, 32], enabling KVS-level information to be propagated to Keigo with minor code changes.

Operation flow. Figure 5 illustrates how Keigo handles KVS operations. KVS files that contain clients' data are created when writing to C_{log}, flushing C_m to disk, and during compactions. These operations are submitted to Keigo via extended `open()` calls that complement the standard POSIX interface with the *context* that triggered such operation at the KVS level, whether it is a `log` write, `flush`, or `compaction` (and respective levels involved) (❶) (§4.4). Based on this context, Keigo creates the file in the corresponding storage device according to the placement scheme (❻), generated offline by the *concurrency profiler* (§4.2). To transparently support POSIX I/O to the KVS across a storage hierarchy, Keigo's file namespace maintains a logical-to-physical mapping of each file, linking the SST file with its physical location, original file identifier (*i.e.,* file descriptor, memory address), and logical file descriptor returned to the KVS, which is used in subsequent I/O operations to that file such as `read()`, `write()`, and `close()`. Writes to the critical data path (❹), coming from C_{log}, flush, and low-level compactions, are handled by the faster storage device (`/dev1`), while foreground reads are served from the tier where the file currently resides, whether cached or placed by the policy (❷). Files may be moved between devices during compactions involving levels placed at different tiers (§4.2); when *caching* hot files to faster devices in the hierarchy (§4.3); and during *migrations* triggered for space management (❸) (§4.2). When a compaction completes, unlinked files are removed from the caching/migration manager and the file namespace (❺).

[1]The current Keigo prototype caches full SST files, but it is possible to further refine the policies for block-level caching.

4.2 Optimizing writes

Keigo integrates the takeaways from §3, by providing two placement policies to optimize writes.

Performance-aware data placement. The *performance placement policy*, applied to the fastest device in the storage hierarchy (`/dev1`), manages the LSM components that directly impact the performance perceived by clients. The policy follows two key rules: *1)* identify the components that lie in the critical data path that must be placed on the faster tier to minimize latency, and *2)* determine the additional components that can be placed without exceeding the device's supported write parallelism and capacity.

1) Performance-critical components are placed in the faster tier. Keigo places C_{log} and lower levels of the LSM (L_0 and L_1) on the faster tier. Our reasoning is threefold. First, writes to C_{log} need to be fast since they incur significant overhead to the critical data path, especially when the OS page cache is bypassed (*e.g.,* `O_DIRECT`) or when crash-consistency guarantees are desirable. Second, LSM workloads have strong temporal locality, where the popularity of objects fades over time [7, 51]. This means that the most accessed keys are stored in the most recent SST files, which are placed at the lower LSM levels. Third, background tasks that involve C_m and the lower levels of the tree are prone to write stalls, especially under write-heavy workloads. Stalls occur when flushes cannot proceed, either because flushes and low-level compactions are slow or on hold, resulting in degraded performance [5]. While placing these components on the faster tier does not avoid write stalls, their duration and performance degradation are hampered [52].

2) Ensure the maximum device-level parallelism is not exceeded. I/O bandwidth cannot serve as the sole determining factor for placing files in the faster tier, as the KVS performance under write-intensive workloads is significantly impacted by the number of active writers in the system, especially on NVMM devices (§3). This is particularly detrimental for higher levels of the LSM, which perform larger compactions and contend the device for longer periods [5]. As such, Keigo ensures that levels placed in the faster device are not solely determined by its bandwidth and space but also by the degree of parallelism among writer threads (*i.e.,* writes to C_{log}, flushes, parallel compactions). This means that under write-heavy workloads, by selectively storing a small subset of KVS files (*e.g.,* C_{log} + L_0 to L_2,

with a combined size of ≈3 GiB (§3.4)) on the faster device, Keigo reduces write contention and improves performance.

Keigo implements these rules through an *offline concurrency profiler* that generates a placement scheme with the LSM levels that must be placed in the faster storage tier. The profiling process is made in three phases, as depicted in Figure 5. First, it profiles the performance of each storage device, using the `fio` benchmarking tool [4], under read and write workloads with increasing number of threads until performance degradation is observed. This allows determining the maximum parallelism supported by each device (§3.1). Second, it profiles the average number of concurrent writes that occur at different levels of the LSM during write-heavy workloads, using the YCSB benchmark [13] (②). We used YCSB as it generates workloads representative of those evaluated in §5. Nevertheless, system designers can use alternative configurations and workloads that more accurately reflect their production environments. Based on these results, the profiler generates a scheme with the LSM levels whose cumulative concurrency demand does not exceed the supported write parallelism of the faster tier (③). Similarly to prior work [9, 40, 42, 50], Keigo assumes the internal state of the LSM remains stable over time (*e.g.,* size of LSM levels, number of active threads). While our experiments validate the effectiveness of this approach, there may be scenarios where the data placement should change over time (*e.g.,* shifting access patterns, varying number of writers). We leave the use of *online profiling* for future work.

Capacity-aware data placement. In the *capacity placement policy*, applied to the remainder devices (`/dev2`, ..., `/devN`), LSM levels are placed in each tier according to their available storage space. Our reasoning is that as new objects are inserted or updated in the KVS, older values are pushed down the stack and stored at the bottom LSM levels, thus accommodating the *main bulk* and *colder* portion of the dataset. To ensure sufficient capacity for incoming files, SST files are migrated across devices following an LRU eviction scheme. Moreover, due to the size of the bottom LSM levels (*i.e.,* hundreds of GBs to TBs), levels can be stored across multiple tiers.

Data movements across the hierarchy. In Keigo, data operations between devices arise from two sources: compactions between LSM levels placed in different devices (managed by the KVS) and explicit operations performed by Keigo for performance and space management, including caching (§4.3) and migrations. For compactions, Keigo performs a standard logical-to-physical mapping mechanism, translating the POSIX calls submitted by the KVS into their corresponding device-specific accesses – namely, reads SST files from the targeted levels, writes the newly generated ones in the corresponding location, and removes obsolete SST files.

For migrations, SST files are managed with an LRU eviction policy based on file access frequency. Candidate SST files for eviction are placed in a dedicated queue and moved by a thread pool once an upper-bound threshold is exceeded. This threshold ensures migrations are only triggered when device utilization surpasses a certain limit, avoiding premature eviction of SST files and minimizing inter-device traffic. To avoid migrating SST files from upper LSM levels, particularly during atypical bursts to older key-value pairs, the eviction policy weighs the level at which SSTs respect to, prioritizing the migration of files from deeper (*i.e.,* colder) LSM levels. Further, to prevent the write concurrency problem observed in §3, the *caching and migration manager* component tracks the number of active writers in each device (*i.e.,* compactions, caching, ongoing migrations) through atomic counters and dynamically adjusts migration parallelism to stay within the device's supported write parallelism. Finally, to reserve space for incoming SST files, Keigo forces migration when free space falls below a lower-bound threshold.[2] While this could, in theory, temporarily exceed the device's supported parallelism, we have not observed this in practice; nevertheless, since capacity devices store the non-critical portion of the dataset, we do not expect any noticeable impact.

4.3 Optimizing reads

Under read-heavy workloads exhibiting medium skew or uniform distributions (§3.4), most requests are made over the slower devices in the hierarchy (`/dev2`, ..., `/devN`). While placing more levels on `/dev1` improves the performance of read-dominated workloads, it would severely impact write-heavy ones (§3.2). Keigo overcomes this challenge by reserving space at each storage device to persistently cache frequently accessed SST files from devices at the lower levels of the hierarchy. For example, hot SST files from device `/devY` (*e.g.,* `/dev2`) are temporarily cached on `/devY-1` (*e.g.,* `/dev1`) as *read-only copies*, while the *original* files remain persisted in `/devY`. Cached files are created in read-only mode, as in traditional LSM KVS (*e.g.,* RocksDB) files become immutable after being fully written [14]. Files are removed from the cache when the *original* file is deleted (*e.g.,* compactions) or when space is needed for hotter SST files. Keigo' caching process is applied to all devices in the hierarchy, except for the last one, and addresses the following questions: *1) which* files should be cached; *2) when* should files be cached; and *3) how* should the actual process of caching be made in the presence of workloads with different read-write proportions.

1) File temperature profiling. Keigo determines *which* files should be copied to the persistent cache by tracking SST access frequency. Access counters are maintained at the file namespace and are updated by the KVS foreground threads (during reads that miss the block cache). To prevent caching stale data due to shifting access patterns or compactions, Keigo applies an aging factor to each SST file, decreasing the access counter in an exponential back-off manner each time a file is copied.

2) Hit ratio maximization. Keigo determines *when* to cache SST files by monitoring the hit ratio of foreground reads in the storage device that hosts the cache (`/devY-1`). To achieve this, a dedicated background thread continuously computes the hit ratio of `/devY-1` and triggers a copy when the value is below a certain threshold. When triggered, Keigo selects the most frequently accessed SST file from `/devY` and places it in an internal queue to be copied. The monitoring granularity and hit ratio threshold are user-configurable.

3) Concurrency-aware copying. Keigo uses a dedicated thread pool to cache files in parallel. However, this operation must be done carefully to avoid exacerbating the write concurrency problem observed in §3. As such, Keigo tracks the number of *active writers* on the storage device that hosts the cache (*e.g.,* `/dev1`) — namely, C_{log}, flush, compactions, and caching — and dynamically adjusts the number of caching threads to ensure the total writers do not

[2] In our experiments, the upper- and lower-bound thresholds were set to 5% and 2% of the device capacity, respectively, but these values are user-configurable parameters.

exceed the device's maximum parallelism level. The number of writers decreases when the corresponding background tasks finish.

Furthermore, in write-only workloads, the number of caching threads may temporarily drop to zero due to frequent flushes and compactions. This behavior is not detrimental, as caching is unnecessary in the absence of read operations.

4.4 Context-based I/O differentiation

General-purpose storage systems are agnostic of the LSM's internal I/O logic, treating all requests in the same manner regardless of their priority and performance cost. On the other hand, building specialized LSM KVS from the ground up to support heterogeneous storage devices requires significant implementation efforts, which are continuously repeated upon the release of new storage hardware. KEIGO strikes a balance between the two approaches by propagating the origin of KVS operations (*e.g.,* C_{log}, flush, L_0 to L_1 compaction) to the storage, enabling the same level of control and performance as LSM-specific optimizations while imposing minimal code changes. It combines ideas from *context propagation*, a commonly used technique that enables a system to forward additional request information along its execution path [31, 32, 49], and applies them to determine which device should handle each request. The process for differentiating requests in KEIGO is twofold.

Instrumentation. First, KEIGO needs minimal instrumentation on the data path of the LSM foreground and background work, including C_{log}, flushes, and compactions at different levels. Whenever these operations are triggered, a tag (or *context*) associated with the corresponding operation type (*e.g.,* `log`, `flush`, `comp_l0_l1`) is registered at a variable residing in the local address space of each thread through the OS's thread-local storage mechanism [15].

Execution. During execution, when a new file is created, an extended `open()` system call is sent to KEIGO with the original arguments of POSIX `open()` and the *context* that originated the request. This *context* determines the device that will persist the file, according to the data placement scheme. Subsequent file requests (*e.g.,* `read()`, `write()`, `close()`) do not need to pass the operation context, as they can access the file through KEIGO's file namespace.

4.5 Implementation

We implemented a prototype of KEIGO in 4K LoC in C++. KEIGO is provided as a user-level library, extending 25 POSIX calls, such as `open()`, `read()`, `pwrite()`, etc; we found that supporting this set of calls is sufficient to enable KEIGO over LSM-based KVS (§5.4).

Storage drivers. We implemented two storage drivers: a NVMM driver for managing byte-addressable and a POSIX driver for block-based storage devices. The former is implemented using Intel PMDK. POSIX operations are converted into their corresponding memory-mapped version: reads are serviced via `memcpy()`, and writes with `pmem_memcpy()` using `ntstore` instructions to avoid polluting the processor cache. KEIGO provides similar consistency guarantees as POSIX, making data durable upon explicit synchronization through `pmem_flush()` and `pmem_drain()` instructions. For the POSIX driver, requests are submitted following standard POSIX semantics.

In-memory structures. The *file namespace* is implemented in a concurrent hashmap that maps *logical* file descriptors to the metadata objects that maintain information about the actual file (*e.g.,* original file descriptor, pathname). Entries are atomically updated at file creation/removal (*i.e.,* flush, compactions, C_{log}) and when moving files across devices (*i.e.,* trivial moves, caching, migrations).

Persistent caches. Each *persistent cache* uses a dedicated thread pool for background copying of SST files and separate threads for monitoring the hit ratio of each storage device. The hit ratio threshold is defined by a fixed, configurable parameter. On file copying, KEIGO prefetches targeted SST files to the OS page cache using `fadvise()` and `readahead()`. After its completion, it hints the OS to evict the pages that hold prefetched data.

Porting LSM-based KVS. We integrated KEIGO with three popular LSM KVS, namely RocksDB [36] (57 LoC), Speedb [43] (81 LoC), and LevelDB [18] (40 LoC), which required updating the thread-local variables for context propagation and replacing POSIX system calls with those supported by KEIGO.

4.6 Discussion

Impact of device-level parallelism. Device-level parallelism is a fundamental characteristic that universally affects storage devices. While its effects are especially pronounced in NVMM due to its inherent read-write asymmetry, our findings demonstrated that block-addressable devices like NVMe and SATA SSDs are also impacted (§3.1), resulting in suboptimal performance under high write concurrency. In §5.5, we demonstrate how KEIGO improves LSM KVS performance even in block-addressable storage hierarchies.

Performance under multiple NVMM devices. While our experiments focus on using a single NVMM, the findings and core principles of KEIGO extend to setups with multiple devices. Previous studies demonstrate that increasing the number of devices enhances overall write bandwidth but does not eliminate the concurrency limit inherent to each device [25, 48], highlighting the importance of KEIGO's concurrency-aware placement scheme.

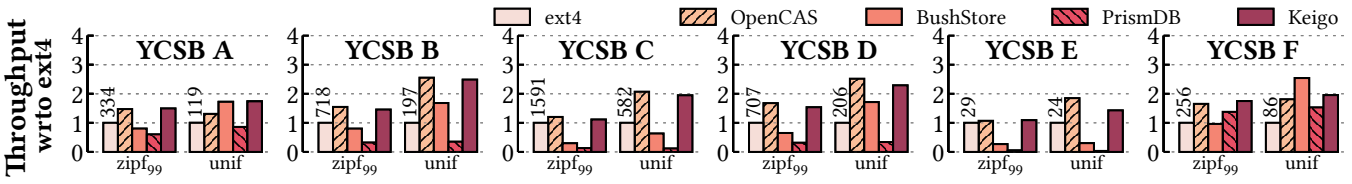
Internal metadata consistency. KEIGO's current implementation is fault tolerant. Upon a crash, KEIGO reconstructs its namespace by scanning the files stored in each device, which allows redirecting operations to the correct location. Cached files may be present during recovery but do not affect KEIGO's correctness, as they are removed during normal operation. We defer the improvement of crash-recovery mechanisms for KEIGO's metadata to future work.

5 EVALUATION

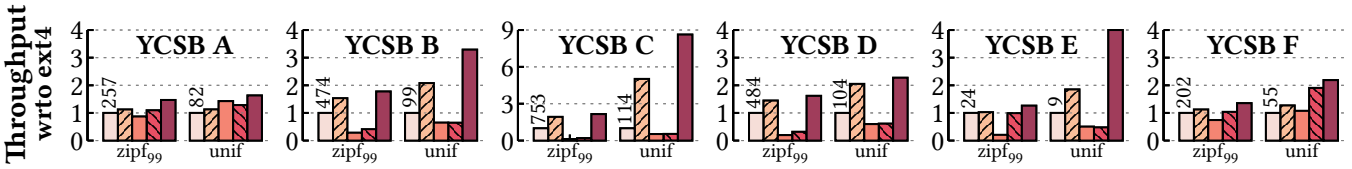
Our evaluation sets out to answer the following questions:

- How does KEIGO perform under varying dataset sizes (§5.1)?
- How does KEIGO handle different levels of concurrency (§5.2)?
- How does KEIGO perform under production workloads (§5.3)?
- Can KEIGO improve the performance of other LSM KVS (§5.4)?
- What is the performance breakdown of the different techniques implemented in KEIGO (§5.5)?

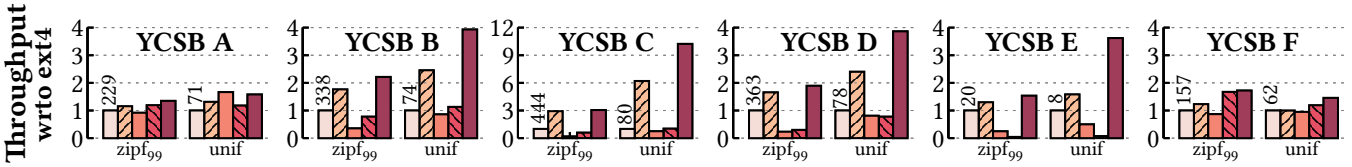
Hardware and OS configurations. We used the same hardware and OS configurations as in §3, with the exception of the SATA SSD, which was replaced by two 480 GiB SATA SSDs configured with RAID-0. Unless stated otherwise, experiments were conducted using the NVMM and NVMe SSD devices.



(a) Relative throughput of all systems, with respect to *ext4*, over a 50 GiB dataset under zipf99 and uniform data distributions.



(b) Relative throughput of all systems, with respect to *ext4*, over a 200 GiB dataset under zipf99 and uniform data distributions.



(c) Relative throughput of all systems, with respect to *ext4*, over a 400 GiB dataset under zipf99 and uniform data distributions.

Figure 6: Relative throughput of *ext4*, *OpenCAS*, *BushStore*, *PrismDB*, and *Keigo* for YCSB workloads (A to F) under distinct data distributions and dataset sizes. The absolute throughput value (in kops/s) of *ext4* is shown above the bar.

Baselines. We evaluate and compare Keigo over two groups of systems: *general-purpose storage systems*, namely ext4 and OpenCAS [39]; and state-of-the-art *LSMs designed for heterogeneous storage devices*, namely BushStore [44] and PrismDB [40]. ***ext4*** corresponds to an ext4 file system mounted over NVMe SSD. ***OpenCAS*** is an in-kernel Linux module that provides transparent and persistent caching to applications. We configured OpenCAS to use the NVMe SSD as main storage (*capacity device*) and NVMM as persistent cache following *write-back* policy – writes are submitted to the cache and acknowledged to the application before being written to the NVMe; periodically, these writes are flushed to the capacity device. ***BushStore*** is a LSM KVS that organizes L_0 and L_1 in a B+Tree structure and stores them in NVMM, while the remainder levels are stored on NVMe SSD. ***PrismDB*** is a LSM KVS that stores hot data items (organized in slab files) and the C_{log} on NVMM, and cold data items (in SST files) on NVMe. The system also uses a multi-tiered compaction scheme to minimize I/O stalls. As for ***Keigo***, unless stated otherwise, NVMM handles C_{log} and L_0 to L_2, while the remainder levels are placed on NVMe SSD, as generated by the offline concurrency profiler. Leftover NVMM space (≈100 GiB) is allocated to the persistent cache. The thread pools for managing caching and migrations are configured with 16 threads.

Experimental setup. The ext4, OpenCAS, and Keigo experiments were conducted over RocksDB configured with two C_m of 128 MiB each, a 1 GiB block cache, and a thread pool of 4 threads for internal operations, including 1 flushing thread. For optimal performance, BushStore was set with 64 threads for background work, and PrismDB with 1 background thread per client. All experiments (except §5.2) use 8 client threads. The data loading phase is single-threaded using a uniform distribution with 16B keys and 1024B values.

5.1 Varying dataset sizes

To understand how each system manages data across the storage hierarchy, we compare their performance under different dataset sizes, which can fit entirely (50 GiB) or only partially (200 GiB and 400 GiB) in the faster storage device (*i.e.,* NVMM). Experiments were conducted using YCSB workloads A to F. Figures 6a, 6b, and 6c depict the relative throughput of all systems with respect to ext4 for 50 GiB, 200 GiB, and 400 GiB datasets, respectively.

Write-intensive workloads. Under write-intensive workloads (A, F), Keigo outperforms all systems. For datasets that do not fit entirely on NVMM, Keigo improves throughput up to 2.2×, 1.7×, 1.6×, and 1.4× over ext4, OpenCAS, BushStore, and PrismDB. The reason behind this performance difference is that in ext4, all requests are handled by the NVMe SSD since it is the only device that can accommodate all datasets; OpenCAS caches requests in NVMM but the choice of which items should be cached is agnostic to the KVS, not considering their level or priority. The closest competitors are BushStore and PrismDB, which similarly to Keigo, handle the critical data path (*i.e.,* C_{log}, L_0, L_1) on NVMM. Keigo's performance difference becomes more pronounced as the dataset size increases. In Keigo, L_2 reads and writes are serviced by NVMM, while other solutions use the NVMe SSD.

Read-intensive workloads. Under read-intensive workloads (B, C, D), Keigo significantly outperforms all systems, especially under datasets that do not fit on NVMM. Keigo show performance improvements of up to 10×, 1.7×, 18×, and 15× over ext4, OpenCAS, BushStore, and PrismDB, a direct result of the persistent caching mechanism (§4.3). In these experiments, ext4 and OpenCAS cache hot data items in the OS page cache and NVMM, being particularly noticeable under the zipf99 distribution. For small datasets,

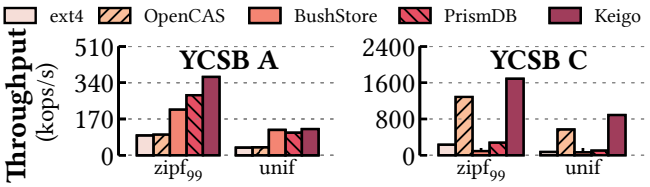


Figure 7: Performance under direct I/O for YCSB A and C.

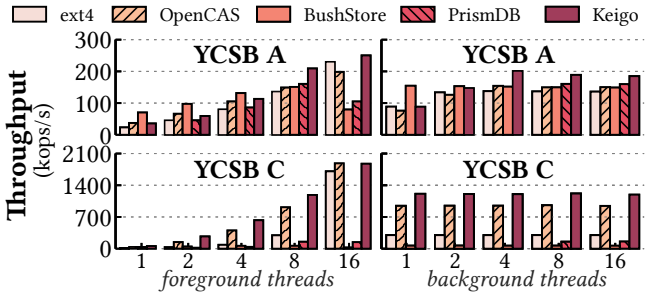


Figure 8: Performance of all systems for YCSB A and C with increasing number of foreground and background threads.

OpenCAS demonstrates similar performance to Keigo, as it caches the entire dataset on NVMM. On the other hand, BushStore and PrismDB experience poor performance due to their inability to cache hot data items either on NVMM or the OS page cache, leading to the majority of requests being serviced by NVMe SSD. Under scan-intensive workloads (E), Keigo improves performance up to 4× over general-purpose systems and 16× over the LSM KVS.

Direct I/O. In some production environments, applications use direct I/O to access their data, bypassing the OS page cache. As such, we assess the performance of each system when the OS page cache is disabled (`O_DIRECT`) under a 200 GiB dataset, depicted in Figure 7. Keigo, BushStore, and PrismDB demonstrate similar performance as with the OS page cache enabled, as they store the critical data path on NVMM with synchronous I/O to reduce CPU utilization by eliminating copies from the OS cache to the application buffer. For YCSB A, Keigo overcomes ext4 and OpenCAS up to 4×, as write operations are now directly submitted to the NVMe SSD. For YCSB C, Keigo overcomes ext4 by up to 12×; OpenCAS exhibits similar performance as with the OS page cache enabled, since hot data items are eventually promoted to NVMM.

5.2 Varying concurrency levels

Figure 8 depicts the performance of all systems under varying concurrency levels. Experiments were conducted for write-intensive (A) and read-intensive (C) workloads using a 200 GiB dataset under a zipf80 distribution with an increasing number of foreground and background threads, ranging from 1 to 16. For the background experiments, all systems were configured with 8 client threads. We were unable to configure PrismDB for 1 to 4 background threads due to its requirement of having at least 1 thread per client.

Foreground concurrency. Under the YCSB A workload, ext4, OpenCAS, and Keigo scale with the number of foreground workers.

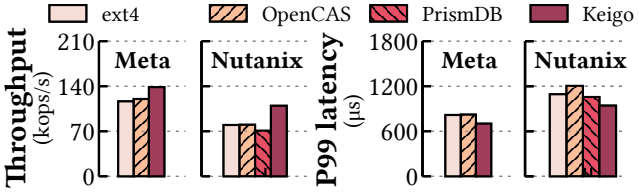


Figure 9: Performance of *ext4*, *OpenCAS*, *PrismDB*, and *Keigo* under Meta and Nutanix production workloads.

For a small number of workers (1 to 4), BushStore exhibits the best performance due to its B+Tree design under L_0 and L_1 LSM levels. Interestingly, contrary to the other systems, both BushStore and PrismDB performance peaks at 8 client threads, and experience a performance decrease of 47% and 34% under 16 threads. The reasons behind this are that: for BushStore, since NVMM only stores the lower levels of the tree, the majority of requests are serviced by the NVMe SSD device; for PrismDB, since it requires at least 1 background thread per client, the number of active writers in the system far exceeds the NVMM's limit (*i.e.,* 4), causing the write concurrency problem observed in §3. For the read-intensive workload, we draw similar conclusions as in §5.1.

Background concurrency. Keigo was configured with different placement schemes generated by the concurrency profiler. Specifically, with up to 2 background threads, Keigo stored in NVMM all files up to L_3, while the other levels were placed on NVMe due to space constraints; for 8 and 16 threads, Keigo stored in NVMM all files up to L_1, as the device's supported write parallelism would be exceeded beyond that level. Under YCSB A, for a small number of threads, since ext4, OpenCAS, and Keigo use native RocksDB, all systems experience write stalls, as discussed in §3.3. Beyond 4 threads, Keigo achieves the best performance across all systems. As for YCSB C, since it is a read-only workload (absent of compactions), all systems show stable performance with increasing number of workers. Interestingly, while Keigo is configured with different placement schemes, its performance remains stable due to the persistent caching mechanism, achieving a performance increase of 4×, 1.25×, 18×, and 7.5× over ext4, OpenCAS, BushStore, and PrismDB.

5.3 Production workloads

We now evaluate how each system performs under production workloads from Meta [7] and Nutanix [28]. Figure 9 captures the throughput and tail latency of all systems.

Meta workloads. We use the *prefix_dist* production workload from Meta [7], a read-dominated workload that combines read, write, and scan operations at a ratio of 83:14:3, uses varying key-value pair sizes, and exhibits realistic key hotness patterns. Experiments ran over a 50M key-value pairs dataset using 8 client threads. Due to its placement scheme and persistent cache, Keigo outperforms all baselines, increasing throughput by 1.2× over ext4 and 1.15× over OpenCAS. We were unable to run BushStore and PrismDB under this workload, as they crash due to the varying key sizes.

Nutanix workloads. The Nutanix workload is a write-intensive workload that combines read, write, and scan operations at a ratio of 40:58:2. The items requested during the execution also vary

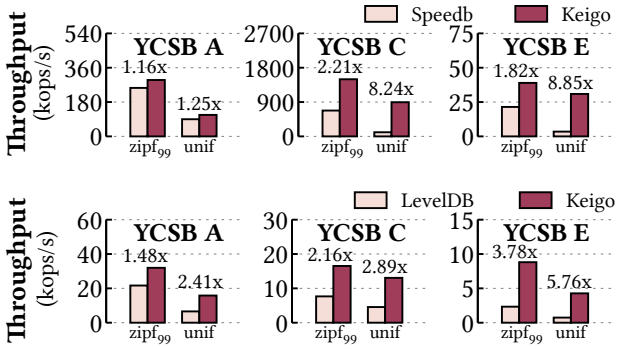


Figure 10: Throughput of Speedb and LevelDB with Keigo.

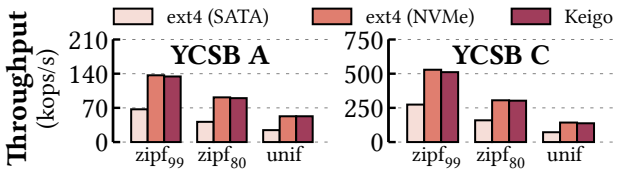


Figure 11: Performance of Keigo under a block-addressable storage hierarchy for YCSB A and C.

in size, ranging from 100B to 4 KiB, with a median size of 400B. Experiments were conducted over a 500M key-value pairs dataset. Keigo outperforms all baselines, achieving a throughput increase of up to 1.6× over other systems. Similarly to the Meta workloads, we were unable to successfully run BushStore.

5.4 Portability

We now demonstrate the portability of Keigo by integrating it with Speedb [43] (81 LoC) and LevelDB [18] (40 LoC). Figure 10 compares the performance of the base systems configured with the ext4 setup and their corresponding Keigo-enabled versions. Experiments were conducted in a 200 GiB dataset with write- (A), read- (C), and scan-intensive (E) workloads under different data distributions. Speedb experiments were conducted using 8 client threads, while LevelDB's were sequential due to the lack of concurrency support.

Keigo improves the performance of both systems across all workloads due to its data placement and persistent caching optimizations. Speedb performance is improved up to 8× for read- and scan-intensive workloads and up to 1.33× for write-intensive workloads. As for LevelDB, Keigo shows a performance increase of up to 2.4× for YCSB A, 2.89× for YCSB C, and 5.76× for YCSB E.

5.5 Sensitivity analysis

We now evaluate the performance of distinct features of Keigo.

Block-addressable storage hierarchy. We first study the performance of Keigo under a block-addressable storage hierarchy. The experiments were conducted for YCSB A and C workloads using a 200 GiB dataset with 16 client threads across three setups: ext4 mounted on the SATA SSD, ext4 mounted on the NVMe SSD, and Keigo using both NVMe and SATA SSDs. For Keigo, the NVMe SSD is limited to 100 GiB to simulate a scenario where the dataset exceeds the capacity of the faster tier. As generated by the offline concurrency profiler, the NVMe SSD handles C_{log} and L_0 to L_3, while the remainder levels are placed on the SATA SSD. Leftover NVMe space (≈70 GiB) is allocated to the persistent cache. To minimize caching effects, the OS page cache was disabled for all systems.

Figure 11 depicts the performance of all setups over different data distributions. Under an NVMe-SATA SSD hierarchy, Keigo improves RocksDB performance by up to 2.2× for write-heavy workloads and 1.9× for read-heavy workloads compared to storing all LSM components on ext4 backed by the SATA SSD. When compared to a scenario where all LSM components reside on the NVMe SSD device, Keigo achieves similar performance while placing only a small portion of the LSM (≈30 GiB, 15%) in the faster, and more expensive, tier. These results highlight Keigo's adaptability to different storage hierarchies and reinforce the broader applicability of its core principles (§4) beyond NVMM-based storage hierarchies.

Multiple storage tiers. We now analyze the performance of Keigo under different storage hierarchies: a single tier composed of a NVMM; a tier using NVMM and NVMe SSD devices; and the combination of NVMM, NVMe SSD, and SATA SSD. Adding more devices to the hierarchy allows the system to handle larger datasets. For these experiments, NVMe capacity was limited to 800 GiB, and the NVMM and NVMe's caches were configured to 80 GiB and 160 GiB, respectively. Figure 12 depicts the performance of the different combinations for write- (A) and read-intensive (C) workloads over increasing dataset sizes, ranging from 50 GiB to 1.6 TiB. All storage hierarchies achieve the best performance for smaller datasets, as the majority of requests are serviced by the faster tiers. Interestingly, the NVMM + NVMe SSD and NVMM + NVMe SSD + SATA SSD hierarchies show similar performance under the same dataset, never exceeding a relative difference of 5%. This is due to Keigo's caching and migration mechanisms, which ensure sustained performance even when adding slower storage devices to the hierarchy.

Impact of the persistent cache. We now analyze the performance impact imposed by Keigo's persistent cache. We ran a read-intensive workload (C) for a 400 GiB dataset stored over a storage hierarchy composed of NVMM, NVMe SSD, and SATA SSD. We considered three setups: when caching is disabled, when NVMM cache is enabled, and when both caches (NVMM and NVMe SSD) are enabled. The storage capacity of each device was limited to 100 GiB, 200 GiB, and 400 GiB, respectively. The NVMM and NVMe SSD's caches were configured to 70 GiB and 50 GiB, respectively. Due to space constraints, the plots for these experiments are omitted.

When caches are disabled, Keigo's performance ranges between 126 kops/s and 815 kops/s for uniform and zipf99 distributions. When the NVMM cache is enabled, its throughput increases by 4.5× for uniform and 1.7× for zipf99. Results also show that, under this dataset, the NVMe SSD cache has a negligible impact on performance since most requests are serviced by the NVMM cache, either due to accesses over the hot SST files originally placed on NVMM or the hot files copied from the NVMe SSD device.

Caching and migration agressiveness. As discussed in §4.2 and §4.3, Keigo automatically controls the number of threads migrating and caching data across devices. Figure 13 shows the performance of such mechanism for YCSB A and C workloads with an

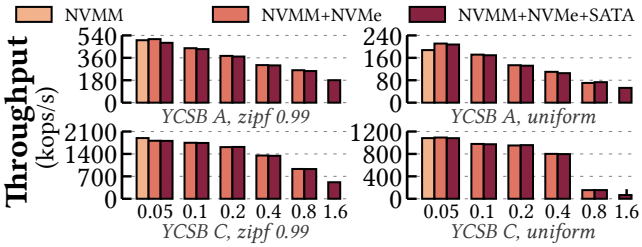


Figure 12: Performance with different tiers and dataset sizes.

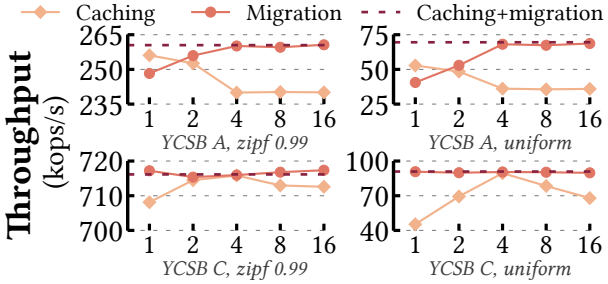


Figure 13: Performance of Keigo's automatic thread control for caching and migrations.

800 GiB dataset. We compare three different setups: *Caching* manually changes the number of caching threads from 1 to 16, while letting Keigo choose the number of migration threads; *Migration* varies the number of migration threads and lets Keigo define the number of caching threads; and *Caching+migration* lets Keigo automatically control the number of both thread types.

For write-intensive workloads, increasing the number of caching threads decreases the KVS throughput, as parallelizing the caching process (combined with ongoing writes from C_{log}, flush, and compactions) causes the number of active writers to exceed NVMM's concurrent writers limit. For read-intensive workloads, since there are no active writers in the system, Keigo scales up to 4 caching threads. As for migrations, since these are performed between the NVMe and SATA SSDs, the performance improves up to 16 threads. When both caching and migrations operate with the automatic thread control mechanism, Keigo achieves the best performance across all experiments. This is because Keigo continuously monitors the number of active writers in each storage device and automatically adjusts the number of concurrent caching and migrations.

6 RELATED WORK

This section describes prior work and places our work in context.

General-purpose hierarchical storage. OpenCAS [39] is a generic block-layer caching mechanism that enables using a fast device as a cache of a slower device. P2Cache [29], FirstResponder [41], and SPFS [45] are in-kernel caching mechanisms that enhance legacy file systems by using NVMM to absorb frequent writes. Orthus [47] introduces a non-hierarchical caching strategy that redirects requests based on device load. Strata [26], Ziggurat [53], and TPFS [54] are file systems that tier data across DRAM, NVMM, and SSD according to the applications' access patterns and consistency requirements. These systems, however, are agnostic of applications' internal I/O logic. For LSM KVS, this means that requests with different priorities and storage performance costs are treated in the same manner across the storage hierarchy, impacting the KVS' end performance as observed in §3 and §5. In contrast, Keigo is an LSM-aware middleware that places LSM components in the storage device that best suits their workload patterns.

KVS with hierarchical storage support. Mutant [51] places SST files based on their popularity over cloud-based storage while enforcing storage cost SLOs. SpanDB [9] uses high-performance SSDs via SPDK to store the C_{log} and lower levels of the LSM tree. PrismDB [40] proposes a multi-tier compaction scheme that spawns across a hierarchy of heterogeneous storage devices. MatrixKV [50] proposes a new matrix-like data structure that manages L_0 and is placed on NVMM. BushStore [44] replaces traditional SSTs with B+Trees, placing L_0 and L_1 in byte-addressable storage, and the rest of the LSM in a block-based device. WaLSM [10] actively profiles data freshness and access frequency to accurately migrate cold data from NVMM to SSD. Prism [42] is a KVS specifically designed for DRAM-NVMM-SSD storage hierarchies, that follows a key-value separation model, where keys are placed on NVMM and values are placed on SSD (first absorbed by NVMM to minimize write latency and later migrated to SSD). Prism introduces techniques that minimize thread synchronization over *"wide"* storage hierarchies composed of multiple similar devices operating in parallel. Replacing LSMs used in production with these systems is not trivial, posing significant implementation efforts. Keigo is independent of specific LSM implementations or storage devices, enabling portability and performance improvements with minimal code changes.

Unlike previous works, Keigo is the first solution to fully leverage heterogeneous storage hierarchies by optimizing LSM I/O workflows for each device's *parallelism*, *bandwidth*, and *capacity*. By doing so, Keigo significantly improves the performance of widely used LSM-based KVS systems under read and write workloads.

7 CONCLUSION

We presented Keigo, a novel storage middleware that accelerates the performance of LSM KVS using a heterogeneous storage hierarchy. Contrary to prior work, Keigo consolidates the inherent properties of LSM with the parallelism, I/O bandwidth, and capacity of different storage devices. Our extensive evaluation shows significant performance improvements over general-purpose systems and specialized KVS with native support for heterogeneous storage.

ACKNOWLEDGMENTS

This work was conducted when Zhongjie Wu was a student at McGill University. We thank our shepherd and the anonymous reviewers for their insightful comments and feedback. This work was partially supported by: the NSERC Discovery Grant RGPIN-2021-02662 (Changjun Zhou and Zhongjie Wu); the European Regional Development Fund through the NORTE 2030 Regional Programme under Portugal 2030, within project BCDSM (ref.14436, NORTE2030-FEDER-00584600) (João Paulo); and by the Portuguese funding agency (FCT) within the PhD grant 2024.02442.BD and project LA/P/0063/2020 (DOI 10.54499/LA/P/0063/2020) (Rúben Adão).

REFERENCES

[1] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *27th ACM Symposium on Operating Systems Principles*. ACM, 353–369. https://doi.org/10.1145/3341301.3359656

[2] Alluxio. 2014. Alluxio/alluxio: Alluxio, data orchestration for analytics and machine learning in the cloud. https://github.com/Alluxio/alluxio. Last accessed Jun 12, 2025.

[3] Apple. 2018. apple/foundationdb: FoundationDB - the open source, distributed, transactional key-value store. https://github.com/apple/foundationdb. Last accessed Jun 12, 2025.

[4] Jens Axboe. 2005. axboe/fio: Flexible I/O Tester. https://github.com/axboe/fio. Last accessed Jun 12, 2025.

[5] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference*. USENIX Association, 753–766.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 753–768.

[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 209–223.

[8] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1087–1098. https://doi.org/10.1145/2882903.2904441

[9] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies*. USENIX Association, 17–32.

[10] Lixiang Chen, Ruihao Chen, Chengcheng Yang, Yuxing Han, Rong Zhang, Xuan Zhou, Peiquan Jin, and Weining Qian. 2023. Workload-Aware Log-Structured Merge Key-Value Store for NVM-SSD Hybrid Storage. In *39th IEEE International Conference on Data Engineering*. IEEE, 2207–2219. https://doi.org/10.1109/ICDE55515.2023.00171

[11] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel D. G. Lee, Jin-Hyeok Choi, and Jaeheon Jeong. 2018. A flash memory controller for 15μs ultra-low-latency SSD using high-speed 3D NAND flash with 3μs read time. In *2018 IEEE International Solid-State Circuits Conference*. IEEE, 338–340. https://doi.org/10.1109/ISSCC.2018.8310322

[12] CockroachDB. 2015. cockroachdb/cockroach: CockroachDB – the cloud native distributed SQL database. https://github.com/cockroachdb/cockroach. Last accessed Jun 12, 2025.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *1st ACM Symposium on Cloud Computing*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[14] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *8th Biennial Conference on Innovative Data Systems Research*.

[15] Ulrich Drepper. 2007. What every programmer should know about memory. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf. Last accessed Jun 12, 2025.

[16] Facebook. 2016. Dragon: A distributed graph query engine. https://engineering.fb.com/2016/03/18/data-infrastructure/dragon-a-distributed-graph-query-engine/. Last accessed Jun 12, 2025.

[17] Alexandra Fedorova, Keith A. Smith, Keith Bostic, Susan LoVerso, Michael Cahill, and Alex Gorrod. 2022. Writes Hurt: Lessons in Cache Design for Optane NVRAM. In *Proceedings of the 13th Symposium on Cloud Computing*. ACM, 110–125. https://doi.org/10.1145/3542929.3563461

[18] Google. 2024. google/leveldb: LevelDB v1.23 . https://github.com/google/leveldb. Last accessed Jun 12, 2025.

[19] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proceedings of the VLDB Endowment* 14, 4 (2020), 626–639. https://doi.org/10.14778/3436905.3436921

[20] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proceedinds of the IEEE* 105, 9 (2017), 1822–1833. https://doi.org/10.1109/JPROC.2017.2731776

[21] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *12th European Conference on Computer Systems*. ACM, 127–144. https://doi.org/10.1145/3064176.3064187

[22] Intel. 2025. Persistent Memory Development Kit. https://pmem.io/pmdk/. Last accessed Jun 12, 2025.

[23] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference*. USENIX Association, 993–1005.

[24] Riduan Khaddam-Aljameh, Milos Stanisavljevic, Jordi Fornt Mas, Geethan Karunaratne, Matthias Brändli, Feng Liu, Abhairaj Singh, Silvia M. Müller, Urs Egger, Anastasios Petropoulos, Theodore Antonakopoulos, Kevin Brew, Samuel Choi, Injo Ok, Fee Li Lie, Nicole Saulnier, Victor Chan, Ishtiaq Ahsan, Vijay Narayanan, S. R. Nandakumar, Manuel Le Gallo, Pier Andrea Francese, Abu Sebastian, and Evangelos Eleftheriou. 2022. HERMES-Core - A 1.59-TOPS/mm^2 PCM on 14-nm CMOS In-Memory Compute Core Using 300-ps/LSB Linearized CCO-Based ADCs. *IEEE J. Solid State Circuits* 57, 4 (2022), 1027–1038. https://doi.org/10.1109/JSSC.2022.3140414

[25] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. 2023. NVM: Is it Not Very Meaningful for Databases? *Proceedings of the VLDB Endowment* 16, 10 (2023), 2444–2457. https://doi.org/10.14778/3603581.3603586

[26] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *26th Symposium on Operating Systems Principles*. ACM, 460–477. https://doi.org/10.1145/3132747.3132770

[27] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010). https://doi.org/10.1145/1773912.1773922

[28] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *27th ACM Symposium on Operating Systems Principles*. ACM, 447–461. https://doi.org/10.1145/3341301.3359628

[29] Zhen Lin, Lingfeng Xiang, Jia Rao, and Hui Lu. 2023. P2CACHE: Exploring Tiered Memory for In-Kernel File Systems Caching. In *2023 USENIX Annual Technical Conference*. USENIX Association, 801–815.

[30] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/S00778-019-00555-Y

[31] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *13th European Conference on Computer Systems*. ACM, 8:1–8:18. https://doi.org/10.1145/3190508.3190526

[32] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. 2022. PAIO: General, Portable I/O Optimizations With Minor Application Modifications. In *20th USENIX Conference on File and Storage Technologies*. USENIX Association, 413–428.

[33] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 3217–3230. https://doi.org/10.14778/3415478.3415546

[34] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *28th Symposium on Operating Systems Principles*. ACM, 243–262. https://doi.org/10.1145/3477132.3483568

[35] Meta. 2023. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide Last accessed Jun 12, 2025.

[36] Meta. 2025. facebook/rocksdb: RocksDB v6.11. https://github.com/facebook/rocksdb/tree/v6.11.6 Last accessed Jun 12, 2025.

[37] MongoDB. 2013. mongodb/mongo: The MongoDB database. https://github.com/mongodb/mongo. Last accessed Jun 12, 2025.

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385. https://doi.org/10.1007/S002360050048

[39] OpenCAS. 2025. *Open Cache Acceleration Software*. Last accessed Jun 12, 2025.

[40] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. 2023. Efficient Compactions between Storage Tiers with PrismDB. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 179–193. https://doi.org/10.1145/3582016.3582052

[41] Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh. 2021. First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage. In *2021 USENIX Annual Technical Conference*. USENIX Association, 839–853.

[42] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 588–602. https://doi.org/10.1145/3575693.3575722

[43] Speedb. 2025. speedb-io/speedb: Speedb v2.6.0. https://github.com/speedb-io/speedb/tree/speedb/v2.6.0. Last accessed Jun 12, 2025.

[44] Zhenghao Wang, Lidan Shou, Ke Chen, and Xuan Zhou. 2024. BushStore: Efficient B+Tree Group Indexing for LSM-Tree in Non-Volatile Memory. In *40th IEEE International Conference on Data Engineering*. IEEE, 4127–4139. https://doi.org/10.1109/ICDE60146.2024.00316

[45] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H. Noh, and Beomseok Nam. 2023. On Stacking a Persistent Memory File System on Legacy File Systems. In *21st USENIX Conference on File and Storage Technologies*. USENIX Association, 281–296.

[46] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association.

[47] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies*. USENIX Association, 307–323.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 169–182.

[49] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Split-Level I/O Scheduling. In *25th Symposium on Operating Systems Principles*. ACM, 474–489. https://doi.org/10.1145/2815400.2815421

[50] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference*. USENIX Association, 17–31.

[51] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 162–173. https://doi.org/10.1145/3267809.3267846

[52] Jinghuan Yu, Sam H. Noh, Young ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies*. USENIX Association, 65–80.

[53] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies*. USENIX Association, 207–219.

[54] Shengan Zheng, Morteza Hoseinzadeh, Steven Swanson, and Linpeng Huang. 2023. TPFS: A High-Performance Tiered File System for Persistent Memories and Disks. *ACM Transactions on Storage* 19, 2, Article 20 (mar 2023). https://doi.org/10.1145/3580280