Vigil-KV: Hardware-Software Co-Design to Integrate Strong Latency Determinism into Log-Structured Merge Key-Value Stores

Miryeong Kwon, Seungjun Lee, and Hyunkyu Choi, *KAIST;* Jooyoung Hwang, *Samsung Electronics Co., Ltd.;* Myoungsoo Jung, *KAIST*

https://www.usenix.org/conference/atc22/presentation/kwon

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Vigil-KV: Hardware-Software Co-Design to Integrate Strong Latency Determinism into Log-Structured Merge Key-Value Stores

Miryeong Kwon[1], Seungjun Lee[1], Hyunkyu Choi[1], Jooyoung Hwang[2], Myoungsoo Jung[1]
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)[1], Samsung[2]
http://camelab.org

Abstract

We propose *Vigil-KV*, a hardware and software co-designed framework that eliminates long-tail latency almost perfectly by introducing strong latency determinism. To make Get latency deterministic, Vigil-KV first enables a predictable latency mode (PLM) interface on a real datacenter-scale NVMe SSD, having knowledge about the nature of the underlying flash technologies. Vigil-KV at the system-level then hides the non-deterministic time window (associated with SSD's internal tasks and/or write services) by internally scheduling the different device states of PLM across multiple physical functions. Vigil-KV further schedules compaction/flush operations and client requests being aware of PLM's restrictions thereby integrating strong latency determinism into LSM KVs. We implement Vigil-KV upon a 1.92TB NVMe SSD prototype and Linux 4.19.91, but other LSM KVs can adopt its concept. We evaluate diverse Facebook and Yahoo scenarios with Vigil-KV, and the results show that Vigil-KV can reduce the tail latency of a baseline KV system by 3.19× while reducing the average latency by 34%, on average.

1 Introduction

Log-structured Merge Key-Value stores (LSM KVs) such as RocksDB [1] and LevelDB [2] are widely adopted in diverse computing domains to handle large-scale data thanks to their simplicity, scalability, and high-performance [3–10]. LSM KVs are also used in many production environments to offer large-scale storage whose capacity is beyond main memory subsystems to latency-sensitive applications. For example, Facebook uses RocksDB as the storage engine of an SQL database, which is used for social graph processing [11–14]. This type of application considers the query latency (e.g., Get) of each social action (e.g., view profile, list friends, etc.) as a first-class citizen. In particular, managing long-tail latency of reads (and latency consistency) is a matter of meeting diverse user demands and service-level agreements (SLA) [15–17].

However, we observe that the long-tail read latency of the Facebook scenario is 10.4× worse than normal read operations, making the user experience inconsistent. The main contributor to this long-tail latency is device-level SSD latency, not software or operating system (OS); the execution times of all the software, including storage stack and user application, account for only 13% of the long-tail latency (99.9^{th} percentile). We will give a detailed analysis of this in Section 3.1. The long-tail latency mainly comes from I/O interferences caused by two different levels of internal tasks: i) *LSM KV's internal tasks* and ii) *SSD's internal tasks*. LSM KVs periodically perform internal tasks such as compaction and flushing for their persistence and effectiveness [18–21]. The compaction merges data from the lower to a higher level of their LSM tree, whereas the flush writes the in-memory buffer back to the underlying storage in securing more space to buffer and making the buffered data persistent. Since the write operations of these internal tasks exhibit long latency and often stall all incoming requests, many prior studies (e.g., TRIAD [22], PebblesDB [23], and SILK [24]) reschedule additional writes of the internal tasks and serve them at future idle times. These LSM KVs would reduce the latency inconsistency imposed by the internal tasks to some extent, but we observe that they lead to serious side-effects, which deteriorate read services and increase memory footprints significantly (cf. Section 3.2).

Even with an ideal case of abolishing all the LSM KV's internal tasks, the long-tail read latency cannot be eliminated because of SSD's internal tasks such as DRAM caching/flushing [25–30], garbage collections [31–39], and read-reclaiming [40–44]. For example, even in cases where LSM KV solely reads the underlying SSD at a certain period, it exhibits long latency on the reads since the SSD internally flushes the buffered/cached data to its backend storage. Similarly, at any given time, a garbage collection or read-reclaiming can introduce a set of reads and writes, which also prevent the incoming requests from being serviced. Note that these internal tasks are scheduled by the underlying SSD firmware, which makes the read latency behaviors non-deterministic at the user-level and increases the latency significantly (cf. Section 2.2).

In this work, we propose *Vigil-KV*, a hardware and soft-

ware co-design to eliminate the long-tail latency of LSM KVs and make their read services consistently deterministic. Vigil-KV hardware offers a scheduling interface to remove SSD's internal tasks, whereas its software is designed toward eliminating the overhead imposed by LSM KV's internal tasks without delaying compaction or flushing in-memory buffer at idle times. To this end, we advocate a *predictable latency mode* (PLM) interface, which is recently added to the standard NVMe protocol [45]. We enable the brand-new interface on a high-performance NVMe SSD and enforce the read latency deterministic on a specific time window. Obviously, PLM cannot deliver the latency consistency indefinitely since SSD's internal tasks are essential to managing the reliability and persistence of the backend's storage media. For the host's finer PLM scheduling, Vigil-KV hardware also implements NVMe's NVM set features by internally partitioning the storage volume into multiple functions.

While PLM has great potential to eliminate the long-tail latency of LSM KVs by having a closer collaboration between a host and storage, there are several constraints that have not been analyzed in the literature yet. Specifically, PLM relies two essential scheduling components, *deterministic window* (DTWIN) and *non-deterministic window* (NDWIN). DTWIN is the time window to offer predictable latency, whereas NDWIN is not. This work reveals three important characteristics of PLM, which should be considered when the host communicates with the underlying SSD to achieve the latency consistency: i) *write-free on DTWIN* ii) *fair-scheduling for PLM windows*, iii) *device lockdown constraint*. First, DTWIN can be guaranteed only if there is no write request in a DTWIN period. The reason behind this DTWIN's write-free constraint is that, even though PLM supports the latency consistency by removing SSD's internal tasks at DTWIN, it cannot completely eliminate the stalls caused by online write requests coming from clients. Second, as SSD's internal tasks should be performed at some point, the longest-serving time of DTWIN is determined at design time, and NDWIN should be preserved and appropriately scheduled before jumping in DTWIN. Lastly, the host curbs I/O requests when the underlying SSD transits from NDWIN to DTWIN. This is because the transition requires a make-ready time, which must not be interrupted by any other I/O activities (i.e., device lockdown).

Based on the restrictions that we observe, the software part of Vigil-KV classifies the requests of LSM KVs at runtime and carefully assigns them to appropriate PLM time windows through our device state scheduling. This device state and request scheduling can make the latency of client-side I/O requests deterministic and have no long-tail all the time. To this end, we introduce a PLM driver atop Vigil-KV hardware, which manages all the device states across different NVM sets but makes them visible as a single storage volume. This driver makes sure that there are always $n-1$ NVM sets having DTWIN (where n is the total number of data NVM sets) while allowing an NVM set to schedule SSD's internal task via NDWIN. During the device state management, it also takes into account the fair-scheduling and lockdown constraints such that a kernel-level scheduler can focus on assigning the I/O requests based on the condition of given PLM time windows. Specifically, Vigil-KV's kernel-level scheduler packs all I/O activities coming from LSM KV's internal tasks into NDWIN (scheduled by the PLM driver), which takes the overhead of all the internal tasks off the critical path in I/O services. In addition, it makes all the incoming read requests (heading to the NDWIN-scheduled set) non-blocking, inspired by a novel memory/storage array-level technique [46–49]. The kernel-level scheduler detects the read requests targeting an NVM set (configured with NDWIN) and directly serves the corresponding data without touching it at all. Since there are $n-1$ NVM sets with DTWIN at an any given time (invisible to the host clients), the scheduler can collect data from them within the deterministic time window and reconstruct the requested data (with the help of the PLM driver) thereby making the target SSD latency consistent constantly.

Even though the Vigil-KV driver and thread can put all the internal tasks into NDWIN and isolate them from the client reads, they unfortunately fail to meet the write-free constraint. This is because of additional writes for providing atomicity and durability at the system-level (e.g., journaling). Since these writes can interfere with the reads on DTWIN, we dedicate an NVM set for the metadata management dealing with the write-ahead log (WAL) and file system journaling. To this end, we have a minor modification of RocksDB (but other LSM KVs can adopt its concept) to give a different priority to each process based on their nature of I/O activities. This technique can address the write-free constraint and make the target SSD be in all DTWIN for client requests consistently.

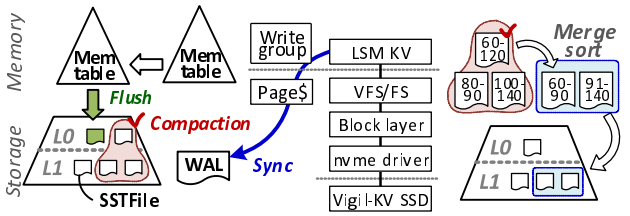
We prototype Vigil-KV hardware on a 1.92TB Datacenter-scale NVMe SSD, while implementing Vigil-KV software using Linux 4.19.91 and RocksDB 6.23.0. To the best of our knowledge, this is the first paper that implements the PLM interface in a real SSD and makes the read latency of LSM KVs deterministic in a hardware-software co-design manner. We evaluate six Facebook and Yahoo scenarios, and the results show that Vigil-KV can reduce the tail latency of a baseline KV system by 3.19× while reducing the average latency on Get services by 34%, on average.

2 Preliminaries

We will explain RocksDB as representative of LSM KVs in this section. We will also explain the internal tasks of LSM KV and SSD in detail and analyze the challenges imposed by those two different levels of internal tasks.

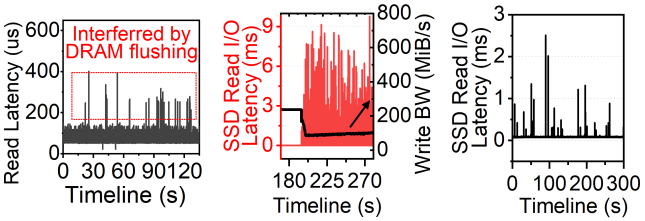
2.1 Log-Structured Merge KV Stores

Figure 1a explains the major data structure and corresponding operations of RocksDB. RocksDB maintains all information



(a) Structure. (b) Internal tasks. (c) Example.

Figure 1: Log-Structured Merge KV Stores.



(a) DRAM flush. (b) Garbage collection. (c) Read reclaim.

Figure 2: SSD internal tasks.

in the log-structured merge (LSM) tree consisting of two separate structures, each of which is optimized to volatile memory and block storage. The in-memory data structure, called *Memtable*, holds data before turning their state into persistent in an unsorted manner. Memtables allow users/clients to quickly update by serving the requests from the memory (rather than storage). The storage data structure manages key and value (KV) pairs, which are managed in an immutable form of *sorted string table* files (SSTFiles). SSTFiles are maintained in hierarchical levels, each being denoted by L0 (level-0), L1 (level-1), ..., LK (level-K).

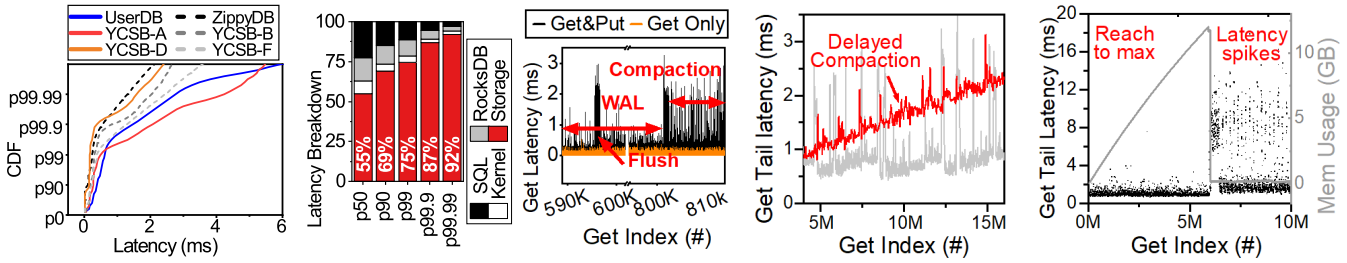
Client operations. RocksDB supports various query services such as Put (writes), Get (reads), Delete, and Scan. Since the majority of the queries are *Put* and *Get*, this work focuses on those two operations. Users' Put requests are inserted to a Memtable as a KV pair by RocksDB if the Memtable is mutable, meaning that it has available room to update. In default, RocksDB maintains two Memtables, each taking 64MB spaces, which are the same as the size of logfiles; we will explain this in detail with LSM KV's internal tasks (i.e., flush and compaction) shortly. If there is no available space in a Memtable, RocksDB locks down and changes its state from mutable to immutable, which does not allow further updates. RocksDB then places another Memtable for the next Put requests while writing the data of the immutable Memtable to L0 by converting the Memtable to an SSTFile in the background. Since it is important to secure Memtable(s) in memory as soon as possible, turning a Memtable into L0 is performed in an unsorted and out-of-order manner. Thus, L0 can contain multiple SSTFiles associated with the same key. Later, the SSTFiles at L0 are migrated into a lower level of storage space (L1) by LSM KV's internal tasks.

On the other hand, Get requests accompany a series of reads the value associated with a given key. RocksDB first searches the key in Memtables and serves the value if there is. In cases where it fails to find the key in the Memtables, RocksDB scans all the SSTFiles residing in L0 and searches for the key. This is because the files are stored in an unsorted way, and it can be possible for L0 to have multiple SSTFiles corresponding to the given key. If RocksDB cannot find the key at L0, it goes L1 and searches again. Since L1's SSTFiles are compacted from L0, each file contains a unique key, making RocksDB faster to search the target KV pair. Note that the unsorted data structure of L0 allows RocksDB to quickly secure in-memory buffers, thereby preventing Put against stalls, but it introduces many storage accesses (reads) on Get services.

Internal tasks. Figure 1b illustrates the detailed procedure of LSM KV's internal tasks and major software components associated with the tasks. While Memtables are well designed toward taking performance advantage of volatile memory media, their data can be lost when there is a power failure. To make the KV pairs in the buffer persistent and durable, RocksDB writes the KV pair as a form of logfiles to a designated area in the underlying storage, called *write-ahead log* (WAL) before its Memtable update. Writing WAL (per request) is performed as a synchronous operation bypassing the page cache of the underlying file system for crash consistency control. Since it is a time-consuming task, RocksDB employs another internal buffer, called *write group* existing in front of Memtables. In the meantime, it checks the space utilization of Memtables and L0, and if there is no available space, Rocks DB enqueues *flush* and/or *compaction* tasks item to reclaim a Memtable and an L0 SSTFile, respectively. These items include an appropriate pointer for the space reclaiming, which is all performed by RocksDB's background threads. For a Memtable flush, the internal task checks all the keys in a Memtable, builds an SSTFile, and flushes the SSTFile to L0. In cases of an L0 compaction, its internal task selects a target SSTFile. Consider Figure 1c as an example, the SSTFile's key ranges from 60 to 120. The task also picks L1's SSTFiles whose keys are associated with the compaction target's keys (e.g., two L1 SSTFiles, each having 80∼90 and 100∼140, in the figure). It then performs a merge sort by checking up all entries of three SSTFiles and letting only the latest information remain, which generates a new L1's SSTFile. Lastly, RocksDB synchronously writes the new SSTFile and removes the three old SSTFiles from the underlying SSD.

2.2 SSD Internal Tasks and Challenges

Internal DRAM flush. Since flash writes are slower than its reads by order of magnitude, high-performance SSDs employ a large size of internal DRAM, and their firmware buffers the writes [25–27, 50]. For example, our baseline NVMe hardware has 3GB DRAM buffering/caching data. These buffered writes are periodically flushed to the storage backend with a specific access pattern in favor of increasing bandwidth. Thus, even though there is no write at all for a certain period,



(a) CDF. (b) Breakdown. (c) Time series.

Figure 3: Long tail analysis.

(a) Compaction. (b) Flush.

Figure 4: Limitations internal tasks.

draining the data (buffered previously) can interfere with incoming read operations. Figure 2a shows the read latency interfered by SSD's internal flush; we write a block (64MB) to SSD before the test and only issue 4KB-sized read requests (in sequential) without any writes for the test period. As shown in the figure, the baseline NVMe hardware suffers from massive latency spikes, which are higher than the typical latency by 7.75× at most, and its latency significantly fluctuates during the read-only time. This is because the writes introduced by the internal flush stall the reads until their service completes.

Note that, since the internal flushes are solely managed by the firmware, host software components cannot unfortunately control the latency consistency of reads. To remove the latency fluctuation analyzed above, it requires a tight collaboration between the host and firmware.

Garbage collection. Flash also has unique device-level characteristics such as erase-before-write and asymmetric I/O granularity for read/write and erase [51–54]. Because of this nature, SSD's firmware writes incoming data into a free block (erased in advance) instead of its actual location. While this address remapping (translation) for out-of-updates makes flash compatible with the existing block devices, it needs to perform a *garbage collection* (GC) if there is no free block [31, 38, 55, 56]. Since GCs are performed on the basis of a flash block containing hundreds of pages, the valid data residing in the target block(s) should be safely migrated into a new location of a block. This internal task introduces block erase operations even longer than the flash writes and many reads/writes for the migration. It exhibits long latency and stalls many incoming requests before completing the task. Figure 2b shows the read latency while performing GCs (from 195 sec). In this test, reads exhibit sustainable latency (16.2 *us*), but their latency sharply increases and reaches as high as 9.8 *ms* once GCs begin.

While these internal tasks significantly hamper the read performance, all their activities are essential to secure more available rooms for further requests and manage the reliability of the storage backend, which cannot be simply removed or scheduled by host-side software modules.

Read-reclaiming. Flash is very well optimized for read services at the low-level [57–60], but a read-only scenario can also introduce additional data migration and block erase operations in certain circumstances. Specifically, when one keeps reading out a set of pages in a block without an erase, it stresses the block even without any writes and affects all data residing in the block together. This *read disturbance* unfortunately increases error rates often beyond the coverage that parity-check codes (e.g., ECC [61–64] and LDPC [65–68]) can correct [69–72]. To address the read disturbance issue, the underlying firmware needs to periodically reset (erase) the block(s) being intensively touched over the past period. Once the firmware erases the block, its internal state returns back to the nominal state, such that the block can endure the stress imposed by subsequent reads again. To erase the block, it requires reading the existing data on the target and copying all of them to a new block. As shown in Figure 2c, this internal task, called *read reclaiming* [69], can deteriorate the read performance seriously. In this test, we intensively read a set of specific blocks four times as a precondition and read them again in a random I/O pattern. One can observe from the figure that the read latency affected by the read reclaiming reaches as high as 2.5 *ms*, which is 32× longer than the typical cases.

Even with the ideal situation that only utilizes the underlying SSDs as read-dedicated storage, this long-tail latency imposed by the read reclaiming are inevitable, and thus, it is necessary to devise new interface and firmware assistance to get them off the critical path in LSM KV's read services.

3 Motivation and Related Work

3.1 Long-tail Latency on Reads

Figure 3a shows the cumulative distribution function (CDF) of Get latency for diverse RockDB usage scenarios of Facebook [12, 73] and Yahoo [74]. In this evaluation, we use RocksDB 6.23 [1] on a baseline 1.92TB NVMe hardware that we will modify in Section 4.1 and use for all the remaining tests. This baseline employs 3GB internal DRAM and includes 64 TLC NAND flash (64 layers), which are connected to eight different channels. The detailed environment descriptions are the same as what we used for Section 7.

Significance of long-tail latency. Thanks to the low device-level latency of flash, the nominal performance trend of them is similar to each other; all their Get latencies are under 200*us*. However, the Get latencies reach a few *ms* from three nine

(P99, 99.9th percental), and all their latencies increase compared to the normal Get latency as high as 15.7×. The main reason why this long-tail latency is observed across all the RockDB usage scenarios that we test does not stem from database or kernel computation but heavy storage accesses. To be precise, we also decompose the execution time of UserDB, Facebook's social graph data processing workload [12, 73], into Get's storage latency (`Storage`), client computation times (`App`), database latency (`RocksDB`), and kernel latencies (`Kernel`). As shown in Figure 3b, the computation of LSM KV's software stack does not sit on a critical path in the Get long-tail latency, but `Storage` takes 87% of the total execution time thereby dominating Get service times at the tails. While the computation of latency of software stack is well balanced with `Storage` (taking half of the total execution time), LSM KV's heavy I/O requests sharply increase the faction of `Storage` when it should reclaim Memtables and/or SSTFiles.

Internal tasks' performance impacts. Figure 3c shows a time series analysis for the UserDB workload and compares its read characteristic with an ideal Get-only workload, which exhibits I/O patterns the same as UserDB but removes all Put queries from its execution. One can observe from this analysis that, when RocksDB flushes Memtable (at 596K index), the baseline read latency increases from 147*us* to 2.97*ms*, and the read latency does not return for a while. Similarly, once RocksDB begins to compact SSTFiles, it introduces many reads and writes to merge KV pairs, which unfortunately block incoming Get requests thereby exhibiting 30× longer latency than the normal cases. Note that the latency of reads being performed in parallel with WAL is not that significant (compared to flush and compaction), but WAL also makes the Get latency 10.6× worse than the usual cases.

3.2 Scheduling Internal Tasks

Challenges of system-level approaches. There are many studies [4, 5, 7–10, 15, 16, 18–20, 22–24] that try to address the performance degradation imposed by RocksDB's internal tasks, such as TRIAD [22], PebblesDB [23], and SILK [24]. There are variant optimization points across these approaches, but their proposals in general reschedule or delay flush and compaction into idle or other available times, thereby removing the long-tail latency. While these system-level approaches can hide the read/write overhead imposed by LSM KV's internal tasks to some extent, they cannot remove the long-tail latency on Get services because of unavailability to handle SSD's internal tasks and side-effects raised by their scheduling. Specifically, postponing the compaction removes the suspending time for incoming Put services, but it enforces LSM KV's L0 accumulatively accommodate SSTFiles without a data migration to L1, thereby increasing the Get latency. Figure 4a compares two tail latency trends on Gets, each being served with and without compaction rescheduling. The Get tail latency is sustainably managed when RocksDB compacts SSTFiles at the right time (lower than 1 *ms*), but its tail latency served with the delayed compaction keeps increasing and reaches 3.1 *ms*, which is 3.6× longer than the no-scheduling case of RockDB compactions. This is because Get services require searching the appropriate values (paired with input keys) from the beginning to the end of RocksDB's L0. Since the SSTFiles on L0 are not sorted, the KV searching introduces many outstanding reads thereby increasing the tail latency.

On the other hand, as shown in Figure 4b, the delayed flush of RocksDB also increases the Get tail latency as high as 27.4×. The reason why the Get tail latency looks more severe than the rescheudled compaction is that delaying Memtable flush gobbles up all the in-memory spaces, allocated to Memtable management. Thus, the writes of RocksDB are all stalled until it secures a Memtable, which in turn makes the read service suspended seriously.

Device-level latency determinism and limits. The aforementioned SSD's internal tasks are well-known challenges to exhibit serious performance drop and long latency [31–44, 75–77]. Since the internal tasks are invoked in an arbitrary time period, they render many productions in diverse computing domains difficult to deploy latency-critical applications in the environment. Recently, the standard NVMe protocol introduces the *predictable latency mode* (PLM) interface in an attempt to make the latency predictable and deterministic. PLM proposes that SSDs operate in either a deterministic performance window (*DTWIN*) or a non-deterministic performance window (*NDWIN*). Based on the NVMe specification [45], NDWIN is the time period to prepare the next DTWIN.

Note that PLM interface is simply a part of interface protocol, which does not enforce specific requirements or design details for the guarantee of deterministic latency. While this young interface presents blueprints on how to handle the unpredictable SSD behaviors in a well-managed manner, *PLM is in practice a just best-effort contract, which only supports soft latency determinism*. For example, we cannot make the underlying SSD always appropriately work with DTWIN because SSD's internal tasks for hiding the flash characteristics are inevitable to invoke. Even in the ideal case where the underlying hardware hides all the SSD's internal tasks with its maximum efforts, the latency determinism can be easily broken according to how the host-side LSM KV behaves at anytime. To support strong latency determinism, it is necessary to have a close collaboration between the host-side LSM KVs and the underlying storage.

4 High-level View of Vigil-KV

The main goal of this work is to secure an LSM KV system that has no long-tail latency on Get services to make their read performance deterministic and consistent. As this strong latency determinism is infeasible to achieve by scheduling

either only LSM KV's or SSD's internal tasks alone, Vigil-KV takes a hardware and software co-design approach. Specifically, Vigil-KV hardware is designed towards offering basic scheduling blocks that allow the host to integrate strong latency determinism into the LSM KV. On the other hand, the software part of Vigil-KV classifies the requests of LSM KVs at runtime and carefully assigns them to appropriate by fully utilizing the scheduling blocks that the underlying hardware provides. This hardware and software co-design approach can make the latency of client-side Get queries consistently deterministic and have no long-tail all the time.

PLM semantics \ NVMe cmd	Field name			
	OP CODE	Feature ID (CDW10)	NVM Set ID (CDW11)	Feature Enable (CDW12)
`PLMConfig()` Arg1: SetID Arg2: Enable	Set Features	PLM Config	SetID	Enable (0: Off, 1: On)
`PLMWindow()` Arg1: SetID Arg2: Enable	Set Features	PLM Window	SetID	Enable (0: NDWIN, 1: DTWIN)
`GetLogPage()` Arg1: SetID	Get Features	Return values Longest-serving time of DTWIN, Preserved NDWIN, Device lockdown		

Table 1: Vigil-KV hardware.

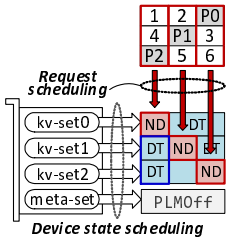


Figure 5: Vigil-KV software.

4.1 Hardware Support for Fine-Granular Performance Windows

PLM interfaces. As shown in Table 1, Vigil-KV hardware implements and provides a set of PLM interfaces that allow the host-side Vigil-KV software to precisely schedule the device states. The functionalities that our PLM interfaces offer are largely classified into three: i) PLM setup (`PLMConfig()`), ii) NDWIN and DTWIN configuration (`PLMWindow()`), and iii) device log queries (`GetLogPage()`). The table also includes how the host-side kernel driver can implement those three semantics using NVMe feature commands. For example, a LSM KV system's kernel driver can turn on or off the target storage's PLM mode by configuring a feature ID (PLM configuration) and enable flag (on/off) through NVMe's `set-feature` [45]. In similar way, it can simply configure the performance window of Vigil-KV hardware using `PLMWindow()`. To query the device state/condition information (that we will reveal in Section 5.2), the LSM KV system can communicate with Vigil-KV hardware through `GetLogPage()` simply returning the results into 512B data package, called a *log page*. Based on a given performance window information, Vigil-KV hardware prioritizes NDWIN to perform SSD's internal tasks as much as possible, and it guarantees that the internal tasks are not scheduled in DTWIN. As discussed in Section 3.2, SSD's internal tasks cannot permanently be postponed, we regulate the longest-serving time of DTWIN and reports it to the host through `GetLogPage`. In addition, Vigil-KV hardware defines the minimum time of NDWIN that should be secured to handle SSD's internal tasks and exposes the configuration time to the host via the log page. Thus, Vigil-KV software can utilize this information by referring into the log page to schedule performance windows appropriately.

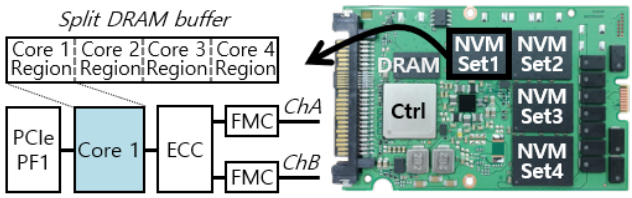
NVM multi-set architecture. To offer a variety of performance scheduling options to the host, our hardware also introduces *NVM multi-set*, which splits the backend storage into multiple volumes, each being exposed to the host as a separate PCIe physical function. Vigil-KV hardware then enables the PLM interface to each physical function, called *NVM set* and makes them work independently by allocating different internal logic/cores across the sets. This NVM multi-set architecture can grant maximum flexibility to the host-side LSM KV's software components, such that they schedule the underlying device states (DTWIN and NDWIN) in a finer granular manner. For example, the LSM KV system can configure different performance windows within a single NVMe device by configuring the NVM Set ID of NVMe's `set-feature` (i.e., the codeword 11 of NVMe's command) differently.

4.2 Software-Defined Strong Latency Determinism for Get services.

Figure 5 shows how Vigil-KV software achieves strong latency determinism by utilizing the finer-granular performance windows that Vigil-KV hardware provides. It consists of three major logical components: i) *metadata separation*, ii) *device state scheduling*, and iii) *request scheduling*.

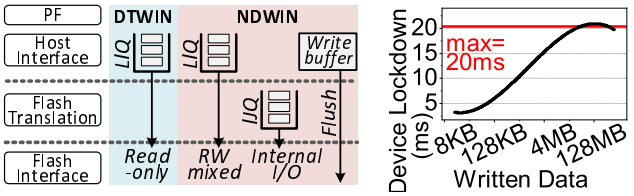
Managing data, devices, and requests. Vigil-KV software excludes a physical function from the storage volume and internally allocates it for metadata management, called *meta-set*. PLM of this meta-set is disabled by `PLMConfig()`, and Vigil-KV software isolates all WAL and journaling activities from LSM KV's regular queries by forwarding the metadata-related requests to the meta-set. This metadata separation allows the kv-sets not to be interfered with by the heavy internal writes for crash consistency management, such that our device state and request scheduling mechanisms can mainly focus on offering strong latency determinisms for Get services.

On the other hand, the remaining physical functions that Vigil-KV hardware exposes are allocated to handle incoming LSM KV's query requests at the kernel-level, which is referred to as `kv-sets`. Vigil-KV software then schedules all the kv-sets device states (i.e., performance windows) to make $n-1$ kv-sets be in DTWIN at any given time (using `PLMWindow()`) while allowing NDWIN to be granted to the underlying kv-sets in a fairly scheduled aspect (round-robin). n is the total number of physical functions that Vigil-KV can assign to the SSTFile management. Vigil-KV software classifies LSM KV's internal tasks and client requests at runtime and schedules them differently by knowing the underlying device's configured performance windows. Specifically, all the client requests are scheduled to be served from the $n-1$ kv-sets, configured with DTWIN. In contrast, Vigil-KV software



(a) Partitioning. (b) Prototype.

Figure 6: Vigil-KV hardware prototype.



(a) Firmware. (b) Device lockdown.

Figure 7: PLM and its constraint.

schedules all the requests coming from LSM KV's internal tasks with a kv-set, scheduled by NDWIN (if there is), but regulates the number of the internal tasks' requests not to make NDWIN be too much long, thereby having always $n-1$ kv-sets configured with DTWIN. We will explain the details of this device state and request scheduling in Section 6.2.

Data reconstruction for NDWIN. Vigil-KV pushes all the LSM KV's and SSD's internal tasks into NDWIN, which are scheduled across different kv-sets in a round-robin manner. While handling requests over NDWIN is essential for both the LSM KV and SSD, the client requests, particularly Get services, targeting the kv-set scheduled with NDWIN can be blocked, thereby exhibiting the long-tail latency. To address this, Vigil-KV encodes parity bits and writes them with internal tasks at NDWIN. Specifically, when Vigil-KV stores an SSTFile, it splits the file into multiple chunks and stripes those chunks across kv-sets at NDWIN. Since we ensure that there are $n-1$ kv-sets configured DTWIN at any given moment, Vigil-KV reads out the data from other kv-sets, reconstructs the original data, and serves them without touching the NDWIN kv-set. This data reconstruction inspired by emerging "array-level" memory and storage techniques [48, 78–81] can obviously remove the long tail latency on Get services, but its reconstruction time can increase the average latency compared to ideal storage making all kv-sets DTWIN consistently. Thus, Vigil-KV also minimizes NDWIN to avoid unnecessary data reconstruction at the physical function level. Note that, as the parity bits are generated per chunk (not per SSTFile) in our scheme, it does not need to recalculate the parities after compaction. The details of this technique and implementation will be described in Section 6.2.

5 Hardware Prototype and Characterizations

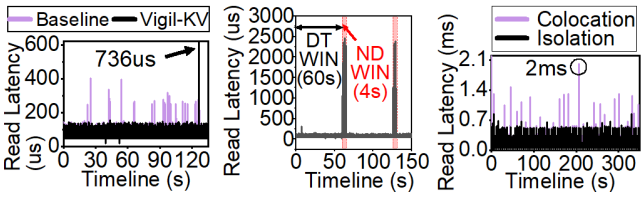
5.1 Enabling PLM with NVM Multi-Sets

Partitioning an SSD. Modern SSDs employ many flash packages, which are connected to multiple embedded cores through multiple memory buses, called *channels*. All flash packages per channel are managed by a specific micro-coded controller, called flash memory controller (*FMC*). For example, our baseline hardware (SSD) contains eight flash packages, each containing eight flash memory banks, and all of them are connected to four cores through eight channels and FMCs. Since each FMC manages the underlying flash packages in a self-governing manner, we modify the baseline hardware to partition the single storage space into multiple spaces. Specifically, as shown in Figure 6a, we allocate each core to every two FMCs and make all the cores work independently as a (separate) physical function. As each physical function should not interfere with each other, we also evenly split the internal DRAM space into multiple spaces, each being allocated to a different physical function. Figure 6b shows our prototype of Vigil-KV hardware. There are four physical functions, each being able to be indicated by a different identifier from the host (cf. Table 1's NVMSet ID). Flash firmware is instantiated per core, such that a physical function performance is not interfered with by other physical functions.

Integrating PLM. To implement DTWIN, each firmware of Vigil-KV hardware employs multiple queues, each being associated with the host command control and internal task management (Figure 7a). Specifically, the internal job queue (*IJQ*) is dedicated to a firmware module that manages address translation while legacy I/O queues (*LIQ*) are allocated to the firmware part that manages the host (NVMe) interface. The requests in Vigil-KV hardware can be therefore classified into legacy and internal tasks and served differently using IJQ and LIQ. Specifically, if a physical function is configured with DTWIN (using `PLMWindow()`), our firmware only handles the requests coming from LIQ and suspends all the requests of IJQ in both foreground and background. This device can immediately serve the incoming (client) read requests without an interruption of SSD's internal tasks. However, the firmware cannot suspend the requests of IJQ if there is no room, which enforces the host schedule DTWIN appropriately. We will explain this constraint in detail shortly.

5.2 PLM Constraint and Behavior Analysis

DTWIN/NDWIN conditions. While resource partitioning and queue isolation (IJQ/LIQ) can remove the read latency spikes imposed by SSD's internal tasks, unfortunately, making deterministic latency consistent is not that simple; it needs a strong collaboration with the host. First, read services on DTWIN suffer interference from a write, which was buffered in a previous NDWIN state. To eliminate this interference, Vigil-KV's firmware explicitly flushes the internal buffer before jumping into DTWIN and disables the buffer for further writes in DTWIN. Note that offering DTWIN with fewer restrictions is the mission that our hardware targets to



(a) vs. Baseline (b) Fair-scheduling. (c) Isolation.

Figure 8: Performance characterization of prototype.

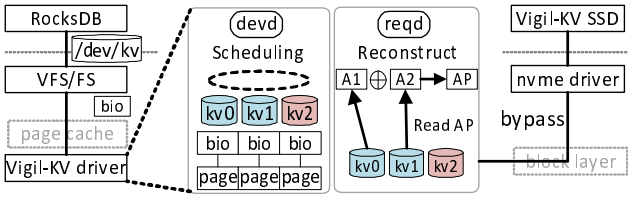


Figure 9: Implementation of Vigil-KV software stack.

achieve, but it should not lose any data during I/O services with DTWIN. Thus, it is necessary to clear the internal buffer and bypass it before DTWIN and in the middle of DTWIN, respectively. During the internal buffer flush, the host should not further write data in order to clearly wipe it out, which is called *device lockdown* condition. Figure 7b analyzes the device lockdown times varying based on how much data were written in the previous NDWIN. All the workloads that we tested [12, 74] write tens of MB during NDWIN. it is sufficient for the host to hold the data (if there is) by under 20 *ms*. Similarly, when there is a write on DTWIN, our hardware returns the performance window from DTWIN to NDWIN in order to guarantee strong durability and consistency of the written data. The host therefore makes sure that there is no write on DTWIN, called DTWIN's *write-free* condition.

DTWIN must also not hurt the current level of reliability management that the existing flash firmware provides. Specifically, the underlying flash media can be stressed only with reads even though there is no write or internal task because of the read disturbance issue (Section 2.2). Thus, Vigil-KV hardware regulates the most extended time window for DTWIN, called *maxDTWIN*, by considering the worst case where the heavy reads on a specific block can corrupt all the page data therein. Similarly, NDWIN should be continued for a certain level of the time duration, called *minNDWIN*, which is the shortest time to complete SSD's internal tasks (data migration and block erases) and the accumulated requests in IJQ during maxDTWIN. Obviously, these maxDTWIN and minNDWIN periods are strongly correlated because IJQ is limited to queue SSD's internal tasks. By considering this, the host should schedule DTWIN and NDWIN fairly, called *fair-scheduling* condition. Based on preliminary profiles, we configure maxDTWIN and minNDWIN as 60 and 4 seconds, respectively.

Note that all these information such as the device lockdown time, maxDTWIN, and minNDWIN are exposed to the host through `GetLogPage()` (cf. Table 1).

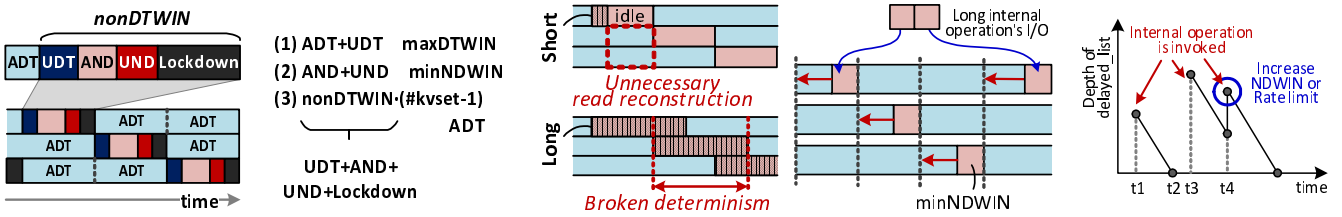
Performance characterization and validation. Figure 8a compares the read latency trends between the baseline device and our Vigil-KV hardware prototype. While the baseline device exhibits multiple latency spikes (∼402*us*), the reads with Vigil-KV hardware are all served by 74*us*, on average, and it is guaranteed for the latency to be under 200*us*. Note that, when we change DTWIN to NDWIN by calling `PLMWindow()` at 128 seconds, the read latency reaches as high as 736 *us* as SSD's internal tasks are scheduled in that performance window correctly. When we schedule DTWIN and NDWIN one by one (by satisfying the fair-scheduling), as shown in Figure 8b, the performance behaviors mentioned above are all guaranteed across multiple DTWINs. At the same time, the hardware is busy handling the accumulated internal tasks in NDWIN. Lastly, Figure 8c compares the baseline device that collocates reads and writes within a single storage space and Vigil-KV hardware isolating the interference across multiple physical functions. One can observe from this figure that the read latency of the baseline device severely fluctuates and reaches as high as 2 *ms*. In contrast, the read latency on a physical function of Vigil-KV hardware is not interfered with by the writes heading to other physical functions even though we turned off the PLM interface for the physical function. This is because we partition each physical function with completely different resources. Note that Vigil-KV software utilizes this performance isolation for metadata management, which can make kv-sets free from managing the write-free condition in cases of writing WAL and journaling to the underlying storage.

6 Details of Vigil-KV Software

While there are constraints for PLM management, Vigil-KV hardware opens the opportunity to schedule performance windows across different physical functions being mapped to NVM sets in a finer granule manner. Vigil-KV software separates LSM KV's internal tasks, including metadata management from client Get services, and schedules them with NDWIN having SSD's internal tasks together. In addition, the software part of Vigil-KV reconstructs data in cases where it cannot serve the data because NDWIN services, which can in turn allow LSM KVs to have DTWIN consistently, providing strong latency determinism.

6.1 Vigil-KV Stack Implementation

Figure 9 shows the implementation of our Vigil-KV software stack. RocksDB connects Vigil-KV hardware through existing file system interfaces and performs SSTFile-related services on `/dev/kv`. Underneath the file systems, we locate our Vigil-KV driver operating with two kernel threads, `reqd` and `devd`, each scheduling block I/O (`bio`) requests and our hardware's device states, respectively. Vigil-KV driver maps



(a) Timeline. (b) Equations. (c) Under/Overflow. (d) Dynamic schedule. (e) Example.

Figure 10: Performance window management.

multiple physical functions that Vigil-KV hardware exposes to different NVM sets (meta-set and kv-sets) at the system's initialization. `reqd` is similar to Linux existing multiple device `md` driver for striping data chunks and bios across different kv-sets, but it schedules them being aware of underlying device states. Specifically, `reqd` ensures the scheduled bio requests satisfy the write-free condition on DTWIN. In addition, it makes sure that the client's read requests are not stalled due to LSM KV's internal tasks by performing the data reconstruction on-the-fly. On the other hand, `devd` schedules the device states for kv-sets by considering the fair-scheduling condition and device lockdown time. More details of this device state scheduling will be explained shortly.

To make read latency predictable, the Vigil-KV driver also bypasses Linux page cache and block layer, which can make the read latency fluctuate and/or be difficult to manage to some extent. For example, Since kv-sets are only managed internally, the bio structures for kv-sets (e.g., logical block address and offsets) are different from the bio requests that the page cache manages. Instead, the Vigil-KV driver employs an internal buffer, called `plm_cache`, which buffers bio requests of kv-sets in a form of Linux `stripe` list. The `plm_cache` size can be configured by the user as a kernel parameter at the boot time. When `reqd` schedules the including bio requests to underlying kv-sets, it thus uses `stripe` requests. Since the Vigil-KV driver bypasses the page cache, it also offers `plm_sync` system call (a variant of file system's `fsyc`) to RocksDB. This `plm_sync` makes sure that the Vigil-KV driver completely flushes `plm_cache` before Memtables, WAL, and SSTFiles are deleted because of LSM KV's internal tasks. The reason why our Vigil-KV driver bypasses the block layer and directly communicates with the `nvme` driver is that the block layer's bio merging and ordering can break determinism. For example, the requests of LSM KV's internal tasks are scheduled for NDWIN, but they can be issued at DTWIN by the block layer. Note that as `reqd` schedules LSM KV's internal tasks and client requests differently, it is required to deliver the priority information from LSM KV to the Vigil-KV driver. To this end, we have a minor modification on RocksDB and journaling block device daemon (`jbd2`), which can be easily applied to other LSM KVs. When RocksDB creates background threads, it calls a system call, `ioprio_set` that configures I/O priority as 'internal task'. `ioprio_set` delivers the priority by storing its information into `io_context` of process control block, `task_struct`. Since Get queries and WAL are managed by all the same thread of RocksDB, we modify `WriteImpl()` such that it configures I/O priority as 'WAL' by calling `ioprio_set` before performing `WriteToWAL()`. Journaling is also classified by `ioprio_set` before committing a transaction (e.g., `jbd2_log_do_checkpoint()`).

6.2 Performance Window Management

Device state scheduling. As shown in Figure 10a, `devd` schedules DTWIN and NDWIN to make sure that there are always $n-1$ kv-sets, configured with DTWIN. Therefore, all the client requests are served from DTWIN or `reqd`'s data reconstruction. When `devd` schedules performance windows to meet the fair-scheduling and device downtime constraints, there are two more technical challenges. Even though `reqd` reads the data from kv-sets configured with DTWIN, the read request can be delayed because of the outstanding reads issued previously and not yet completed. These delayed reads can be served at NDWIN, which cannot in turn offer the strong latency determinism. Similarly, the writes issued to NDWIN can be practically served at DTWIN because of the outstanding writes as well as the time delay caused by SSD's internal tasks to some extent. This situation is less desirable than the former as it can break the write-free condition on DTWIN.

We classify DTWIN and NDWIN more specifically by considering those two unavailabilities further. DTWIN is split into *ADT* (available DTWIN) and *UDT* (unavailable DTWIN), and similarly, NDWIN is also separate into available/unavailable NDWIN (*AND*/*UND*). Since UDT and UND can have such outstanding operations on `reqd`, `devd` schedules NDWIN and DTWIN with a time unit as long as *nonDTWIN*. nonDTWIN includes UDT, AND, UND, and the device lockdown time windows (*Lockdown*), and each of the windows should satisfy the condition described by Equations 10b. `devd` profiles kv-sets' bandwidth and then estimates UDT and UND by dividing the total amount of data volume for the outstanding requests with the read and write bandwidth of kv-sets, respectively. Note that, in contrast to the read bandwidth, the write bandwidth on DTWIN can vary. We thus use the worst-case bandwidth of writes for the UND estimation.

Dynamic adjustment for NDWIN. When `devd` controls performance windows of the underlying kv-sets per nonDTWIN

(=ADT), there are two challenges that it needs to address as shown in Figure 10c: NDWIN *underflow* and *overflow*. In cases where the amount of internal tasks of LSM KV and/or SSD at NDWIN, `reqd` wastes computation for data reconstruction and can increase the read latency when there is heavy read traffic; we observed that, when one increases QPS (queries per second) from 60K to 150K, the read latency increases by 26%, on average. While minimizing the data reconstruction involvement (NDWIN) is the matter, NDWIN can break determinism if too many the internal tasks are issued. We also preliminary evaluated all our workloads and observed that 48.3% of compaction scheduled at NDWIN (excluding UDT, UND, and Lockdown) are executed at DTWIN.

As shown in Figure 10d, `devd` adjusts NDWIN at runtime to address the underflow and overflow situation. Specifically, `devd` begins scheduling NDWIN (per ADT) by setting it as long as minNDWIN to minimize the involvement of `reqd`'s data reconstruction and spreads buffered bio requests (stored on `plm_cache`) across different minNDWIN. If the outstanding requests are accumulated more than a threshold, it maximizes NDWIN to serve LSM KV internal tasks' I/O as fast as possible. For example, as shown in Figure 10e, `devd` examines `reqd`' queues containing outstanding bio requests at the time epoch 1 ($t1$). In this case, all the outstanding requests are served/completed before $t2$. However, since the requests associated with LSM KV internal tasks at $t3$ are not resolved by $t4$, `devd` maximizes NDWIN. Note that, Vigil-KV applies dynamic NDWIN adjustment for `plm_cache` by using the user-defined memory limits as adjustment threshold.

7 Evaluation

7.1 Experimental Setup

Prototype and environments. We implement a prototype of Vigil-KV hardware on a 1.92TB datacenter-scale NVMe SSD for research purposes. Vigil-KV hardware employs four physical functions, and they equally divide the hardware resources such as 3GB LPDDR4 DRAM, eight channels, and 64 TLC NAND flash dies into four. Note that the baseline hardware is not that different from the Vigil-KV hardware. It has hardware resources the same as Vigil-KV hardware, but it only employs a single physical function. We perform the evaluation on a 12-core AMD Ryzen 9 5900X, 96GB DRAM, and Vigil-KV hardware by running Vigil-KV software implemented on RocksDB 6.23.0 and Linux 4.19.91. For the evaluation, we set the size of `plm_cache` as 2GB.

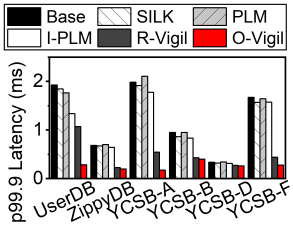
		Short description.	Get (%)	Put (%)	Get $ hit (%)	Flush write (GB)	Compaction write (GB)
FB	**UserDB**	All social graph actions	54	46	25	2.4	8.8
	ZippyDB	Read ObjStorage meta	42	58	86	8.9	17.5
YCSB	**A**	Log user action	66	34	17	3.6	19.2
	B	Update/read photo tag	95	5	23	0.2	1.3
	D	Read latest record	95	5	83	0.5	1.8
	F	Update user record	74	26	45	3.1	15.6

Table 2: Important characteristics of evaluated workloads.

Workloads. We evaluate six workloads that use an LSM KV as their backend storage engine. (two from Facebook [12] and four from Yahoo [74]) For social network services, Facebook uses `UserDB` and `ZippDB` workloads that serve social graph data and object (e.g., image or video) storage metadata as a form of key-value, respectively. The key-value cache hit ratio of `UserDB` is only 25% due to its irregular key access pattern, whereas that of `ZippyDB` is 86% because of its read-latest characteristics of social contents. Yahoo also provides representative LSM KV access patterns of various cloud services, such as write-intensive (`YCSB-A`), read-intensive (`YCSB-B`), read-latest (`YCSB-D`), and read-modify-write intensive (`YCSB-F`). In the table, also analyze the amount of writes caused by internal tasks (compaction) during our evaluations.

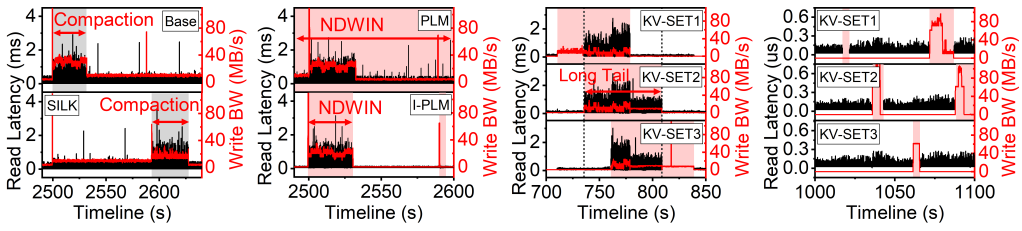
Configurations. We evaluate six different LSM KV hardware and software combinations.

- `Base` [1]: the representative conventional LVM KV (e.g., RocksDB) with the baseline hardware.
- `SILK` [24]: the state-of-the-art software supports of LSM KV with the baseline hardware.
- `PLM`: Vigil-KV hardware with a simple driver, which utilizes the PLM interfaces (cf. Section 4.1).
- `I-PLM`: based on `PLM`, we add the metadata isolation support (cf. Section 6.1).
- `R-Vigil`: based on `I-PLM`, we add the device state scheduling support (cf. Section 6.2).
- `O-Vigil`: based on `R-Vigil`, we add the dynamic non-determinism scheduling support (cf. Section 6.2).



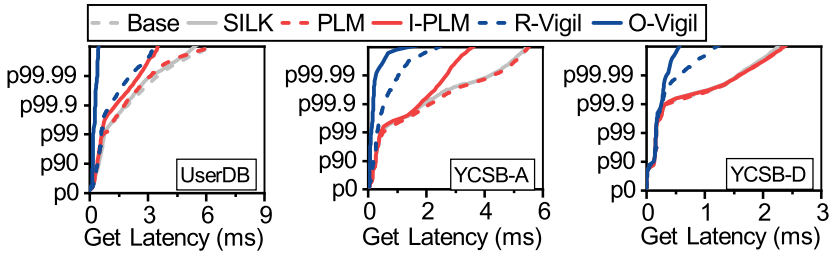
(a) p99.9 tail latency.

Figure 11: Tail latency.



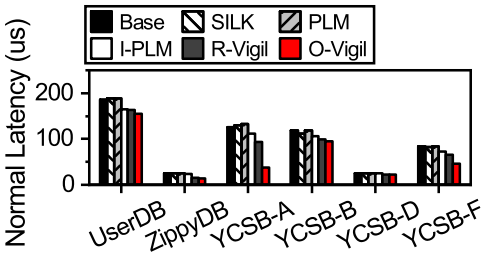
(a) `Base` and `SILK`. (b) `PLM` and `I-PLM`. (c) `R-Vigil`. (d) `O-Vigil`.

Figure 12: Time series analysis of representative workload (`UserDB`).



(a) UserDB. (b) YCSB-A. (c) YCSB-D.

Figure 13: CDF graphs.



(a) Average latency.

Figure 14: Normal cases.

Since Vigil-KV adopts the concept of array-level memory and storage techniques making three NVM sets as a single storage volume, we use conventional multiple device (`md`) driver for `Base` and `SILK` as well to satisfy the fair performance comparison among the configurations that we tested.

7.2 Long-tail Latency Analysis

We analyze the long-tail latency on Get services by executing all the Facebook and Yahoo workloads atop the six configurations. As shown in Figure 11, `UserDB`, `YCSB-A`, and `YCSB-F` show longer p99.9 tail latency than others due to their high Put service ratio, which increases the LSM KV and SSD internal tasks. Meanwhile, `ZippyDB` and `YCSB-D` further exhibit shorter tail latency than `YCSB-B` thanks to their high key-value cache hit ratio, which can reduce the number of reads interfered with by internal tasks. To understand how six different configurations impact long-tail latency, Figure 12 analyzes time series by selecting `UserDB` as a representative workload. It shows both storage read latency (left axis with black line) and write throughput (right axis with red line), which allow us to infer how much LSM KV and SSD internal tasks interfere with Get queries and when the internal tasks occur.

`Base` vs. `SILK`. As shown in Figure 11, `SILK` only reduces 5% of the tail compared to `Base`, on average. Even though `UserDB` can capture the user idle behaviors while YCSB workloads cannot, still there was not an enough idle time to schedule LSM KV internal tasks as shown in Figure 12a. Thus, the delayed compaction starts to interfere with Get services from 2600 seconds in `SILK`, and the storage read latency spikes.

`SILK` vs. `PLM`. `PLM` experiences the tail similar to `Base` (5.4% longer than `SILK`), which indicates that Vigil-KV hardware cannot guarantee latency determinism without Vigil-KV software. As shown in the top of Figure 12b, Vigil-KV hardware is in NDWIN most of the time (red background) since WAL or journaling breaks the DTWIN's write-free condition.

`PLM` vs. `I-PLM`. Therefore, `I-PLM` isolates WAL and journaling to dedicated meta-set and securing NDWIN (white background) as shown in the bottom of Figure 12b. Since `UserDB` has a higher Put service ratio than others, it experiences 24.4% shorter long-tail latency compared to `PLM`, while others achieve 12% shorter long-tail latency, on average.

`I-PLM` vs. `R-Vigil`. As shown in Figure 12c, `R-Vigil` schedules NDWIN across three kv-sets and guarantees latency determinism as much as possible by reconstructing reads with DTWIN kv-sets. Thus, it can reduce long-tail latency by 48.2% compared to `I-PLM`, on average. As shown in Figure 11, especially for high Put service workloads (`YCSB-A` and `YCSB-C`), they exhibit 70% shorter long-tail latency than `I-PLM`. However, `R-Vigil` still exhibits long-tail latency due to NDWIN overflow (cf. 736 $\sim$ 809 seconds in Figure 12c).

`R-Vigil` vs. `O-Vigil`. Therefore, `O-Vigil` strongly guarantees the latency determinism with out dynamic NDWIN adjustment. `O-Vigil` reduces long-tail latency by 33.5% than `R-Vigil`, on average, while high Put service workloads (`UserDB`, `YCSB-A` and `YCSB-F`) can achieve shorter long-tail latency 59%. As shown in Figure 12d, `O-Vigil` dynamically schedules LSM KVs internal tasks.

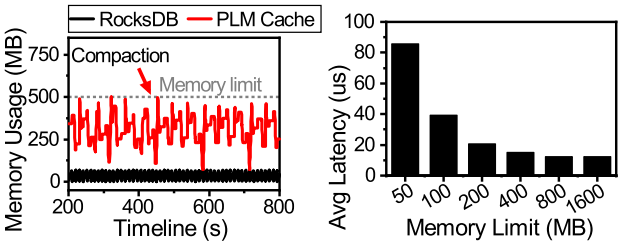
Note that `O-Vigil` not only reduces the long-tail latency of the `Base` by 3.19×, on average, but also guarantees under 500*us* Get service latency across all the workloads.

7.3 Analysis of Different-level Latency

Tail latency distribution. To understand the impact of Vigil-KV for the different levels of tail-latency, we analyze the CDF of Get latency for three representative workloads, such as `UserDB`, `YCSB-A`, and `YCSB-D`. Since `UserDB` and `YCSB-A` are high Put service ratio workloads, the long-tail of Get latency start from p99 as shown in Figures 13a and 13b. On the other hand, as `YCSB-D` has higher Get service ratio workloads and most of the Get is serviced from the key-value cache, the long-tail of Get latency start from p99.9 as shown in Figure 13c. Thus, for the `YCSB-D` workload, `O-Vigil` achieves 78% shorter p99.99 long-tail latency than `Base`, while only 11% of the long-tail is reduced at p99.9 latency. Not only for `YCSB-D`, `O-Vigil` can further mitigates the p99.99 long-tail latency of `UserDB` and `YCSB-A` by 69% and 94%, respectively.

Note that Vigil-KV can also guarantee the strong latency determinism (*us*-scale latency of Get service) more than four nines long-tail latency (p99.99) as shown in Figure 13.

Normal cases. Since Vigil-KV adds more software supports than conventional LSM KV, it is important to analyze the Get service latency of normal cases to check whether Vigil-KV slows down the average Get latency or not. Figure 14 shows the average latency of Get services for all the workloads and



(a) Usage. (b) Limit.

Figure 15: Memory consumption.

configurations, and we observed that Vigil-KV (`O-Vigil`) reduces the average Get latency by 34% compared to `Base` (not increases the average Get latency). This is because of two reasons: 1) Vigil-KV isolates the write I/O traffic of metadata (e.g., WAL and filesystem journaling) from the read I/O traffic of Get service, and 2) Vigil-KV minimizes the data reconstruction as much as possible. Note that, the reason why `ZippyDB` and `YCSB-D` show shorter average Get latency compared to other workloads is that they have a high key-value cache hit ratio (which is not related to Vigil-KV solutions).

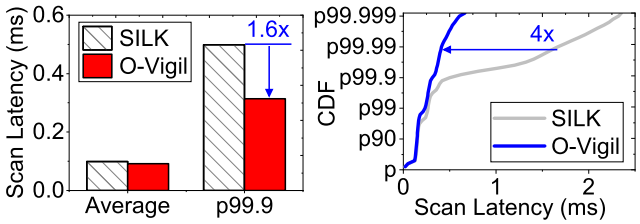
7.4 Memory Consumption and Scan Service

Time series analysis. Figure 15a shows memory consumption of application-level (e.g., RocksDB's Memtable) and kernel-level (e.g., Vigil-KV driver's `plm_cache`) during workload execution. We select `ZippyDB` as a representative workload since there is a large amount of storage writes by compaction internal tasks (cf. Table 2). While RocksDB periodically flushes Memtables to the underlying storage to maintain a certain threshold (e.g., 64MB) of the application-level write buffer, Vigil-KV has to cache/buffer write requests until the target kv-set reaches NDWIN. Thus, the memory usage of `plm_cache` increases when LSM KV internal tasks (e.g., Memtable flush and compaction) occur. However, Vigil-KV supports dynamic adjustment for NDWIN being aware of the memory limits of `plm_cache` which can regulate the maximum memory consumption.

Sensitivity test. To understand the impact of `plm_cache`'s memory limits to the average latency of Get services, we perform sensitivity tests by increasing the memory limits from 50MB to 1.6GB as shown in Figure 15b. While extremely low memory limit (e.g., 50MB) exhibits long normal latency of more than 80*us*, a few hundreds of MB `plm_cache` is enough to serve normal Get latency without performance degradation.

Note that, although Vigil-KV delays LSM KV internal tasks until target kv-sets reach NDWIN, it does not increase memory footprints or degrade the Get latency, unlike prior studies (e.g., TRIAD [22], PebblesDB [23], and SILK [24]).

Performance with Scan. Vigil-KV mainly considers improving the performance of Get queries as a first-class citizen. This is because Get of most large-scale workloads (Facebook [12] and Yahoo [74]) account for 78% of the total queries in the workloads that we tested. While Scan in contrast accounts



(a) `SILK` vs. `O-Vigil`. (b) CDF analysis.

Figure 16: Performance analysis of Scan queries.

for 3% of the total queries, its query latency may also be important for a specific workload such as scanning many posts in parallel (e.g., YCSB-E).

In this subsection, we compare the latency behaviors of Vigil-KV (`O-Vigil`) with those of `SILK` by evaluating `YCSB-E` as the representative of Scan-sensitive workloads (95% and 5% for Scan and Put, respectively). Figure 16a shows the comparison for the average (p50) and p99.9 latency. Since RocksDB performs readahead and prefetch in default, the sequentially of Scan exhibits many cache hits. Similar to `YCSB-D`, this in turn benefits the average latency marginal as Scan operations of both `O-Vigil` and `SILK` from memory than devices in most cases. However, such readahead and prefetch techniques cannot avoid every I/O request and hide all the read latency that the underlying SSD exposes. Vigil-KV can support deterministic latency for such cases, thereby offering 1.6× shorter p99.9 latency of `SILK`. This performance benefit becomes more promising as the degree of the long-tail latency gets higher. As shown in Figure 16b, even though there are few Put operations, they lead LSM KV internal tasks, which can make p99.9∼p99.999 tail-latency much longer. Vigil-KV can remove such long-tail latency with hardware/software co-designed strong latency determinism, thereby improving the tail latency by 1.6× ∼ 4×.

8 Conclusion

In this paper, we propose Vigil-KV, a hardware and software co-designed framework that eliminates long-tail latency by introducing strong latency determinism into LSM KVs. We evaluate diverse Facebook and Yahoo scenarios with Vigil-KV, and our empirical evaluation shows that Vigil-KV can reduce the tail latency of the baseline KV system by 3.19× while reducing the average latency by 34% as well.

Acknowledgement

The author thanks to anonymous reviewers for their constructive feedback. This work is mainly supported by Samsung (G01200447) and Samsung HiPER. This work is also in part supported by NRF's 2021R1A2C4001773, IITP's 2021-0-00524 & 2022-0-00117, KAIST start-up package (G01190015), and KAIST IDEC. Myoungsoo Jung is the corresponding author.

References

[1] Facebook. Rocksdb: A persistent key-value store for fast storage environments. https://rocksdb.org.

[2] Sanjay Ghemawat and Jeff Dean. Leveldb. https://github.com/google/leveldb.

[3] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021.

[4] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[5] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment*, 2017.

[6] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017.

[7] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 2017.

[8] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, 2020.

[9] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated key-value storage management for balanced i/o performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[10] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[11] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment*, 2020.

[12] 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)*, 2020.

[13] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[14] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 2021.

[15] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. Slm-db: single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[16] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[17] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[18] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[19] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX-ATC 18)*, 2018.

[20] 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 (USENIXATC 20)*, 2020.

[21] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021.

[22] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, 2017.

[23] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[24] 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)*, 2019.

[25] Hyojun Kim and Seongjun Ahn. Bplru: A buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.

[26] Ping Huang, Pradeep Subedi, Xubin He, Shuang He, and Ke Zhou. Flexecc: Partially relaxing ecc of mlcssd for better cache performance. In *2014 USENIX Annual Technical Conference (USENIXATC 14)*, 2014.

[27] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[28] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. *Acm Sigplan Notices*, 2009.

[29] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. Fab: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 2006.

[30] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *IEEE Transactions on Computers*, 2008.

[31] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[32] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 2014.

[33] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Taylan Kandemir. Design of a host interface logic for gc-free ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[34] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, 2019.

[35] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 2004.

[36] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. Reducing garbage collection overhead in ssd based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[37] Narges Shahidi, Mahmut T Kandemir, Mohammad Arjomand, Chita R Das, Myoungsoo Jung, and Anand Sivasubramaniam. Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016.

[38] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*. Boston, USA, 2008.

[39] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 2017.

[40] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 2017.

[41] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Errors in flash-memory-based solid-state drives: Analysis, mitigation, and recovery. *arXiv preprint arXiv:1711.11427*, 2017.

[42] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2015.

[43] Bryan S Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for ssd reliability. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[44] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. An integrated approach for managing read disturbs in high-density nand flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.

[45] NVM Express, Inc. NVM express specification. https://nvmexpress.org/specifications.

[46] Gyuyoung Park, Miryeong Kwon, Pratyush Mahapatra, Michael Swift, and Myoungsoo Jung. Bibim: A prototype multi-partition aware heterogeneous new memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[47] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Nam Sung Kim, Mahmut Taylan Kandemir, and Myoungsoo Jung. Revamping storage class memory with hardware automated memory-over-storage solution. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.

[48] Jie Zhang, Gyuyoung Park, David Donofrio, John Shalf, and Myoungsoo Jung. Dram-less: Hardware acceleration of data processing with new memory. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020.

[49] Huaicheng Li, Martin L Putra, Ronald Shi, Xing Lin, Gregory R Ganger, and Haryadi S Gunawi. loda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[50] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. Bcw: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[51] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, 2012.

[52] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program interference in mlc nand flash memory: Characterization, modeling, and mitigation. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013.

[53] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: Improving nand flash lifetime by balancing page endurance. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[54] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, 2006.

[55] Bryan S Kim, Hyun Suk Yang, and Sang Lyul Min. Autossd: an autonomic ssd architecture. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*, 2018.

[56] Myoungsoo Jung, Ramya Prabhakar, and Mahmut T. Kandemir. Taking garbage collection overheads off the critical path in ssds. In *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*, volume 7662 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2012.

[57] Guanying Wu and Xubin He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, 2012.

[58] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of ultra-low latency solid state drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[59] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. Faster than flash: An in-depth study of system challenges for emerging ultra-low latency ssds. In *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019*. IEEE, 2019.

[60] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 477–492. USENIX Association, 2018.

[61] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association. ISBN 978-1-931971-28-7. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder.

[62] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012.

[63] Alaa R Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. Energy-efficient cache design using variable-strength error-correcting codes. *ACM SIGARCH Computer Architecture News*, 2011.

[64] Chun-Yi Liu, Jagadish B Kotra, Myoungsoo Jung, Mahmut T Kandemir, and Chita R Das. Soml read: Rethinking the read operation granularity of 3d nand ssds. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[65] Qiao Li, Liang Shi, Chun Jason Xue, Kaijie Wu, Cheng Ji, Qingfeng Zhuge, and Edwin H-M Sha. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[66] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. Ldpc-in-ssd: Making advanced error correction codes work effectively in solid state drives. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[67] Hongbin Sun, Wenzhe Zhao, Minjie Lv, Guiqiang Dong, Nanning Zheng, and Tong Zhang. Exploiting intracell bit-error characteristics to improve min-sum ldpc decoding for mlc nand flash-based storage in mobile device. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2016.

[68] Meng Zhang, Fei Wu, Xubin He, Ping Huang, Shunzhuo Wang, and Changsheng Xie. Real: A retention error aware ldpc decoding scheme to improve nand flash read performance. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2016.

[69] M Jung and M Kandemir. Revisiting widely-held expectations of ssd and rethinking implications for systems. *SIGMETRICS'13*, 2013.

[70] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013.

[71] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R Nevill. Bit error rate in nand flash memories. In *2008 IEEE International Reliability Physics Symposium*. IEEE, 2008.

[72] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. Vulnerabilities in mlc nand flash memory programming: Experimental analysis, exploits, and mitigation techniques. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017.

[73] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (MOD)*, 2013.

[74] 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*, 2010.

[75] Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. Parallelizing garbage collection with i/o to improve flash resource utilization. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018.

[76] Benny Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 2013.

[77] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *2014 International Conference on Smart Computing*. IEEE, 2014.

[78] Sangwon Lee, Miryeong Kwon, Gyuyoung Park, and Myoungsoo Jung. Lightpc: hardware and software co-design for energy-efficient full system persistence. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 289–305, 2022.

[79] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 2010.

[80] Bo Mao, Hong Jiang, Suzhen Wu, Lei Tian, Dan Feng, Jianxi Chen, and Lingfang Zeng. Hpda: A hybrid parity-based disk array for enhanced performance and reliability. *ACM Transactions on Storage (TOS)*, 2012.

[81] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity ssd arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021.