An Efficient Memory-Mapped Key-Value Store for Flash Storage

Anastasios Papagiannis[*]
Institute of Computer Science, FORTH
Heraklion, Greece
apapag@ics.forth.gr

Giorgos Saloustros
Institute of Computer Science, FORTH
Heraklion, Greece
gesalous@ics.forth.gr

Pilar González-Férez
Department of Computer Engineering, University of Murcia, Spain
pilargf@um.es

Angelos Bilas[†]
Institute of Computer Science, FORTH
Heraklion, Greece
bilas@ics.forth.gr

ABSTRACT

Persistent key-value stores have emerged as a main component in the data access path of modern data processing systems. However, they exhibit high CPU and I/O overhead. Today, due to power limitations it is important to reduce CPU overheads for data processing.

In this paper, we propose *Kreon*, a key-value store that targets servers with flash-based storage, where CPU overhead and I/O amplification are more significant bottlenecks compared to I/O randomness. We first observe that two significant sources of overhead in state-of-the-art key-value stores are: (a) The use of compaction in LSM-Trees that constantly perform merging and sorting of large data segments and (b) the use of an I/O cache to access devices, which incurs overhead even for data that reside in memory. To avoid these, *Kreon* performs data movement from level to level by performing partial instead of full data reorganization via the use of a full index per-level. In addition, *Kreon* uses memory-mapped I/O via a custom kernel path with Copy-On-Write.

We implement *Kreon* as well as our custom *memory-mapped I/O* path in Linux and we evaluate *Kreon* using commodity SSDs with both small and large datasets (up to 6 billion keys). For a large dataset that stresses I/O, *Kreon* reduces CPU cycles/op by up to 5.8x, reduces I/O amplification for inserts by up to 4.61x, and increases insert ops/s by up to 5.3x, compared to RocksDB, a state-of-the-art key-value store that is broadly used today.

[*]Also with the Department of Computer Science, University of Crete, Greece
[†]Also with the Department of Computer Science, University of Crete, Greece

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6011-1/18/10. . . $15.00
https://doi.org/10.1145/3267809.3267824

CCS CONCEPTS

• **Information systems** → **Key-value stores**; **Flash memory**; *B-trees*; *Hierarchical storage management*; • **Software and its engineering** → *Virtual memory*;

KEYWORDS

Key-Value Stores, LSM-Tree, Memory-Mapped I/O, mmap, SSD, Copy-On-Write

ACM Reference Format:
Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 13 pages.
https://doi.org/10.1145/3267809.3267824

1 INTRODUCTION

Persistent key-value stores [1, 12, 17, 18] are a central component for many analytics processing frameworks and data serving systems. These systems are considered as write-intensive because they typically exhibit bursty inserts with large variations in the size of data items [7, 37]. To better serve write operations, key-value stores have shifted from the use of B-trees [3], as their core indexing structure, to a group of structures known as write-optimized indexes (WOIs) [23]. This transition took place because even though B-trees [3] are asymptotically optimal in the number of block transfers required for point and range queries their write performance degrades significantly as the index grows [24].

A prominent data structure in the WOIs group is LSM-Tree [31]. LSM-Tree has two important properties: (a) it amortizes device write I/O operations (I/Os) over several insert operations and (b) it is able to issue only large I/Os to the storage devices for both reads and writes, essentially resulting in sequential device accesses. These properties have made LSM-Tree appropriate for hard disk drives (HDDs) that suffer from long seek times and their throughput drops by more than two orders of magnitude in the presence of random I/Os. However, these desirable properties come at the expense of significant CPU overhead and I/O amplification. LSM-Tree needs to constantly merge and sort large data segments, operations that lead to both high CPU utilization and increased I/O traffic [33, 41].

In addition, modern key-value stores incur significant CPU overhead for caching data in their address space [21]. Key-value stores need to cache data in user-space to avoid frequent user-kernel crossings and accesses to devices. Therefore, at runtime, there is a need to maintain a lookup structure for data items that reside in memory. Lookup operations occur in the common path and are required not only for misses but also for hits, when data reside in memory. These common path lookup operations incur significant cost in CPU cycles. Harizopoulos *et.al.* [21] claim that about one-third of the total CPU cycles of a database system is spent in managing the user-space cache when the dataset fits in memory. Furthermore, the cache needs to manage I/O to the devices via the system call interface that is expensive for fine-grain operations and requires data copies for crossing the user-kernel boundary. In our work, we find that cache and system call overheads in RocksDB [17], a state-of-the-art persistent key-value store, are up to 28% of the total CPU cycles used (Table 3).

With current technology limitations and trends, these two issues of high CPU utilization and I/O amplification are becoming a significant bottleneck for keeping up with data growth. Server CPU is the main bottleneck in scaling today's infrastructure due to power and energy limitations [25, 28, 36]. Therefore, it is important to increase the amount of data each CPU can serve, rather than rely on increasing the number of CPUs in the datacenter. In this context, flash-based storage, such as solid state drives (SSDs), introduces new opportunities by narrowing the gap between random and sequential throughput, especially at higher queue depths (number of concurrent I/Os). Figure 1 shows the throughput of an SSD and two NVMe devices with random I/Os and increasing request size. At a queue depth of 32, an I/O request size of 32 KB for SSDs and 8 KB for NVMe achieve almost the maximum device throughput. Therefore, increased traffic due to I/O amplification is becoming a more significant bottleneck than I/O randomness. This trend will be even more pronounced with emerging storage devices that aim to achieve sub-μs latencies.

In this paper we present *Kreon*, a key-value store that aims to reduce CPU overhead and I/O traffic by trading I/O randomness. *Kreon* combines ideas from LSM [31] (multilevel structure), bLSM [37] (B-Tree index), Atlas/WiscKey [25, 29] (separate value log), and Tucana [32] memory mapped I/O. Additionally, it uses a fine-grain spill mechanism which partially reorganizes levels to provide high insertion rates and reduce CPU overhead and I/O traffic. *Kreon* uses a write optimized data structure that is organized in N levels, similar to LSM-Tree, where each level i acts as a buffer for the next level $i+1$. To reduce I/O amplification, *Kreon* does not operate on sorted buffers, but instead it maintains a B-tree index within each level. As a result, it generates smaller

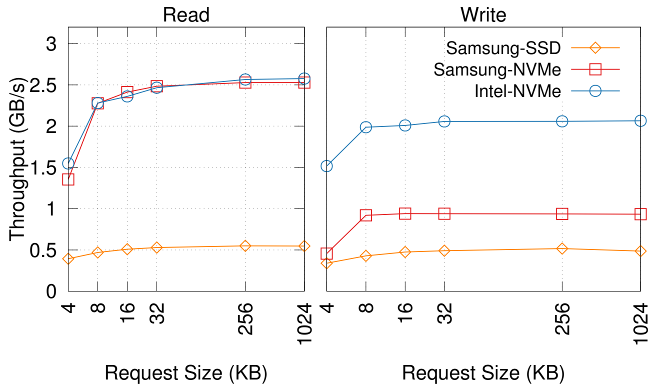


Figure 1: Throughput vs. block size (using iodepth 32) for Samsung SSD 850 Pro 256 GB, Samsung 950 Pro NVMe 256 GB, and Intel Optane P4800X NVMe 375 GB devices, measured with FIO [2].

I/O requests in favor of reduced I/O amplification and CPU overhead. *Kreon* still requires and uses multiple levels to buffer requests and amortize I/O operations.

Furthermore, *Kreon* uses *memory-mapped I/O* to perform all I/O between memory and (raw) devices. *Memory-mapped I/O* essentially replaces cache lookups with valid memory mappings, eliminating the overhead for data items that are in memory. Misses incur a page fault and require an I/O operation that happens directly from memory without copying data between user and kernel space. However, the asynchronous nature of *memory-mapped I/O* means that I/O happens at page granularity, resulting in many and small I/Os, especially for read operations. In addition, *memory-mapped I/O* does not provide any type of consistency, recoverability, nor the ability to tune I/O for specific needs. To overcome these limitations, we implement a custom *memory-mapped I/O* path, *kmmap*, as a Linux kernel module. *kmmap* addresses these issues and provides all the benefits of memory-mapped storage: it removes the need to use DRAM caching both in kernel and user space, eliminates data copies between kernel and user space, and removes the need for pointer translation.

We implement *Kreon* and evaluate its performance by using YCSB and large datasets of up to 6 billion keys. We compare *Kreon* with RocksDB [17], a state-of-the-art, LSM-Tree based, persistent key-value store which has lately been optimized for SSDs [13]. Our results show that using both datasets that stress I/O and datasets that fit in memory, *Kreon* reduces the amount of cycles/op by up to 8.3x. Additionally, *Kreon* reduces I/O amplification for insert-intensive workloads by up to 4.6x and increases ops/s by up to 5.3x. Finally, our analysis of CPU overheads shows that a saturated *Kreon* server can achieve up to 2.4M YCSB insert requests/s.

Overall, the contributions of this paper are:

(1) The combination of multilevel data organization with full indexes at each level and a fine-grain spill mechanism that all together reduce CPU overhead and I/O traffic at the expense of increased I/O randomness.
(2) The design and implementation of *kmmap* a custom *memory-mapped I/O* path to reduce the overhead of explicit I/O and address shortcomings of the native *mmap* path in Linux for modern key-value stores.
(3) The implementation and detailed evaluation of a full key-value store compared to a state-of-the-art key-value store in terms of absolute performance, CPU and I/O efficiency, execution time breakdown, tail latencies, and device behavior.

The rest of this paper is organized as follows: Section 2 presents our design and implementation of *Kreon*. Section 3 presents our evaluation methodology and experimental results. Section 4 reviews related work and Section 5 concludes the paper.

2 DESIGN

2.1 Overview

Kreon, similar to Atlas [25], Tucana [32], and Wisckey [29], stores key-value pairs in a log to avoid data movement during reorganization from level to level. It organizes its index in multiple levels of increasing size and transfers data between levels in batches to amortize I/O costs, similar to LSM-Tree. Unlike LSM-Tree, within each level, it organizes keys in a B-tree with leaves of page granularity similar to bLSM [37]. However, unlike bLSM, *Kreon* transfers data between levels via a *spill operation*, rather than full reorganization of the data in the next level. Spills are a form of batched data compaction that merge keys of two consecutive levels [L_i, $L_i + 1$]. However, spills do not read the entire L_{i+1} during merging with L_i and do not reorganize data and keys on a sequential part of the device [37]. Instead, *Kreon* spills read/write level L_{i+1} partially using the full B-tree index of each level.

The trade-off is that during spills, *Kreon* generates random read I/O requests at large queue depth (high I/O concurrency) to significantly reduce I/O traffic and CPU overhead. On the other hand write I/O requests are relative large for writing updated parts of L_{i+1} index. This is because *Kreon* B-tree uses Copy-on-Write for persistence [19] and a custom segment allocator so updated leaves are written close on the device.

Furthermore, *Kreon* uses memory mapped I/O to eliminate redundant copies between kernel and user space and constant pointer translation. *Kreon*'s *memory-mapped I/O* path is designed to provide efficient support for managing I/O memory addressing shortcomings of the default *mmap* path in the Linux kernel. These shortcomings are: (a) It does not provide explicit control over data eviction, as with an application-specific cache, (b) it results in an I/O even for pages that include garbage, and (c) it employs eager evictions to free memory, which results in excessive I/O, in order to avoid starving other system components.

Figure 2 depicts the architecture of *Kreon* showing two levels of indexes, the key-value log, and the device layout. Next, we discuss our design for the system index and *memory-mapped I/O* in detail.

2.2 Index Organization

Kreon offers a dictionary API (insert, delete, update, get, scan) of arbitrary sized keys and values stored in groups named *regions*. Each region can map either to a table or shards of the same table. For each region it stores key-value pairs in a single append-only *key-value log* [29, 32] and keeps a multilevel index. The index in each level is a B-tree [3], which consists of two types of nodes: internal and leaf nodes. Internal nodes keep a small log where they store pivots, whereas leaf nodes store key entries. Each key entry consists of a tuple with a pointer to the key-value log and a fixed-size key prefix. Prefixes are the first M bytes of the key used for key comparisons inside a leaf. They reduce significantly I/Os to the log since leaves constitute the vast majority of tree nodes. If the effectiveness of prefixes is reduced due to low entropy of the keys, existing techniques discuss how they can be recomputed [4].

During inserts, *Kreon* appends the key-value pair to the key-value log, then it performs a top-down traversal in its L_0 B-tree, from the root to the corresponding leaf, and adds a key entry to the leaf. Get operations examine hierarchically levels from L_0 to L_N and return the first match. Since inserts propagate with the same order as get operations, the version of the retrieved key is the most recent. Delete operations mark keys with a tombstone and defer the actual delete operation. During system operation we use the marked key entries for subsequent inserts that reuse the index entry and mark as free the deleted (old) key-value pair in the log. Marked and unused entries in the index are reclaimed during spills. Marked space in the log is reclaimed asynchronously, as discussed in Section 2.2.2. Update operations are similar to a combined insert and delete. Scan operations create a scanner per-level and use the index to fetch keys in sorted order. They combine the results of each level to provide a global sorted view of the returned keys.

Each region supports a single-writer/multiple-readers concurrency model. Readers operate concurrently with writers using Lamport counters [26] per tree node for synchronization. Scans, similar to other systems [17], access all data inserted to the system up to the scanner creation time and they operate on an immutable version of each tree which is facilitated by the Copy-On-Write approach used by *Kreon* (Section 2.4).

Similar to LSM-Tree, L_0 in *Kreon* always resides entirely in memory. Portions of $levels \geq 1$ are brought in memory on demand. *Kreon* enforces memory placement rules for different levels by using *kmmap* and explicit priorities (Section 2.3).

2.2.1 Spill Operations. When level i, L_i, fills up beyond a threshold, *Kreon* merges L_i into L_{i+1} via a *spill* operation. Spills are conceptually similar to LSM-Tree compactions [17, 18, 37], however, they operate differently. Spills avoid sorting

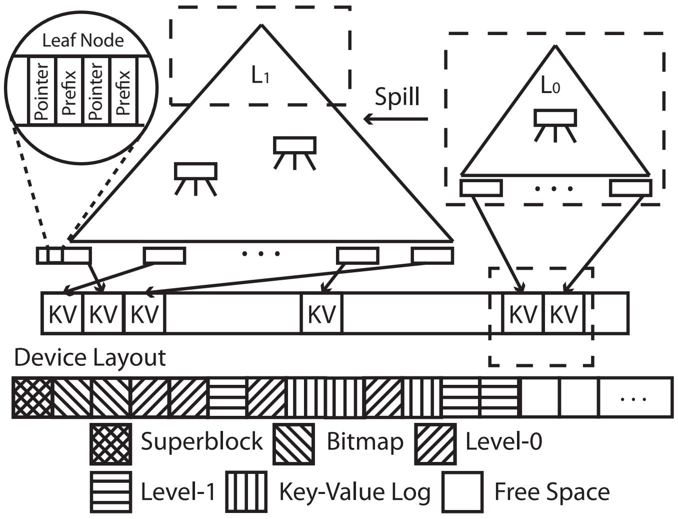


Figure 2: The main structures of *Kreon* showing two levels of indexes, the key-value log, and the device layout. Dashed rectangles include portions of the data structures that are kept in memory via *kmmap*.

by using the B-tree of the level to scan L_i keys in lexicographic order and to insert them in L_{i+1}. Spills effectively move a large portion of keys from one level to the next. This batching of insert operations results in amortizing device I/Os over multiple keys due to the lexicographic retrieval of L_i keys: *Kreon* fetches a leaf of L_{i+1} once and performs all updates in the batch related to this leaf before writing it back to storage. Furthermore, *Kreon* spills involve only metadata while data remain in the append-only log. Compared to LSM based key-value stores [17, 27, 37], where compactions move and reorganize the actual data as well, this reduces overhead at the expense of leaving unorganized data on the device.

During spills, *Kreon* produces random and relatively small read requests (4 KB) for leaves of L_{i+1}. However, due to the use of Copy-on-Write in *Kreon* (Section 2.4) writes to the next level happen always to newly allocated blocks within contiguous regions of the device, which results in efficient merging of write I/Os into larger requests. Additionally, during spills, *Kreon* creates many concurrent I/Os by using multiple spill threads.

For spills to be effective, each level needs to be able to buffer a substantial amount of keys compared to the size of the lower (and larger) level, similar to compactions in LSM-Tree. We determine empirically that buffering about 5-10% of the metadata of the next level (key-value pairs themselves are not part of the indexes) results in effective amortization of I/O operations. This growth factor of 10-20x between successive levels refers only to metadata and depends also on the distribution of the inserted keys. Zipf-like distributions, that are considered more typical today compared to uniform, behave well with buffering a (relatively) small percentage of the next level. We evaluate the impact of the growth factor in Section 3.5.

To achieve bounded latency for inserts during spills, *Kreon* allows inserts to L_0 to be performed concurrently with spills, as follows. It creates a new L_0' tree where it performs new inserts, while spilling from L_0 to L_1. Pages freed from the spill operation can be reused by the new L_0' index. Therefore, L_0' grows at the same rate as L_0 shrinks. Freeing pages from the old index and adding them to the new index involves memory unmap and remap operations (via *kmmap*) but no device I/O.

2.2.2 *Device Layout and Access.*

Kreon manages storage space as a set of segments. Each segment is a contiguous range of blocks on a device or a file. To further reduce overhead we access devices directly rather than use a file system in between. Our measurements show that files result in a 5-10% reduction in throughput due to file system overhead. Each segment hosts multiple regions and it has its own allocator to manage free space.

Kreon's allocator stores its metadata at the beginning of each segment, which consists of a *superblock* and a *bitmap*. The *superblock* keeps pointers to the latest consistent state of the segment and its regions. The *bitmap* contains information about the allocation status (free or reserved) of each 4 KB block. The *bitmap* is accessed directly via an offset and at low overhead, while for searches we use efficient bit parallel techniques [5].

Kreon allocates space eagerly for regions in large units, currently 2 MB, consuming them incrementally in smaller units. This approach avoids frequent calls to the allocator that is shared across regions in each segment. It also improves average write I/O size by letting each region grow in a contiguous part of the device.

Similarly, the key-value log in *Kreon* is organized in large chunks, also 2 MB. At the start of each chunk we keep metadata about the garbage bytes as done in other systems [30]. Delete operations update the deleted bytes counter of the corresponding chunk. When this counter reaches a threshold the valid key-value pairs are moved to the end of the log. We locate these keys in the index via normal lookups and we update the leaf pointers accordingly. Finally, we release the chunk to be available for subsequent allocations.

2.2.3 *Partial Reorganization.*

Scan operations in *Kreon* for small key-value pairs (less than 4 KB) produce read amplification due to page size access granularity. To address this, *Kreon* reorganizes data during scan operations, at leaf granularity. Reorganization takes place only for $L \geq 1$ leaves, since L_0 leaves are always in memory. During reorganization the key-value pairs belonging to the same leaf are written in a continuous region of the key-value log and their previous space is marked free. The reorganization criterion is currently based on a counter per leaf, which is incremented every time a leaf is written. During scans, if this counter exceeds a threshold (currently, half the leaf capacity) the leaf is reorganized and the counter is reset. We leave as future work additional adaptive policies for data reorganization.

2.2.4 Number of Levels. In our work, we claim that two levels in *Kreon* are adequate for most practical cases, given current DRAM and Flash prices. If we assume a growth factor R of about 10-20x between levels, we can calculate the dataset that can be handled with M bytes of memory devoted to L_0, which needs to fit in memory. If we assume that space amplification in B trees is 1.33 [24] and N keys are buffered in L_0 then the size of L_0 is $M = 1.33 * N * P_k$, where P_k is the size of the metadata for each key (pointer and prefix). *Kreon* uses 20 bytes of metadata for each key, which results in $M = 26 * N$. Similarly, the size of the dataset is $D = R*N*(S_k+S_v)$, where S_k and S_v are the size of the keys and values respectively, in the dataset. If we conservatively assume $R = 10, S_k = 10$, and $S_v = 100$, then $D = 1100 * N$ and $M/D = 0.02$. However, more typical sizes for keys and values are $S_k = 20$ and $S_v = 1000$. If we also assume $R = 20$, then $D = 20600 * N$ and $M/D = 0.001$. Assuming that the cost ratio of DRAM over Flash is about 10x per GB, then the cost of DRAM for L_0 in a 2-level *Kreon* configuration is conservatively 20% (M/D=0.02) cost of Flash to store the data and more realistically 1% (D/M=0.001) or less.

Similar to our analysis, previous work has claimed that three levels are adequate for most purposes [27, 37]. However, in previous cases the index contains the key-value pairs as well, while in *Kreon* key-value pairs are placed in a separate log, further reducing the index size. Finally, if two levels are not adequate, *Kreon* introduces additional levels to the hierarchy. In this case however, there will be a need to also provide bloom filters for avoiding out of memory lookups for all levels, similar to other systems [11, 17, 37].

2.3 Memory-Mapped I/O

Most key-value stores and other systems that handle data use explicit I/O to access storage devices or files with read/write system calls. In many cases, they also employ a user-space cache as part of the application to minimize accesses to storage devices and user-kernel crossings for performance purposes. The use of a user-space cache is important to avoid frequent system calls for lookup operations that need to occur for every data item, regardless if it eventually hits or misses. However, even the use of an application user-level cache incurs significant overhead in the common path [21, 22, 32].

The use of *memory-mapped I/O* in *Kreon* reduces CPU overhead related to the I/O cache in three ways: (a) It eliminates cache lookups for hits by using valid virtual page mappings. Memory-mapped I/O does not require cache lookups because virtual memory mappings distinguish data that are present in memory from data that are only located on the device. All device data are mapped to the application address space but only data that are present in memory have valid virtual memory mappings. Accesses to data that are not present in memory result in page faults that are then handled by *mmap*. Given that many operations in key-value stores, such as get operations with a Zipf distribution, complete from memory, *Kreon* avoids all related cache lookup overheads. (b) There is no need to copy data between user and kernel space when performing I/O. Pages used for data in memory are used directly to perform I/O to and from the storage devices. (c) There is no need to serialize/deserialize data between memory and the storage devices. Finally, *memory-mapped I/O* uses a single address space for both memory and storage, which eliminates the need for pointer translation between memory and storage address spaces and therefore, the need to serialize and deserialize data when transferring between the two address spaces.

2.3.1 Kreon's Memory-Mapped I/O. *Kreon* provides its own custom *memory-mapped I/O* path to address the shortcomings of *mmap* in Linux.

First, in *mmap* there is no explicit control over data eviction, as with an application-specific cache. Linux uses an LRU-based policy, which may evict useful pages, for instance, pages of L_0 instead of L_1 pages. L_0 has to reside in main memory to amortize write I/O operations. Linux *mmap* does not provide a mechanism to achieve this. A possible solution is to lock important pages with *mlock*. However, Linux does not allow a large number of pages to be locked by a single process because this affects other parts of the system.

Second, each write operation in an empty page is effectively translated to a read-modify-write because *mmap* does not have any information about the status (allocated or free) of the underlying disk page and the intended use. This results in excessive device I/O. Instead, if applications can inform *mmap* whether a page contains garbage and will be written entirely, *mmap* can map this page without reading it first from the device, eliminating unnecessary read traffic.

Third, *mmap* employs aggressive evictions based on memory usage and time elapsed since pages marked as dirty to free memory and avoid starving other system components. Mapping large portions of the application virtual address space creates pressure to the virtual memory subsystem and results in unpredictable use of memory and bursty I/O. Furthermore, eager and uncoordinated evictions do not facilitate the creation of large I/Os through merging. Empirically, we often observe large intervals (of several 10s of seconds) where the system freezes while it performs I/O with *mmap* and applications do not make progress. Furthermore, we observe similar behaviour with *msync*. This unpredictability and large periods of inactivity are an important problem for key-value stores that serve data to online, user-facing applications.

To overcome these limitations, we implement a custom *mmap*, as a Linux kernel module, called *kmmap*. Figure 3 shows the overall design and data structures of *kmmap*.

Kmmap bypasses the Linux page cache and uses a priority-based FIFO replacement policy. As priority we define a small, per-page number (0 to 255). During memory pressure, a page with a higher priority is preferred for eviction. Priorities are kept only in memory and are set explicitly by *Kreon* with *ioctl* calls. Priorities are set as follows: *Kreon* assigns priority 0 to index nodes of L_0, 1 to index nodes of L_1, 2 to leaf nodes of L_1, and 3 to the log. L_0 fits in memory and it will not be evicted. Generally if we have more than two levels L_0 always uses priority 0 and the log maximum priority. We calculate

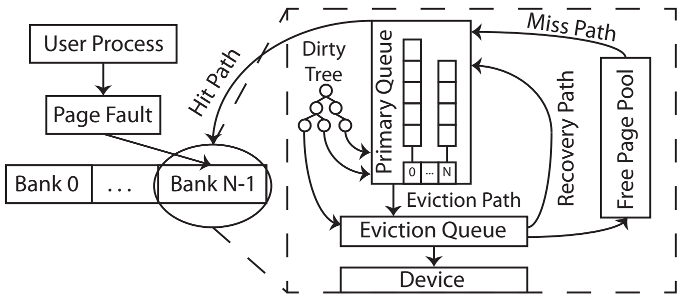


Figure 3: The main structures of *kmmap*.

the priority of level L_N as $(2*N-1)$ for index nodes and $(2*N)$ for leaves.

To increase parallelism, *kmmap* organizes memory in independent banks, similar to DI-MMAP [14]. Pages are mapped to banks by hashing the page fault address. To place consecutive pages in the same bank, the page fault address is first shifted. Unlike DI-MMAP, *kmmap* uses fine-grain locking inside banks, which results in higher concurrency and eliminates periods of inactivity (long freezes).

When *Kreon* accesses a page (for read or write), that does not reside in main memory, a page fault occurs. On a page fault, *kmmap* retrieves a free page from an in-memory list (*Free Page Pool*), it reads the data from the device if required, and finally enqueues the page to the *Primary Queue* based on its priority. *kmmap* keeps a separate FIFO per priority inside the *Primary Queue*. In the case where the *Primary Queue* is full of pages, it dequeues a fixed number of entries for batching purposes, with preference to entries with higher priority. Then it unmaps them from the process address space and moves them into the *Eviction Queue*. The *Eviction Queue* is organized as an in-memory red-black tree structure, keeping keys sorted based on page offset at the device. For evictions, it traverses the *Eviction Queue* and merges consecutive pages to generate as large I/Os as possible. It keeps dirty pages that belong to the *Primary Queue* or the *Eviction Queue* in another in-memory red-black tree structure (*Dirty Tree*) sorted by their device offset. The *Dirty Tree* is used by *msync*, to avoid scanning unnecessary (clean) pages.

Kmmap compared to *mmap* keeps pages in memory for a longer period of time and does not evict them, unless there is a need to do so. This allows *Kreon* to generate larger I/Os during spill operations by merging more requests. When a spill is completed, *Kreon* sets the priority of pages from the previously spilled L_0 to 255 (smallest priority) so they get evicted as soon as possible.

To avoid unnecessary reads that occur when a new page is written in *Kreon*, *kmmap* detects and filters these read-before-write operations, whereas write and read-after-write operations are forwarded to the actual device. To achieve this, it uses an in-memory bitmap, which is initialized and updated by *Kreon* via a set of *ioctl* calls. The bitmap uses a bit per device block, so a 1 TB SSD requires 32 MB of memory for the bitmap.

Kmmap provides a non-blocking *msync* call that allows the system to continue operation while pages are written asynchronously to the devices. For this purpose we keep a timestamp for each page that indicates when it became dirty. To write dirty pages, we iterate the *Dirty Tree* and write only pages with timestamp older than the timestamp of *msync*. We use fine grain locking in *Dirty Tree* and we allow to add new dirty pages into it during *msync*. However, there can be pages that are already dirty and changed after *msync*, which should not be written. *Kreon* uses Copy-On-Write to ensure that after a commit dirty pages will not change again as we need to allocate new pages.

Finally, *Kreon* significantly reduces unpredictability with respect to memory management during system operation by limiting the maximum amount of memory it occupies throughout its operation. It uses a configuration parameter to calculate the size of L_0 in memory and based on this it preallocates all *memory-mapped I/O* structures.

2.4 Persistence

Kreon uses Copy-On-Write (CoW) [35] to maintain its state consistent and recoverable after failures. *Kreon*'s state includes the data section of each segment (metadata and data of the tree) and the allocator metadata. To persist a consistent version of its state *Kreon* provides a commit operation. This operation first writes the dirty (in-memory) data into the device and then switches atomically from the old state to the new state. More specifically, *Kreon* stores a pointer to the latest persistent state in the superblock. At the end of a commit operation, *Kreon* updates this pointer to the newly created persistent state which becomes immutable. In case of a failure, the new state that is not committed will be discarded during startup, resulting in a rollback to the last valid state.

In *Kreon* we use CoW for different purposes at L_0 and the rest of the levels. The index of all levels except L_0 is kept on the device and only brought to memory on demand. Therefore, typically, only a small part of these indexes is in memory. For these indexes, *Kreon* uses CoW to ensure consistency of the index on the device during failures. These levels are only written to the device during spills. Therefore, the only time when commits occur (besides L_0), is at the end of each spill operation.

L_0 is different and can always be recovered by replaying a subset of the key-value log. This subset is always the latest portion of the log and is easy to identify via markers placed in the log during the spill operation from L_0 to L_1. Therefore, after a failure, L_0 can be reconstructed. However, L_0 can grow significantly due to the large amount of memory available in modern servers. *Kreon* uses CoW to checkpoint L_0 to the device and to reduce recovery time. Therefore, *Kreon*'s commits of L_0 are not critical for recovery. L_0 checkpoints do not have to be very frequent. Infrequent L_0 commits do not lead to data loss because the L_0 index can be reconstructed through the replay of the key-value log. The log is written

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write
G	100% scans

Table 1: Workloads evaluated with YCSB. All workloads use a query popularity that follows a Zipf distribution except for D that follows a latest distribution.

to the device more frequently, when a log segment (2 MB) becomes full.

Essentially, *Kreon* uses L_0 commits at a coarse granularity to improve recovery time, without however, a negative impact on the recovery point. The tradeoff introduced is that commits incur overhead during failure free operation. Overall, we expect that *Kreon* L_0 commits will be issued periodically at a time scale of minutes, which has a low impact on performance. Section 3.5 evaluates commit overhead in *Kreon*.

3 EXPERIMENTAL RESULTS

In this section we evaluate *Kreon* against RocksDB [16, 17]. Our goal is to examine the following aspects of *Kreon*:

(1) What is the efficiency in cycles/op achieved by *Kreon* compared to LSM-based key-value stores? Does higher efficiency come at the cost of worse absolute throughput or latency?
(2) How much does the new index design and *memory-mapped I/O* contribute to reducing overheads?
(3) How does *Kreon* improve I/O amplification? How much does it increase I/O randomness?
(4) How do the growth factor across levels and L_0 checkpoint interval affect performance?

Next, we discuss our methodology and each aspect of *Kreon* in detail.

3.1 Methodology

Our testbed consists of a single server which runs the key-value store and the YCSB client. The server is equipped with two Intel(R) Xeon(R) CPU E5-2630 v3 CPUs running at 2.4 GHz, with 8 physical cores and 16 hyper-threads, for a total of 32 hyper-threads and with 256 GB DDR4 at 2400 MHz. It runs CentOS 7.3 with Linux kernel 4.4.44. During our evaluation we scale-down DRAM as required by different experiments. The server has six Samsung 850 PRO 256 GB SSDs, organized in a RAID-0 using Linux *md* and 1 MB chunk size.

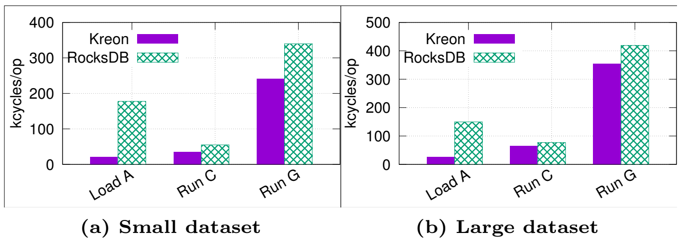


Figure 4: Efficiency of *Kreon* and RocksDB in cycles/op.

We use RocksDB[1] v5.6.1, on top of *XFS* with disabled compression and jemalloc [15], as recommended. We configure RocksDB to use direct I/O because we evaluate experimentally that in our testbed results in better performance. Furthermore, we use RocksDB's user-space LRU cache, with 16 and 192 GB depending on the experiment. We use a C++ version of YCSB [34] with the standard workloads proposed by YCSB [9, 10]. Table 1 summarizes these workloads. We add a new workload named *G* which is similar to *E* but consists only of scans. In all cases we use 128 YCSB threads for each client and 32 regions.

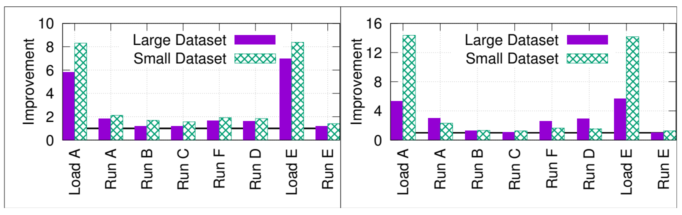
We emulate two datasets a small dataset that fits in memory and a large dataset that does not by using two different memory configurations for our system. In the small dataset we boot the server with 194 GB of memory, 192 GB for key-value store and 2 GB for the OS. For the large dataset, and to further stress I/O we boot the server with 18 GB of memory, 16 GB for key-value store and 2 GB for the OS. The dataset consists of 100M records and requires about 120 GB of storage. YCSB by default generates 10 columns for each key. We keep these 10 columns inside a single value. We use a 100M keys (recordcount and operationcount equals to 100M) * 10 columns which results in 1 billion columns.

In the small dataset, both the key-value log and the indexes fit in memory, so I/O is generated by commit operations. In the large dataset, neither the key-value log nor the indexes fit in memory and only L_0 is guaranteed to reside in memory. Therefore, the small dataset demonstrates more clearly overheads related to memory accesses whereas the large dataset stresses the I/O path.

We calculate efficiency in cycles/op as follows:

$$cycles/op = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}},$$

where $CPU\_utilization$ is the average of CPU utilization among all processors, excluding idle and I/O wait time, as given by *mpstat*. As $cycles/s$ we use the per-core clock frequency. $average\_ops/s$ is the throughput reported by YCSB, and *cores* is the number of system cores including hyper-threads.



(a) Efficiency (b) Throughput

Figure 5: Efficiency and throughput improvement of *Kreon* compared to RocksDB for all YCSB workloads.

3.2 CPU Efficiency and Performance

We evaluate the efficiency of *Kreon* in terms of cycles/op required to complete each operation, excluding YCSB overhead. To exclude the overhead of the YCSB client, we profile the average cycles/op required by YCSB and we subtract this overhead from the overall value for both RocksDB and *Kreon*.

Figure 4 shows our overall results for *Kreon* and RockDB. For the small dataset *Kreon* requires 8.3x, 1.56x, and 1.4x fewer cycles/op for *Load A*, *Run C*, and *Run G*, respectively. For the large dataset *Kreon* requires 5.82x, 1.2x, and 1.18x fewer cycles/op for *Load A*, *Run C*, and *Run G*, respectively. In addition, for the small dataset and *Load A* we compare *Kreon* when using *kmmap* and when using vanilla *mmap*. Although we do not show these results for space purposes, using *kmmap*, *Kreon* achieves 1.47x fewer cycles/op compared to vanilla *mmap*, indicating the importance of proper and customized *memory-mapped I/O* for key value stores.

In terms of absolute numbers, we see that *Kreon* requires 21, 35, and 241 kcycles/op for each of *Load A*, *Run C*, and *Run G* for the small dataset and 25, 64, and 354 kcycles/op for each of *Load A*, *Run C*, and *Run G* for the large dataset.

We now show results from a complete run for all YCSB workloads. We run the workloads in the recommended sequence [9]: Load the database using the configuration file of workload A, run workloads A, B, C, F, and D in a row, delete the whole database, reload the database with the configuration file of workload E and finally run workload E.

For both the small and large dataset, Figure 5a shows the improvement in efficiency compared to RocksDB, whereas Figure 5b shows the improvement in throughput. Regarding efficiency, *Kreon* improves RocksDB efficiency, on average, by 3.4x and 2.68x, for the small and large dataset, respectively. Regarding throughput, the improvement in *Kreon* compared to RocksDB is, on average, 4.72x and 2.85x for the small and large datasets, respectively.

3.2.1 Latency analysis. First, we examine the average latency per operation for the small dataset. For *Load A*, RocksDB achieves 1162 μs/op, *Kreon* with vanilla *mmap* achieves 346 μs/op, and *Kreon* with *kmmap* achieves 72 μs/op.

[1]Options file: https://goo.gl/NJNLNr.

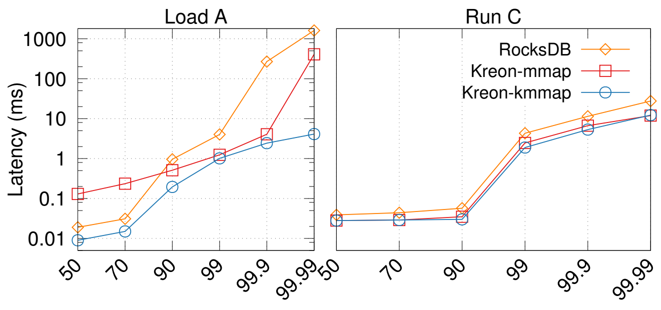


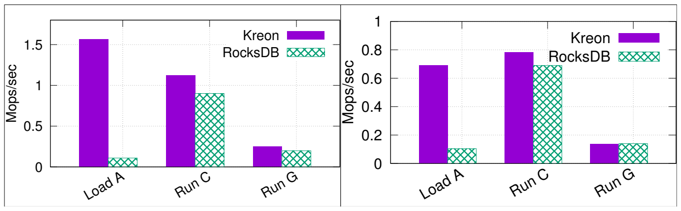
Figure 6: Tail latency for Load A and Run C for RocksDB, *Kreon* with vanilla *mmap*, and *Kreon* with *kmmap*.

This shows that *kmmap* provides significant reduction in latencies compared to vanilla *mmap*. For *Run C*, RocksDB achieves 174 μs/op, *Kreon* with vanilla *mmap* achieves 119 μs/op, and *Kreon* with *kmmap* achieves 109 μs/op. Generally, *Kreon* with *kmmap* achieves 16.1x and 1.5x lower latency on average for *Load A* and *Run C* compared to RocksDB.

Figure 6 shows the tail latency for *Kreon* using both *kmmap* and vanilla *mmap* and RocksDB. For *Load A*, for 99.99% of requests, *Kreon* with *kmmap* achieves 393x lower latency compared to RocksDB. Furthermore, *kmmap* results in 99x lower latency compared to vanilla *mmap*. In our design we remove blocking for inserts during *msync* and during spilling of L_0. Unlike *Kreon*, RocksDB blocks inserts during compaction operations for longer periods. For *Run C*, *Kreon* results in almost the same latency with and without *kmmap* and about 2x better than RocksDB. This is because in a read-only workload most overheads comes from the data structure, as we use a dataset that fits in memory and removes the need for I/O. In the case of RocksDB this overhead includes also a cache lookup while in *Kreon* it only accesses already mapped memory. The use of *mmap* and *kmmap* results in almost the same performance as this experiment does not stress *memory-mapped I/O* path.

3.2.2 Very large dataset. To examine *Kreon*'s behavior with a very large dataset we run *Load A* using 6 billion keys with one column per key (key size of 30 bytes and value size of 100 bytes). For this experiment we use 192 GB of DRAM for both *Kreon* and RocksDB. *Kreon* reduces cycles/op by 8.75x, increases ops/s by 12.11x, reduces write volume by 4.25x, and read volume by 3.14x.

3.2.3 Absolute operation throughput. Next, we examine if *Kreon*'s increased efficiency in cycles/op comes at the expense of reduced absolute performance. This is important for understanding if *Kreon* trades device and host CPU efficiency in the right manner. For *Kreon* and RocksDB, Figure 7 shows the throughput (ops/s), achieved by YCSB. For the small dataset, *Kreon* achieves 14.35x, 1.24x, and 1.25x more ops/s for *Load A*, *Run C*, and *Run G*, respectively.



(a) Small dataset **(b) Large dataset**

Figure 7: Throughput for *Kreon* and RocksDB in ops/s.

For the large dataset, *Kreon* achieves 5.33x and 1.05x more ops/s for *Load A* and *Run C*, respectively, than RocksDB. However, *Kreon* is 2% worse for *Run G*. In this case, both RocksDB and *Kreon* are limited by device throughput and this is the reason that both systems are comparable. On the other hand, *Kreon* results in much lower CPU utilization: on average *Kreon* has a utilization of 13.8% while RocksDB has a utilization of 39.5%. Therefore, *Kreon* is able to support more clients given an adequate number of storage devices.

For the small dataset and *Load A*, we compare *Kreon* with *kmmap* and with vanilla *mmap*. We see that *kmmap* improves throughput by 4.34x compared to vanilla *mmap*.

3.3 Execution Time Breakdown

In this section we examine the main components that contribute to overhead in *Kreon* and RocksDB. Our purpose is to identify what are the main sources of improvement in *Kreon* compared to RocksDB and what are the remaining sources of overhead.

We examine two workloads a write-intensive (Load A) and a read-intensive (Run C) using both the small and large datasets. We profile *Load A* and *Run C* workloads and we use stack traces from *perf* and Flamegraph [20] to identify where cycles are spent. We divide overhead in the following components: index operations (updates/traversals for put/get operations), caching, I/O, and compaction/spill. I/O refers to explicit I/O operations in RocksDB and *memory-mapped I/O* in *Kreon*. Caching refers to the cycles needed for cache lookups, fetching new data for misses and page evictions when the cache becomes full. RocksDB uses a user-space LRU cache whereas in *Kreon* cache resides in kernel-space as part of *kmmap*.

Table 2 shows the breakdown for the write-intensive *Load A* workload. The number of cycles used by the YCSB client is roughly the same in all cases. In the small workload, index manipulation incurs about 44% lower overhead in *Kreon* (∼13K cycles/op in *Kreon* vs. 24K cycles/op in RocksDB). Caching overhead for the write-intensive workload is lower for the large dataset whereas for the small dataset *Kreon* spends more 0.23 Kcycles/op. For I/O *Kreon* requires 61% fewer cycles. For compaction/spill *Kreon* dramatically reduces the cycles required per operation from 63.41K to 0.78K. In the large workload, index manipulation requires 51% fewer cycles in *Kreon* (from 26K to 13K) and for I/O 80% fewer cycles. Similarly to the small dataset, *Kreon* significantly reduces the number of cycles per operation for compaction/spill from 60.87K to 0.64K.

kcycles/ operation	Load A (16GB)			Load A (192GB)		
	RocksDB	Kreon	Impro vement	RocksDB	Kreon	Impro vement
index	24.15	13.46	44%	26.76	13.1	51%
cache	0.33	0.56	-69%	0.82	0.45	45%
I/O pfault	2.92	5.84	61%	1.66	2.61	80%
I/O syswrite	12.20	0		11.91	0	
compaction/spill	63.41	0.78	98%	60.87	0.64	98%
Total	103.1	20.64	79%	102.02	16.8	83%
YCSB	26.67	25.34	-	22.79	21.37	-

Table 2: Breakdown of cycles per operation for workload Load A (write only). Numbers are in kcycles.

kcycles/ operation	Run C (16GB)			Run C (192GB)		
	RocksDB	Kreon	Impro vement	RocksDB	Kreon	Impro vement
index	4.87	4.28	12.3%	25.59	10.29	59%
cache	8.61	0.41	95%	9.79	0.74	92%
I/O pfault	0.12	3.16	-6%	0.54	5.9	23%
I/O sysread	2.86	0		7.21	0	
Total	16.46	7.85	52%	43.13	16.93	60%
YCSB	13.9	12.11	-	54.04	53.11	-

Table 3: Breakdown of cycles per operation for workload Run C (read only). Numbers are in kcycles.

Table 3 shows the breakdown for the read-intensive workload (*Run C* benchmark). In the small dataset, index manipulation incurs 12% fewer cycles (from 4.87K in RocksDB to 4.28K in *Kreon*). Caching overhead is reduced by 95% (from 8.61K cycles/op in RocksDB to 0.41K cycles/op in *Kreon*) whereas I/O requires 6% more cycles in *Kreon*. In the large dataset, index manipulation overhead is reduced by 59% in *Kreon*, caching overhead by 92%, and I/O by 23%.

Overall, we see that *Kreon*'s design significantly reduces overheads for index manipulation, spills, and I/O. We also see that all proposed mechanisms for indexing, spills that involve only metadata, and *memory-mapped I/O*-based caching, have important contributions. Finally, we see that in *Kreon* the largest number of cycles is consumed by index manipulation (up to 13K cycles/op) both for both datasets in both workloads and secondarily by page faults (up to 5.9K cycles/op).

3.4 I/O Amplification and Randomness

In this section we evaluate how *Kreon* reduces amplification at the expense of reduced I/O size and increased I/O randomness. To reduce amplification, *Kreon* generates by design smaller and more random I/Os compared to RocksDB and traditional LSM trees. We measure the average request size for *Load A* using the large dataset. For writes, *Kreon* has an average request size of 94 KB compared to 333.2 KB for RocksDB. However, even at 94 KB, most SSDs exhibit high throughput with a large queue depth (Figure 1). For reads, *Kreon* produces 4 KB I/Os, compared to 126 KB for RocksDB. Because of compactions, RocksDB reads large

	Load A	Run C	Run G
RocksDB-Read	669	138	296
Kreon-Read	112	127	1237
RocksDB-Write	869	0	8
Kreon-Write	221	0	139

Table 4: Total I/O volume (in GB) for Load A, Run C, and Run G using the large dataset.

	R_t	R_r	R_w
RocksDB	0.001780	0.003878	0.000112
Kreon	0.009851	0.033648	0.000325

Table 5: I/O randomness using the large dataset and *Load A*. The higher the value of R, the more random the I/O pattern.

chunks of data in order to merge them. This results in a large request size but it also results in high read amplification, 4.8x more data compared to *Kreon*.

Table 4 shows the total amount of traffic to the device using the large dataset. We see that for *Load A Kreon* reduces both read traffic by 5.9x and write traffic by 3.9x, while the total traffic reduction is 4.6x. *Kreon* reads 1.08x less data for *Run C*. On the other hand, *Kreon* reads 4.1x more data for *Run G*, due to data re-organization. This cost is related only to scans and for leaves that are not re-organized. On the other hand, in RocksDB data reorganization takes place in every compaction.

To examine randomness, we implement a lightweight I/O tracer as a stackable block device in the Linux kernel that keeps the device offset and size for *bios* issued to the underlying device. The tracer stores this information to a ramdisk to reduce overhead and avoid interfering with the key-value store I/O pattern. Tracing reduces average throughput of YCSB by about 10%. We analyze traces after each experiment and calculate a metric for I/O randomness based on the distance and size of successive *bios*, as follows:

$$R = \frac{\sum_{i=0}^{nb-1} |bs[i+1].off - (bs[i].off + bs[i].size)| + bs[i].size}{device\_size\_in\_pages * \sum_{i=0}^{nb-1} bs[i].size},$$

where bs is the array that contains bio information and nb its length. R is the randomness metric and takes values between [0,1]. The larger R is, the more random the I/O pattern. Finally, we compute three versions of R, one for all *bios* (R_t), one for reads (R_r), and one for writes (R_w).

Table 5 shows our results for *Kreon* and RocksDB. For calibration purposes, we run *fio* with queue depth of 1 and block size of 4 KB: a sequential pattern is 0 and a random pattern is close to 0.33. *Kreon* produces overall about 5.53x more random I/O patterns than RocksDB. Reads exhibit a larger difference in randomness, about 10x, because *Kreon* moves data between levels at smaller granularity than RocksDB. For writes, *Kreon* exhibits a 3x more random pattern.

Overall, during inserts, *Kreon* reduces write traffic by 2.8x and read traffic by 4.8x. In both cases, queue depth is about 30 on average. Figure 1 shows that, at this queue depth, commodity SSDs achieve their maximum throughput with at 32 KB requests, so *Kreon*'s 94 KB write requests result in little or no loss of device efficiency, while there is a 2.8x gain from reduced write traffic. For read traffic, *Kreon*'s 4K requests result in a small percentage drop of SSD throughput at a queue depth of 32, but at a 4.8x gain in traffic. Therefore, *Kreon* properly trades randomness and request size for amplification. The calculation is somewhat different for our NVMe devices, but still favorable to *Kreon*.

Finally, *Kreon* achieves an average read throughput of 123 MB/s and an average write throughput of 743 MB/s at an average queue depth of 21.2. On the other hand RocksDB achieves 707 MB/s for reads and 889 MB/s for writes at an average queue size of 26.2. In both cases queue depth is large enough for devices to operate at high throughput, although *Kreon* exhibits lower throughput for reads due to the smaller request sizes it generates. This loss of device efficiency is compensated by the reduced amplification (by 4.6x) and the reduced CPU overhead, eventually resulting in higher performance and data serving density.

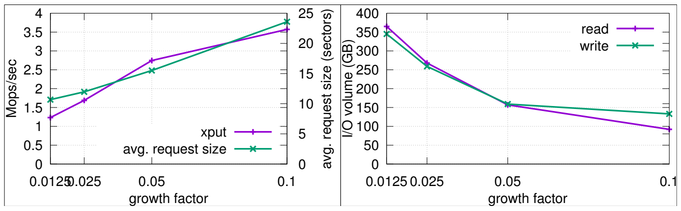
3.5 Growth Factor and Commit Interval

An important parameter for key value stores that use multi-level indexes is the ratio of the size between two successive levels (growth factor). The growth factor in *Kreon* represents the amount of buffering that happens for inserts in one level before keys are spilled to the next level. This affects how effectively I/Os are amortized across several inserts and reduces write amplification.

Figure 8 shows *Load A* with varying growth factor using the large dataset. A growth factor of 0.1 means that L_1 is 10x larger than L_0 and therefore L_0 can buffer about 10% of the keys in L_1. Figure 8b shows that a growth factor between 0.05 and 0.1 is appropriate, meaning that each level should buffer between 5-10% of the next level. A smaller growth factor results in significant increase in traffic and reduces device efficiency. Increasing the growth factor beyond 0.1 reduces traffic further, however, this also requires more memory for L_0. Figure 8a (right y-axis) shows that average request size increases as buffering increases and combined with the reduced traffic, results in increasing throughput (ops/s), as shown in Figure 8a (left y-axis).

Figure 9 shows how the commit interval for L_0 affects ops/s, read volume, and write volume in *Kreon*. For *Run C* the commit interval does not affect any of the metrics, therefore, we examine only *Load A* with the large dataset.

Increasing the commit interval decreases the total amount of data read and written to the device. This is due to Copy-on-Write. For each commit we create a read-only version of our tree, thus an insert has to allocate new nodes and copy data from the immutable copy. Additionally, we see that



(a) xput & avg. request size **(b) I/O volume**

Figure 8: Results with varying growth factor from 1.25% to 10% (x-axis) using the large dataset.

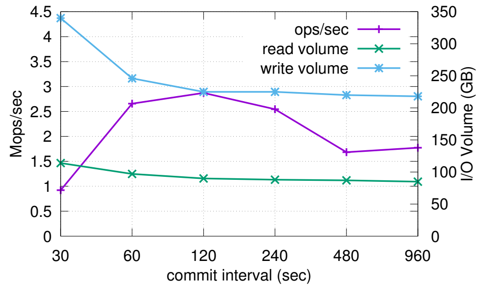


Figure 9: Results with varying the commit interval (x-axis) for *Load A* and the large dataset.

commit intervals longer than 120s have a small impact for read and write volume.

For throughput, a small commit interval results in larger read and write volume which reduces performance. Interestingly, a value larger than 240 seconds reduces throughput significantly as well. This is due to the behavior of *msync*. In *kmmap*, *msync* is optimized to generate many large and asynchronous I/Os from all dirty pages, which means that it is more efficient compared to the eviction path *mmap* where we evict less amount of data. Overall, we see that a good value for the commit interval is about 2 minutes, which we use in all our other experiments.

4 RELATED WORK

bLSM [37] uses a B-tree index per level and bloom filters to reduce read amplification. It also introduces gear scheduling, a progress-based compaction scheduler that limits write latency. *Kreon* shares the idea of a B-tree index per level but keeps an index only for the metadata and it does not fully rewrite levels during spills trading I/O randomness for CPU efficiency. FD-tree [27] is an LSM tree for SSDs, which uses fractional cascading [6] to reduce read amplification. VT-tree [38] reduces I/O amplification by merging sorted segments of non-overlapping levels of the tree. LSM-trie [41] uses a hashing technique to replace sorting but does not support range queries. Contrary to these systems, *Kreon* replaces sorting with indexing and introduces a spill mechanism to reduce CPU overheads and I/O amplification.

Atlas [25] is a key-value store that aims to improve data-serving density and data replica space efficiency. To achieve these, Atlas employs an LSM–based approach and separates keys from values to avoid moving values during compactions. Similarly, WiscKey [29] proposes the separation of keys and values to reduce write amplification. It stores values in a data log and keeps an LSM index for the keys. Furthermore, it implements a prefetching mechanism for speeding up range queries because values are written randomly on the device.

PebblesDB [33] identifies as the main problem of write amplification in the LSM-tree the repeated merges of files at each level during compaction. To fix this, it keeps overlapping sorted files at each level instead of non-overlapping. However, this approach adds overhead in the read path since multiple files need to be checked instead of a single. To improve this, PebblesDB introduces guards which act as a coarse grain index per level inspired by skip lists. *Kreon* shares the idea of using an index per level with the differnce that in *Kreon* case is full. Furthermore, it uses *memory-mapped I/O*, keeps both keys and values on a separate log, and executes spill operations only on pointers to keys and values.

TokuDB [40] implements at its core a B$^{\epsilon}$–Tree structure. It keeps a global B-tree index in which it associates a small buffer per B-tree node. Buffers are relatively small so it keeps them unsorted and scans them during look-up queries. When a buffer fills it is spilled to its N children, where N is the fan out of the B-tree. Tucana [32] uses a B$^{\epsilon}$–Tree which buffers keys only at the last level of the tree and relies on a ratio of memory/data to operate efficiently. *Kreon* keeps a buffer per level in order to achieve better batching and is able to server larger datasets with smaller memory/data ratio.

DI-MMAP [14] proposes an alternative FIFO based replacement policy that targets data-intensive HPC applications. *kmmap* shares the same goals as DI-MMAP and introduces priorities for pages in memory. This gives applications fine grain control similar to user-space application specific caches. Authors in [39] optimize the free page reclamation procedure and make use of extended vectored I/O to reduce the overhead of write operations. Finally, in [8] the authors propose techniques that reduce the overhead of page faults and page-table construction. These techniques are orthogonal to our design and they can be used in *Kreon* as well.

5 CONCLUSIONS

In this paper, we design *Kreon*, a persistent key-value store based on LSM trees that uses an index within each level to eliminate the need for sorting large segments and uses a custom *memory-mapped I/O* path to reduce the cost of I/O. Compared to RocksDB, *Kreon* reduces CPU overhead by up to 8.3x, I/O amplification by up to 4.6x at the expense of increasing randomness of I/Os. Both index organization and *memory-mapped I/O* contribute significantly to the reduction of CPU overhead, while index manipulation and page faults emerge as the main components of per operation cost in *Kreon*.

ACKNOWLEDGMENTS

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the Vineyard (GA 687628) and ExaNeSt (GA 671553) projects. Finally, we thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] Apache. 2018. HBase. https://hbase.apache.org/.

[2] Jens Axboe. 2017. Flexible I/O Tester. https://github.com/axboe.

[3] Rudolf Bayer and Edward McCreight. 2002. *Organization and maintenance of large ordered indexes*. Springer.

[4] Philip Bohannon, Peter Mcllroy, and Rajeev Rastogi. 2001. Main-memory Index Structures with Fixed-size Partial Keys. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/375663.375681

[5] Randal Burns and Wayne Hineman. 2001. A bit-parallel search algorithm for allocating free space. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on.* IEEE, 302–310.

[6] Bernard Chazelle and Leonidas J Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 1 (1986), 133–162.

[7] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1802–1813.

[8] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotstorage17/program/presentation/choi

[9] Brian F. Cooper. 2018. Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[11] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[13] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[14] Brian Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. 2015. DI-MMAP–a Scalable Memory-map Runtime for Out-of-core Data-intensive Applications. *Cluster Computing* 18, 1 (March 2015), 15–28. https://doi.org/10.1007/s10586-013-0309-0

[15] Jason Evans. 2018. jemalloc. http://jemalloc.net/.

[16] Facebook. 2015. RocksDB Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks.

[17] Facebook. 2018. RocksDB. http://rocksdb.org/.

[18] Google. 2018. LevelDB. http://leveldb.org/.

[19] Goetz Graefe. 2004. Write-optimized B-trees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 672–683. http://dl.acm.org/citation.cfm?id=1316689.1316748

[20] Brendan Gregg. 2016. The Flame Graph. *Queue* 14, 2, Article 10 (March 2016), 20 pages. https://doi.org/10.1145/2927299.2927301

[21] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713

[22] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, et al. 2012. MonetDB: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering* 35, 1 (2012), 40–45.

[23] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 301–315. https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen

[24] B Kuszmaul. 2014. A comparison of fractal trees to log-structured merge (LSM) trees. *White Paper* (2014).

[25] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data.. In *MSST*. IEEE Computer Society, 1–14. http://dblp.uni-trier.de/db/conf/mss/msst2015.html#LaiJYLSHCC15

[26] Leslie Lamport. 1977. Concurrent reading and writing. *Commun. ACM* 20, 11 (1977), 806–811.

[27] Y. Li, B. He, Q. Luo, and K. Yi. 2009. Tree Indexing on Flash Disks. In *2009 IEEE 25th International Conference on Data Engineering*. 1303–1306. https://doi.org/10.1109/ICDE.2009.226

[28] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. 2012. Scale-out Processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 500–511. http://dl.acm.org/citation.cfm?id=2337159.2337217

[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[30] Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '99)*. USENIX Association, Berkeley, CA, USA, 43–43. http://dl.acm.org/citation.cfm?id=1268708.1268751

[31] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[32] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 537–550. https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis

[33] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[34] Jinglei Ren. 2016. YCSB-C. https://github.com/basicthinker/YCSB-C.

[35] Ohad Rodeh. 2008. B-trees, Shadowing, and Clones. *Trans. Storage* 3, 4, Article 2 (Feb. 2008), 27 pages. https://doi.org/10.1145/1326542.1326544

[36] Allen Samuels. 2018. The Consequences of Infinite Storage Bandwidth. https://goo.gl/Xfo7Lu.

[37] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 217–228.

https://doi.org/10.1145/2213836.2213862
[38] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building Workload-Independent Storage with VT-Trees. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, San Jose, CA, 17–30. https://www.usenix.org/conference/fast13/technical-sessions/presentation/shetty
[39] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. 2016. Efficient Memory-Mapped I/O on Fast Storage Device. *Trans. Storage* 12, 4, Article 19 (May 2016), 27 pages. https://doi.org/10.1145/2846100
[40] INC TOKUTEK. 2013. TokuDB: MySQL Performance, MariaDB Performance.
[41] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu