The End of Moore's Law and the Rise of The Data Processor

Niv Dayan
Pliops
Ramat Gan, Israel
nivd@pliops.com

Moshe Twitto
Pliops
Ramat Gan, Israel
moshet@pliops.com

Yuval Rochman
Pliops
Ramat Gan, Israel
yuvalr@pliops.com

Uri Beitler
Pliops
Ramat Gan, Israel
urib@pliops.com

Itai Ben Zion
Pliops
Ramat Gan, Israel
itaib@pliops.com

Edward Bortnikov
Pliops
Ramat Gan, Israel
ebortnik@pliops.com

Shmuel Dashevsky
Pliops
Ramat Gan, Israel
shmueld@pliops.com

Ofer Frishman
Pliops
Ramat Gan, Israel
oferf@pliops.com

Evgeni Ginzburg
Pliops
Ramat Gan, Israel
evgenig@pliops.com

Igal Maly
Pliops
Ramat Gan, Israel
igalm@pliops.com

Avraham (Poza) Meir
Pliops
Ramat Gan, Israel
pozam@pliops.com

Mark Mokryn
Pliops
Ramat Gan, Israel
markm@pliops.com

Iddo Naiss
Pliops
Ramat Gan, Israel
iddon@pliops.com

Noam Rabinovich
Pliops
Ramat Gan, Israel
noamr@pliops.com

ABSTRACT

With the end of Moore's Law, database architects are turning to hardware accelerators to offload computationally intensive tasks from the CPU. In this paper, we show that accelerators can facilitate far more than just computation: they enable algorithms and data structures that lavishly expand computation in order to optimize for disparate cost metrics. We introduce the Pliops Extreme Data Processor (XDP), a novel storage engine implemented from the ground up using customized hardware. At its core, XDP consists of an accelerated hash table to index the data in storage using less memory and fewer storage accesses for queries than the best alternative. XDP also employs an accelerated compressor, a capacitor, and a lock-free RAID sub-system to minimize storage space and recovery time while minimizing performance penalties. As a result, XDP overcomes cost contentions that have so far been inescapable.

PVLDB Reference Format:
Niv Dayan, Moshe Twitto, Yuval Rochman, Uri Beitler, Itai Ben Zion, Edward Bortnikov, Shmuel Dashevsky, Ofer Frishman, Evgeni Ginzburg, Igal Maly, Avraham (Poza) Meir, Mark Mokryn, Iddo Naiss, and Noam Rabinovich. The End of Moore's Law and the Rise of The Data Processor. PVLDB, 14(12): 2932 - 2944, 2021.
doi:10.14778/3476311.3476373

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476373

1 INTRODUCTION

A storage engine is the software component that lays out data on a storage device on behalf of a database system. For decades, storage engines have benefited from Moore's law, which has accurately predicted that computer chips would double in speed every two or so years on account of a doubling in transistor density [50]. Over the past decade, however, Moore's law has stagnated [37]. This means that storage engine designers can no longer rely on CPU advances to alleviate computational overheads.

More recently, the rise of SSDs brought a thousandfold improvement in storage bandwidth. Hence, CPU overheads are no longer negligible relative to storage access. In addition, SSDs have idiosyncrasies that must be carefully managed [2]: writes are slower than reads, and random writes are particularly slow (as they lead to internal garbage-collection). Furthermore, writes wear out the device. It has therefore become important to avoid random writes, and to economize the use of writes in general. These trends pose new challenges for modern storage engines and their users [11, 34].

Challenge 1: Data Structure Trade-Offs. In order to optimize computation and storage writes, recent storage engines [3, 46, 48, 51, 52] employ an index+log architecture, which is in turn inspired by Log-Structured File Systems [49]. Index+log flushes all application writes to a log in storage and maps each entry to its location in the log using an in-memory index, typically a hash table. Compared to more traditional storage engines based on B-tree [10, 15, 44] or LSM-tree [5, 6, 27, 33, 45], index+log exhibits lower write costs and is less compute-heavy due to the hash table access. However, it requires a hefty memory footprint for the index. To reduce memory footprint, some variants map fingerprints instead of keys in the index, but this introduces false positives and thus redundant storage I/Os for queries [13]. The deeper problem is that any data structure design choice that prioritizes certain performance metrics tends to also penalize others [8].

Challenge 2: Compression. Compression has become painful to manage due to the recent hardware trends. On one hand, it improves storage bandwidth, lifetimes and utilization as fewer bits in the SSD are written for any unit of application data [56]. On the other hand, it can be a heavy CPU bottleneck [38].

Challenge 3: Resilience. Similarly painful cost contentions have emerged with respect to resilience. (1) The various RAID designs, which stripe data along with error correction codes across SSDs, must hold mutexes at a considerable CPU cost to provide atomicity. (2) Recovering a failed SSD using RAID entails overwriting a whole new SSD at considerable write cost and degradation to the overall storage bandwidth, even if some parts of the original SSD were empty or comprised of invalid data. (3) With respect to safe-guarding against power failure, write-ahead logging (WAL) and double-write buffering both impose runtime penalties by contributing to SSD write-amplification and CPU usage [32].

Hardware Accelerators. With Moore's law slowing down, hardware accelerators such as FPGAs, ASIC and GPUs have become attractive means of alleviating CPU bottlenecks. Such accelerators exhibit intrinsic parallelism, and they are cheaper and more energy-efficient than commodity CPUs. However, they require more effort and know-how to program. They can be used to speed up a variety of compute-intensive database operations [29], including selection [43], projection [42], aggregation [26], etc. Dedicated compression accelerators are also becoming common-place [1]. We observe that until today, accelerators have been applied mostly as bionic limbs that offload specific compute-heavy tasks from the CPU. In this paper, we argue that accelerators can have a more profound impact on storage engine design.

Insight 1: Using Compute to Solve Non-Compute Problems. Trade-offs in storage engines and in computer science in general are often intrinsic and inescapable. Optimizing for one metric (e.g., memory or I/O) usually requires a sacrifice on another (e.g., computation). Our insight is that hardware accelerators offer a way out. They make it possible to design algorithms that lavishly expand computation while in exchange guaranteeing a smaller overhead on other competing metrics than ever before. While such algorithms would have been prohibitively expensive on commodity CPU, carefully crafted hardware accelerators make them viable. We show that this design philosophy addresses a host of storage engine problems, including the challenges above.

Insight 2: A Unified Box. The number of physcial interfaces on a motherboard is limited. This, in turn, limits the number of devices (e.g., accelerators, NVRAM, SSDs, etc.) that can be plugged in. Moreover, with a greater number of devices plugged in, the overheads of CPU orchestration, data movement, and system administration increase. Our second insight is that by encapsulating as much functionality as possible within one device, we can increase the number of tasks offloaded from the CPU while simultaneously scaling their orchestration and administration overheads. The unified whole becomes more useful than the sum of parts that it replaces.

The Pliops Extreme Data Processor (XDP). We introduce XDP, a novel storage engine designed from the ground up lend itself to hardware acceleration. XDP consists of a thin software component in the host and a hardware device connected to the host through a PCIe port. XDP is interoperable with any commodity SSDs, which it visualizes as a RAID sub-system. XDP addresses each of the above-mentioned challenges using customized hardware.

Contribution 1: Alleviating Data Structure Contentions. XDP implements an index+log architecture. While index+log offers good SSD write performance and longevity, its core problem, as mentioned above, is a high memory footprint for the index. XDP tackles this problem in three ways. (1) It succinctly encodes the difference between fingerprints within a hash bucket instead of storing the actual fingerprints. This not only saves memory but also eliminates false positives to keep average and tail latency for queries low. (2) XDP partially sorts data in storage to reduce the size of pointers. (3) XDP uses a dense hash bucket format to keep overflow chains rare. While these techniques are computationally costly, we show that hardware acceleration makes them viable. The outcome is that XDP requires 10x less memory than popular index+log systems used in industry [52], all while outperforming them by eliminating false positives from queries.

XDP also has a hardware-accelerated indexing layer drawing from [19, 21–23, 39] to support richer storage engine operations (e.g., range reads), but we leave its description to a future article.

Contribution 2: Removing Compression Contentions. XDP compresses data as it arrives using ZSTD, a compute-heavy algorithm that achieves top-of-the-line compression rates. Our hardware implementation prevents ZSTD from burdening the CPU while still dramatically reducing SSD writes and space utilization.

Contribution 3: Penalty-Free Resilience. XDP takes exclusive control of the storage devices and only issues large sequential writes. This allows us to implement a customized RAID sub-system that does not hold mutexes, resulting in substantially lower CPU overheads. Furthermore, our RAID sub-system recovers a failed SSD using XDP's mapping of the underlying data to only copy valid data to a replacement SSD while ignoring empty space or invalid data. This, compounded with the fact that data is smaller due to compression, implies quicker and less obtrusive recovery after an SSD fails. In addition, XDP streamlines new data to a capacitor-backed memory module to support durable commits without issuing double writes to storage (e.g., WAL).

Contribution 4: Seamless Integration. XDP can expose a key-value interface or a block device interface to applications running on top. While the key-value interface gives better performance, the block device interface allows applications to seamlessly integrate with XDP without having to reformat their data. As XDP is lean and hardware accelerated, it does not amplify CPU overheads or memory footprint as a rich layer of indirection normally would.

2 HIGH-LEVEL OVERVIEW

This section presents XDP's high-level index+log architecture, shown in Figure 1. For now, we leave out details on how and where each

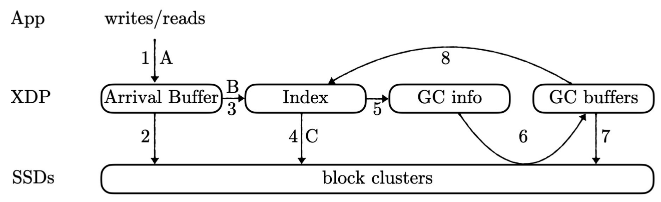


Figure 1: Logical system overview and write/read paths.

of the components in the figure is implemented, focusing instead on their functions and highlighting some important properties.

Application data arrives in the form of key-value entries. XDP compresses and inserts them into an in-memory arrival buffer (Arrow 1). When this buffer reaches capacity, XDP sorts its constituent entries based on the hash of their keys and flushes them to storage as a contiguous area called a *block cluster* (Arrow 2). For every entry in a new block cluster, XDP maintains a mapping entry within the in-memory index, and as we show later, sorting data within a block cluster allows to reduce the index size. The block cluster size is configurable, though we treat it as 2GB in this paper.

Index Design. Some index+log designs store the full key of every data entry in their index along with a pointer to its location in storage [3, 51]. This requires a lot of memory. Other designs store a smaller *fingerprint*, which is a hash digest of a key, instead of storing the key itself [4, 13]. This, however, leads to redundant storage accesses during reads on account of coincidentally matching fingerprints that belong to other keys, i.e., false positives.

To eliminate false positives while driving memory footprint farther down, we devise a novel structure called Delta Hash Table (DHT). DHT encodes only the first bit that differentiates between the fingerprints of entries that collide within the hash table. For example, consider three 5-bit fingerprints $\underline{1}0000$, $\underline{0}00\overline{1}0$ and $\underline{0}00\overline{0}0$ that fall into the same hash table slot. DHT only materializes enough information to signify that the first fingerprint differs from the other two in terms of its first bit (underlined) while the remaining two fingerprints differ in terms of their fourth bit (overlined). XDP encodes these fingerprint deltas as a succinct trie, which occupies much less space than the full fingerprints. Moreover, since it differentiates between all colliding fingerprints, no false positives can take place. To drive memory footprint even lower, DHT employs a novel and highly dense bucket format. It also exploits the fact that block clusters are sorted to index them using smaller address pointers. A DHT bucket must be parsed bit-by-bit, and so it is computationally expensive to access. Our parallel customized hardware implementation, however, makes it viable. Section 4 describes DHT and how it implements the XDP index in detail.

Index Maintenance. As entries in the arrival buffer get flushed to storage (Arrow 2), XDP checks whether each of them has a previous version in storage (Arrows 3 and 4). We refer to this as a fetch-existing-entry operation (fee-op), and it involves one index access and one storage read I/O. If an older version of the entry exists, we change its mapping address within the index to point to the new version. If the entry is new, however, we insert a new mapping entry to the index. This further reduces the index size (i.e., relative to other designs that continue to index obsolete versions of entries until garbage-collection [13, 25]).

XDP performs fee-ops asynchronously, and so they do not contribute to write latency. Note that fee-ops are not unique to XDP; all fingerprint-based index+log designs use them for index maintenance and for garbage-collection bookkeeping (as described below). Some designs, like XDP, perform fee-ops on the write path [52]. Other designs perform them during garbage-collection [13, 25]. The overhead in terms of read I/Os is the same.

Garbage-Collection (GC). The garbage-collector identifies and reclaims block clusters with mostly invalid data by relocating any remaining valid data. The reclaimed space can be used to store new application data. Many index+log designs perform *circular GC*, whereby the block cluster that was written the longest time ago is reclaimed [13]. The rationale is that the oldest block cluster is likely to contain the least amount of valid data. This approach, however, is known to perform poorly in the presence of non-uniformly updated data. The reason is that colder data winds up mixing with hot data on the same block clusters and has to be reclaimed at a significant overhead yet without yielding much space in return [52].

To alleviate this problem, XDP employs counters to keep track of the exact amount of valid data (measured in bytes) at each block cluster so that the one with the least amount of valid data can always be chosen as a reclamation victim. These counters are maintained via fee-ops, which access outdated versions of entries and can thus identify the block cluster containing them and subtract every outdated entry's size from the correct counter (Arrow 5). Furthermore, XDP maintains a separate GC buffer to rewrite data to storage as a result of GC (Arrows 6 and 7). This broadly separates cold and hot data and thus reduces write-amplification. As entries are migrated during GC, their corresponding pointers within the index are updated to reflect their new locations (Arrow 8).

Read Path. An application read first checks if the arrival buffer contains the target entry (Arrow A). If not, it checks the index (Arrow B) and then the appropriate storage location (Arrow C).

3 PHYSICAL SYSTEM OVERVIEW

We now elaborate on XDP's physical architecture. As shown in Figure 2, each of the components previously illustrated in Figure 1 consists of multiple physical substructures, each of which resides on the host or on embedded SRAM, DRAM, or flash modules.

The XDP device contains customized hardware that operates on all the substructures shown in Figure 2 to offload work from the host CPU. Figure 3 outlines some of the more notable hardware operations and ranks them based on their computational intensity. We proceed to elaborate on these substructures and operations roughly in the order that data flows through the system.

PCIe Channel. The XDP device connects to the host through a PCIe channel. This is the core system bottleneck as all reads and writes pass through it. To keep this channel consistently saturated, XDP materializes as many parallel hardware engines as needed for each of the operations shown in Figure 3 to ensure that none of them is a bottleneck relative to the channel's bandwidth.

Fast Commits Using SRAM. As application data arrives, it is first placed within the Welcome-Buff, a small internal capacitor-backed SRAM module. SRAM is extremely fast, and so it allows XDP to make new data persistent and acknowledge to the host that it had been committed extremely quickly.

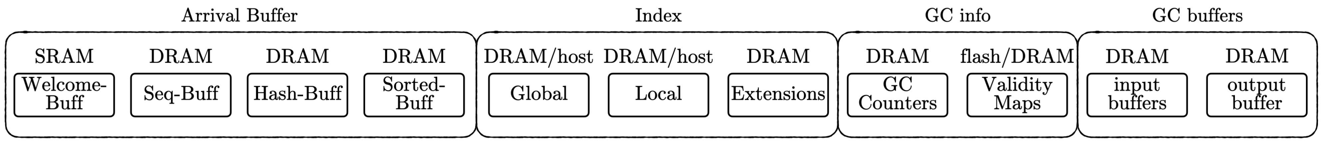


Figure 2: The different XDP physical substructures and where they reside.

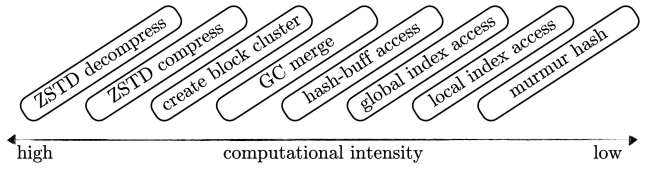


Figure 3: Accelerated XDP operations.

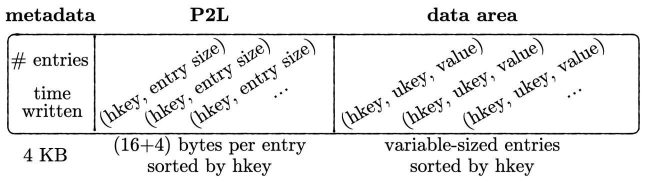


Figure 4: Block Cluster Structure.

Compression. XDP compresses the value of every data entry using ZSTD, a compute-intensive compression algorithm known to achieve high compression rates [28]. ZSTD is often passed over in favor of less compute-heavy algorithms to reduce CPU pressure (e.g., LZ4 [14] or Snappy [31]). Our customized hardware, however, implements multiple parallel ZSTD engines to prevent bottlenecks and thus to achieve a high compression rate while still operating at channel bandwidth. Compression expands the underlying SSDs' storage capacity by a factor of the *compression ratio* (i.e., the original data size divided by the compressed data size).

Identifying Entries. XDP identifies every entry in the system using a 16B hash digest, referred to as an hkey. An hkey is generated based on the entry's user key, referred to as its ukey. 16B hkeys are large enough to make the probability of any two hkeys colliding for different ukeys negligible. To generate hkeys, XDP uses murmur hash [7] as it is simple to implement on customized hardware [36], and it is adept at creating uniformly random hash keys.

Buffering in DRAM. New entries are eventually moved from the Welcome-Buff to an append-only capacitor-backed 2GB buffer called Seq-Buff in DRAM. To support queries for entries within Seq-Buff, the hkey of each of these entries is added to a hash table called Hash-Buff that maps the location of each entry within Seq-Buff.

Bucket-Sort. The Hash-Buff *bucket-sorts* [17] entries based on their hkeys. It consists of multiple buckets, and it inserts mapping entries into these buckets based on the most significant bits of their hkeys. Within each bucket, it truncates common hkey prefixes and sorts mapping entries based on the remaining bits of their hkeys. An important property of bucket-sort is that its run-time is linear when applied over uniformly distributed data [18] (i.e., as opposed to, say, merge-sort, which is more robust for non-uniform data but also more expensive). This allows XDP to sort incoming data with minimal computational pressure on the customized processor.

Flushing to Storage. XDP in fact contains multiple pairs of Seq-Buff and Hash-Buff. Once one pair fills up with data, it is sealed and all subsequent arriving data goes into the next pair. A block cluster is then created from the sealed pair by copying entries from Seq-Buff in the sorted order of hkeys in Hash-Buff to another buffer called Sorted-Buff. At the end, Sorted-Buff is flushed as a sorted block cluster to storage, and the three buffers are cleared.

Structure in Storage. Figure 4 illustrates the layout of a block cluster (BC). A BC begins with one 4KB page containing metadata such as the number of data entries within the BC. It is followed by a *physical to logical mapping* (P2L). P2L is an array of fixed-sized pairs sorted by hkey, where each pair consists of an hkey of one of the data entries within the BC as well as the corresponding data entry's size. The metadata block and P2L are used during recovery to quickly reconstruct the index without having to traverse the whole data set. P2L is followed by a *data area*, whereon data entries sorted by hkey are laid out contiguously across the remaining space in the BC as slotted 4KB database pages.

Global and Local Indexes. As shown in Figure 2, the index consists of three substructures. The *Global Index* is a hash table that maps each entry to the ID of the block cluster that contains it. For every BC, there is a *Local Index*, which maps the hkey of every entry in the BC to the data page/s that contain the entry (an entry may span one or more pages). The reason for having a separate local index for each BC (i.e. as opposed to mapping the full page address of each entry from the Global Index) is to reduce memory footprint. A local index takes advantage of the fact that data is sorted within its BC to encode page address offsets as opposed to full page addresses. The global and local indexes are both implemented using the novel Delta Hash Table. As such, they do not contain the full hkeys of constituent entries but only enough information to distinguish between hkeys that coincide in the same hash table slot. The third substructure is an array of *extension buckets*, which may be utilized if some of the buckets in the global or local indexes overflow. The detailed index design is given in Section 4.

Using Host Memory. While the XDP device's customized processor is in charge of operating on the local and global indexes, Figure 2 shows that there are two options for where to store these indexes: on the XDP device's internal DRAM or on the host's memory. The former option leads to lower latency as it keeps these structures closer to the customized processor. On the other hand, since the sizes of the global and local indexes depend on the number of entries in the system, storing them on the host gives more flexibility with respect to the maximum number of entries (or conversely, the minimum entry size) that the system can support. Users can configure the system either way depending on their workload characteristics and performance needs.

Garbage-Collection Bookkeeping. As discussed in Section 2, XDP maintains a counter for each BC to facilitate GC victim selection. In addition, XDP maintains a validity map for each BC that marks which of the entries on the BC are valid. These maps are maintained through fee-ops, and they get accessed during garbage-collection to determine which entries to reclaim. In fact, the validity maps are also used to recover XDP from power failure (as described

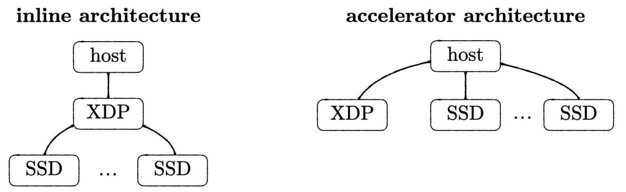


Figure 5: Inline vs. accelerator architecture.

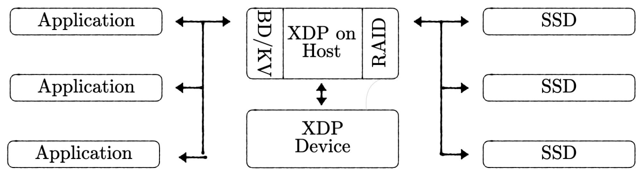


Figure 6: The XDP interfaces to the host and to storage.

shortly), and so they must be durable. The simplest way to implement them is as a capacitor-backed bitmap on the XDP device's internal DRAM, where the i^{th} bit in a given map indicates whether the entry at offset i of a given block cluster is valid. An alternative approach is to log the offsets of invalidated entries in storage and to later reorganize them for efficient access (e.g., as an LSM-tree of bitmaps [20]). The former approach performs better. However, since the size of the validity maps depends on the number of entries in the system, the latter approach gives more flexibility with respect to the number of entries that the system can support, and it allows for a lower-cost capacitor. The implementation can be tailored to the needs of a particular application.

Garbage-Collection Merge-Sort. XDP requires all block clusters to be sorted based on hkey in storage. This means that the garbage-collector has to effectively merge-sort several BCs while also discarding invalid entries from them. The XDP device contains a fixed number of GC input buffers that can be sort-merged into an output buffer. Since the number of input buffers is fixed, the output buffer may not be full after the merge-sort if the input buffers do not contain enough valid data. In this case, rather than wasting space by writing the resulting BC partially empty, we transform the output buffer into an input buffer and merge-sort it with several new GC victims, which we read in from storage, into a new output buffer. This process continues iteratively until we have a full output buffer.

Maintaining Block Cluster Uniformity. In addition to requiring each BC to be sorted, XDP also requires all entries within each BC to be uniformly randomly distributed in the hkey space. As we will see later, this requirement is necessary to keep the memory overheads of the local indexes modest. BCs that contain new data from the application naturally follow a uniformly random distribution. However, BCs created during GC do not necessarily maintain this property. For example, consider three uniformly randomly distributed BCs that get garbage-collected but contain more valid entries than can fit into the output buffer. In this case, a simple merge-sort process would cause the new BC to skew towards smaller hkey values. To correct for this effect, XDP fully merge-sorts only a sub-set of the input buffers while carefully randomly sampling valid entries from them to ensure that the hkey distribution in the output buffer stays uniform.

Rate Limiter. While XDP is able to ingest and persist application writes very quickly via its capacitor-backed SRAM module, its write bandwidth is limited by also having to reclaim space for these new writes via garbage-collection. In order to prevent drastic performance slumps while GC is taking place, XDP limits the rate at which writes are transmitted from the application. It does this using a first-order control loop, which takes into account statistics collected during run-time about the overheads of GC so that writes can be evenly interspersed. This allows XDP to provide a stable and predictable write bandwidth in time.

Power Failure & Recovery. The XDP device contains a small internal flash module to facilitate recovery. When power fails, the capacitor-backed Welcome-Buff, Seq-Buff and validity maps are flushed to this module. During recovery, the validity maps are first restored. The metadata block and P2L for each BC are then read in order to (1) reconstruct the BC's local index, (2) check the BC's P2L against its GC bitmap to determine which entries are valid, (3), insert valid entries into the global index, and (4) subtract the size of the invalid entries from the BC's GC counter. The Welcome-Buff and Seq-Buff are then recovered from flash, and a Hash-Buff is created for the Seq-Buff.

Inline vs. Accelerator Architecture. An important design consideration is how to connect the XDP device to the host and to the underlying SSDs. As shown in Figure 2, there are broadly two options. With an *inline* architecture, the SSDs are connected directly to the XDP device. This option minimizes CPU orchestration overheads for cross-device communication, and it occupies fewer interconnect slots on the host machine. In the *accelerator architecture*, the XDP device and the SSDs interact through the host. As all communication between XDP and storage takes place through battle-tested host-side drivers, this option allows to seamlessly inter-operate with a greater diversity of storage devices. It also entails lower setup overheads as it does not require creating customized inter-connections among peripheral devices within the host server. XDP supports both options to allow users to choose between performance and flexibility.

Interfacing with Storage. Figure 6 zooms in on the accelerator architecture, focusing on how XDP interacts with applications and with storage. XDP is compatible with SSDs that expose either the standard block device (BD) interface or the newer zoned namespace (ZNS) interface, which offers better storage utilization to applications that only issue sequential writes (a beta version of XDP at one point also employed the open-channel interface [47]).

RAID Sub-System. XDP implements a RAID5 sub-system to virtualize the SSDs, striping block clusters across them for better bandwidth along with distributed error correction codes. RAID5 is considered prohibitively expensive for systems that issue small random writes, as each write entails a read-modify-write operation across the stripe. Since XDP transforms all random writes into sequential ones, however, it obviates this problem. This allows us to implement a lock-free RAID sub-system. Furthermore, in contrast to standard RAID solutions, XDP exploits its awareness of the data when an SSD fails to only recover non-empty and valid data within each BC. This, coupled with the fact data is compressed, leads to significantly faster and less obtrusive recovery when an SSD fails.

Interfacing with Applications. The left-hand side of Figure 6 shows that XDP can expose either a key-value (KV) interface or a block device interface to applications running on top. Internally, the block device interface treats each 4KB page as a key-value pair

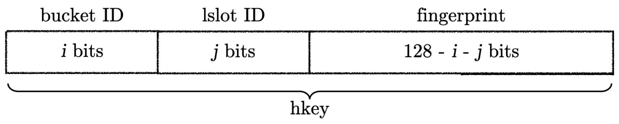


Figure 7: An entry' 128-bit hkey maps the entry to a bucket, an lslot within that bucket, and a fingerprint.

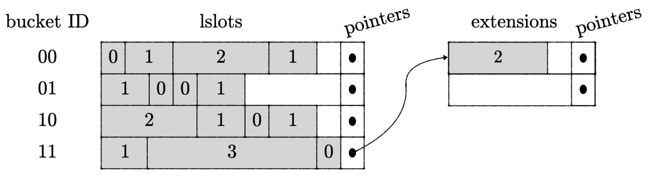


Figure 8: DHT consists of fixed-size buckets, each of which consists of variable-size lslots.

where the key is the page address. The KV interface allows for better performance as less data tends to flows between the host and the XDP device. However, the block device interface allows applications to quickly migrate to XDP without having to reformat their data. This is especially useful for any application that issues random write I/Os to the SSD (e.g., MariaDB). XDP internally converts these random writes into sequential writes. This improves storage performance, utilization and longevity without the CPU overheads that such a rich layer of indirection would normally involve.

4 THE DELTA HASH TABLE

This section describes the Delta Hash Table (DHT), which is used by XDP to map each key to the location of the matching entry in storage. Nevertheless, DHT has broader applicability as a generic accelerated hash table. DHT addresses the following two challenges, which are intrinsic to the design of hash tables.

Challenge 1: Entry Representation. Hash tables exhibit an innate trade-off between memory and performance dictated by how entries are represented. Some designs represent each entry using its full key. This takes up a lot of memory, especially if keys are large. Other designs instead use fingerprints, which take up less space but lead to false positives and thus to redundant storage accesses. DHT introduces a new approach that uniquely identifies entries by encoding the differences (i.e., deltas) among their fingerprints. This requires even less memory than with full fingerprints while also eliminating false positives for queries to existing entries.

Challenge 2: Bucket Sizing. Hash tables exhibit yet another intrinsic performance vs. memory trade-off dictated by the bucket size. With smaller buckets, the variability in the number of entries mapped to each bucket causes many buckets to remain underutilized and others to overflow. This leads to poor memory-utilization. On the other hand, with larger buckets, there are more entries on average in each bucket, and so more entries need to be compared against during queries. This leads to higher data movement and processing overheads. With DHT, the small delta encodings allow us to squeeze more entries into buckets and thus achieve good memory utilization and modest data movement overheads at the same time. Furthermore, we use customized processors to tilt the balance towards larger buckets to improve memory utilization without incurring computational bottlenecks.

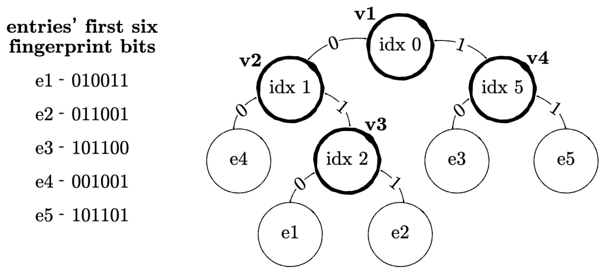


Figure 9: A delta trie distinguishes between entries based on the most significant bits that differ in their fingerprints.

Overview. DHT consists of 2^i contiguous, fixed-size buckets. Each bucket comprises 2^j variable-size adjacent logical slots (lslots). An entry is mapped to a bucket based on the first i bits of its hkey and to an lslot within that bucket based on the subsequent j bits of its hkey. All succeeding bits within the hkey comprise the entry's fingerprint, as shown in Figure 7.

Figure 8 illustrates a DHT instance where i and j are both set to two. Hence, there are four buckets, each of which contains four lslots. The lslots are illustrated in gray, and the number on top of each lslot indicates the number of entries that have been mapped to it. For example, an entry for which the first four hkey bits are 0111 belongs to the last lslot within Bucket 01. White areas in the figure indicate un-utilized space.

Within a bucket, a query or a put/delete (PD) operation searches for the matching lslot by traversing the lslots serially. A PD operation that changes the size of an lslot must also rewrite all subsequent lslots within the bucket to keep them adjacent to the lslot whose size changed. As we later show, DHT works best when there are few entries per lslot and tens of lslots per bucket on average.

To accommodate bucket overflows, each bucket has an extension pointer. In Figure 8, for example, Bucket 11 had overflown causing its last lslot to be stored in an extension bucket. Each extension bucket also has an extension pointer to support chaining.

4.1 LSlot Structure

Delta Trie. Figure 9 illustrates the logical structure of an lslot containing five entries. The left-hand side of the figure shows the first six bits of these entries' fingerprints, while the right-hand side illustrates a binary *delta trie* that encodes the differences between these entries' fingerprints. Each leaf node in the trie corresponds to one of the five entries, while an internal node corresponds to the index (idx) of the first fingerprint bit that differentiates between some two sets of entries. For example, the root node v1 indicates that the bit at index zero is the first that differentiates between the fingerprints of entries in the left sub-tree (e4, e1 and e2) and the right sub-tree (e3 and e5). As another example, the node v4 indicates that the bit at index five is the first that differentiates between the fingerprints of entries e3 and e5. As shown in the figure, the delta trie effectively sorts the entries based on their fingerprints.

Lslot Encoding. Figure 10 illustrates an lslot's physical encoding and how it evolves starting from an empty state as we insert the five entries previously shown in Figure 9. As shown in Figure 10, an lslot consists of four fields. The *tenancy* field encodes the number of entries in the lslot. The *structure* field encodes the delta trie's

	tenancy	structure	indices	payloads
empty	0			
e1 - 0100111	10			p1
e2 - 0110010	110	~~00~~	110	p1, p2
e3 - 1011001	1110	10 ~~00~~	0 ~~1~~10	p1 p2 p3
e4 - 0010010	11110	10 01 ~~00~~	0 ~~1~~0 ~~11~~0	p4 p1 p2 p3
e5 - 1011010	111110	11 01 00 ~~00~~	0 ~~1~~0 ~~11~~0 ~~1~~11110	p4 p1 p2 p3 p5

Figure 10: An lslot is a physical encoding of a delta trie. In this example, we add five entries to an initially empty lslot to illuminate how the encoding evolves.

topology, meaning how nodes are connected. The *indices* field encodes the contents of each of the trie's internal nodes, namely the index of the most significant bit that differentiates between the fingerprints of entries at that internal node's left-hand sub-tree vs. its right-hand sub-tree. Lastly, the *payloads* field gives the payload of each of the entries sorted by fingerprint (in the case of XDP, the payloads encode information about the location of each data entry in storage). We now elaborate on each of these fields.

Tenancy Field. The tenancy field is unary encoded. For example, lslot sizes of 0, 1, 2 and 3 are encoded as 0, 10, 110, 1110, respectively. We use unary encoding because the probability distribution of entries per lslot is approximately Poisson. Unary leads to the optimal (i.e., as small as possible) average code length for this distribution. For an lslot with l entries, the tenancy field takes up $l + 1$ bits.

Structure Field. The structure field captures the topology of the delta trie by encoding whether the children of each internal node are internal or leaf nodes. A given internal node is represented as 11 if its two children are also internal nodes, 10 if the left-child is an internal node while the right-child is a leaf node, 01 if the left-child is a leaf node while the right-child is an internal node, and 00 if both children are leaf nodes. The representations for the different internal nodes are laid out in a depth-first leftwards order. For example, the bottommost structure field in Figure 10 encodes the trie topology in Figure 9 as 11 01 00 00 where 11 corresponds to v1, 01 to v2, 00 to v3, and 00 to v4.

As shown in Figure 10, with zero or one entry in an lslot, the structure field is vacant as the delta trie has no internal nodes. We also observe that with two or more entries in an lslot, the last two bits of the structure field are always set to 00. This is because the children of the last internal node visited in a depth-first traversal are always both leaf nodes. We exploit this property to save space by always truncating these two bits as they are implicit. Figure 10 illustrates this by crossing out the last two structure bits from all lslots with two or more entries. Overall, for an lslot with $l \geq 2$ entries, the structure field occupies $2 \cdot (l - 2)$ bits.

Indices Field. For each internal node in the delta trie, the indices field contains the index of the first bit that differentiates between fingerprints of entries in the left-hand sub-tree and the right-hand sub-tree. Each of the indices is unary encoded, and they are laid out in a depth-first leftwards order. For example, the bottommost indices field in Figure 10 is 0 10 110 111110 because the first differentiating bit is at index zero for v1, one for v2, two for v3, and five for v4.

Since the indices strictly increase as we descend the trie, we can in fact encode only the difference between each internal node's index and its parent's index to save space. Using this optimization, the bottommost indices field in Figure 10 becomes 0 0 0 11110. Figure 10 crosses out the bits saved by using this optimization.

To analyze the size of the indices field, note that the first bit index differentiating between any parts of two uniformly randomly independent hashes is geometrically distributed with a mean of two. Unary encoding is optimal for this distribution and leads to an average code length of two bits per entry. Hence, for an lslot with $l \geq 1$ entries, the average indices field size is at most $(l - 1) \cdot 2$ bits. For an empty lslot, the indices field is vacant.

Payloads Field. The payloads field contains a payload for each of an lslot's entries. The payloads are laid out contiguously in the order of their corresponding fingerprints as shown in Figure 10. In XDP, the payloads contain information about the locations of entries in storage (as described in detail in Section 4.4). However, they can assume other types of data for different applications.

4.2 Memory Analysis

Equation 1 upper bounds the average size of an lslot with l entries, derived by adding up the sizes of the tenancy, structure and indices fields. We disregard the payloads field in this analysis as its size depends on particular application use-cases and is therefore not intrinsic to the DHT design. We observe from Equation 1 that the lslot size per entry (i.e., $\frac{lslot\_size(l)}{l}$) strictly increases with respect to the number of entries l as there are more entries to have to differentiate between. DHT is therefore space-optimal when every lslot contains one entry on average. XDP therefore strives to allocate each instance of DHT such that there is one entry per lslot.

$$lslot\_size(l) \leq \begin{cases} l + 1 \text{ bits} & 0 \leq l \leq 1 \\ 5 \cdot (l - 1) \text{ bits} & 2 \leq l \end{cases} \tag{1}$$

Average LSlot Size. To reason about the average lslot size, we first observe that the structure of our *"entries into lslots"* problem is analogous to the classic *"balls into bins"* problem in probability theory. It follows that the number of entries that fall into a given lslot approximately follows a Poisson distribution for which the mean parameter λ is equal to the number of entries divided by the number of lslots [40]. Equation 2 derives the average size of an lslot by taking a weighted Poisson average over the lslot size from Equation 1. We use $\lambda = 1$ for this analysis to reflect the case where there is one entry per lslot on average. The result is that the tenancy, structure and indices fields cumulatively occupy slightly less than three bits per entry on average.

$$\text{average lslot size} \leq \sum_{l=0}^{\infty} \text{Poisson}(l, 1) \cdot lslot\_size(l) \approx 3 \text{ bits} \tag{2}$$

Overprovisioning Buckets. Occupancy variability across buckets is inevitable in a hash table. Too much of it can compromise space-utilization due to bucket overflows, which necessitate the use of extension buckets. To mitigate this problem, we use large buckets, and we over-provision the bucket size to be slightly larger than the average content size per bucket. The goal is to prevent buckets with slightly more content than average from overflowing. On the x-axis in Figure 11, we vary the number of extra bits per entry assigned

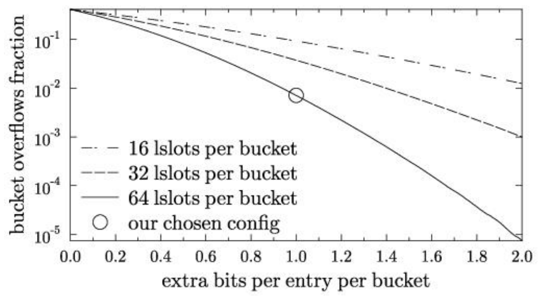


Figure 11: With larger and slightly over-provisioned buckets, DHT prevents most bucket overflows.

to each bucket from zero to two. On the y-axis, we measure the resulting percentage of bucket overflows. We repeat this for three configurations with 16, 32 and 64 lslots per bucket, and where there is one entry per lslot on average in each configuration. We observe that the larger the buckets, the more rapidly the percentage of overflows drops with respect to the number of extra bits per entry. The statistical intuition is that with larger buckets, the standard deviation is smaller relative to the mean bucket occupancy.

While all configurations in Figure 11 are a part of the DHT design space, we tend favor configurations with larger buckets and less extra space per bucket, for example 64 lslots per bucket and one extra bit per entry. Note that on a CPU, a DHT configuration with 64 lslots per bucket is less viable. Such large buckets are expensive to process on a CPU, not least due to the complex DHT bucket structure, which has to be parsed bit by bit. A customized processor, however, eliminates such computational bottlenecks and thereby allows to optimize for disparate metrics such as memory.

Total Memory Overhead. With one over-provisioned bit per entry used to mitigate bucket overflows and with three bits per entry on average used for the tenancy, structure, and indices fields, the total memory overhead of DHT is a modest four bits per entry.

4.3 Efficient LSlot Decoding

In order to parse a bucket efficiently using the customized processor, it is crucial to scan it in one pass and to only use simple primitive data types to maintain state. We meet both requirements using Algorithm 1. Algorithm 1 is used for parsing a single lslot, though its logic can be applied iterativly to parse a whole bucket. It takes a fingerprint as a parameter and returns the leaf offset of the matching entry in the lslot. The returned leaf offset can then be used to access the corresponding entry's payload. Algorithm 1 is the core component of get, put and delete operations in DHT.

Decoding in One Pass. To make an lslot decodable in one pass, we interleave the structure and indices fields, as shown in Figure 12 for the bottommost row from Figure 10. By virtue of interleaving, Algorithm 1 can process an lslot in one pass by first decoding the tenancy field (Line 2) and then decoding the interleaved structure and indices fields for one internal node in each iteration (Lines 7-8). As the unary encodings are self-delimiting while the tenancy field tells us how many internal and leaf nodes to expect, Algorithm 1 infers precisely where each field begins and ends.

By extension, a bucket parser on top can exploit the self-delimiting nature of an lslot to tell precisely where each lslot within a bucket begins and ends. In this way, a query targeting lslot number q within

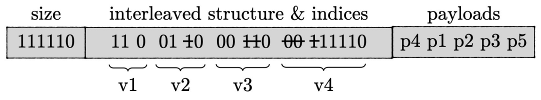


Figure 12: The structure and indices fields within an lslot are interleaved to support parsing in one pass.

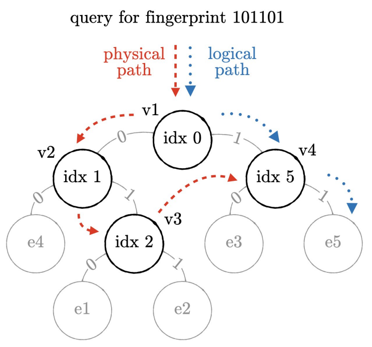
```
1  Function getLSlotOffset(string fingerprint):
2      int size = parseNextUnaryCode()
3      if size == 0 then return -1 end
4      int skip = 0, offset = 0, index = -1
5      bool foundTargetOffset = false
6      for $i = 0$; $i < size - 1$; $i$++ do
7          string structure = (i == size − 2) ? ‘00’ : parseTwoStructureBits()
8          int indexDelta = parseNextUnaryCode()
9          if skip == 0 and !found then
10             index += indexDelta + 1
11             if fingerprint[index] == ‘1’ then
12                 if structure == ‘11’ or structure == ‘10’ then
13                     skip++
14                 else if structure == ‘01’ then
15                     offset++
16                 else if structure == ‘00’ then
17                     offset++
18                     found = true
19             else if fingerprint[index] == ‘0’ then
20                 if structure == ‘01’ or structure == ‘00’ then
21                     found = true
22         else if skip > 0 and !found then
23             skip−−
24             if structure == ‘11’ then
25                 skip += 2
26             else if structure == ‘01’ or structure == ‘10’ then
27                 offset++
28                 skip++
29             else if structure == ‘00’ then
30                 offset += 2
31 return offset
```

Algorithm 1: An lslot is decoded in one pass and using primitive variables to lend itself to acceleration.

a bucket parses and ignores the first $q-1$ lslots while invoking Algorithm 1 on the q^{th} lslot.

Primitive State Maintenance. Figure 13 illustrates how Algorithm 1 processes the lslot encoding in Figure 12. The algorithm traverses the internal nodes in the depth-first leftwards order with which they are encoded in the lslot. We refer to this as the *physical path*, and it is illustrated in dashed red arrows. The logical path, shown in dotted blue arrows in Figure 13, shows the path from the root to the target leaf node. Each iteration of the algorithm parses one internal node along the physical path. The objective is to meanwhile also efficiently keep track of progress along the logical path. Algorithm 1 achieves this using four primitive variables. (1) The *offset* variable counts the number of leaf nodes encountered along the physical path before finding the target leaf. (2) The *skip* variable counts the number of consecutive incoming internal nodes along the physical path that are not part of the logical path. (3) The *index* variable is the differentiating fingerprint index of the current or previous internal node visited along the logical path. (4) The *found* variable indicates whether we have found the correct leaf node and can therefore ignore all subsequent nodes on the physical path.

Query Example. The right-hand side of Figure 13 illustrates Algorithm 1 in action by showing how its state evolves as it runs. At Node v1, the logical path turns right since the target fingerprint's



traversal state

	skip	index	offset	found
v1	1	0	0	false
v2	1	0	1	false
v3	1	0	3	false
v4	0	5	4	true
return			4	

Figure 13: A query physically traverses an lslot's internal nodes depth-first while keeping track of the logical path from the root to target leaf using four primitive variables.

bit at index zero is set to one (Line 11). Since the left child of v1 is an internal node, however, the skip variable is incremented (Lines 12-13). The next Node, v2, is not on the logical path as evidenced by skip variable being greater than zero (Line 22). Since one of v2's children is a leaf node and the other is an internal node, we increment both the offset and skip variables (Lines 26-28). The next Node, v3, is also not on the logical path. As both of v3's children are leaf nodes, we add two to the offset variable (Lines 29-30). The next Node, v4, is on the logical path since the skip variable is back to zero (Line 9). The logical path turns right since the target fingerprint's bit at index five is set to one (Line 11). Since both children of v4 are leaf nodes, we increment the offset variable by one to count the left child, and we then set the found variable to true as the right child is our target (Lines 16-18). At this point, the offset variable's value is four, which corresponds to the target leaf's position in the lslot.

Modifiers. A put(key, payload) operation invokes Algorithm 1 to find the closest key within the target lslot. It then retrieves the full hkey using a fee-op so that it can be compared with the new key's hkey. If they are the same, we infer that the put is an update and change the payload of the existing entry within the lslot. Otherwise, the put is an insert and so we (1) find the most significant differentiating fingerprint bit among the two entries' fingerprints, (2) add an internal node to the lslot to differentiate between the entries based on this bit, and (3) add the new entry's payload to the lslot at the offset matching the new leaf for the entry.

Similarly, a delete(key) operation invokes Algorithm 1 and retrieves the target entry to check if it exists. If so, the existing entry is deleted from the lslot by removing its payload and replacing its parent node with its sibling node in the delta trie.

4.4 DHT Instantiations

This section describes how XDP utilizes DHT.

Global Index. The Global Index (GI) is a DHT instance that maps each data entry in the system to the ID of the block cluster (BC) in which it resides. Fee-ops keep GI up-to-date as discussed in Section 3. In terms of space, GI takes up $\approx Y - 27$ bits per entry for an underlying SSD storage capacity of 2^Y bytes. The reason is that with each BC being 2GB (2^{31} bytes), each block cluster ID takes up $Y - 31$ bits while each DHT entry has an additional overhead of 4 bits per entry as discussed in Section 4.2.

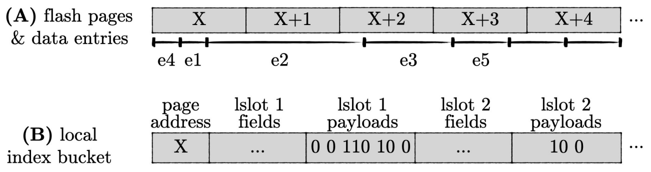


Figure 14: The local index encodes the page address deltas between adjacent entries in a block cluster.

Local Index. The local index for each BC in the system is a static instance of DHT that maps each entry in the given BC to the 4KB flash page whereon the entry begins (an entry may cross one or more page boundaries). A local index is created for a new BC right before the BC is written to storage. At this point, all data entries belonging to the BC are in memory (i.e., in the sorted-buff or the GC output buffer) and so it is possible to create a DHT instance without storage I/Os. XDP keeps a local index throughout its corresponding BC's existence and resets it when its BC gets garbage-collected.

As a payload for each entry in a local index, the simplest option is to use a page address. Such addresses, however, would take up at least $\log_2(31\text{GB}/4\text{KB}) = 19$ bits per entry to uniquely identify all pages within a BC. This would be a hefty toll. To save space, we exploit the fact that entries in a BC are sorted by hkey. Since DHT also effectively sorts entries by hkey, it means that adjacent entries within DHT correspond to entries on either the same flash page or on adjacent flash pages. As a payload for each entry, we therefore encode the number of 4KB pages that the entry crosses in unary. Thus, 0 means an entry is wholly contained within one page, 10 means it spans two pages, 110 means it spans three, on so on. Since most entries are smaller than a 4KB page, the encoded deltas stay small on average. In addition, we augment DHT to allow storing a page address at the start of each bucket. We set this address to the flash page whereon the first entry in the bucket begins.

Figure 14 Part (A) illustrates how the five entries e1 to e5 from our running example may be laid out sorted by hkey across several flash pages. Part (B) shows how the corresponding local index bucket would be structured starting with page address X and proceeding with lslots that store address deltas as payloads. To derive an entry's address with this design, we exploit the fact that a DHT query scans the whole bucket up to the target entry. During this process, we add the address at the start of the bucket to the sum of deltas up to the target entry's delta to derive the starting page address. For instance, a query to e3 in Figure 14 deduces that the fourth payload at Lslot 1 belongs to e3 and adds the initial bucket address X to all deltas before e3's delta (0+0+110 in unary) to obtain the correct starting page address of X+2. Since e3's delta is 10, we know to read both pages X+2 and X+3 to obtain the full entry.

As long as the average entry size is smaller than 4KB, the address deltas occupy less than two bits per entry on average. By adding the standard DHT mapping overhead of 4 bits per entry, the space overhead of the local indexes is ≈ 6 bits per entry.

Hash-Buff. Hash-buff is an instance of DHT whereby the payload of each entry comprises the entry's full hkey as well as a pointer to the corresponding data entry within seq-buff. Its memory footprint is minor as the seq-buff spans at most 2GB.

5 MEMORY & PERFORMANCE PROPERTIES

We now summarize XDP's memory and performance properties.

Memory Footprint. By adding up the sizes of the global and local indexes, the XDP system requires $Y - 21$ memory bits per entry to manage 2^Y bytes of data. For 32TB, for instance, this amounts to 3 bytes per entry. This is a significant memory reduction relative to existing index+log systems. BitCask [51] and ForestDB [3] store the full keys of data entries in their indexes thus requiring orders of magnitude more memory. Aerospike [52] uses 20 byte fingerprints and stores other auxiliary metadata (e.g., time to live and time last written) about each entry in its index leading to a total of 64B per entry. FASTER [13] uses a chained hash table with 64 byte buckets each storing 7 entries (6B pointer and 2B fingerprint), thus leading to an overhead of $\approx$ 10 bytes per entry before the impact of bucket overflows is considered. FAWN [4] uses a chained hash table with 4 byte pointers and 2 byte fingerprints leading to a 6 byte per entry overhead before the impact of overflows is considered. FlashStore [24] uses a cuckoo hash table variant with 4B pointers and 2B fingerprints leading to a 6-7 byte overhead. Hence, XDP at least halves the memory footprint relative to existing index+log architectures. It achieves this by (1) storing fingerprint deltas as opposed to full fingerprints, (2) using large hash buckets to make overflows rare, and (3) sorting each block cluster to allow encoding address deltas as opposed to full addresses.

Query Cost to Existing Entry. XDP performs a query to an existing entry in strictly one storage I/O. The reason is that the global and local indexes are DHT instances that encode the differences across all coinciding fingerprint. Therefore, a query to an existing entry cannot result in false positives. This is an improvement over fingerprint-based index+log systems [4, 13, 24], which incur false positives and thus exhibit higher average and tail latency.

Query Cost to Non-Existing Entry. In contrast to non-empty queries, an empty query (i.e., to a non-existing entry) that lands on a non-empty lslot within the Global Index always leads to one false positive and thus one redundant storage I/O. XDP does not prevent this I/O as DHT only encodes information about the differences between existing fingerprints and no information that allows filtering out non-existing fingerprints. Assuming one entry per lslot on average, the fraction of empty lslots in the global index is Poisson(0, 1) = e^{-1} (as per the classic "balls in bins" problem with load factor one). Hence, the probability of an empty query landing on a non-empty lslot and thus incurring a false positive is $1 - e^{-1} = 0.63$. While this rate is higher than with a fingerprint-based index, empty queries are less common in our target workloads and their performance is therefore less critical. For workloads with many empty queries, XDP can support adding a few fingerprint bits to each payload in the Global Index to reduce false positives.

Write-Amplification. XDP keeps track of the amount of live data in each BC to allow GC to identify and reclaim the BC with the least amount of live data left. Furthermore, XDP separates newly written application data from garbage-collected data to prevent cold data from constituting a constant dead-weight migration overhead. In this way, XDP achieves a write-amplification akin to state-of-the-art designs [52]. With 20% of storage space being used for over-provisioning, write-amplification is $\approx$ 2.6 for workloads with uniformly random writes [53] and tends to be lower for workloads with more sequential and/or skewed writes.

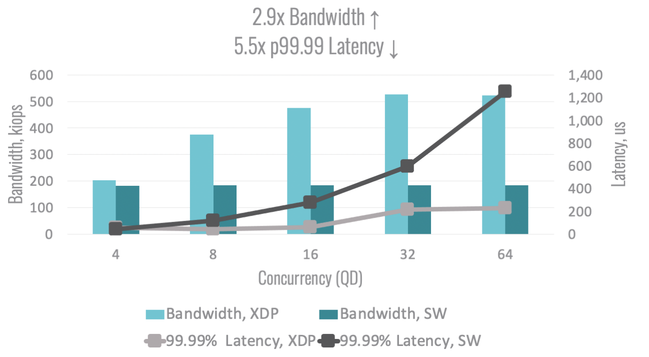


Figure 15: FIO benchmark with XDP RAID5 vs software RAID0 on top of four SSDs and with varying queue depth. XDP offers superior bandwidth and tail latency while at the same time providing failure-protection.

Space-Amplification. XDP compresses data as it arrives and stores it contiguously in storage. Compression rates of x4 are typical in our target workloads. Hence, XDP effectively enlarges the storage capacity by the compression factor and yet without introducing any CPU overheads by virtue of using customized hardware.

6 ENSURING FAST SEQUENTIAL READS

SSD Sequential vs. Random Reads. On modern SSDs, sequential reads are known to be faster than random reads by as much as 40% or more [12, 35]. Storage bandwidth tends to gradually improve as we increase a read request's size until leveling off at about eight flash blocks per request (i.e., $\approx$ 32-128KB).

The Problem: Losing Sequentiality. The XDP block device treats each 4KB block as a key-value entry for which the logical block address (LBA) is the key. It sorts these entries in block clusters based on the hash keys (hkeys) of their LBAs. The outcome is that blocks with adjacent LBAs are not physically adjacent in storage. This means that the XDP block device as described so far would translate sequential reads into slower random reads.

Reinforcing Sequentiality. We alleviate this problem by modifying XDP's block device such that small groups of blocks with consecutive LBAs and written at the same time are laid out contiguously in storage. We do this by truncating the least significant b bits from each LBA before deriving its hkey, and then appending the truncated b bits to the resulting hkey. This ensures that groups of up to 2^b blocks in the logical address space stay adjacent physically. We typically set b to 3 or 4. This optimization allows applications running on top to saturate storage bandwidth.

Efficient Header Encoding. Within the global and local indexes, we treat each group of sequentialized pages as a single entry identified by their common hash prefix. The payload for each header contains additional information about whether each of the constituent blocks in the group exists and if so where it resides. We leave out the precise encoding but note that it contributes at most two bits per entry to the overall memory footprint, a modest toll.

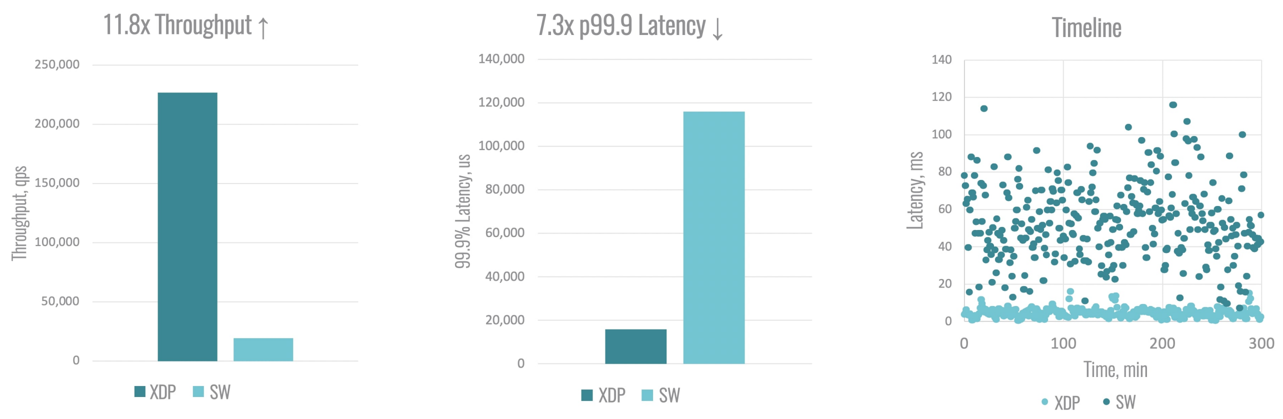


Figure 16: MariaDB with and without XDP under a TPC-C transactional workload. XDP provides superior throughput (left), lower average latency (middle), and lower latency variability (right).

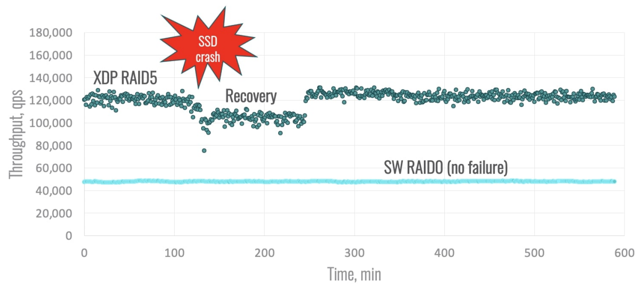


Figure 17: TPC-C benchmark on MariaDB with XDP RAID5 vs. plain RAID0 on four SSDs. XDP performance even during recovery is significantly improved relative to the baseline under normal operation.

7 EVALUATION

We evaluate the performance of multiple popular system benchmarks on top of the XDP block device. The benchmarks include raw block I/O in Section 7.1, the SQL database MariaDB in Section 7.2, and the NoSQL database MongoDB in Section 7.3.

The testbed is a Dell R740 server with 512 GB RAM and a dual-socket Intel Xeon Gold CPU with 40 cores and hyperthreading enabled. The OS is Ubuntu Linux 18.04.4 LTS. Unless stated otherwise, all experiments use industry-standard NVMe drives of 1.9TB raw capacity based on TLC NAND flash. We also run experiments with both RAID0 and RAID5, both of which stripe data across four identical SSDs, but where the latter sacrifices one SSD's worth of storage bandwidth and capacity to provide failure protection.

7.1 Block I/O

The flexible I/O (FIO) [9] benchmark measures disk bandwidth and tail I/O latency under customizable workloads. We focus on a random I/O benchmark with 4KB blocks, 70% reads and 30% writes. The data is synthetically generated and 3x-compressible. We compare (1) FIO on top of XDP with RAID5 to (2) plain FIO on top of RAID0. RAID5 gives the XDP baseline a harder time as it is known to perform worse than RAID0. We vary the queue depth (QD) from 4 to 64 to test scalability with load.

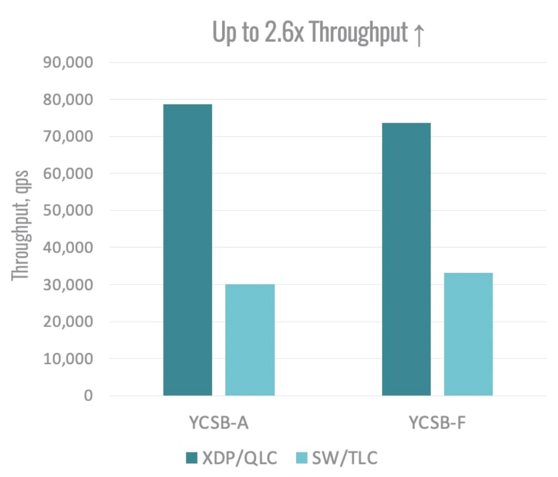
Figure 15 summarizes the results. Under light load, the two systems exhibit similar performance. However, as the load grows, XDP's throughput scales up to QD=32 (526 kIOPS) whereas the baseline fails to scale beyond QD=4 (183 kIOPS). In terms of 99.99% percentile latency, XDP RAID5 stays under 230us with QD=64 while the baseline is as slow as 1.25ms. The reason is that XDP internally transforms the random I/Os issued by FIO into compressed sequential I/Os to the underlying SSDs. As a result, XDP with RAID5 provides superior performance relative to RAID0 while at the same time providing failure protection.

7.2 MariaDB

We now evaluate MariaDB [30] on top of XDP using the TPC-C benchmark [54]. As a baseline, we use plain MariaDB with no acceleration layer. Each experimental run lasts five hours. The data is 4x-compressible. The raw data volume before compression is 6.4TB. Each experimental run comprises of eight database instances with identical datasets running for five hours. Each database instance serves 128 concurrent clients and employs a 50GB buffer pool.

Figure 16 depicts the results when all databases share a single SSD. On the left-hand side, we observe that the XDP-accelerated system delivers a 11.8x throughput increase (in queries per second, or qps). The middle part shows a 7.3x tail latency reduction for the 99.9% percentile, while the right-hand side illustrates performance stabilization in time. The reason for these performance characteristics is that MariaDB's storage engine, InnoDB, is based on a B-tree with a node size of 32KB. The I/O pattern driven by TPC-C is largely random, thus leading to random B-tree reads and writes to the underlying SSDs. XDP speeds up the random writes by transforming them into compressed sequential writes. Compression also shrinks each B-tree read from 32KB down to 8-16B, thus enabling better read throughput and latency.

We now turn to evaluate performance and reliability in a scaled-up setting with four SSDs and a 4x larger dataset. We compare MariaDB on top of XDP with RAID5 to plain MariaDB with RAID0. Figure 17 depicts the results. Under normal operation, the XDP-accelerated system is 2.5x faster than the baseline. The throughput gain is smaller than in the single-SSD experiment for three reasons. First, the execution is more I/O bound (the dataset grows whereas



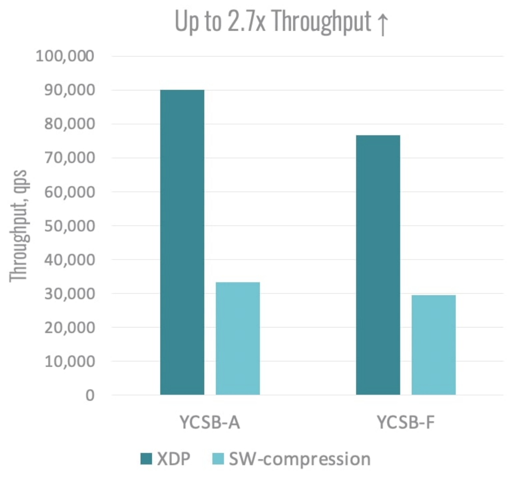


Figure 18: Write-intensive YCSB benchmarks on MongoDB with and without XDP. XDP provides improved performance even while running on a slower QLC SSD (left) as well as while running against software ZSDT compression (right).

the cache size remains the same). Second, the I/O load is spread over multiple SSDs thus making device access faster and the acceleration comparatively less meaningful. Third, the RAID5 error correction codes take a toll on writes.

We now detach one of the drives under RAID5 and keep the TPC-C clients running. The system keeps functioning with less than a 10% transient throughput loss. The drive rebuilds at a rate of 1TB/hour and completes in less than two hours. This demonstrates that the RAID5 drive protection with XDP enables uninterrupted service with minimum performance loss upon recovery.

7.3 MongoDB

We now study how XDP affects MongoDB [41]. Its storage engine, WiredTiger [55], is a copy-on-write B-tree. While it transforms random writes into sequential ones, it exhibits high write-amplification under random writes as each randomly written entry, however small, causes a large node (4K to 16K) to be rewritten.

In this experiment, we include a QLC SSD. While such SSDs have a larger storage capacity than TLC SSDs, they are significantly slower in terms of write speed. We compare two storage settings for Mongo: (1) an XDP-accelerated 16TB QLC drive, and (2) a plain 4TB TLC drive with no acceleration layer. The QLC SSD gives XDP a harder challenge due to its inferior performance.

We use the YCSB benchmark [16], a synthetic workload generator. We experiment with its two core workloads, YCSB-A (50% get and 50% put) and YCSB-F (100% get-modify-put). Keys are sampled from the dataset uniformly at random. The initial dataset size is 1TB. The data is 4x-compressible. The primary key size is 16B, and the value size is 1KB. We run 16 MongoDB instances, each serving 64 concurrent clients.

The left-hand side of Figure 18 shows that with XDP acceleration, the overall throughput gain (in queries per second, or qps) is 2.6x for YCSB-A and 2.2x for YCSB-F. The 99.9% tail latency gains are as much as 3.5x lower for puts and 3.2x lower for gets. These results are achieved despite the slower storage media. XDP achieves these results by seamlessly compressing the data internally using ZSTD and thus reducing the read and write volume.

Finally, we compare the effect of hardware compression versus software compression over the same (TLC) media. We configure the baseline MongoDB deployment to use software ZSTD compression. The storage capacity savings are similar across the two deployments. On the right-hand side of Figure 18, however, we observe that the performance gap is significant. The XDP-accelerated MongoDB instance is 2.7x faster under YCSB-A (90K vs 33.3K qps) and 2.6x faster under YCSB-F. We conclude from this experiment that software-based storage engines must choose between performance and compression. Hardware-accelerated storage such as XDP, however, allows achieving the best of both worlds.

8 CONCLUSION

We introduced the Pliops Extreme Data Processor (XDP), a novel storage engine that lends itself fully to hardware acceleration using a customized processor. XDP seamlessly compresses application data and logs it in storage. It leverages a novel hardware-accelerated hash table to (1) index the data while requiring a 2x lower memory footprint than the best alternative, (2) eliminating redundant storage accesses due to false positives, and (3) making the recovery of a failed SSD faster and less obtrusive. XDP demonstrates that hardware acceleration can do far more than offloading computation from the host CPU: it allows to also optimize for seemingly disparate performance metrics including storage reads and writes, memory footprint, and recovery time. In so doing, it allows overcoming cost contentions that have traditionally been inescapable.

ACKNOWLEDGMENTS

We thank the entire Pliops team, without whom this effort would not have been possible.

REFERENCES

[1] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J Starke, et al. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors: Industrial Product. In *ISCA*.
[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. *ATC* (2008).

[3] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *TC* 65, 3 (2016), 902–915. https://doi.org/10.1109/TC.2015.2435779

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. *SOSP* (2009). https://doi.org/10.1145/1629575.1629577

[5] Apache. [n.d.]. Cassandra. *http://cassandra.apache.org* ([n. d.]).

[6] Apache. [n.d.]. HBase. *http://hbase.apache.org/* ([n. d.]).

[7] Austin Appleby. [n.d.]. Murmur Hash. *https://github.com/aappleby/smhasher* ([n. d.]).

[8] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. *EDBT* (2016).

[9] Jens Axboe. [n.d.]. Flexible I/O Tester. *https://github.com/axboe/fio* ([n. d.]).

[10] Rudolf Bayer and Edward M. McCreight. 1970. Organization and Maintenance of Large Ordered Indexes. *Proceedings of the ACM SIGFIDET Workshop on Data Description and Access* (1970).

[11] Matias Bjørling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. 2013. The Necessary Death of the Block Device Interface. *CIDR* (2013).

[12] Luc Bouganim, Björn THór Jónsson, and Philippe Bonnet. 2009. uFLIP: Understanding Flash IO Patterns. *CIDR* (2009).

[13] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. *SIGMOD* (2018). https://doi.org/10.1145/3183713.3196898

[14] Yann Collet. [n.d.]. LZ4. *https://github.com/lz4/lz4* ([n. d.]).

[15] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comp. Surv.* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. *SoCC* (2010). https://doi.org/10.1145/1807128.1807152

[17] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[18] Edward Corwin and Antonette Logar. 2004. Sorting in linear time-variations on the bucket sort. *Journal of computing sciences in colleges* 20, 1 (2004), 197–202.

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. *SIGMOD* (2017). https://doi.org/10.1145/3035918.3064054

[20] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *SIGMOD* (2016). https://doi.org/10.1145/2882903.2915219

[21] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. *SIGMOD* (2018). https://doi.org/10.1145/3183713.3196927

[22] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *SIGMOD*.

[23] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD*. 365–378.

[24] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: high throughput persistent key-value store. *PVLDB* 3, 1-2 (2010), 1414–1425.

[25] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. *SIGMOD* (2011). https://doi.org/10.1145/1989323.1989327

[26] Christopher Dennl, Daniel Ziener, and Jürgen Teich. 2013. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *FCCM*.

[27] Facebook. [n.d.]. RocksDB. *https://github.com/facebook/rocksdb* ([n. d.]).

[28] Facebook. [n.d.]. ZSTD. *https://github.com/facebook/zstd* ([n. d.]).

[29] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal* (2020).

[30] MariaDB Foundation. [n.d.]. MariaDB. *https://mariadb.org/* ([n. d.]).

[31] Google. [n.d.]. Snappy. *https://github.com/google/snappy* ([n. d.]).

[32] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. 2018. OLTP through the looking glass, and what we found there. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*.

[33] H. V. Jagadish, P. P. S. Narayan, Sridhar Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. *VLDB* (1997).

[34] Ryan Johnson and Ippokratis Pandis. 2013. The bionic DBMS is coming, but what will it look like?. In *CIDR*.

[35] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting widely held SSD expectations and rethinking system-level implications. *ACM SIGMETRICS Performance Evaluation Review* 41, 1 (2013), 203–216.

[36] Kaan Kara and Gustavo Alonso. 2016. Fast and robust hashing for database operators. *FPL* (2016). https://doi.org/10.1109/FPL.2016.7577353

[37] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* (2020).

[38] Weiqiang Liu, Faqiang Mei, Chenghua Wang, Maire O'Neill, and Earl E Swartzlander. 2018. Data compression device based on modified LZ4 algorithm. *IEEE Transactions on Consumer Electronics* (2018).

[39] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. *FAST* (2016).

[40] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.

[41] MongoDB. [n.d.]. Online reference. *http://www.mongodb.com/* ([n. d.]).

[42] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Data processing on FPGAs. *PVLDB* (2009).

[43] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: a query compiler for FPGAs. *PVLDB* (2009).

[44] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. *ATC* (1999).

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

[46] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, and Others. 2010. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Op. Sys. Rev.* 43, 4 (2010), 92–105.

[47] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. 2020. Open-Channel SSD (What is it Good For).. In *CIDR*.

[48] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. 2017. Fast Scans on Key-Value Stores. *PVLDB* 10, 11 (2017), 1526–1537.

[49] Mendel Rosenblum and John K Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *TOCS* 10, 1 (1992), 26–52. https://doi.org/10.1145/146941.146943

[50] Robert R Schaller. 1997. Moore's law: past, present and future. *IEEE spectrum* (1997).

[51] Justin Sheehy and David Smith. 2010. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. *Basho White Paper* (2010).

[52] V Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a real-time operational dbms. *Proceedings of the VLDB Endowment* (2016).

[53] Radu Stoica and Anastasia Ailamaki. 2013. Improving Flash Write Performance by Using Update Frequency. *PVLDB* 6, 9 (2013), 733–744.

[54] TPC. [n.d.]. Specification of TPC-C benchmark. *http://www.tpc.org/tpcc/* ([n. d.]).

[55] WiredTiger. [n.d.]. Source Code. *https://github.com/wiredtiger/wiredtiger* ([n. d.]).

[56] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and how?. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads ({INFLOW} 14)*.