Retina: Cross-Layered Key-Value Store using Computational Storage

Naga Sanjana Bikonda

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Changwoo Min, Chair
Cameron D. Patterson
Haibo Zeng

February 18, 2021
Blacksburg, Virginia

Keywords: Computational storage, Key-Value store, Crash consistency

Retina: Cross-Layered Key-Value Store using Computational Storage

Naga Sanjana Bikonda

(ABSTRACT)

Modern SSDs are getting faster and smarter with near-data computing capabilities. Due to their design choices, traditional key-value stores do not fully leverage these new storage devices. These key-value stores become CPU-bound even before fully utilizing the IO bandwidth. LSM or B+ tree-based key-value stores involve complex garbage collection and store sorted keys and complicated synchronization mechanisms. In this work, we propose a cross-layered key-value store named Retina that decouples the design to delegate control path manipulations to host CPU and data path manipulations to computational SSD to maximize performance and reduce compute bottlenecks. We employ many design choices not explored in other persistent key-value stores to achieve this goal. In addition to the cross-layered design paradigm, Retina introduces a new caching mechanism called Mirror cache, support for variable key-value pairs, and a novel version-based crash consistency model. By enabling all the design features, we equip Retina to reduce compute hotspots on the host CPU, take advantage of the on-storage accelerators to leverage the data locality on the computational storage, improve overall bandwidth and reduce the bandwidth network latencies. Thus when evaluated using YCSB, we observe the CPU utilization reduced by **4x** and throughput performance improvement of **20.5%** against the state-of-the-art for read-intensive workloads.

Retina: Cross-Layered Key-Value Store using Computational Storage

Naga Sanjana Bikonda

(GENERAL AUDIENCE ABSTRACT)

Modern SSDs are getting faster and smarter with near-data computing capabilities. Due to their design choices, traditional key-value stores do not fully leverage these new storage devices. These key-value stores become CPU-bound even before fully utilizing the IO bandwidth. LSM or B+ tree-based key-value stores involve complex garbage collection and store sorted keys and complicated synchronization mechanisms. In this work, we propose a cross-layered key-value store named Retina that decouples the design to delegate control path manipulations to host CPU and data path manipulations to computational SSD to maximize performance and reduce compute bottlenecks. We employ many design choices not explored in other persistent key-value stores to achieve this goal. In addition to the cross-layered design paradigm, Retina introduces a new caching mechanism called Mirror cache, support for variable key-value pairs, and a novel version-based crash consistency model. By enabling all the design features, we equip Retina to reduce compute hotspots on the host CPU, take advantage of the on-storage accelerators to leverage the data locality on the computational storage, improve overall bandwidth and reduce the bandwidth network latencies. Thus when evaluated using YCSB, we observe the CPU utilization reduced by **4x** and throughput performance improvement of **20.5%** against the state-of-the-art for read-intensive workloads.

Dedication

To my parents and friends.

For all their love and support.

Acknowledgments

I am deeply grateful to Dr.Changwoo Min for providing great opportunities and always believing in me. His endless support and guidance allowed me to grow academically and personally. Thank him for all his patience and kindness.

I would also like to thank my committee members, Dr.Cameron Patterson and Dr.Haibo Zeng, for their guidance and support.

I want to thank Dr.Wookhee Kim and Madhava Krishnan for being such great mentors. Apart from helping with technical issues, Madhava has been a good friend who made my master's journey worthwhile.

My support system, my friends Balvansh Heerekar, Parthiv Kativarapu, Shagun Johri, Vaibhav Sundharam, and Keerthana Bhogi. I am grateful for their constant love and support.

Finally, I cannot thank my parents enough for their sacrifices and never-ending support.

Contents

List of Figures

Chapter 1

Introduction

With the exponential increase in data, social media, big data, streaming applications, etc., datastores need to scale and perform with sub-millisecond response time. In this regard, NoSQL data stores offer high performance when dealing with large unstructured data compared to relational databases. Key-value databases [5] are a category of NoSQL databases that are often deemed as highly performant, efficient, and scalable. This class of datastores has been widely deployed for cloud applications to handle cache and session management, message queueing, and online shopping. In this work, we target persistent database designs where the entire data set cannot be accommodated in the main memory.

Based on the definition, key-value stores are implemented as a simple map/dictionary of unique keys and blobs of type agnostic values. Traditionally, to bridge the disparity of speeds between CPU and the storage devices, key-value store designs used techniques to use the CPU's computing power to optimize disc accesses. Thus most key-value stores are implemented as either B+tree structures [12, 38, 43] or Log-Structured Merge (LSM) trees [39], where metadata and hot key-value pairs are stored in caches to minimize data access. Moreover, key-value pairs are sorted on the disc to improve performance with sequential writes and support range operations. While costing extra CPU cycles, these optimizations resulted in improved performance as they mitigate the storage bottleneck.

With the advent of the new generation of NVMe SSD devices which provide with 800K IOPS for read and 135K IOPS for writes [7], the traditional key-value store designs cannot saturate

the IO bandwidths available. In a state-of-the-art work, KVELL [34], the authors conducted an experiment where a YCSB workload(with 50% put and 50% get) is run on RocksDB to collect the CPU and I/O bandwidth utilization. The average bandwidth utilized in RocksDB was measured to be much less than the maximum achievable bandwidth. Furthermore, for the same experiment, the CPU utilization is nearly 100%. The authors conclude that such under utilization of IO bandwidth is because the CPU becomes the bottleneck. Most traditional key-value stores are designed to use CPU cycles to perform heavyweight tasks such as compute-intensive garbage collection, expensive logic to store sorted keys on disk, and complex synchronization mechanisms to achieve higher IO performance. As the new storage has gotten faster, the CPU becomes the bottleneck even before the IO bandwidth is entirely saturated.

To facilitate full saturation of IO provided by NVMe SSD's, the research community has been exploring alternative designs away from key-value stores based on block addressable SSD [19, 45]. The key-value management layer is completely offloaded onto the SSD. This set of new key-value stores are called KV-SSD [25] as they handle object management directly on the SSD to reduce the use of host-side resources and IO overhead. The datastore logic is handled inside the firmware processer of the KV-SSD to achieve the benefits of isolated execution within each storage drive, avoiding multiple levels of IO abstractions on the host and eliminating the need for expensive logging based consistency mechanisms. This work proves that restructuring the datastore design to leverage the compute near storage has great potential.

With the slow down of CPU and DRAM scaling and stagnation of interconnect bandwidth, excessive data movement over the network is becoming a bottleneck for data-intensive applications. A potential solution to address this problem is to offload compute near storage. This field of research is called Near Data Computation or Computational Storage. The

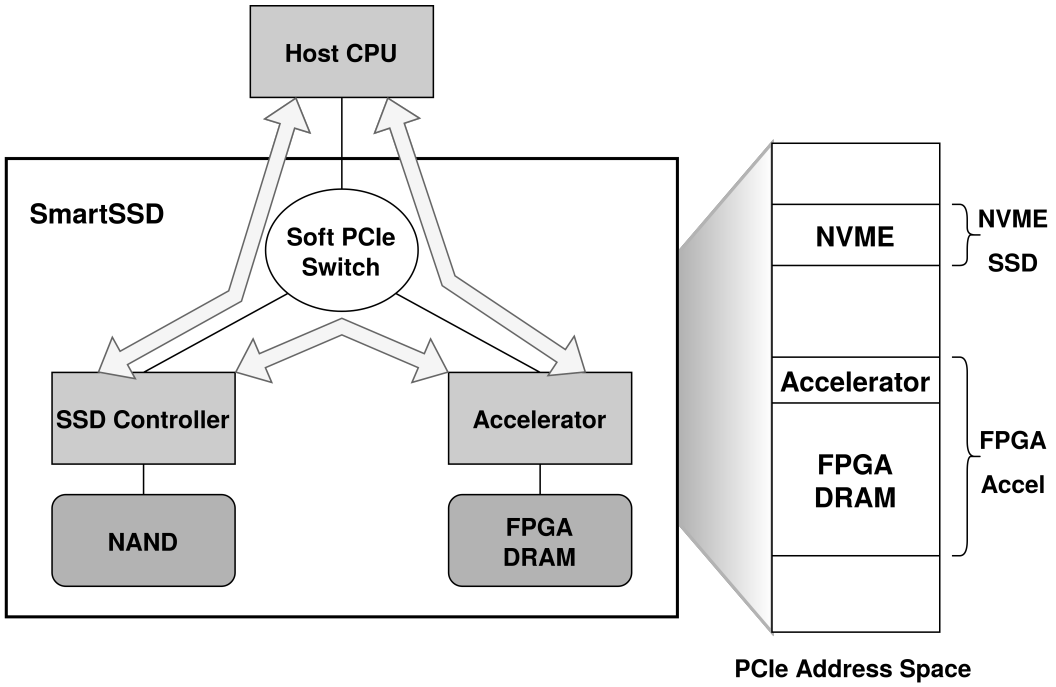

Figure 1.1: SmartSSD Computational Storage Device (CSD) hardware architecture.

problem in identifying computational storage is that an exact definition was not available until recently. The definitions are now set out by the SNIA on Computational Storage drive research and standardization in storage and networking, with a number of standards being drafted [8, 9]. Most commercially available computational storage either amped up the SSD controller or attached an external accelerator to the SSD to realize the compute near storage.

Samsung CSD (Computational Storage Drive) is launched with a Xilinx Kintex FPGA to bring computational storage capabilities in a standard form factor (refer to Figure 1.1) [26]. In SmartSSD, a part of FPGA is used as a PCIe switch so that the host system can view SSD and FPGA as two distinct PCIe endpoints. Thus all the PCIe traffic goes through the FPGA before reaching the SSD. Internally, the computational storage uses the on-platform PCIe switch and FPGA memory to route data transfers between the NVMe SSD and FPGA. The data movement across the internal data path is termed *peer-to-peer (P2P) memory.*

Until now SmartSSD has been used in the following three main categories:

- Storage services: compression/decompression, metadata management, etc,
- Big data applications: DB query, log analytics, etc,
- AI/ML applications: object detection, streaming, etc,

Though SmartSSD has not been used to implement datastores, peer-to-peer memory creates an opportunity to offload compute to FPGA with the added advantage of high internal bandwidth between SSD and FPGA. Thus in this work, we propose a new key-value store based on computational storage to remove CPU bottlenecks and fully utilize the SSD's IO bandwidth. With Retina, we aim to take a *cross-layered approach* where the datastore implementation and scope of control is divided between the host system and SmartSSD. With the help of the peer-to-peer memory, host CPU can offload data plane operations to FPGA and issue read/write operations from SSD to FPGA (vice-versa). Thus reducing network (*i.e.*, PCIe) traffic significantly.

In summary, the following are a list of Retina's main contributions:

- We propose and implement a new cross-layered design that provides high resource utilization and performance; we achieve this by splitting the key-value store into two separate layers to saturate the host CPU in the the control plane and IO bandwidth on NVMe SSD.
- We show that by splitting the cache between the host system and SmartSSD we can fully isolate control, improve data locality, and reduce network latencies.
- We propose a novel version-based crash-consistency model that avoids logging to save the number of synchronous IO calls to SSD.
- We evaluate our Retina performance against state-of-the-art key-value stores using YCSB workloads.

The rest of the thesis is organized as follows. §2 presents the background and motivation. In §3, we review the overview of Retina, and in §4, we describe Retina's system design in detail. We evaluate Retina in §6, discuss related work in §8, present future work in §9, and finally conclude in §10.

Chapter 2

Background and Motivation

2.1 Conventional Key-value Stores

Many conventional key-value storage systems scale inefficiently to achieve high SSD bandwidth as they incur high data management and IO overheads. For instance, RocksDB [20] processes user requests asynchronously by using buffered IO with log-structured merge design. However, it needs a background process to handle garbage collection called compaction. As part of compaction, the key-value store reads stale and new versions of data to merge and reorganize the stable data back into the LSM-based structure. While the user requests optimize to exploit SSD's performance, the backend garbage collection tasks constantly run to restructure the data store without slowing down the frontend tasks. However, to saturate the IO performance of the SSD, the user tasks take full advantage of buffered IO, but this indeed slows down the system by spawning many more background cleanup tasks [13]. It becomes difficult to scale as the system needs more CPU power to saturate increasing number of storage devices [34].

Generally, key-value stores promise data consistency using write-ahead-logging (WAL). Consistency with such a technique is guaranteed by first writing any updates in the datastore to a log file, synchronously persisting it to the disk, and updating the actual file in place. In this process, WAL amplifies the number of writes to the disk by (at least) 2x and reduces the throughput by half. Unlike the foreground processes that use buffered IO and burst read-

/write to maximize the SSD performance, WAL introduces high-latency synchronous IO access into the request processing. Because key-value stores are implemented as an abstraction layer over block layer, they require separate metadata (undo or redo log file) when using WAL to ensure consistency between metadata and data persistence. Thus, using a logging-based consistency model results in increased write amplification, IO interdependencies, and slow request processing.

To achieve high performance, traditional key-value stores need expensive garbage collection processes, which increase the CPU resource utilization and IO accesses to SSD. Each compaction task involves reading a range of pages into the page cache. The cache can be quickly polluted to result in added read-write amplification and negatively affecting the throughput. Hence, achieving high performance and scalability is hindered by CPU utilization and read-write amplification limitations.

The issues caused by compaction/defragmentation, logging-based consistency, and added IO amplification get more significant in the case of SSD. Consider a commodity SSD that provides a bandwidth of 2GB/s. To saturate such a device with 4KB requests, the key-value store needs to process each request in 6.5 microseconds. The KV-SSD paper [27] experimented with quantifying CPU requirements for maximizing SSD bandwidth utilization. They measure the bandwidth and CPU utilization with fio benchmark([3]) while running sequential IO on NVMe SSD for 4KB and 64KB sized pages. The authors conclude the outcome with the following two takeaways:

- To saturate one SSD, the host system needs at least one CPU.
- Increase in number of I/O abstraction layers on the host also increases the need for CPU processing power.

Therefore with an increase in the number of SSD's, the demand for CPU also amplifies,

resulting in high resource contention on the host.

2.2 KV-SSD

KV-SSD [27] completely offloads datastore features onto SSD devices to not only improve the key-value store's performance but also reduce host resource contentions. Though the central concept of KV-SSD is promising, all the existing models are hash-based, which results in poor tail latency. Conventional hash-based KV-SSD maintains a mapping of a signature (key) and a pointer that holds the location of the value in a hash table in the SSD controller's DRAM.

The biggest downside of the hash-based design is to accommodate the entire key-value store inside the limited DRAM on the SSD. To optimize the DRAM usage, some designs store signatures of keys instead of keys themselves on controller DRAM. Though this optimization alleviates the space issue, it results in the signature collision when two or more keys have the same signature. As a further optimization, an implementation can store the hot key-value pairs on the DRAM and the remaining on the SSD. However, this type of design can negatively affect the read latency when a key is not found in the controller's DRAM. In such a case, the lookup time complexity would be $O(1 + a)$, where a is the miss rate. The performance can worsen even more in the signature collision with DRAM miss, where the read tail latency would be unbounded.

As an optimization to bound the inconsistent tail latency, one can use advanced collision resolution policies such as Hopscotch [24] and Cuckoo [35] hash tables. However, using such schemes can cost the expensive application rehashing and reduced write bandwidths. It is predicted that the capacity of DRAM only increases up to 1.13 times a year [36], while flash memory increases 1.43 times a year. When the data set size increases, a lesser fraction of

total key-value pairs can be accommodated in the controller's DRAM, which exacerbates all the issues mentioned above.

Further, the hash-based structure is not suitable to support range operations.

Another widely used key-value store implementation is LSM-tree. While traditionally, it is fully implemented on the host side, it is considered a highly optimized design that supports varied set of workloads. In recent times, new LSM-based designs such as LightStore [15] and iLSM-SSD [31] were introduced with the same motivation as KV-SSD. Using LSM-tree design for KV-SSD can be beneficial as it bounds the worst-case latency to $O(height - 1)$, where *height* signifies the number of levels in the LSM tree structure. To optimize the lookups, LSM-tree implementations store the key-value pairs sorted and use bloom filters to narrow down the key-value ranges.

As discussed earlier, some of the drawbacks of conventional host-based LSM-tree design such as expensive garbage collection, read-write amplification, etc, still persist. New challenges arise with the LSM-tree being completely implemented on the SSD and its controller's DRAM. From the traditional host-based LSM key-value store, we know that CPU bottlenecks result in throughput degradation. Nevertheless, the write bandwidth further suffers due to the device-side wimpy ARM CPU's slow sorting of key-value pairs during compaction. Moreover, it is shown that using bloom filters to cater to fast lookup is no more beneficial because of slow filter reconstruction and limited capacity on the controller's DRAM.

To address the performance bottleneck is LSM-based KV-SSD, Pink [25] avoids using bloom filters to control the performance degradation but instead uses level pinning. Using level pinning provides fast DRAM lookup for the top k levels and improves compaction by eliminating IO read/writes from flash as these levels already reside on the controller's DRAM. The work uses hardware acceleration to improve sorting operation in the compaction phase.

Finally, to reduce compaction costs, Pink uses asynchronous garbage collection by storing merged key-value pair indices on the top layer, L0. Thus curbing read-write amplification.

Although KV-SSD's inherently eliminate consistency overheads, Pink's design requires logging of its L0 level due to the constraint on the number of capacitors in SSD's DRAM. As the flash memory scales fasters than DRAM, the design eventually can pin fewer levels on the DRAM resulting in degraded lookup performance.

2.3 SmartSSD and Peer-to-Peer Memroy

Taking the design path similar to traditional key-value stores involves resource contention in host CPU, expensive data consistency, and read-write amplifications. On the other hand, taking the KV-SSD approach also has issues regarding the limit on capacity on DRAM and the roadblock of DRAM not being able to scale equal to flash. Therefore, implementing the entire key-value store on just the host or SSD is suboptimal.

We propose a new computational storage-based key-value store that splits the host and the computational storage design. Thus fully leveraging the combined compute power of host CPU and storage and reducing the network latencies by utilizing peer-to-peer memory, which we will discuss below.

The SmartSSD Computational Storage Drive(CSD) connects the SSD and the FPGA over the internal data path to enable high-speed data transfer called the peer-to-peer data transfer. This is achieved with the help of an on-chip soft PCIe switch and the FPGA's device memory. The SmartSSD maps both the NVMe SSD and FPGA DRAM onto the PCIe bar memory. The FPGA DRAM part exposed to the PCIe Bar is called the Common Memory Area(CMA). Thus the host can issue I/O calls between the common memory area and the SSD, where

the p2p transfer directly copies data between the two mapped memory regions to enable near-data computation and avoid network latencies.

Thus peer-to-peer memory is a critical component in offloading compute to storage, reducing network overhead and fully realizing our new cross-layered key-value design.

Chapter 3

Overview of Retina

Retina's key-value store based on SmartSSD, a Samsung's Computational Storage Device, has the following design goals:

- **Maximize IO utilization and throughput**: Retina should maximize the IO bandwidth utilization while not overloading the host CPU. The key-value architecture should maximally utilize the P2P memory on computational storage to realize improved overall throughput. Compute should be offloaded from the host CPU to SmartSSD's FPGA to leverage its data locality.
- **Improve tail latency**: Retina should have a worst-case read latency of $O(1)$. The datastore implementation should allow a consistent tail latency and control the read-write amplifications.
- **Lightweight log-free consistency**: Retina should avoid WAL or journaling to provide data consistency. The scheme used should reduce the number of IO accesses involved.
- **Achieve scalability and reduce critical section**: Retina should be able to scale across multiple computational storage drives efficiently. The architecture should be split between host and storage to allow seamless disaggregation.

No prior key-value store has achieved all the design goals. In the following sections, we discuss how Retina realizes all the mentioned goals with the below three main contributions: (1) cross-layered design paradigm (§3.1), (2) mirror cache (§3.2), and (3) version-based crash

consistency (§3.3).

3.1 Cross-Layered Design Paradigm

Retina adopts a design paradigm that splits the underlying datastore into layers to best fit host and computational storage capabilities. It is named *cross-layered approach* as the design separates the implementation and scope of control between host and storage nodes to fully leverage the near-data accelerator and reduce network overhead. To understand the benefits of a cross-layered approach, one needs to understand the underlying data structure of a general key-value store. Most of them are either tree-based or hash-based. Though hash-based structures achieve high lookup performance, they suffer due to rehashing in the case of an increased data set, non-deterministic tail latencies, and are not suitable for range operations. B+-tree based designs can be split into the following two layers([37]):

- Search layer: consists of the internal nodes of a tree structure
- Data layer: made up of the collection of leaf nodes

Maximize computational storage's bandwidth utilization: Now, we discuss how this basic underlying structure made up of search and data layer is manipulated to maximize SmartSSD's bandwidth utilization. Firstly, we want to mitigate the CPU bottleneck by offloading compute to SmartSSD's FPGA. To achieve this, we split the scope of control to assign the search layer manipulations to host CPU and data layer is handled by SmartSSD's FPGA, as shown in Figure 3.1. Secondly, we further split the implementation to place the search layer on host DRAM and the data layer on SmartSSD. Dividing the design in such a manner maximizes locality between the compute resources and storage. Finally, with the help of peer-to-peer memory, FPGA on SmartSSD can directly access the flash memory, thus reducing data hops over the PCIe bus.

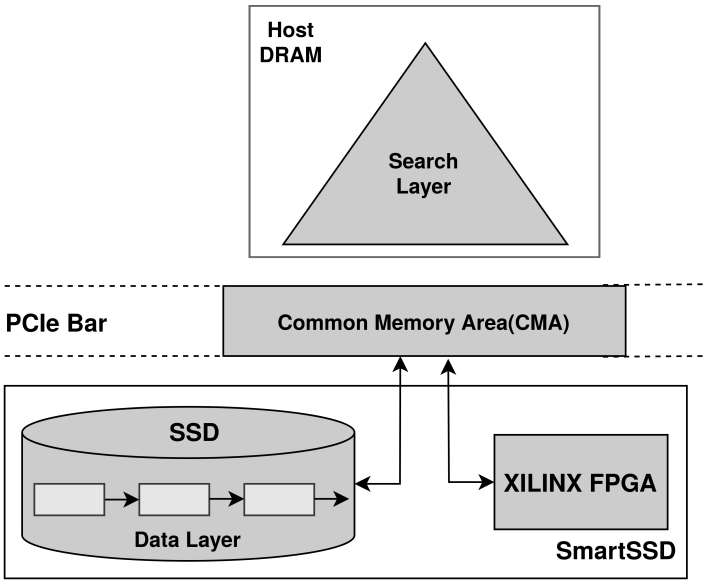

Figure 3.1: Overview of cross-layered key-value store design in Retina.

Maximally utilize host CPU to act as the control plane: Instead of using the host's computing resources for actual data manipulation, we propose to use the host CPU only on the control plane. Like in traditional designs, the host CPU receives requests from applications to lookup the key, but instead of fetching the data to host DRAM, it passes control to the FPGA on SmartSSD. Thus by acting in the control plane, the host CPU triggers FPGA kernels and issues IO requests for data transfer between SSD and P2P memory. Though FPGA handles the actual data manipulation, the host CPU handles the corresponding metadata to provide light-weight concurrency control, cache management, crash consistency, and garbage collection.

Maximally utilize FPGA on SmartSSD in the data plane: SmartSSD's FPGA is used in the data plane to receive requests from the host CPU directly. The host CPU makes sure that the P2P memory is populated with the necessary data pages at every key-value store request. Once the data is populated, the FPGA takes control to execute the respective kernel to read/write the key-value pairs. IO-intensive operations such as crash consistency

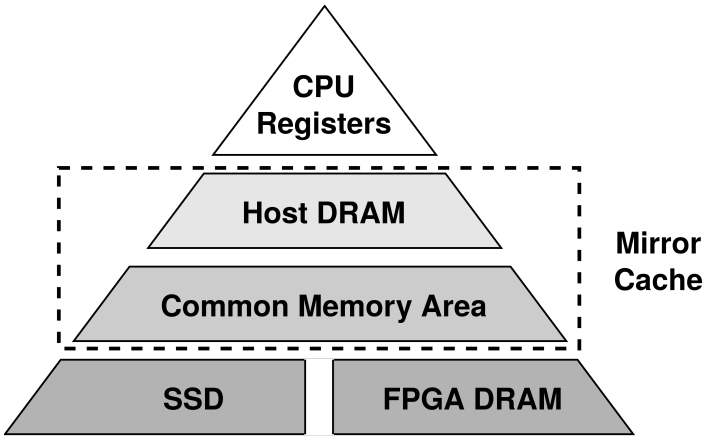

Figure 3.2: Logical memory hierarchy with Retina.

and garbage collection no longer incur high data hops over the network with such a design. Further, the addition of crucial compression and decompression engines on host and FPGA can potentially result in higher overall throughput.

3.2 Mirror Cache

With the cross-layered design, we have IO requests directly happening between the SSD and the P2P memory without the involvement of the host DRAM. While FPGA handles all the data operations, using a host-based cache can increase data hops over the network. Thus Retina proposes a new caching mechanism in alignment to the cross-layered design paradigm called *Mirror cache*. One key observation is that the common memory area (CMA, the P2P memory) can be logically added to the memory hierarchy right below the host DRAM and above the SSD as shown in Figure 3.2. As both the SSD and FPGA can access the p2p memory directly, we can use this layer of memory to design a new cache.

We use CMA to extend the cross-layered design paradigm to propose a new layered cache architecture. The cache is named Mirror cache as it uses the CMA to cache the recently

accessed pages and host DRAM to cache its corresponding metadata at the same offset.

Mirror cache design allows the host CPU to manage all the cache management operations (fetch, invalidate, and flush) and concurrency control by maintaining metadata that is just a fraction of the actual data pages. On the other hand FPGA directly accesses data pages as dictated by the host CPU without worrying about the cache state. Therefore, Mirror cache provides a distributed caching mechanism that fully leverages data locality, reduces network traffic, and achieves scalability.

3.3 Version-based Crash Consistency

Traditional key-value stores use the write-ahead logging (WAL) technique to guarantee data durability and operation atomicity. In WAL, any write made to the datastore first stored in a log file persisted onto the disk, and only then the actual data location is modified.

Though WAL guarantees data consistency, it has a negative impact on the overall datastore performance. WAL involves writing two sets of data to the disk for every update, reducing the overall datastore throughput by (at least) half. Even in buffered IO, the background data logging falls in the critical path to make the operations progress sequentially.

We propose a new lightweight *version-based crash consistency (VCC)* to avoid expensive logging-based crash consistency. Our crash consistency model bases its design on versioning data pages and taking advantage of 4KB atomic read/writes in the filesystem. Each data page is embedded with lightweight version metadata, which is used to track the live pages on the storage. With VCC, maintaining consistency does not result in write amplification. Our hybrid scheme allows in-place updates for 4KB updates and out-of-place updates for pages of size greater than 4KB. Using VCC, one update operation results in only one IO

request to the disk, thus improving the overall key-value store throughput.

Chapter 4

Retina Design

In this chapter, we present a detailed design of the Retina. First, we will start by defining the best-suited structure used to implement our cross-layered architecture (§4.1). Then we dwell on the internals of Mirror cache and the concurrency model (§4.2). Further, we discuss how Retina supports variable-length keys (§4.3), the working of the VCC model (§4.4), and the step-by-step process of crash recovery (§4.5). Next in Figure 4.15, we discuss Retina's concurrency model to set some foundation in OpenCL, individually tackle host and kernel side concurrency and go through end-to-end multi-threading flows. Finally, we present Retina supported API calls and their logic flows in §4.7.

4.1 Cross-Layered Architecture

When choosing an ideal data structure to implement the key-value store, we must consider the following requirements:

- Fast lookup time complexity.
- Increasing dataset size should not degrade lookup performance.
- Should support both point and range operations.

Generally, key-value stores are either hash-based or tree-based. In hash-based structures, every value is indexed by a unique key. Though hash-based structures offer point lookups

with $O(1)$ time complexity, they are not optimal for range operations. Moreover, they run into expensive collision resolution and rehashing in the event of increased dataset size. Any tree-based structure stores the actual key-value pairs in its leaf nodes and its corresponding partial key in the tree's internal nodes. Further, the leaf nodes can also be chained to support range operations. We implement Retina as a tree structure where we split the design into two layers: (1) the *search layer* consisting of the partial keys and (2) the *data layer* consisting of the actual key-value pairs.

4.1.1 Search Layer

The time lookup time complexity of a B+tree-based index is $O(log(n))$, where n is the number of key-value pairs stored. With the increasing size of the datastore, the time to lookup keys with a B+tree-based index would also increase. To eliminate such dependency on the number of keys, we consider a trie-based index whose time complexity depends on the size of the key. Thus trie based structures can offer faster lookups in comparison to B+tree structures. However, trie structures suffer performance when dealing with long keys. This shortcoming can be relieved with the help of path compression, where internal nodes with single branches are merged to reduce the tree height (thus the length of the partial key compressed). Adaptive Radix tree (ART) [32, 33] is a trie-based structure that uses path compression to resize the internal nodes and provide fast lookup performance and high space efficiency. Thus we implement our Search layer by extending the ART structure.

4.1.2 Data Layer

Among the three indexes, the B+tree-based index provides the best scan performance as the structure clusters keys in a leaf node such that it results in minimal pointer chases for

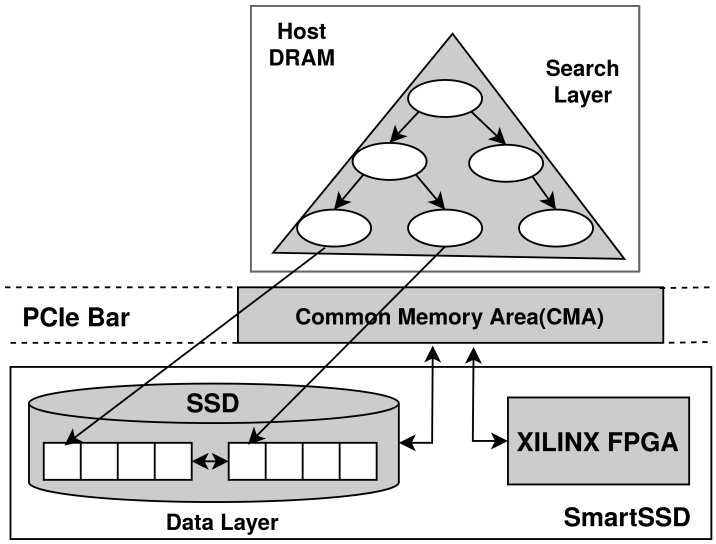

Figure 4.1: Cross-layered key-value store design.

range operations. B+tree's leaf chaining also simplifies reaching the next leaf node within the range. On the other hand, the ART (trie-based) index performs poorly as it does not store any key order information to allow fast-range queries. Because of the lack of chaining in ART, range requests are treated as a set of separate point lookups in the worst case. Thus in order to design the best-suited structure, we modify the ART's implementation to adopt leaf node clustering chaining. We pack the leaf nodes in the data layer as a slotted doubly linked list to realize this. Furthermore, the data layer is a list of sorted key ranges where each leaf node internally has a cluster of unsorted keys. We refer to the leaf node as a data node throughout the current work. The search layer stores the location of each data node in the data layer with the help of its leaf nodes called the *Anchor key*. An invariant to the working of the cross-layered approach is: *each data node stores keys in a particular range and maintains the anchor key as its minimum key that marks the start of that range*

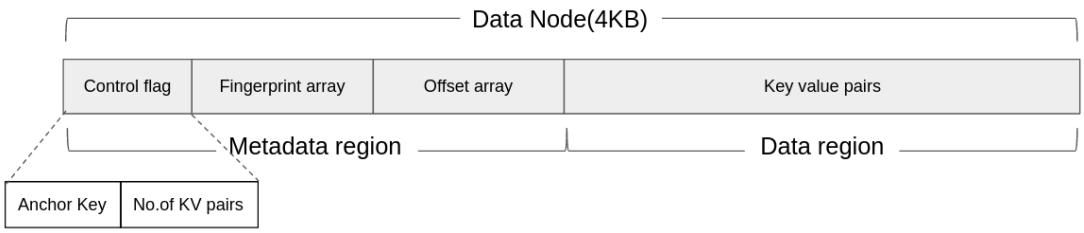

Figure 4.2: Data layout: Data Node with fixed-size key-value support.

4.1.3 Cross-Layered Structure

To realize the benefits of SmartSSD's computational capabilities, we split the implementation and scope of control between host and SmartSSD. The search layer consisting of the internal nodes of ART is stored on the host DRAM. The data layer consisting of the leaf nodes (*i.e.*, slotted doubly linked list) of the ART is stored on the SSD as shown in Figure 4.1. The control on each layer is also separated such that the host CPU handles the search layer, and FPGA on the SmartSSD controls the data layer.

To align with SSD access granularity and simplify crash consistency by taking advantage of 4KB atomic read/writes, we set the size of each data node in the data layer to 4KB. As the data node stores a range of keys in an unsorted manner, random key lookup can be expensive. Thus we maintain light-weight fingerprints (1-byte hash of a key) at the beginning of the page to speed up key lookup. Our data layout is similar to a slotted-paging structure where variable-sized key-value pairs are stored on a fixed-sized page. As shown in Figure 4.2, the data node consists of a metadata region at the beginning of the page and a data region adjacent to metadata. The metadata region maintains control flags, fingerprints of keys inserted, and offsets specifying the actual KV pairs' location. Whereas the data region stores the actual key-value pairs.

When a key is inserted into the data node, the below steps are followed:

- Increment the number of keys flag in the metadata.

- Calculate the fingerprint of a key and insert at the end of the fingerprint array.
- Get the free offset from the metadata, calculate offsets for the key-value pair, and insert the key-value pair offsets in the offset array.
- Insert the key-value pair at the free offset in the data region.

4.2 Mirror Cache: Cross-Layered Cache Design

Mirror cache is a new caching technique that splits the cache into two layers, namely, the *host cache* and *kernel cache* to maximize computational storage's bandwidth utilization (shown in Figure 4.3).

Host Cache is implemented as a lossy hash table. Instead of chaining the bucket with new entries at hash collision, our implementation uses an LRU (Least Recently Used) replacement policy. We simplify the design by obviating the costly collision resolution mechanism. Moreover, with the lossy nature of our hash table, we also improve performance by maintaining hot data in the Mirror cache. The host cache uses the logical block address as the key to navigate the hash table to narrow it down to a cache metadata packet. The metadata stores important information such as logical block address (LBA), timestamp to allow cache replacement, and readers-write lock (RWLock) to handle concurrent read and write accesses. With this lightweight metadata, the host cache enables the host CPU to take control of complex operations such as cache eviction and concurrency control.

Kernel Cache is implemented as a flattened array of data nodes that are physically created with an array of 4KB sized buffers in the P2P (CMA) memory. As the kernel cache mirrors the host cache, operations like cache population, invalidation, and flush are handled by the host CPU. Thus the host CPU issues IO read/write from/to kernel cache while the kernel cache directly consumes the data to perform manipulations. Thus, the data node unit does

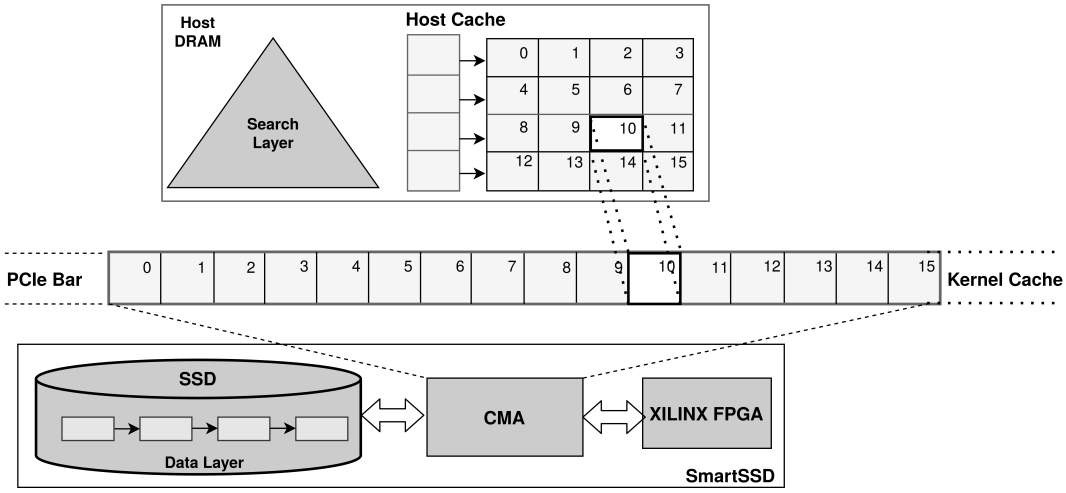

Figure 4.3: Mirror cache: cross-layered cache architecture.

not maintain any caching or concurrency-related information on the kernel cache.

This structure is called mirror cache, as each entry in the kernel cache stores the actual data page at the same offset as its respective metadata on the host cache. This design abides by the cross-layered paradigm as the host CPU takes care of all the cache maintenance operations and concurrency control, whereas kernel cache is directly acting as the input to functions on FPGA.

In Figure 4.4, we discuss the flow of logic for a cache hit in the Mirror cache. As a first step of processing any request in Retina, the host CPU, looks up the partial key in the search layer ❶. Once the leaf node is found, we use its logical block address to navigate the host cache ❷. We have a mirrored cache hit if metadata corresponding to that LBA is found. Based on whether we cater to a read/write operation, we acquire a read/write lock on the metadata entry to increment the timestamp and progress with the request ❸. Once the host cache entry is locked, the host CPU moves forward and triggers the FPGA kernel bypassing the kernel cache buffer at the same offset as the host cache as the input ❹. By the end of kernel execution, the kernel cache buffer is populated with the latest version of the data

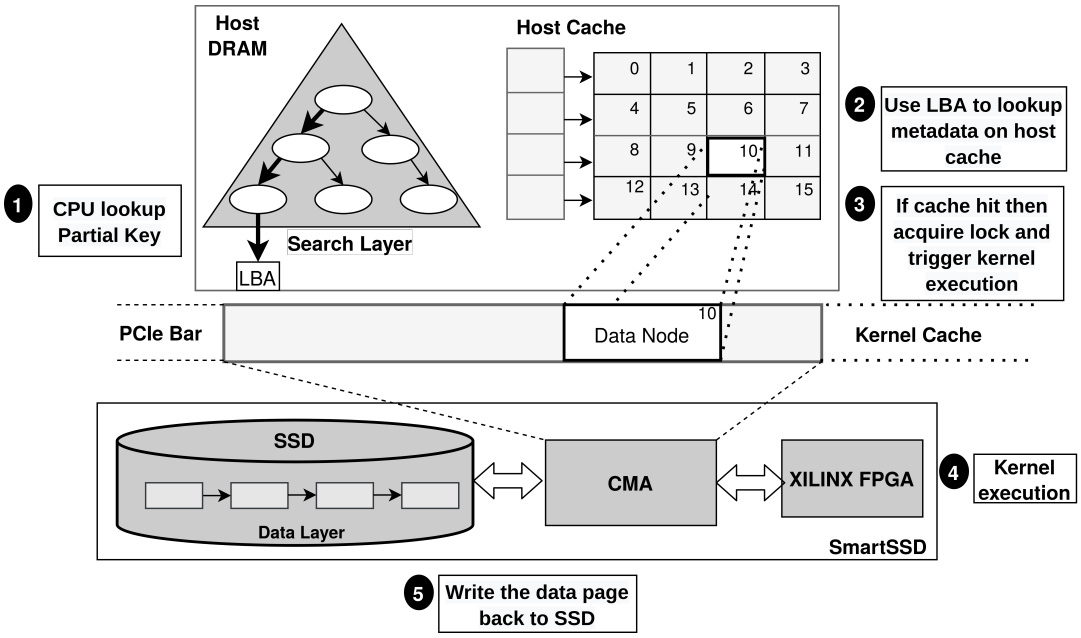

Figure 4.4: Illustration of cache hit in Retina's Mirror cache.

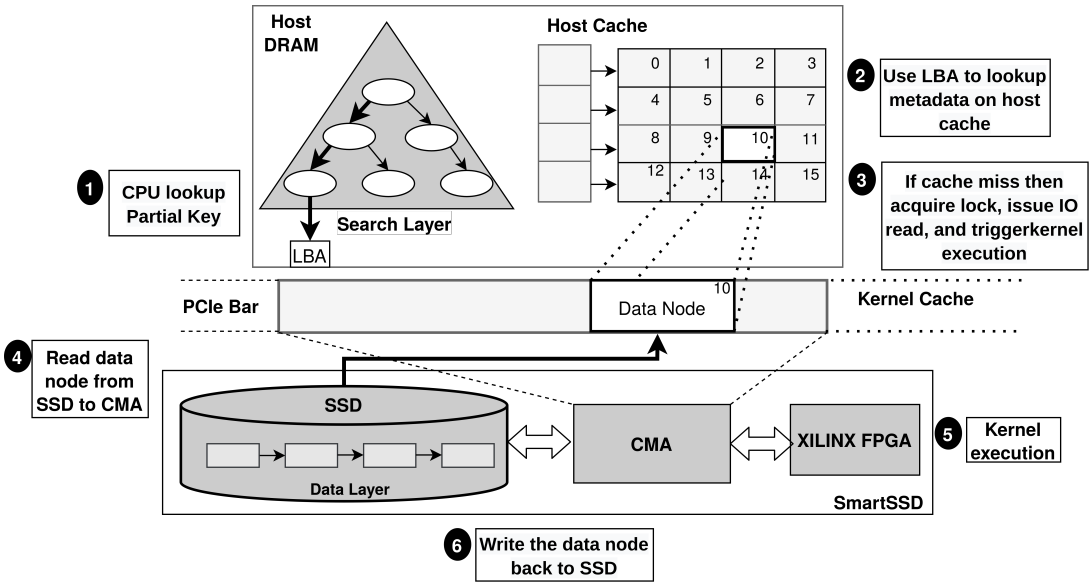

Figure 4.5: Illustration of cache miss in Retina's Mirror cache.

node. Finally, the control is passed back to the host to persist the data node to the SSD ❺.

In Figure 4.5, we present the cache-miss scenario of Mirror cache. Like the cache hit scenario, all the operations first look up the search layer to find the logical block address ❶ and then

use the LBA to find a valid metadata entry in the host cache ❷. If the LBA is not found in the host cache, we have a cache miss. In such a case, we use the least recently used replacement (LRU) policy to assign a metadata slot to the incoming operation after evictin a page if necessary. Acquire either a read/write lock based on the operation to populate the metadata with the new LBA and current timestamp ❸. As the next step, the host CPU issues an IO read operation from the SSD to the kernel cache buffer at the specified offset ❹. Once the kernel cache is populated, the host CPU triggers the FPGA kernel ❺. After kernel execution, like the cache hit case, the control is passed back to the host CPU to persist the changes from the kernel cache to the SSD ❻.

4.3 Variable-Length Key-Value Support

Most AI/ML applications deal with large data such as images, videos, etc., spanning multiple pages. As each data node has a fixed size of 4KB, accommodating bigger KV needs an extension to the design. Thus to support variable KV pairs, we implemented a slab allocator (a fixed-size block allocator, similar to host DRAM's memory allocator) that assigns extension pages (of the size 4KB, 8KB, 16KB, etc.) to each data node to accommodate large KV pairs. Whenever a data node cannot fully accommodate a KV pair, we request a suitable extension page from the slab allocator and chain the extension page to the data node. To support chaining in the data node, we add more control flags to the data node's metadata.

Note that the slab allocator physically implements a slab as a file on the SSD with fixed-sized pages. Moreover, the parent data nodes and extension nodes are maintained in different slabs. Within extension nodes, each available size of extension page (*e.g.*, 4, 8, 16, 32KB) has its own slab. The slab allocator maintains a bitmap to track each slab's live nodes, thus maintaining the file offset of valid pages in that specific slab. Whenever a data node

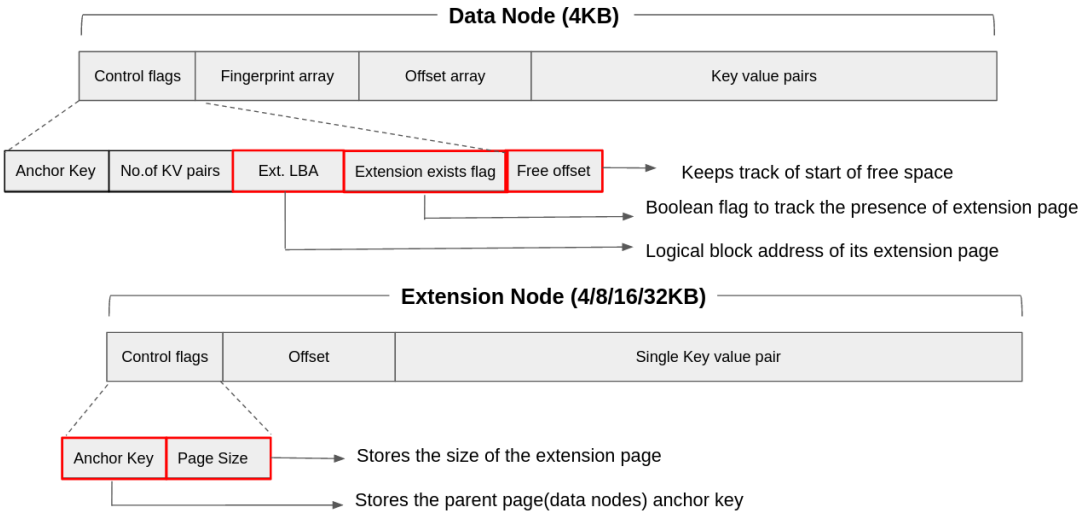

Figure 4.6: Data layout: Data Node with variale-length key-value support.

is deleted, its metadata "isAlive" flag is marked false to inform the slab allocator that the page is deleted. Similar logic is applied to the slabs maintaining the extension nodes as well.

In Figure 4.6, we present the extension page and data node layout supporting variable-length key-value pairs. The data node's metadata now holds three new flags that store,

- Extension node LBA.
- Boolean flag to track the presence of the extension page.
- Free offset to keep track of the start of free space in the data node.

The extension node also consists of a metadata and data region. The metadata region has a set of control flags and an offset region. The control flags store the parent data node's anchor key and the extension page size. The metadata also holds the key-value offset information. The rest of the extension node stores the key-value pair. It is important to note that our design sets the following constraints while supporting variable key-value insertions:

- Data node can chain to at most one extension node.
- Extension node can hold only one key-value pair.

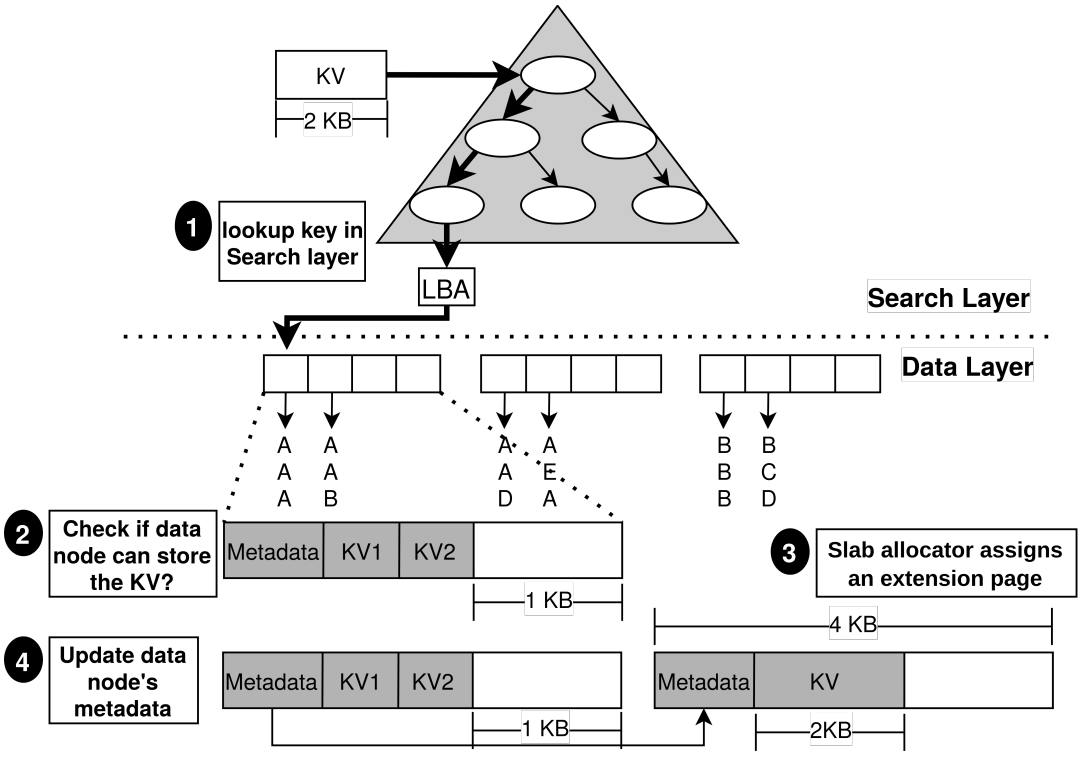

Figure 4.7: Working of variable-length key-value support in Retina.

- The data node undergoes split in the case of either reaching the maximum key-value count or when data node has reached its capacity and extension node already hold a key-value pair. Details of this scenario will be discussed in §4.7.

In Figure 4.7, we present a logic flow of an example to explain the variable key-value feature. To illustrate the feature, we insert a key-value pair that cannot be accommodated in the respective data node. To begin the insert operation, the host CPU first looks up the search layer to narrow down the LBA and then checks the mirror cache for the data node ❶. Once the data node is located, the host checks if the data node can hold the incoming key-value pair ❷. In the case of insufficient space, the host CPU calls the slab allocator to assign an extension page (of the size 4, 8, 16, 32KB) to accommodate the new key-value pair ❸. Later the FPGA is triggered to update the data node's metadata to mark the presence of an extension, store the extension page's LBA, and add the new fingerprint to the fingerprint

array. Finally the FPGA populates the extension page to store necessary metadata such as parent node anchor key, page size, etc., along with the key-value pair ❹.

4.4 Version-Based Crash Consistency

Initially, we maintained crash consistency by strategically accessing 4kB sized data nodes atomically from the disk. Nevertheless, with the introduction of extension pages of size 4KB and greater, there needs some addition to the design. We avoid using logging (*e.g.*, Write-Ahead Logging) instead proposing a novel versioning-based model to guarantee crash consistency. The basic rules of our model are:

- Assign a version number to all data nodes (4KB) and extension nodes (> 4KB).
- Increment the version number after every update.
- Extension node must have the same version number as its parent data node. If there is an update to the extension node, then increment the data node and extension node version.
- If a data node is deleted, then mark its state flag to deleted, and if an extension node is deleted, then mark its state to deleted and mark its data node's extension exist flag to false.

With versioning, we take advantage of atomic 4KB read/writes by using two schemes: (1) Data nodes are written in-place at the same file offset leveraging 4KB atomic page write in SSD and (2) Extension nodes whose size is greater than 4KB are written to the disk in an out-of-place manner to avoid logging and achieve version based crash consitency.

To support Version-based Crash Consistency (VCC), as shown in Figure 4.8, our design adds versioning information to the data node and extension page. The data node's metadata stores

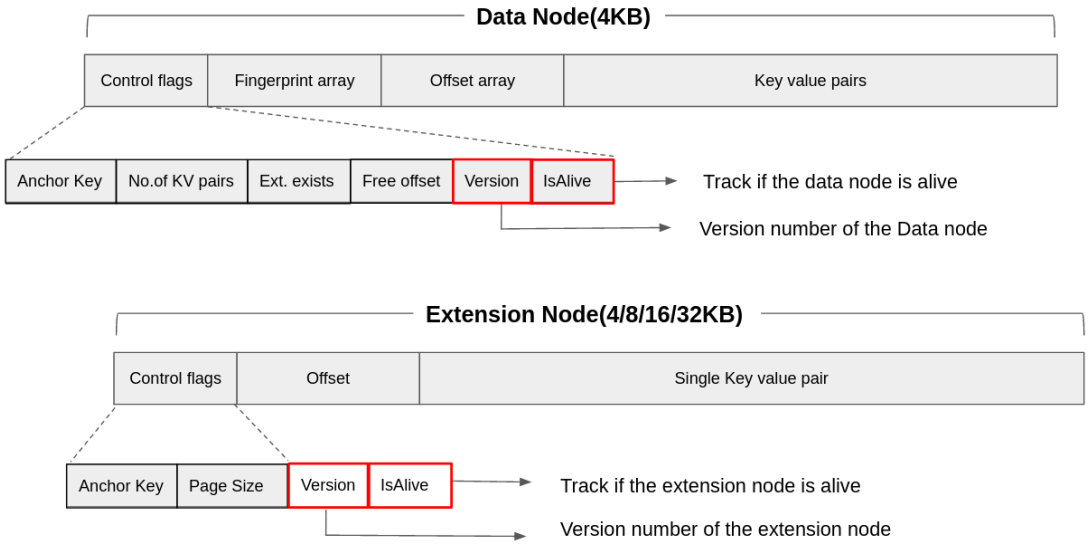

Figure 4.8: Data layout: Data Node with Version-based Crash Consistency (VCC).

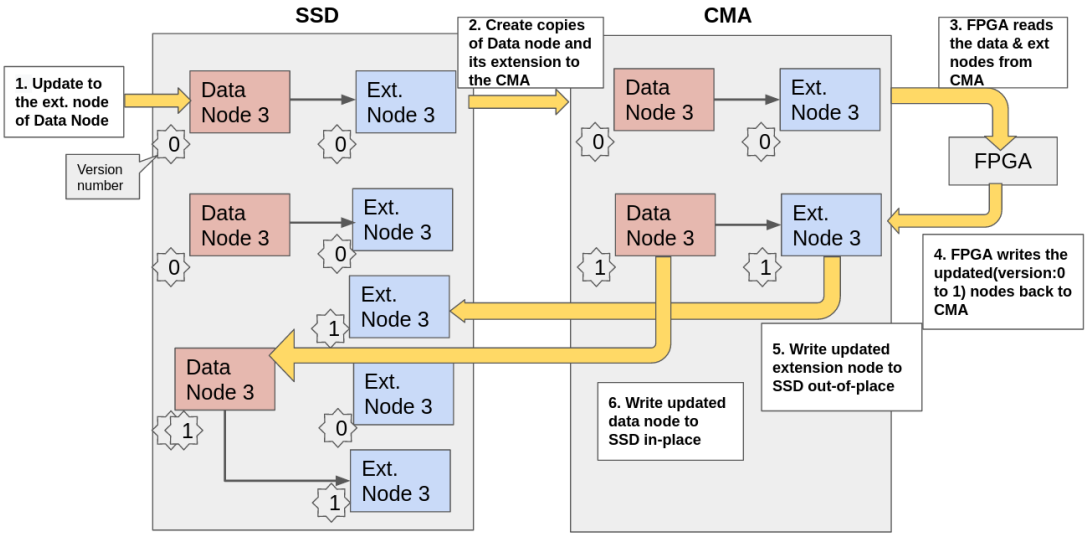

Figure 4.9: Working of Version-based Crash Consistency (VCC) model.

two extra flags: data node "version number" and "isAlive" flag to check if the data node is alive. The extension node holds a version number the same as the parent node and an isAlive flag to track if the data node is alive. With the help of this lightweight metadata, our VCC model promises data consistency and recovery.

No data will be lost or left in an inconsistent state with this VCC scheme. The invariant

to promise correctness is: *In the presence of an extension node, the data node is written in place only after the extension node is fully persistent on the SSD. In the case of a crash in any state, the old version of the data node is not disturbed, and if a newer version of the extension node is present on a disk, it is garbage collected by the slab allocator.*

In Figure 4.9, we present the best-case scenario of guaranteeing consistency without any crash. Consider an update operation to a data node that is chained to an extension node. Say both are at version 0 ❶. The data and extension node are copied to CMA ❷, and the FPGA is triggered ❸. After processing the request, the FPGA writes the updated data and the extension node with the new version number back to the CMA ❹. The new extension node is written to the SSD in an out-of-place manner ❺. Furthermore, the data node is written in place only after persisting the extension node ❻. Thus the updated data node would point to the new extension node safely.

Other essential scenarios of promising crash consistency in the presence of crashes:

- **Case 1:** *If a crash happens at any of the following stages, restart the operation,*
  - *Data node is being read from SSD to CMA.*
  - *FPGA kernel execution is underway.*
  - *FPGA is populating the updated data node back to CMA.*

  Figure 4.10 presents an example for the case 1 crash scenario. Consider an update operation issued to a data node chained to an extension node, similar to the example discussed above ❶. Before triggering the FPGA kernel, an IO read from SSD to CMA is issued if a mirrored cache is missed ❷. Next, the kernel function executes the update logic ❸. In this case, a system crash occurs when the FPGA populates the data and extension node back to the CMA ❹. At this stage, the API call exits, and the slab allocator takes care of the garbage collection ❺.
- **Case 2:** *If the system crashes when the extension page is partially written, abort and*

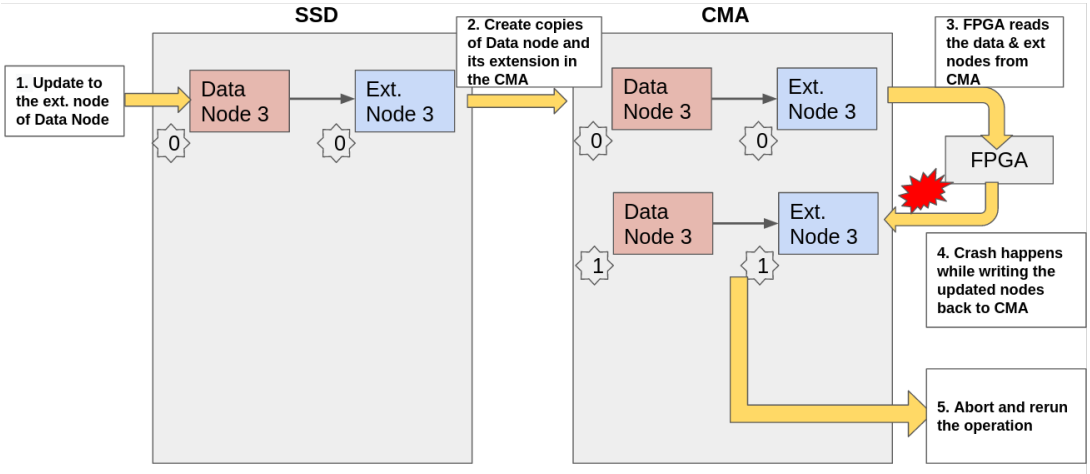

Figure 4.10: Crash case 1.

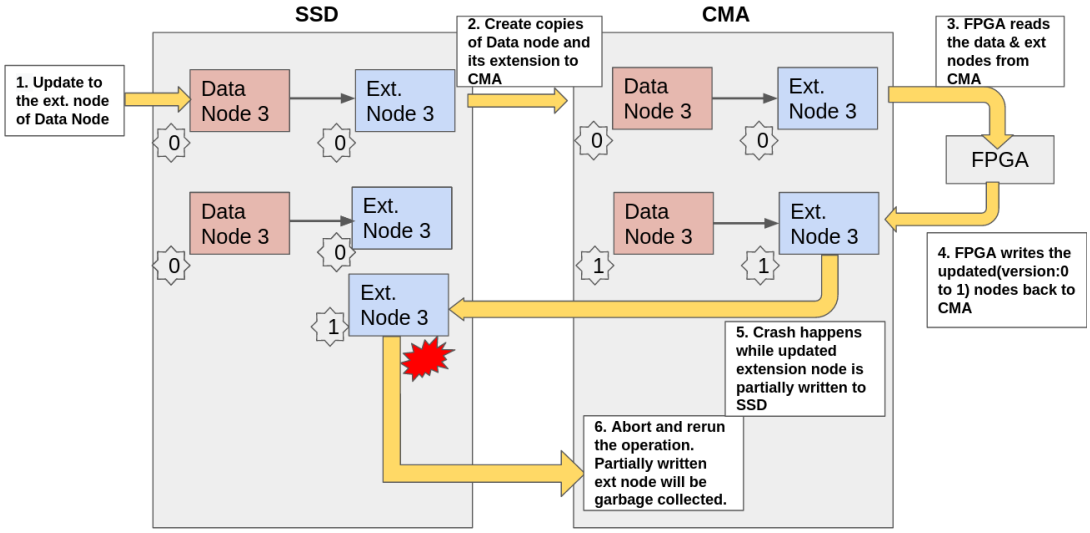

Figure 4.11: Crash case 2.

let the slab allocator garbage collect by checking version numbers.

In Figure 4.11, we illustrate the crash case 2. Consider a similar setup as discussed in case 1, where we issue an update operation. Let us suppose that the kernel executes successfully ❸ and the updated data and extension nodes are written back to the CMA ❹. Once the FPGA kernel passes the control back to the host, the CPU issues IO write calls to persist the updates. Based on our policy of data consistency, the extension page persists before the data node. In this example, a system crash happens at this stage

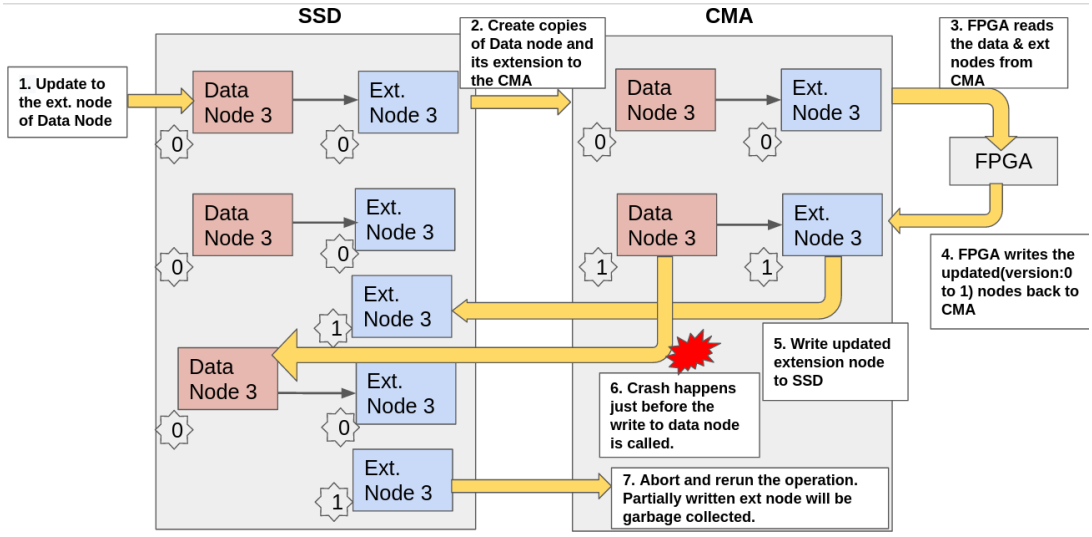

Figure 4.12: Crash case 3.

❺. Thus the operation aborts, and the slab allocator takes care of garbage collecting the updated extension page ❻. This way, we avoid tampering with the original data.

- **Case 3:** *If the system crashes after the extension page is fully persisted and just before writing to the data node, we abort and let the slab allocator garbage collect.*

  Figure 4.12 presents a logic flow for case 3, where we run an update operation on a data node with an extension. Suppose our execution progresses to the stage where the kernel execution is complete ❸❹, and the control is passed back to the host CPU. Say the extension node is successfully persisted onto the SSD ❺. Suppose the system crashes when the data node is written in place to the SSD ❻. In that case, the execution is aborted, and the slab allocator garbage collects both the updated data and extension nodes ❼.

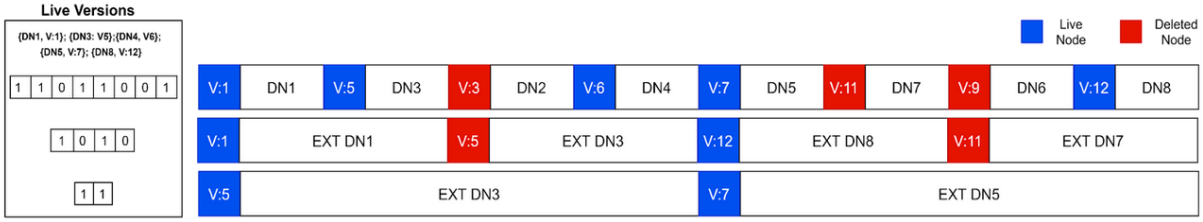

Figure 4.13: Crash recovery.

4.5 Crash Recovery Process

In the previous section, we discussed the working of VCC and how to handle different stages of system crashes. Now we discuss how Retina restores the key-value store in the case of a system crash. If any operation aborts due to a crash, none of the inconsistent data is persisted to the stable versions. Thus to restore the key-value structure and metadata on the host DRAM, Retina scans the entire SSD in two passes. The version numbers in parent data nodes are the latest versions, either alive/dead. Thus we use the following scheme:

- **First pass:** Scan the entire data node slab file (size: 4KB) to fetch all live data nodes. Then create a map of anchor keys and their corresponding valid version number and page offset. Simultaneously, start inserting the valid anchor keys into the search layer to build the host side metadata internally.
- **Second pass:** Scan each of the extension node slab files (size: 4, 8, 16, 32, 64KB), lookup their anchor keys in the live nodes map constructed in the first pass, and track the live extension nodes by matching their version numbers. Once a live node is encountered, we add the metadata of the extension node to the host.

Garbage collection

In the second pass of crash recovery, the slab allocator may find live/non-live extension pages whose corresponding version number does match their parent data page. In such a case, the

slab allocator will garage collect the pages and recycle them for future writes. As discussed in §4.3, the slab allocator maintains a bitmap to keep track of the live (i.e., in-use) nodes in each slab. Thus during garbage collection, it reset the bits on the bitmap to restore the pages by following the below steps,

- First fetching the parent data node's version number,
- Then scanning the entire slab to check for extension nodes(live/non-live) that have the same anchor key as the parent data node but different version number, and
- Mark these nodes as dead in the bitmap

To understand the crash recovery, we present Figure 4.13 to discuss the entire flow. As we can observe from Figure 4.13, the first bar indicates the data node slab, and the subsequent bars represent the extension slabs. When a crash occurs, as the first step, the data node slab is scanned to construct the live nodes mapping along with the slab's bitmap that marks their location. In our example, we observe that DN1 (Data Node 1) is live, and hence we add DN1 to the live versions map with its version number(V:1) and capture its offset by setting the bit in the slab bitmap. Similar to Data Node 1(DN1), Data Node 3(DN3) is also live thus, we add the mapping of (DN3, Version number 5) to the live versions map. To mark DN3's presence, we set the bit in the bitmap at offset two(where DN3 is stored). At the next offset is, Data Node 2(DN2) is found to be deleted; hence DN2 is not inserted into the live versions map, and the bitmap is reset corresponding to its location.

Similarly, we traverse the entire data node slab to populate the live map and slab bitmap. In the second pass, we scan all the extension nodes to look up their anchor key in the data node live version map. If the anchor key has a mapping, we match the version number to mark the extension node valid in the slab's bitmap. As shown in the Figure 4.13, the first extension node(Ext DN1) has found an entry with its anchor key in the live version map, and also it has the same version number as its parent data node. Thus the bitmap for this

specific slab notes the presence of EXT DN1 by setting the bit. While marking the slab's bitmap, the data and extension nodes are inserted into Retina to reconstruct the Search layer and the host metadata.

4.6 Concurrency Model

The cross-layered design of Retina splits the scope of search and data layer between the CPU and FPGA. Thus it is intuitive to separate the concurrency control into host-side and kernel-side concurrency models. In this section, we fisrst introduce the concurrency model in OpenCL (§4.6.1), which we used to implement Retina kernel. We then describe Retina's host-side concurrency (§4.6.2) and kernel-side concurrency (§4.6.3). Finally, we explain the entire flow (§4.6.4).

4.6.1 Introduction to OpenCL

OpenCL [6] is a framework to program heterogenous systems (consisting of CPU, GPU, DSP, FPGA, etc). It provides API to fine tune communication and offload logic to an accelerator by taking advantage of data and task parallelism. OpenCL is an open standard by Khronos Group. In the rest, we discuss the programming and memory model of OpenCL.

OpenCL programming model

- **Platforms:** A platform is a structure that encapsulates each vendor's openCL implementation of their specific hardware. The devices can be programmed and manipulated with the help of platforms. For instance, one can program an Intel device(CPU/GPU/FPGA) through the Intel OpenCL platform.
- **Contexts:** A context is an abstraction layer that the host maintains to handle the set

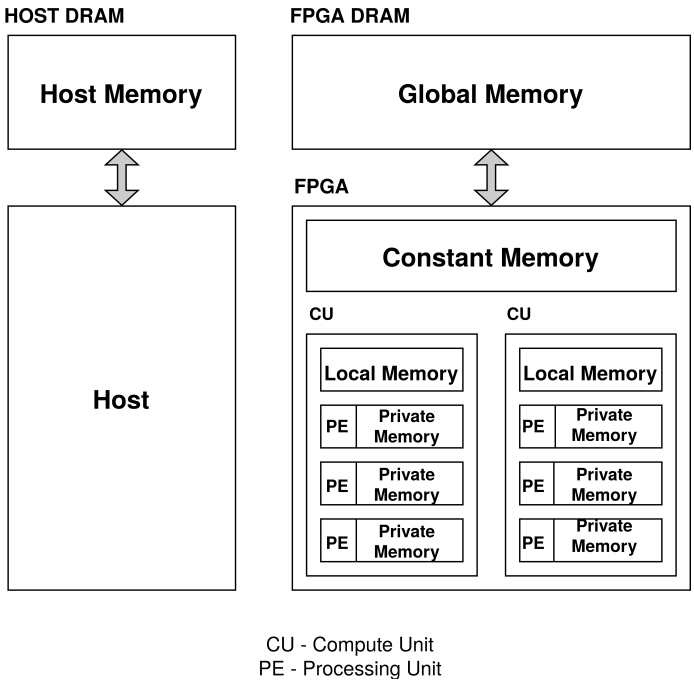

Figure 4.14: OpenCL memory model.

of devices being used from a platform. A context at a time can support only devices from one specific platform.

- **Devices:** A device represents the actual computing hardware provided by a vendor. For example, three FPGAs by Xilinx will contribute to three devices in a Xilinx platform.
- **Programs:** A program physically represents a .cc or .cpp file that can hold numerous kernel functions to be executed on specific computing hardware.
- **Kernels:** A kernel is an accelerator function that is issued along with the input data onto the computing device to execute on the accelerator.
- **Command queues:** A command queue is a pipe structure that enables the host to the accelerator communication where the kernel function and data transfer commands are enqueued.

OpenCL memory model:

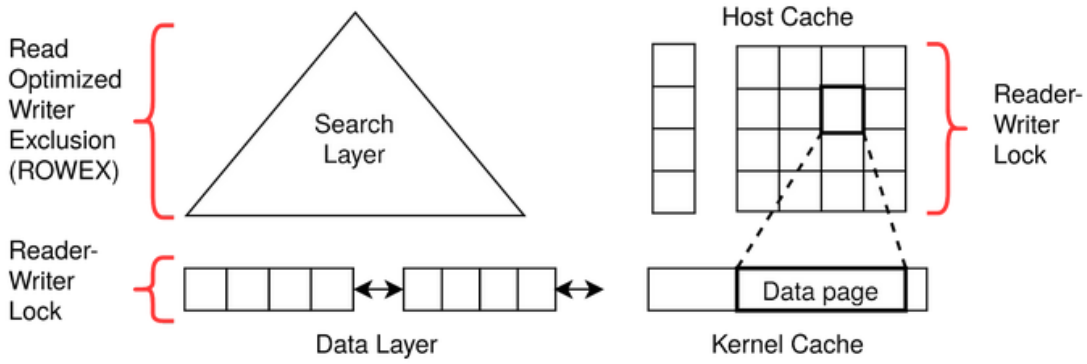

Figure 4.15: Retina's concurrency model.

- **Host memory:** As in Figure 4.14, this memory is physically located on the host DRAM. OpenCL allows the users to create memory objects on the host memory. These host-based buffers can be copied or memory mapped onto the device using OpenCL API calls.
- **Global memory:** It is the accelerators DRAM memory that is shared by the host and all the compute units on the device. Both host and device have read-write access.
- **Constant memory:** This is a part of the on-chip memory that allows read-write access to the host but only read access to compute units.
- **Local memory:** Implemented using On-chip BRAM memory and is logically shared by all processing entities on a computing unit. This memory is local to the given compute unit.
- **Private memory:** Physically stored in On-chip BRAM/SRAM memory and logically accessible to the attached processing entity.

4.6.2 Host-Side Concurrency

The host side concurrency handles the search layer and mirror cache concurrent accesses. As the search layer simultaneously deals with a single writer, the reader-writer synchronization

is essential. Due to data skewness in production databases, the read operations are more frequent than write operations. Thus multithreading model should allow non-blocking reads. Hence, the Read Optimized Write Exclusion (ROWEX) [33] protocol is the best suited to synchronize the search layer in Retina. With this algorithm, writers acquire a lock on a node and then update the node to exclude other writers. Whereas readers that use no locks offer safety by atomically updating the concurrently read fields. Similar to ART, the nodes are modified using an atomic compare-and-swap (CAS) instruction.

Though the data layer is physically stored and manipulated in the computational storage, the host maintains some skeletal metadata to traverse the data layer to narrow down a valid anchor key and thus a logical block address. While this implementation detail makes the job easy for the FPGA kernel, the host now needs to handle concurrency control for this data layer meta structure. As the structure is a simple doubly linked list, we use a reader-writer lock to allow a single writer or multiple readers at a time.

The flow of logic for all API calls in Retina goes through looking up the search layer, the data layer meta-structure, and finally looking up the Mirror cache using the LBA. Thus as a next step, we discuss synchronization in the Mirror cache. The Mirror cache is a hash structure implemented as a 2D array, where each row is a bucket. To best suit the Mirror cache's implementation, we choose a reader-writer lock to allow either multiple readers or a single writer.

4.6.3 Kernel-Side Concurrency

Although the host-side concurrency controls most of the synchronization in Retina, we achieve a fine-grain control on concurrency by leveraging FPGA's parallelism. Retina achieves such control by issuing multiple compute units for kernel functions onto an *out-*

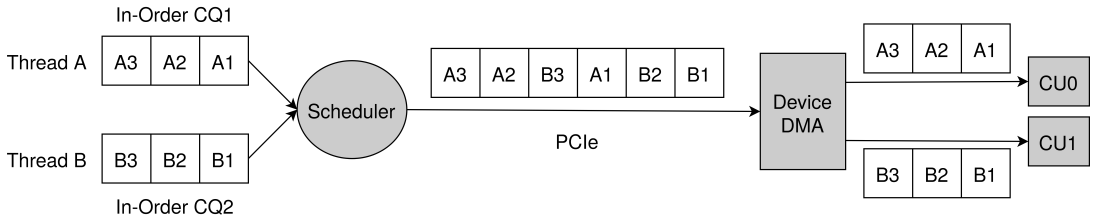

Figure 4.16: An example of in-order command queue in OpenCL.

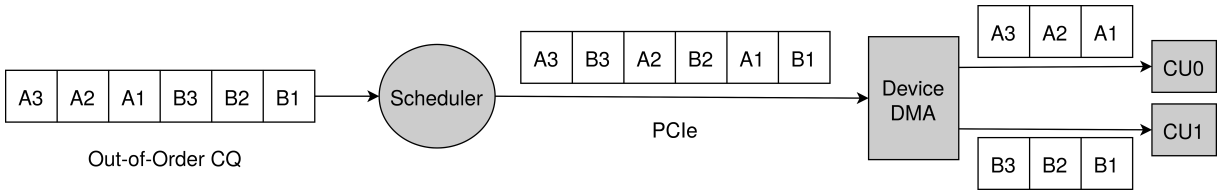

Figure 4.17: An example of out-of-order command queue in OpenCL.

of-order command queue. At compile time, the user application can allocate a set number of computing units to each kernel function. As each compute unit implements the kernel logic on separate hardware resources and has its local memory, increasing compute units provide with linear scaling of speedup. OpenCL programming model provides two ways of scheduling concurrent kernel calls [1]:

- Multiple in-order command queues
- Single out-of-order command queues

While both the approaches result in the same output, using a single out-of-order queue proves to be more advantageous due to the flexibility of issuing non-blocking asynchronous data transfers and kernel task enqueues. With out-of-order queues, we simplify the event synchronization by avoiding the creation of too many command queues.

In Figure 4.16, we present a simple example of the working of OpenCL's in-order command queue. Suppose two host threads are trying to issue commands to their respective compute units on the FPGA. The host application needs to create one in-order command queue for each host thread (Thread A and B). Though the scheduler dequeues the tasks to compute

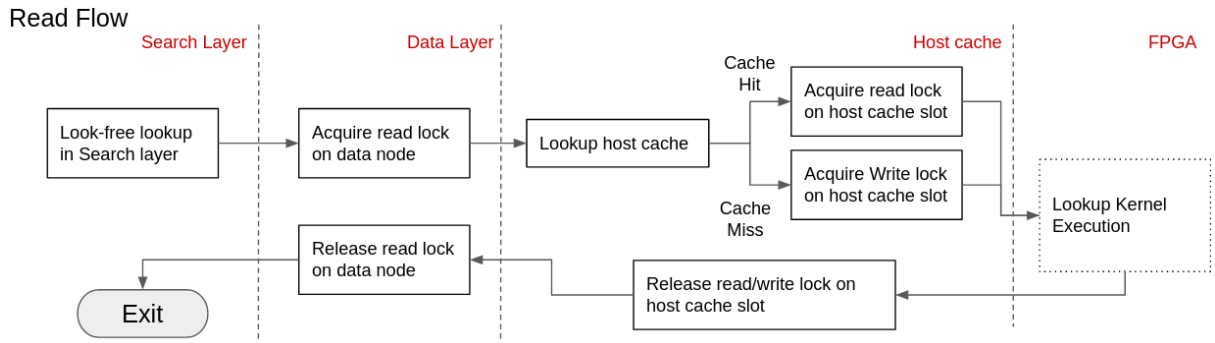

Figure 4.18: Concurrency control for a read operation.

units in order, it does not maintain any order from the two command queues (CQ1 and CQ2). Thus, the host application needs to handle any race conditions between the two threads carefully.

In Figure 4.17, we discuss an example of OpenCL's out-of-order command queue. Let us assume the same setup as the above example, where we consider two host threads (Thread A and B) issuing tasks on two compute units. Though both the host threads issue tasks onto a single out-of-order command queue, the order of operations within each thread is maintained by explicitly defining event dependencies. In the case of any race conditions, similar to when using an in-order queue is, the host application needs to synchronize explicitly. Using events in OpenCL can provide this extra fine-grain control of synchronizing within the same thread and across multiple threads.

4.6.4 End-to-End Concurrency Flow

We not present an end-to-end flow of concurrent read and writes operations. As shown in Figure 4.18 to cater to a read operation. Retina first incurs a lock-free lookup on the Search layer. Then the host traverses the data node meta-structure to acquire a read lock on the data node. Using the LBA found, it looks up the host cache to find the location of the

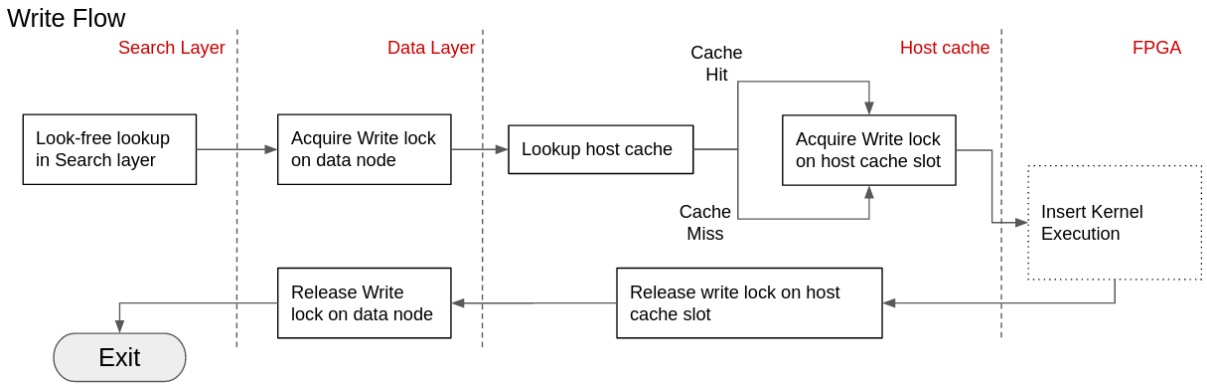

Figure 4.19: Concurrency control for a write operation.

data node on the kernel cache. In the case of a cache hit, it acquires a read lock on the cache slot; otherwise, if a cache misses, acquire a write lock. During cache miss, the host acquires a write lock to read the valid data node from SSD to the kernel cache and progress with the read operation. Once the kernel cache is ready with the data node at the found offset, the lookup kernel on the FPGA is triggered. Retina issues multiple compute units for each kernel function to achieve concurrency on the kernel side. After the lookup kernel execution completes, the control is passed back to the host with the output. Before exiting the application, the host first releases the read/write mirror cache lock and the read lock on the data node meta structure.

The Search layer lookup logic for the write operation, as shown in Figure 4.19, is the same as the read operation discussed above. As the write operation modifies the data node, Retina blocks other writers or readers to access that data node by acquiring a write lock on the data node meta-structure and host cache slot. In the case of a host cache hit, the application can trigger the kernel function, whereas, in the case of a cache miss, the host reads the data node into the valid kernel cache location. Once the kernel cache is populated, the insert kernel is called. After completing the execution, the host takes control to release the write lock on the host cache and then the data node meta structure.

4.7 Supported API Calls

In this section, we present Retina key-value store's supported API operations and their respective logic flow. Retina provides with the following five API calls:

- *Insert(key, value)*
- *Update(key, value)*
- *Remove(key)*
- *Lookup(key)*
- *Scan(key, range)*

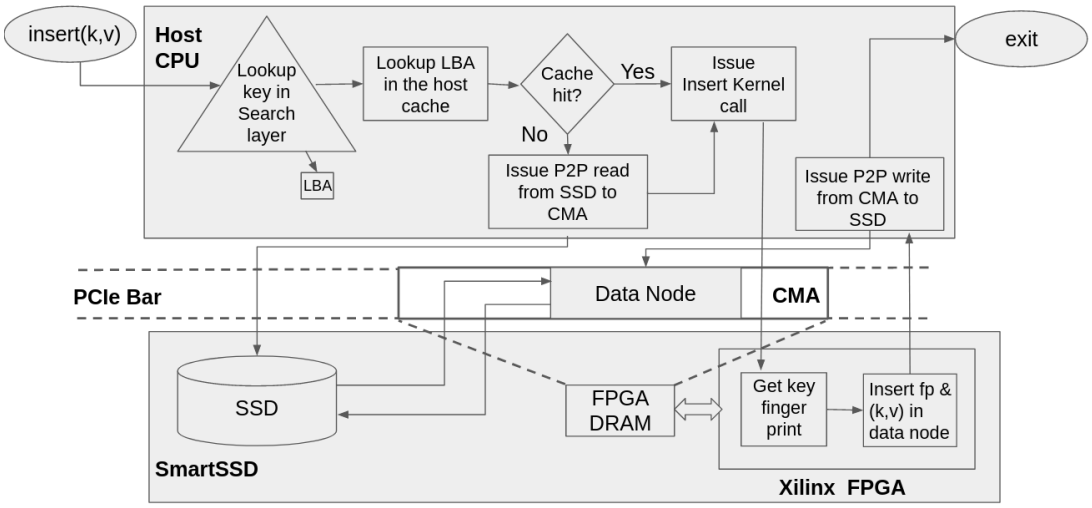

Figure 4.20: Retina key-value store: Insert API flow.

Figure 4.20 presents Retina's "Insert API call". The first step to inserting a new key-value pair into Retina is to lookup the partial key in the Search layer to find the data node's logical block address (LBA). Using the LBA, we look up the Mirror cache to check if the LBA is already present, implying the presence of a data node in the kernel cache. In the case of Mirror cache miss, the host CPU issues an SSD read to populate the kernel cache with the required data node and then triggers the FPGA Insert kernel call. Else in the case

of Mirror cache hit, the FPGA Insert kernel call is directly initiated. Internally, the Insert kernel call generates the fingerprint for the key, finds the valid offset to store the key-value store, updates the metadata, and finally adds the key-value pair to the data node at the right offset. Once the kernel execution is complete, the control is passed back to the host CPU. As our crash-consistency design is based on a versioning system, the host CPU persists any modifications at every operation. Thus the host CPU issues an SSD write for the data node in the kernel cache and finally exits the API execution.

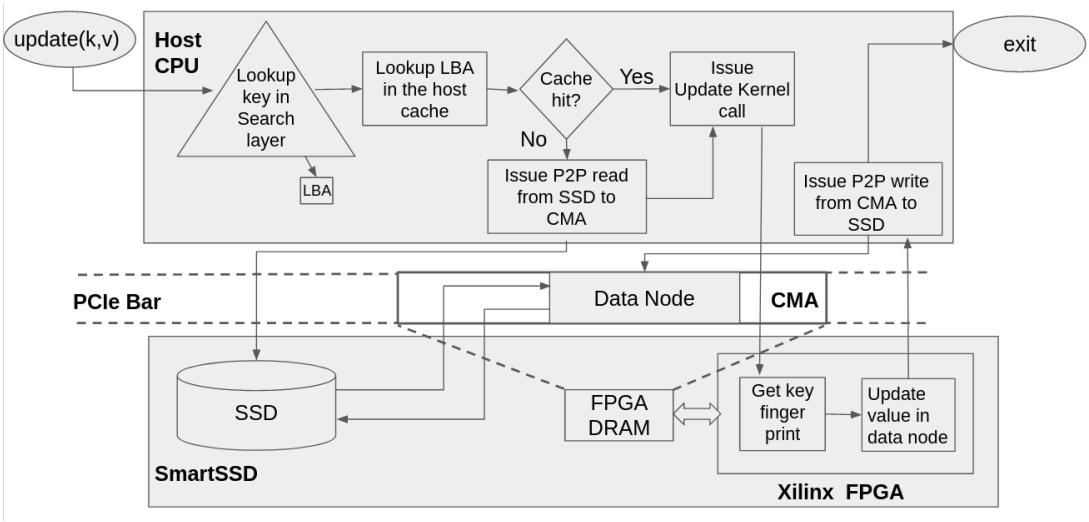

Figure 4.21: Retina key-value store: Update API flow.

The "Update API call" and "Remove API call" shown in Figure 4.21 and Figure 4.22, respectively, follow similar logic-flow as that of the "Insert API call". All of the host the CPU portion of the logic comprises looking up the Search layer, fetching data node location from the Mirror cache, handleing Mirror cache hit/miss, and populating kernel cache remains constant. Although the kernel logic for the Update call involves generating the fingerprint, looking up the key offset, and updating the old value to the new value in place. The kernel logic for the Remove call involves generating the fingerprint, finding the key offset, deleting the key-value pair, and updating the metadata. Once the kernel execution is complete, the host CPU can issue an SSD write to persist any modifications. This step is again the same

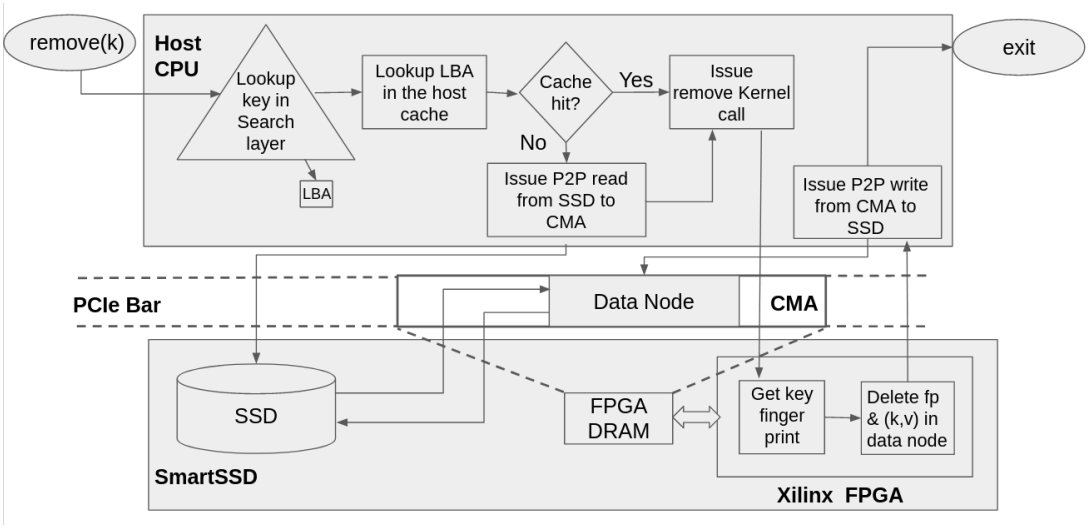

Figure 4.22: Retina key-value store: Remove API flow.

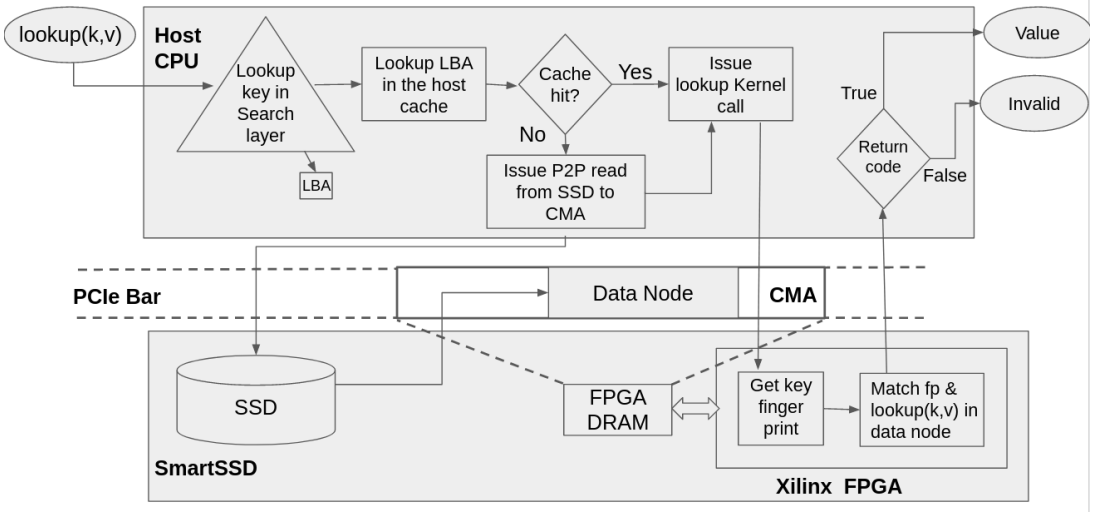

Figure 4.23: Retina key-value store: Lookup API flow.

for both the "Update API call", and "Remove API call".

Figure 4.23 presents Retina's "Lookup API call". Once the Lookup call is issued, the host CPU looks up the Search layer to narrow down the LBA. With the LBA, we look up the Mirror cache to check if the data node is already cached in the kernel cache. While the host-side overall logic for "Insert API call" and "Lookup API call" is similar so far, the Lookup call acquires a read lock while parsing the Mirror cache, whereas the Insert call

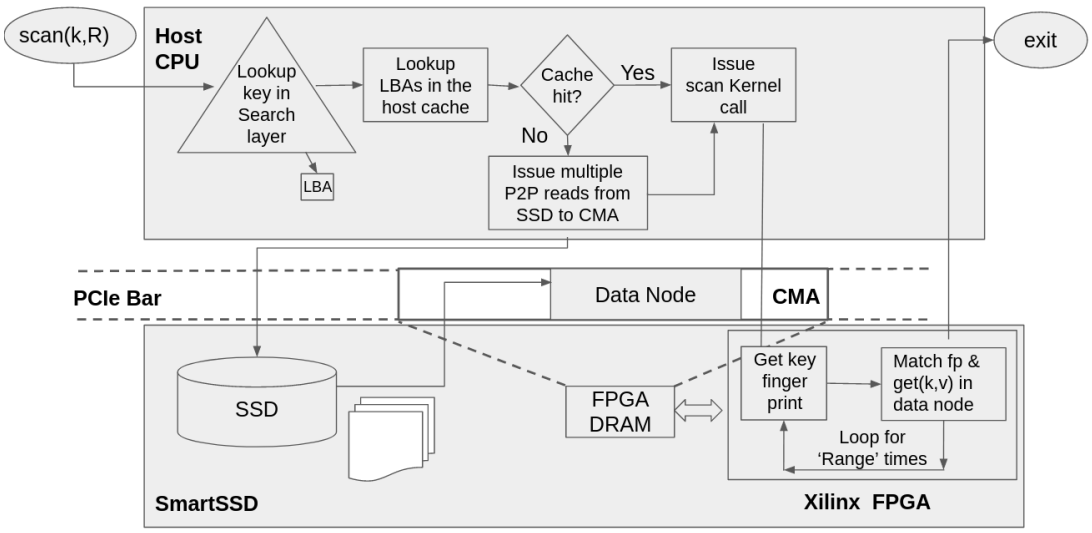

Figure 4.24: Retina key-value store: Scan API flow.

acquires a write lock. Based on whether it is a Mirror cache miss/hit, the host CPU decides to issue/not issue an SSD read to populate the kernel cache. Then the host CPU triggers the Lookup kernel call, which internally generates the fingerprint, finds the key offset and returns the value at that index. The host CPU receives the value from the Lookup kernel call and returns it to the API user.

"Scan API call" is similar to the "Lookup API call", where the initial of host-side logic flow is similar. However for this operation, after the LBA is found from the Search layer, a set of subsequent data node LBA's are fetched. These LBAs will be used to lookup the Mirror cache and facilitate the kernel cache population in the cases of cache misses. Once all the necessary data nodes accommodating the range of key-value pairs is stored in the kernel cache, the FPGA Scan kernel is triggered (as shown in Figure 4.24). Inside the Scan kernel,

- (Step 1): Generate the fingerprint of the key.
- (Step 2): Match the fingerprint to find the key-value offset.
- (Step 3): Store the value found at the key-value offset into an output value array.
- (Step 4): Increase the key by one and repeat (Step 1) until we exhaust the range.

Chapter 5

Implementation

Retina provides with API calls as mentioned in §4.7, which internally consists of C++ code and OpenCL API calls to trigger FPGA kernels. Retina uses Vitis Accel [10] toolchain to configure and trigger FPGA kernels. Vitis Accel execution model [11] dictates an application be split into host code executing on a host processor and hardware-accelerated kernels running on Xilinx FPGA, with a PCIe communication channel. Host applications can use OpenCL API calls, managed by the Xilinx Runtime (XRT), to communicate with the hardware accelerator. Data transfers between the host x86 machine and the accelerator board occurs across the PCIe bus. Thus Retina implements the search layer, Mirror cache (host-side), metadata for the data layer on the host-side. For each API call, there is a respective kernel logic (insert, update, remove, lookup, scan as mentioned in the §4.7) that manipulates the data nodes in the data layer. The host-side code consists of around 4000 lines of C++ code and 1200 lines of HLS kernel code. To implement the Least Recently Used (LRU) replacement policy in the Mirror cache, we maintain a timestamp using the RDTSC hardware clock that scales without any overhead [28, 29]. Concurrency on the kernel-side is statically defined in the config file by setting a set number of computing units to be allocated to each kernel function.

Chapter 6

Evaluation

In this chapter, we present the evaluation of Retina key-value store. In §6.1, we define the evaluation goals. Then we present the evaluation environment in §6.2. Finally, discuss the YCSB performance numbers to evaluate Retina against the state-of-the-art key-value store in §6.3.

6.1 Goals

- Run YCSB benchmark [17] on Retina for all workloads to test for the production-like environment. Compare Retina performenace against the state-of-the-art key-value store, RocksDB. Measure the CPU utilization to profile the savings on resource utilization.
- Evaluate the advantages of the Mirror cache and VCC features with micro-benchmarks.
- Profile the Retina's insert and update flow to analyze the latency breakdown in-depth. Identify the performance bottlenecks, provide reasoning, and set the scope for Retina design.

6.2 Evaluation Enviroment

6.2.1 Hardware

To evaluate Retina's design, we use Samsung SmartSSD. The computational device integrates a Samsung V-NAND SSD of 3.84TB capacity with a Xilinx Kintex Ultrascale (KU15P) FPGA. The SSD uses the NVMe protocol with a single port Gen3x4 PCIe. Also, the flash memory provides with 800k IOPS random write and 110k IOPS random read performance. The Xilinx FPGA provides 1.143 Million system logic cells with 300K LUT's, 1.9k DSP slices, 34.6Mbit internal distributed RAM, 36.0 Mbit URAM, and 4GB of accelerator dedicated DDR SDRAM (with 2.4Gbps read/write bandwidth). The host system being used is Intel Xeon Gold 6152 CPU with eight-core (256KB L1 cache, 4MB L2 cache, 60MB L3 cache), distributed among two NUMA nodes with 263GB DRAM memory.

6.2.2 Workload

We choose the YCSB benchmark [17] to evaluate Retina as it helps gauge performance for a range of varied workloads and also mimics production-like requests. Among the YCSB workloads, in this section, we present numbers for the YCSB A (50% GET and 50% PUT), YCSB B (95% GET 5% PUT), and YCSB C (100% GET 0% PUT). We use key-values of the size 1024 bytes with the dataset size of 5 million for uniform and Zipfian key distributions.

6.2.3 System Configuration

As Retina currently does not support asynchronous user API calls, we enable a fair performance comparison by tweaking the RocksDB configuration. RocksDB uses buffered IO to

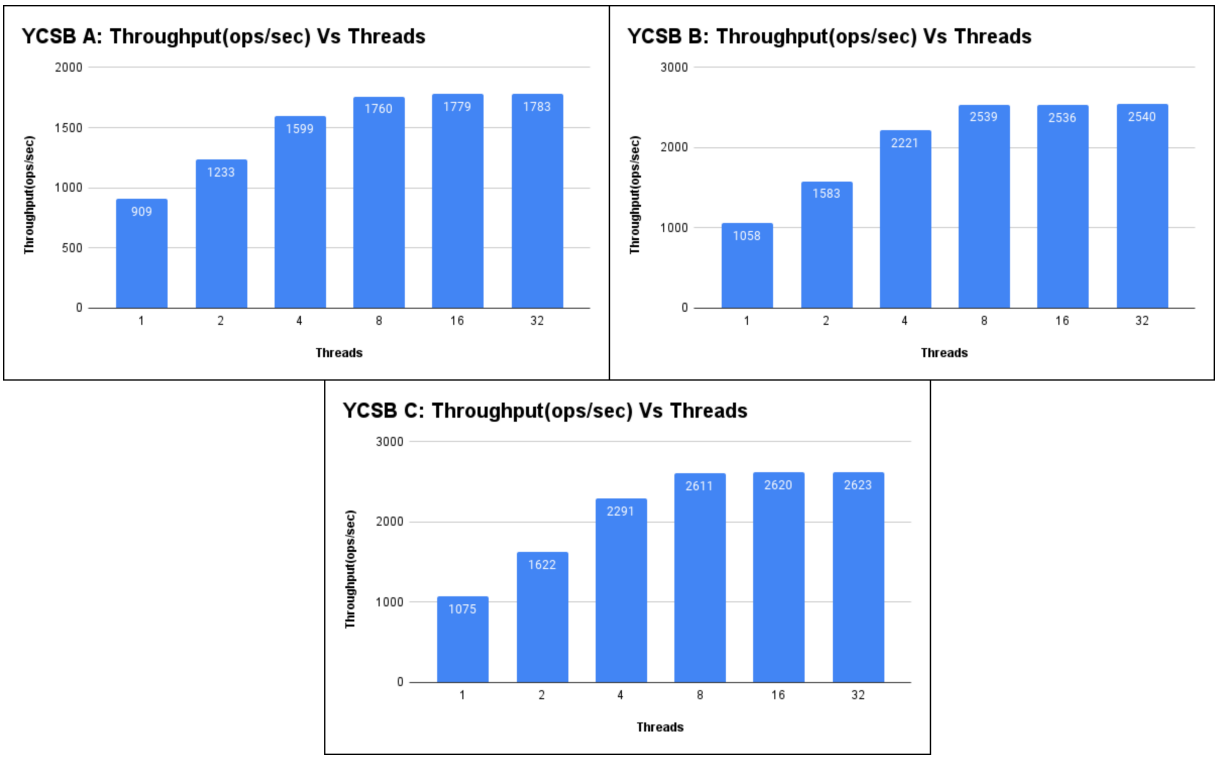

Figure 6.1: YCSB A,B, and C workload performance number on Retina. The benchmark is run with 16 threads, key-value size of 1024 bytes for 5 million dataset size.

achieve asynchronous read/write user requests. Thus we disabled the use of OS page cache and block cache to switch RocksDB to Direct IO. In addition, we also disable bloom filters stored in each level.

6.3 Performance Evaluation

6.3.1 Benchmark with YCSB

Figure 6.1 measures Retina's throughput performance for YCSB A, B, and C workloads with increasing host-side threads. Throughout this evaluation the number of kernel threads (instances) for each API call is set to 2. The size of each key-value pair is maintained to

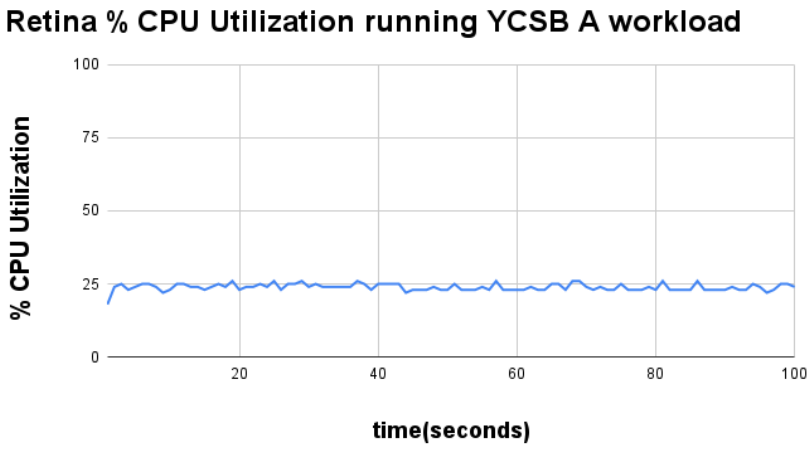

Figure 6.2: Retina's CPU utilization when running YCSB A workload with 16 threads and key-value size of 1024 bytes for 100 seconds.

be 1024 bytes, and the dataset size is fixed to 5 Million key-value pairs. As we can infer from all the YCSB graphs, the throughput also increases with the increase in the number of host-side threads. This trend is persistent when increasing the thread count from 1 to 4. After thread 4, the throughput saturates. This behavior can be attributed to limited kernel instances restricting the concurrency achieved by spawning more host-side threads. As our experimental setup fixes the number of kernel instances to two, there is a near-linear increase in throughput performance when increasing the number of host threads from 1 to 4. However, that performance is slightly degraded when the number of threads increases due to extra thread creation/maintenance overhead exceeding scaling. The scope and limitations of kernel-side concurrency is discussed in detail in the next section §7.

Figure 6.2 presents the % CPU utilization of Retina when running the YCSB A workload over 100 seconds. To run the experiment, we use 16 threads, 1024 byte-sized key-value pairs, and a dataset of 5 million key-value pairs. As discussed in §1, the CPU resource contention hinders the IO bandwidth saturation in most key-value store designs. Thus Retina proposes a Cross-Layered approach §4.1 that offloads data-intensive compute logic from the CPU onto

the computational storage, hence achieving performance by reducing the CPU bottleneck. As it can be observed from Figure 6.2, Retina accomplish this goal by reducing the overall CPU utilization to 25%.

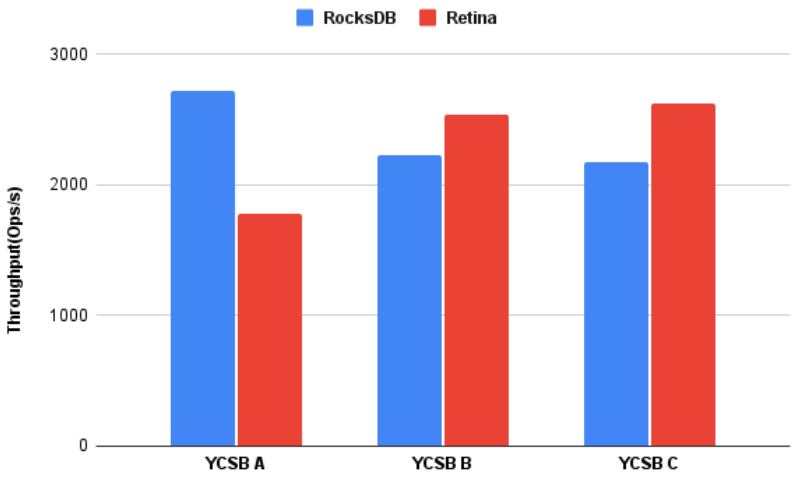

Figure 6.3: Running YCSB A,B, and C workloads to compare throughput performance of RocksDB and Retina. The benchmark is run with 16 threads, key-value size of 1024 bytes for 5 million dataset size.

We compare the performance of RocksDB (configured as mentioned in the system configuration §6.2.3) and Retina by running YCSB A,B, and C workloads. The two key-value stores are evaluated using 16 threads and 1024 bytes key-value pairs for 5 million key-value pairs. As shown in Figure 6.3, Retina performs better than RocksDB in more read-intensive workloads(such as YCSB B and YCSB C workloads§6.2.2).

6.3.2 Profiling Mirror Cache and Version-based Crash Consistency (VCC)

In this section, we profile Retina to measure performance improvements with the addition of Mirror Cache and Version-based Crash Consistency. The experiment is performed using

the following three configurations of Retina,

- *Base*: No Mirror Cache and Write-Ahead-Logging (WAL) for crash consistency.
- *VCC*: No Mirror Cache and Version-based Crash Consistency
- *MC + VCC*: With Mirror Cache and Version-based Crash Consistency.

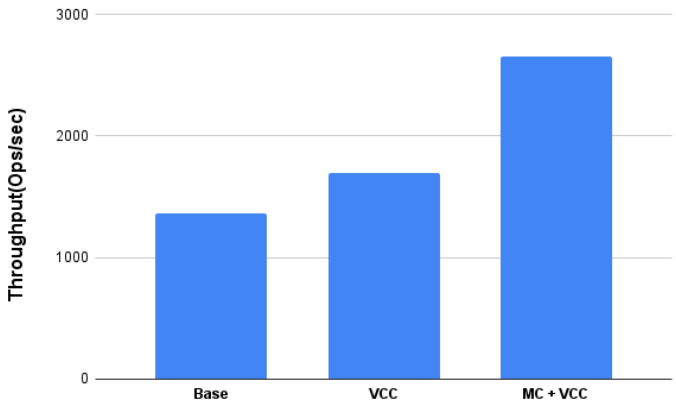

Figure 6.4: Running YCSB A benchmark (50% GET and 50% PUT) with single thread and 128 bytes key-value size on Retina on three configurations Base, VCC, MC+VCC. Enabling each feature on top of the Base config results in improved throughput performance.

The single-threaded performance of the configs mentioned above is measured on YCSB A workload (§6.2.2) where KV size is set to 128 bytes. In Figure 6.4, the Y-axis represents the throughput which is captured in operations per second and on the X-axis, the base config and its two incremental additions are listed. We can infer from Figure 6.4 that the base config which uses WAL performs poorly as it persists each update to the log file before persisting the contents to the original file in the storage. The VCC config, which uses Version-based crash consistency performs better than WAL by avoiding logging and using versioning to keep track of valid pages. As discussed in §4.4, the VCC scheme takes advantage of 4KB atomic reads/writes to persist changes to disk atomically and maintains light-weight version information on every page to store valid extension pages in the disk. Finally, the third config (MC + VC) uses the Mirror cache, which improves performance. Furthermore, it reduces

the overhead of reading pages into device memory or writing pages to SSD at each operation by maintaining a cache that leverages the data locality.

6.3.3 Profiling End-to-End User API

In this section, we profile Retina's read and write API calls using a single thread to narrow down the following host-side scenarios

- Read cache hit: Read API call with Mirror cache hit
- Read cache miss: Read API call with Mirror cache miss
- Write cache hit: Write API call with Mirror cache hit
- Write cache miss: Write API call with Mirror cache miss
- Split: Write API call that triggers a data node split

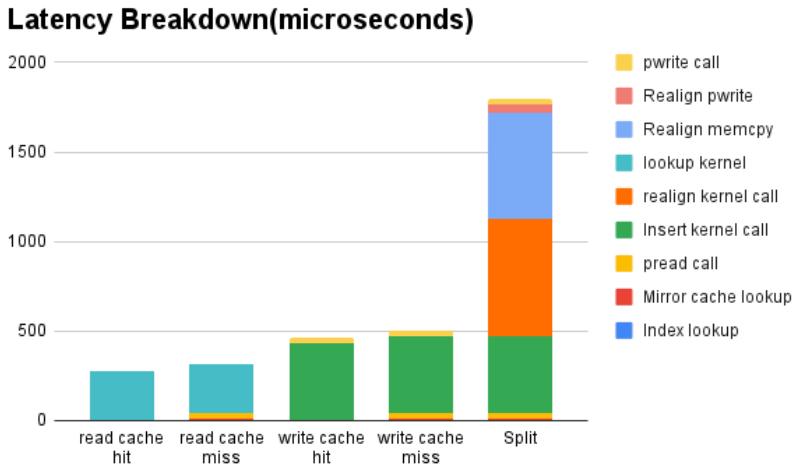

Figure 6.5: Latency breakdown for Retina's end-to-end read & write API calls.

To understand the latency breakdown in each of the above cases, we need to refer to §4.6.4. All the API calls enter Retina to first lookup the search layer to find the logical block address (LBA). This stage is termed "index lookup" and is a constant part of all the above-mentioned scenarios. Thus, when we consider the first case of "Read cache-hit", a small amount of time

is attributed to “index lookup” (search layer lookup) as shown in Figure 6.5. Then the LBA is looked up in the Mirror cache to find the entry with a matching LBA. In this case, as it is a cache hit, the entry is found, and thus the “Mirror cache lookup” time is insignificant. A cache hit indicates that the data page is already populated and ready to be consumed in the device cache. Thus the next step is to issue the “lookup kernel call”. As shown in Figure 6.5 this step takes the most time to execute. Once the lookup kernel returns to the host, the resultant value is returned to the user, and API call ends.

Like the read cache hit case, the second case read cache miss requires the constant “index lookup” (search layer lookup) to narrow down the LBA. But now, when looking up the Mirror cache using the LBA, we incur a cache miss, thus resulting in a higher “Mirror cache lookup” time as shown in the Figure 6.5. Due to the cache miss, the data page needs to be read from the SSD to the device memory, thus contributing to “pread call”. Once the data page is populated in the device memory, the “lookup kernel call” is executed, and the results are returned to the user to end the API call.

For the write cache hit case, the first step would again be “index lookup” (search layer lookup), then the Mirror cache is looked up using the LBA. The current case is a cache hit, so the “Mirror cache lookup” time is insignificant. Due to the cache hit, the flow directly skips to “insert kernel call” execution, contributing the most time. Once the insert kernel call returns, the update is persisted to the SSD resulting in a “pwrite call”. Similarly, the write cache miss case incurs constant “index lookup” time before looking up the Mirror cache. As this case results in a cache miss, the “Mirror cache lookup” time is significant. The API call issues a “pread call” to read the data page from SSD into the device memory. Then the “insert kernel call” is executed to make the update. Finally, the control is passed back to the host, and the host issues a “write call” to persist the changes to the disk.

The last case listed above discusses the split case, which occurs when the data node reaches

the capacity and cannot accommodate any more key-value pairs in the data node. As split is triggered as part of an insert API call, even in this case, the first step is to issue an "index lookup." With the LBA, the Mirror cache is looked up to find the entry with the matching LBA. This particular case presents the worst-case scenario where there is a cache miss. Thus the "Mirror cache lookup" contributes to some amount of time. Next, the host issues a "pread call" to read the data page into the device memory. The "Realign kernel call" is triggered once the data is ready on the kernel cache. After the execution, the control is passed back to the host to persist the old and newly created pages. Thus the host CPU issues two "pwrite calls" and transfers metadata for both the pages from device memory to host memory. Once the data node is split, the insert API call retries and follows the traditional logic as mentioned above.

The trends observed from the breakdown in all the above execution scenarios are that the kernel execution time attributes a significant chunk in the total API call execution time. To scale performance in a concurrent setting, scaling host threads plays an important role, and increasing kernel instance is essential. As there is no limit on the number of host threads that can be spawned, the number of Kernel instances that can be assigned to each logic flow determines the level of concurrency achieved. In the next section, we further discuss the limitations and the scope of the current work.

Chapter 7

Discussion and Limitations

As discussed in the evaluation section §6.3.1, the CPU utilization shown in Figure 6.2 is reduced to a maximum of 25%. Thus, we reduce the CPU bottleneck by offloading logic to the SmartSSD. While Retina key-value store is primarily designed and optimized to store and retrieve data efficiently from the computational storage, it also serves as an important module to integrate AI/ML preprocessing pipelines to storage. Thus alleviating the CPU bottleneck not only provides an opportunity to Retina's key-value store to maximize IO bandwidth utilization [34] but also aids in realizing computational storage based preprocessing pipelines.

Another facet of the design is to maximize throughput performance which is usually achieved by issuing concurrent requests. In traditional key-value stores, this goal can be achieved by spawning more host-side threads to scale the read/write request performance. But as observed by §6.3.3, the level of concurrency is also controlled by the kernel-side scalability. Unlike concurrency on the host-side, where 100's threads can be spawned, there is a limit on the number of computing units allotted to each kernel function due to the hardware resource limitations. In Retina, the below two factors affect the scalability of kernel functions:

- LUT and FF resource saturation
- Maxing out on the number of ports drawn from the FPGA's DDR memory

The Figure 7.1 presents the FPGA resource utilization for all the kernel functions. As we can see that the number of LUT's (Look-Up Tables) and FF's (Flip-Flops) are determining

Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	139286 [26.69%]	11248 [6.99%]	211267 [20.24%]	380 [38.62%]	0 [0.00%]	9 [0.46%]
User Budget	382594 [100.00%]	149768 [100.00%]	832493 [100.00%]	604 [100.00%]	128 [100.00%]	1959 [100.00%]
Used Resources	134358 [35.12%]	4228 [2.82%]	154045 [18.50%]	102 [16.89%]	0 [0.00%]	6 [0.31%]
Unused Resources	248236 [64.88%]	145540 [97.18%]	678448 [81.50%]	502 [83.11%]	128 [100.00%]	1953 [99.69%]
getKernel	34219 [8.94%]	1365 [0.91%]	44989 [5.40%]	34 [5.63%]	0 [0.00%]	2 [0.10%]
getKernel_1	34219 [8.94%]	1365 [0.91%]	44989 [5.40%]	34 [5.63%]	0 [0.00%]	2 [0.10%]
putKernel	41742 [10.91%]	1383 [0.92%]	47401 [5.69%]	33 [5.46%]	0 [0.00%]	2 [0.10%]
putKernel_1	41742 [10.91%]	1383 [0.92%]	47401 [5.69%]	33 [5.46%]	0 [0.00%]	2 [0.10%]
realignKernel	58397 [15.26%]	1480 [0.99%]	61655 [7.41%]	35 [5.79%]	0 [0.00%]	2 [0.10%]
realignKernel_1	58397 [15.26%]	1480 [0.99%]	61655 [7.41%]	35 [5.79%]	0 [0.00%]	2 [0.10%]

Figure 7.1: Retina's Kernel functions resource utilization report.

factors of the number of possible compute units that can be created. With further optimizations to the kernel functions, the per kernel resource consumption can be reduced, attributing to an increase in computing units. Although the resultant concurrency achieved may not be comparable to the host-side concurrency. Also, due to the limited DDR capacity, there is a restriction on the number of ports drawn from the global memory as kernel arguments. Retina proposes to tackle these issues as follows:

- Extend the kernel implementation to follow the dataflow model instead of just relying on scaling compute units.
- Integrate an Arbiter IP to multiplex input arguments to more number of kernel functions.

Each kernel implementation can internally be split into multiple sub-tasks ingesting requests as streams to leverage task-level parallelism. This design is further bolstered by integrating an AXI interconnect (in the crossbar mode) to route requests to more compute units. To tie everything together, we will shift the concurrency model towards a producer-consumer model in the future. With this model, all the API requests will be catered asynchronously, where the user threads will enqueue the tasks to a request buffer and return. Further a set of background worker threads will process the request in a batched manner taking advantage of the kernel pipelines.

Chapter 8

Related Work

The integration of accelerators to datastore systems can be categorized [21] as follows,

- *Type1:* Accelerator as a co-processor
- *Type2:* Accelerator on-the-side (IO attached)
- *Type3:* Accelerator in-the-data-path (Near-Data-Processing)

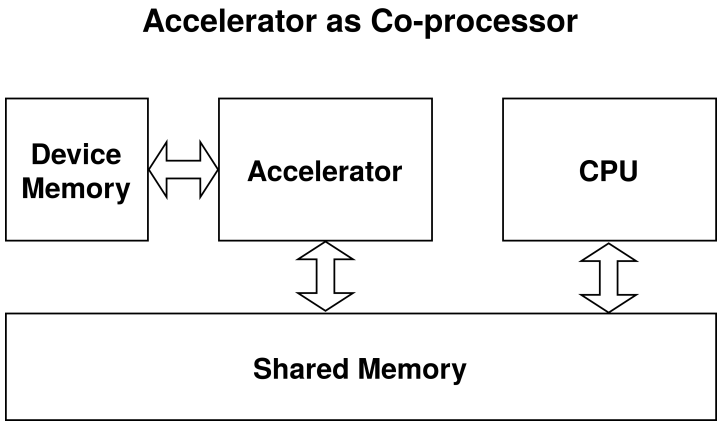

Figure 8.1: An architecture where the accelerator and the host system share a common memory region to seamlessly offload compute from host to accelerator by avoiding memory transfers.

In the co-processor-based architecture, the accelerator and the host CPU share the host memory. Thus the accelerator can access the host memory coherently without any extra memory copies to and from the accelerator's device memory (refer to Figure 8.1). This concept is realized in one of the below two ways:

- First, attaching the accelerator and CPU in the same socket.

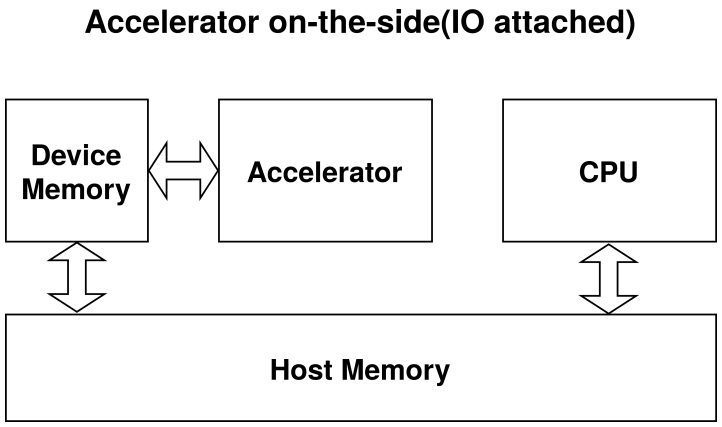

Figure 8.2: In this design, the accelerator is attached to the host system as an IO device. Thus offloading computation onto the accelerator requires explicit data transfer between the host memory and the accelerator's device memory.

- Second, attaching the host and the accelerator with high bandwidth hardware interconnects enables coherent shared memory capabilities.

ZYNQ [18] realized the co-processor based design by connecting the CPU and the FPGA using Accelerator Coherency Port (ACP) that creates a unified shared memory between the two processors. Open Coherent Accelerator Processor Interface (OpenCAPI) [16] by IBM and Compute Express Link (CXL) [2] by Intel is some examples. MetalFS [42] is a near-storage-compute-aware file system(FPGA-based), where the CPU and the FPGA Overlay share the host memory using IBM's OpenCAPI.

The IO attached accelerator-based design is the most prevalent type where compute-intensive tasks are offloaded from the CPU to the accelerator. Figure 8.2 shows the architecture, where the accelerator connects to the host system by attaching to the PCIe bus. The accelerator is treated as another IO device that needs to copy data to/from the host memory to its device memory whenever processing any computation. This design technique has been used in numerous works across academia and the industry [14, 22, 41]. In the prior work [46], FPGA-Accelerated compactions for LSM-based key-value stores, the expensive compute-intensive

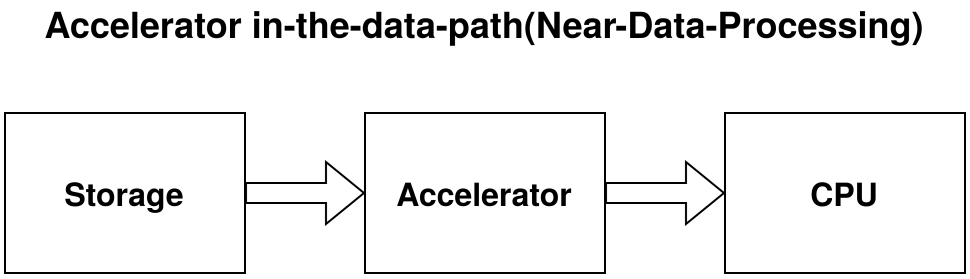

Figure 8.3: This design is based on Near-Data-Computation, where the accelerator is directly attached to the storage in-the-data-path. Any logic offloaded from the host system to the accelerator results in input data fetched into the accelerator's memory and processed before reaching the host system's main memory.

compaction operations are offloaded from the host CPU onto an IO attached FPGA.

The third design pattern originally started as a bandwidth amplifying architecture (shown in Figure 8.3), where a simple compute unit would compress, decompress and filter the data to maximize the overall bandwidth. This would be achieved by connecting the compute unit to the storage using SATA, SCSI, etc. The basic idea is that the accelerator fetches the compressed data from the storage, decompress it, and filter it before sending it back to the host system. As the amount of data being transferred over the network is compressed and filtered the adequate number of bytes is reduced, and thus the overall bandwidth increases.

Many previous works have extended this design to realize near-data computation fully. The underlying goal is still improving the overall bandwidth but with different approaches of integrating the accelerator in the data path of the storage. For instance, INSIDER [40] presents a new file system on a remodeled accelerator-based storage system. To saturate the increasing speeds of SSD's, they integrate an FPGA as the in-storage-computing unit. The nKV [44] key-value store is another work where near-data-processing is realized by completely offloading the IO stack onto the accelerator to minimize host intervention and access data directly from the physical addresses. A similar idea is standardized to define, KV-SSD's [25, 27], which are specialized SSD drives that implement the entire key-value

store on the storage system, altogether avoiding host intervention. Retina also uses the same design pattern of near data computation. However, instead of localizing on storage resources, it decouples the architecture to maximize both the host and compute resource utilization.

Chapter 9

Future Work

In today's Deep Learning (DL) applications, while GPUs run model training algorithms, the CPU preprocess data and feeds it to the GPU. Over the years CPU preprocessing step is becoming a bottleneck due to the reasons below,

- While GPU performance has been improving 1.5x every year, CPU performance improvement has been saturated for a decade.
- The performance gap between GPU and CPU is expected to be 1000x by 2025 [4].
- Larger performance gap means that GPU will be idle longer and the entire DL training will be stalled.

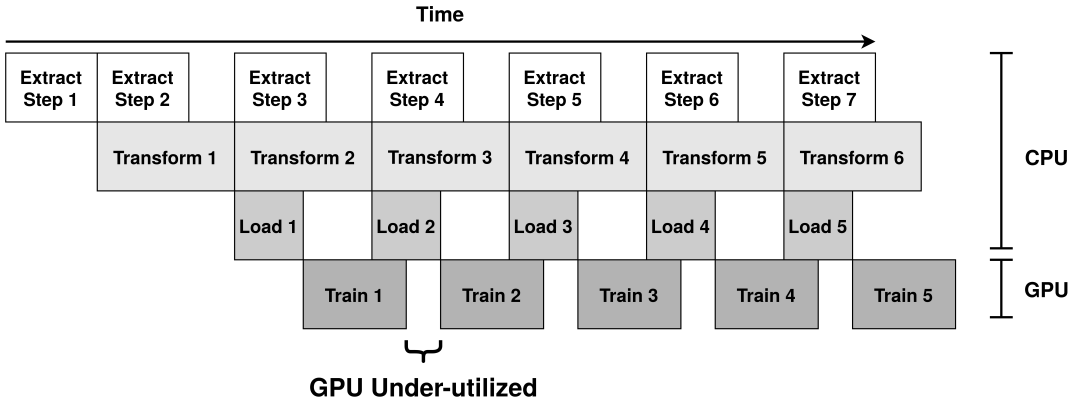

Figure 9.1: Deep learning input pipeline: ❶ Extract: data fetching from SSD to the host DRAM, ❷ Transform: preprocessing with a set of functions on host CPU, ❸ Load: loading data from host DRAM to GPU DRAM, and ❹ Train: model training on the GPU.

Figure 9.1 illustrates a DL input pipeline to discuss the windows of improvement. The first

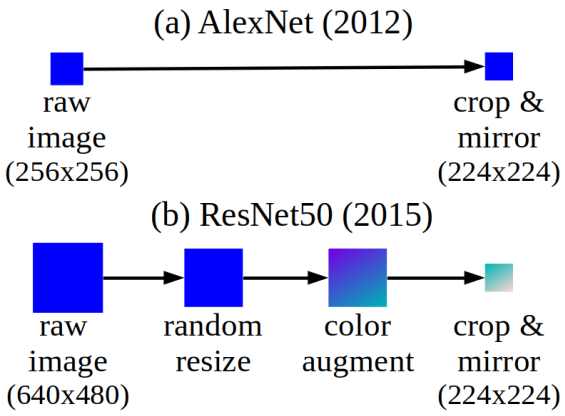

Figure 9.2: Comparison of pre-processing pipeline of AlexNet [30] and ResNet50 [23]. Complex DL models require more complex pre-processing piplelines to avoid overfitting.

step in the pipeline is the ❶ Extract stage, where input data is fetched from the SSD to the host DRAM. Once the input data is ready, in the stage ❷ Transform, the host CPU passes the raw data through a set of functions to adhere to a format. Soon after, the processed data is Loaded in the stage ❸ from host DRAM to GPU DRAM. Finally, in the ❹ Train stage, the processed data is consumed by the DL training model.

Though there is a dependency between the tasks mentioned above, there is scope for interleaving different stages to process distinct data sets parallelly. Despite pipelining, there are gaps in between training stages where the GPU is under-utilized. Due to the widening performance gap between CPU and GPU, the host CPU causes data to fetch stalls and preprocessing stalls. Unfortunately, adding more CPUs to a GPU server is not viable because a multi-socket machine (*e.g.*, 4- or 8-socket machine) is expensive while not providing sufficient computation power for preprocessing. In addition, the preprocessing in deep learning models is getting more complex within a span of few years (see Figure 9.2). That is because more sophisticated DL models, which have better generalization capability for accuracy, require more complex preprocessing steps to avoid the well-known overfitting problem. In the current scenario, relying on the CPUs for the preprocessing pipeline will further increase the

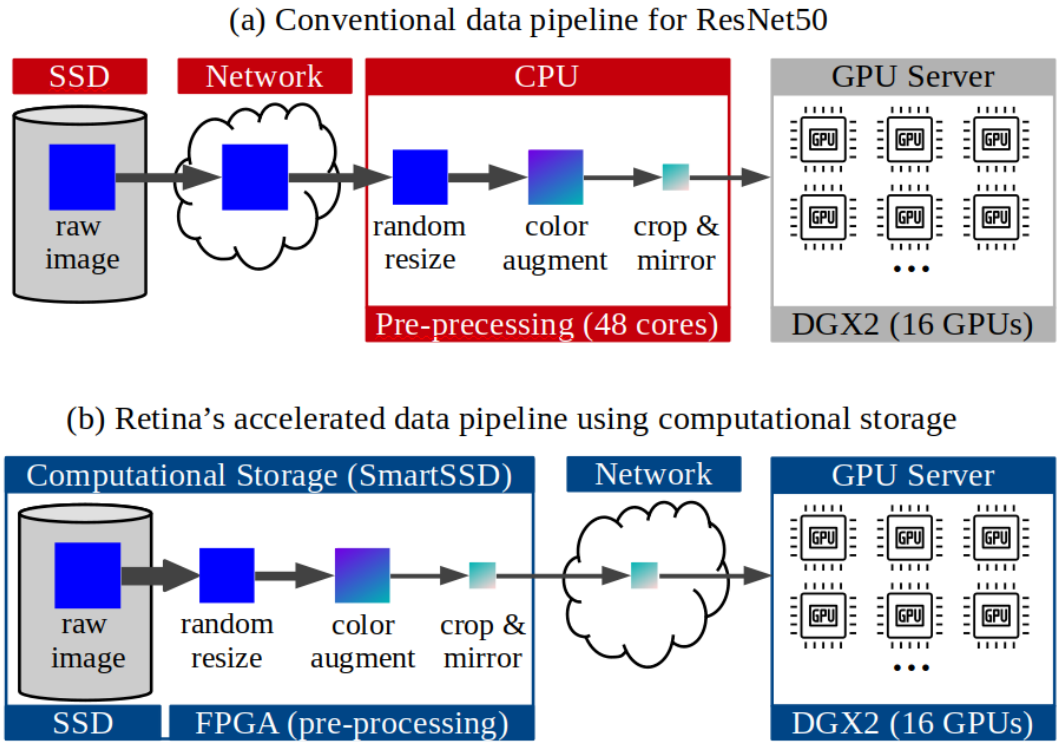

Figure 9.3: Comparison of conventional data pipeline and Retina's accelerated pipeline for ResNet50. Retina removes three major bottleneck in DL data pipeline, namely (1) CPU bottleneck by exploiting FPGA in computational storage, (2) network bottleneck by transferring compact pre-preprocessed data, and (3) storage bottleneck by leveraging high internal bandwidth in computational storage.

GPU idle time, which directly impacts the performance and cost of the DL training.

With Retina, we will further study to improve the DL pipeline performance by

- Leveraging near-data-computation by integrating computational storage to the host system. Offloading preprocessing computation from CPU to on-storage accelerator.
- Reduce fetch stalls by transferring input data directly to the accelerator's DRAM to leverage the high on-chip IO bandwidth. Further, improve overall throughput and reduce network latencies by transferring processed data back to the host DRAM.

As shown in the Figure 9.3, with Retina, we plan to integrate the preprocessing pipeline on

SmartSSD to alleviate CPU, IO, and network bottlenecks. To realize the Retina's computational pipeline, we plan to deploy a user-space framework to integrate with TensorFlow-based AI/ML applications. The framework would receive information on input data and preprocessing pipeline stages to construct the computational pipeline on the SmartSSD. It then pass control to Retina's key-value store to fetch data from the SSD to the accelerator's DRAM. Later the data is processed on the accelerator and passed to the host DRAM.

Chapter 10

Conclusion

Our cross-layered key-value store is used for: Frist, to efficiently store and retrieve data from the computational storage. The second and most important use case is integrating AI/ML preprocessing pipelines to storage. While offloading preprocessing onto SmartSSD can provide a performance boost, using traditional file systems or key-value stores hinders the full utilization of the near-data computation capabilities. Thus, our design avoids CPU bottleneck by offloading data plane control to FPGA, reducing network latencies, fully utilizing SmartSSD high internal bandwidth with peer-to-peer data movement, and enabling data to be fetched directly from SSD to the preprocessing pipeline.

Bibliography

[1] Using multiple compute units, . URL https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/yih1504034306435.html.

[2] Cxl specification, . URL https://www.computeexpresslink.org/download-the-specification.

[3] fio: Flexible i/o tester. URL https://github.com/axboe/fio.

[4] fast data pipelines for deep learning training - nvidia. URL https://on-demand.gputechconf.com/gtc/2018/presentation/s8906-fast-data-pipelines-for-deep-learning-training.pdf.

[5] MongoDB. https://www.mongodb.org/.

[6] OpenCL. https://www.khronos.org/opencl/.

[7] Product Brief: Samsung SmartSSD Computational Drive. https://samsungsemiconductor-us.com/smartssd-archive/pdf/SmartSSD_ProductBrief_13.pdf.

[8] SNIA: Computational Storage. . https://www.snia.org/computational.

[9] Computational Storage Architecture and Programming Model. . https://www.snia.org/sites/default/files/technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.8R0-2021.06.09.pdf.

[10] Vitis unified software platform documentation: Application 238 acceleration development, . URL https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis.

[11] Vitis unified software platform documentation: Application acceleration development, . URL https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Execution-Model.

[12] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1(1):598–609, 2008.

[13] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 753–766, Renton, WA, July 2019.

[14] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In Vaughn Betz and George A. Constantinides, editors, *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, pages 151–160. ACM, 2014.

[15] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. Lightstore: Software-defined network-attached key-value drives. pages 939–953. ACM, 2019.

[16] OpenCAPI Consortium. Opencapi - a new standard for high performance memory, acceleration and networks, Apr 2017. URL http://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM*

Symposium on Cloud Computing (SoCC), pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM. ISBN 978-1-4503-0036-0.

[18] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014. ISBN 099297870X.

[19] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 ACM SIGMOD/PODS Conference*, pages 449–466, Amsterdam, The Netherlands, June 2019.

[20] Facebook. RocksDB. http://rocksdb.org/.

[21] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on fpgas: a survey. *VLDB J.*

[22] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, 2009.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, page 1026–1034, 2015.

[24] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. DISC08, pages 350–364, 2008.

[25] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. Pink: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 173–187, Boston, MA, July 2020.

[26] JooHwan, Hui Lee, Veronica Zhang, Praveen Lagranges, Xiaodong Krishnamoorthy, YangSeok Zhao, and Ki. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. *IEEE '20*, 19(2):114–117, 2020.

[27] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. SYSTOR19, page 144–154.

[28] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 34:1–34:15. ACM, 2018.

[29] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1675–1687. ACM, 2016.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[31] Chang-Gyu Lee, Hyeongu Kang, DongGyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. ilsm-ssd: An intelligent lsm-tree based key-value SSD for data analytics. pages 384–395. IEEE Computer Society, 2019.

[32] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 38–49, Brisbane, Australia, April 2013.

[33] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of Practical Synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 3:1–3:8, San Francisco, California, June 2016.

[34] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 447–461, Ontario, Canada, October 2019.

[35] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, pages 27:1–27:14, Amsterdam, The Netherlands, April 2014.

[36] IC K NOWLEDGE LLC. Lithovision-2020: Economics in the 3D Era, 2020. https://semiwiki.com/wp-content/uploads/2020/03/Lithovision-2020.pdf.

[37] Ajit Mathew and Changwoo Min. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[38] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the cell distributed b-tree store. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 451–464, Denver, CO, June 2016.

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

[40] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: designing in-storage computing

system for emerging high-performance drive. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 379–394. USENIX Association, 2019.

[41] Behzad Salami, Oriol Arcas-Abella, and Nehir Sönmez. HATCH: hash table caching in hardware for efficient relational join on FPGA. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, page 163. IEEE Computer Society, 2015.

[42] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 28:1–28:12. ACM, 2020.

[43] Benjamin Sowell, Wojciech M. Golab, and Mehul A. Shah. Minuet: A scalable distributed multiversion b-tree. *Proc. VLDB Endow.*, 5(9):884–895, 2012.

[44] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. nkv: near-data processing with kv-stores on native computational storage. In Danica Porobic and Thomas Neumann, editors, *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 10:1–10:11. ACM, 2020.

[45] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H. C. Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 603–615, Boston, MA, July 2020.

[46] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. Fpga-accelerated compactions for lsm-based key-value store. In Sam H. Noh and Brent Welch,

editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 225–237. USENIX Association, 2020.