What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store

Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu, *Alibaba Group*

https://www.usenix.org/conference/fast24/presentation/zhang-weidong

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store

Weidong Zhang, Erci Xu[*], Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, Jiesheng Wu

Alibaba Group

Abstract

In this paper, we qualitatively and quantitatively discuss the design choices, production experience, and lessons in building the Elastic Block Storage (EBS) at ALIBABA CLOUD over the past decade. To cope with hardware advancement and users' demands, we shift our focus from design simplicity in EBS1 to high performance and space efficiency in EBS2, and finally reducing network traffic amplification in EBS3.

In addition to the architectural evolutions, we also summarize development lessons and experiences as four topics, including: (i) achieving high elasticity in latency, throughput, IOPS and capacity; (ii) improving availability by minimizing the blast radius of individual, regional, and global failure events; (iii) identifying the motivations and key tradeoffs in various hardware offloading solutions; and (iv) identifying the pros/cons of alternative solutions and explaining why seemingly promising ideas would not work in practice.

1 Introduction

Elastic Block Storage (EBS) service is a cornerstone in today's cloud [16, 18, 19]. In EBS, the storage service is in the form of virtual block devices with high performance, availability, and elasticity. The most outstanding characteristic of EBS architecture is the compute-to-storage disaggregation where the virtual machines (compute end) and disks (storage end) are not physically co-located but interconnected via datacenter networks.

In this paper, we start by revisiting the evolutions behind the three generations of EBS at ALIBABA CLOUD [16]. EBS1 marks our initial step in adopting the compute-to-storage philosophy. In EBS1, there are two notable design choices: in-place update from virtual disks (VDs) to physical disks, and the exclusive management of virtual disks. First, EBS1 directly maps a VD inside the virtual machine (VM) as a series of 64 MiB Ext4 files in the backend storage server. Moreover, EBS1 employs a fleet of stateless BlockServers to manage VDs where each VD is exclusively handled by a BlockServer. While EBS1 had been successfully deployed on more than 300 HDD-backed clusters, its limitations also unfolded. The straightforward virtualization led to severe space amplification and performance bottlenecks.

We then developed EBS2 with two significant changes: the log-structured design, and VD segmentation. First, we employed the Pangu [35] distributed file system as our storage backend, and redesigned the BlockServers to convert VDs' all writes to sequential appends. By switching to a log-structured layout, EBS2 still used three-way replication for incoming writes but could transparently perform data compression and erasure coding (EC) in the background during garbage collection (GC). Moreover, EBS2 split VDs into finer segments (32 GiB each), thus shifting the mapping between VDs and BlockServers from VD level to Segment level. With the above two changes, EBS2 was able to reduce the space efficiency from 3 (i.e., three-way replication) in EBS1 to 1.29 on average in the field. Moreover, supercharged with SSDs, an EBS2-backed VD can achieve up to 1 M IOPS and 4,000 MiB/s throughput with 100 μs-level latency on average. Unfortunately, EBS2 also faced a significant challenge. That is, the traffic amplification factor increased to 4.69, namely 3 (foreground replication write) plus 1 (background GC read) and 0.69 (background EC/compression write).

Hence, we built EBS3 to reduce traffic amplification using online (i.e., foreground) EC/compression via two techniques: Fusion Write Engine (FWE), and FPGA-based hardware compression. FWE aggregates write requests from different segments (if necessary) to meet the size requirement of EC and compression. Moreover, EBS3 offloads the compute-intensive compression to a customized FPGA for acceleration. As a result, EBS3 can reduce the storage amplification factor from 1.29 to 0.77 (after compression) and the traffic amplification factor from 4.69 to 1.59 while still maintaining performance similar to EBS2. Since release, EBS3 has been deployed on more than 100 clusters, serving over 500K VDs.

Figure 1 outlines the chronological progression of Alibaba EBS since 2012. We highlight the time of major releases (i.e., EBS1 to EBS3), the integration of key techniques (e.g., Luna, our user-space TCP stack [46]) and the adoption of advanced hardware (e.g., Persistent Memory in EBSX). The evolution of EBS demonstrates a shift in focus from performance to space

[*]Corresponding author.

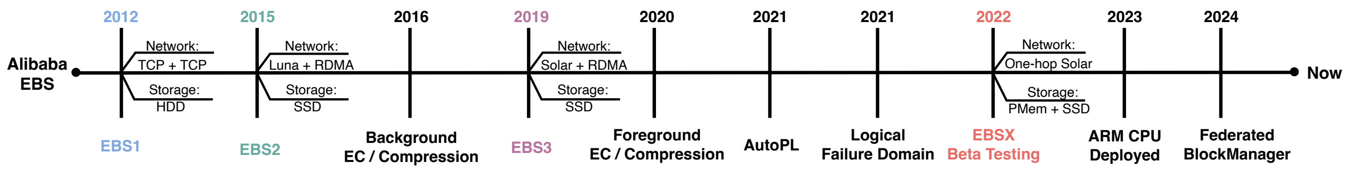


Figure 1: Alibaba EBS Timeline

and traffic efficiency. Nevertheless, simply altering the high-level architecture is not enough. Next, we further discuss our lessons in building a high-performance and robust EBS from four perspectives, including elasticity, availability, hardware offloading, and alternative solutions.

One representative feature of a cloud block store is elasticity—providing VDs with varying performance and capacity levels. There are two key aspects: identifying boundaries and achieving fine-grained tuning. First, we discover that the average and tail latency are dominated by different factors—hardware overhead and software processing, respectively. Thus, we build corresponding solutions including EBSX, a one-hop architecture backed by persistent memory for minimizing average latency, and the use of dedicated threads for I/O to alleviate tail latency. Furthermore, we realize that throughput and IOPS are bounded by similar mechanisms. In the frontend BlockClient, we optimize the stack by moving the processing from kernel to user-space and then to hardware offloading in FPGA. In the backend BlockServer, we utilize high parallelism to achieve efficient throughput/IOPS control. As for space elasticity, EBS not only provides wide ranges of storage space (i.e., from 1 GiB to 64 TiB) but also supports features including flexible resizing and fast cloning.

We then move on to discuss threats to the availability of EBS, especially under large-scale deployment. We begin by categorizing three levels of failure events, including individual, regional, and global, that can lead to one, several, or all VDs in a cluster (temporarily) ceasing service. With VD segmentation and segment migration since EBS2, field diagnosis indicates that regional events become more frequent and an individual event can now easily cascade into regional or even global ones. Therefore, on the control plane, we developed a Federated BlockManager to organize the VDs into mini groups and use CentralManager for coordination. Additionally, on the data plane, we have built the logical failure domain to limit the destinations of segment migration.

In the third topic, we highlight the importance of offloading key control/data paths to hardware for acceleration. We use the offloading evolutions of both BlockClient and BlockServer as examples to discuss the tradeoffs between different options. Specifically, BlockClient started with FPGA offloading to accelerate storage/network virtualization. However, impacted by FPGA instability under large deployment (e.g., 22% of downtime caused by FPGA-related issues), our BlockClient dropped the FPGA-based approach and adopted the ASIC-based solution. Conversely, BlockServer, which also initially chose FPGA for speeding up EC/compression, opted for the many-core ARM CPU as the next stop due to the flexibility and cost requirement.

The final topic is organized as a series of "What If?" questions (§6). Through three Q&A, we explain why seemingly promising ideas, such as extending EBS1 with segmentation but without a log-structured design, eventually failed, and discuss the possibilities of alternative solutions (e.g., building EBS with open-source software). We end this paper with a short discussion of related work and a conclusion.

2 Architecture Evolution: A Shift of Focus

2.1 EBS1: An Initial Foray

EBS1 marked our first step into offering an elastic block store based on a disaggregated architecture. By placing the compute and storage in different clusters and connecting them via the datacenter network, this design offers flexibility in deployment, scaling, and evolution. Such philosophy has been widely adopted by many vendors, such as Microsoft Azure [25] and Google datacenters [30].

Compute end. Figure 2 provides a high-level overview of EBS1. A compute cluster comprises multiple servers where each server runs one BlockClient and can host several VMs. In addition, a VM can mount one or more VDs. Users can access the VD as a normal block device and the host server forwards the I/O requests to the storage clusters via the BlockClient.

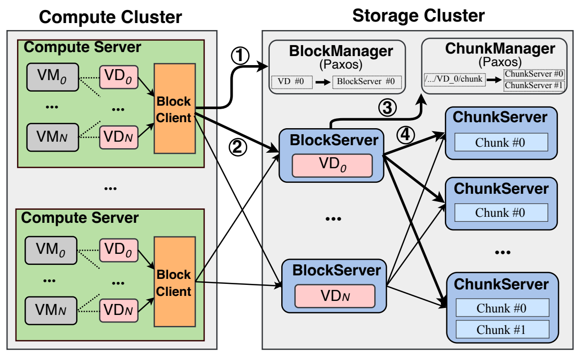


Figure 2: The system architecture of EBS1 (§2.1). ***VD***: *Virtual Disk.* ***VM***: *Virtual Machine. BlockManagers and ChunkManagers all run three-instance Paxos groups. Each VM can host multiple VDs.*

Storage end. EBS1 used a different fleet of dedicated servers for storage. First, we build the BlockManager (a set of three nodes backed by Paxos) and a group of BlockServers. The BlockManager maintains VDs' metadata, such as the capacity and snapshot versions. The BlockServers handle I/O requests of multiple VDs assigned by the BlockManager. Note that the BlockManager can reassign a VD to another BlockServer during failover. Second, we further introduce a data abstraction, called *chunk*. The Logical Block Address (LBA) space of a user's VD is divided into a series of 64 MiB chunks. Similarly, the ChunkManager (a set of three nodes backed by Paxos) manages the chunks' metadata. The ChunkServer stores a 64 MiB chunk as a 64 MiB Ext4 file (called DataFile) and performs in-place updates to the chunk for write requests. Each chunk is three-way replicated on three different ChunkServers. For efficiency, we use thin provisioning—allocating space only when the user writes data to the VD.

Network. There are mainly two sets of network in `EBS1`. The frontend network connects the compute and storage clusters. The backend network inside the storage clusters connects the BlockServers to the ChunkServers. Both use Clos topology and rely on the 10 Gbps network with the kernel TCP/IP stack.

Data-flow. When a VM issues a new write request to its VD, BlockClient first contacts the BlockManager to locate the corresponding BlockServer for this request (① in Figure 2). Then, the BlockClient forwards the write request to the BlockServer (②). The BlockServer further asks the ChunkManager to determine the three ChunkServers (③) and persists the data accordingly (④). In practice, the BlockClient caches the VD-to-BlockServer mappings, and BlockServers cache chunk-to-ChunkServer mappings (i.e. skipping ① and ③).

Limitations. `EBS1`, released in 2012, has served over 1 million VDs and stored hundreds of PBs of data across hundreds of deployed clusters. Its straightforward and mature designs (e.g., in-place update and N-to-1 mapping between VDs and BlockServers)—while expediting development and deployment—limit the performance and efficiency. For example, to reduce the space overhead, we wish to use data compression and EC. However, compression non-deterministically alters the size of data which breaks the direct mapping of in-place updates. In addition, EC has a minimum size requirement (e.g., when the stripe unit is 4 KiB, EC(4,2) requires at least 16 KiB) and thus can result in significant write amplification, especially for small I/O requests. Another limitation is that, under the N-to-1 mapping, the performance of a VD is ultimately bounded by the performance of the corresponding BlockServer which can suffer from hotspot issues under burst workloads. In addition, with HDD and traditional kernel TCP/IP stack, we find it difficult to quantify and guarantee SLOs to users.

2.2 `EBS2`: Speedup with Space Efficiency

Overview. Figure 3 presents the high-level architecture of `EBS2`. The most significant change is that `EBS2` no longer directly handles the data persistence or manages the consensus protocol. Instead, it builds on top of a distributed storage system—named Pangu [35]—which provides append-only file semantics and distributed lock services (based on a customized Raft protocol). The BlockServers employ a log-structured design [38] and translate the VDs' write requests into Pangu append-only writes, thus enabling efficient data compression and EC during background garbage collection. We also split the VD address space into fixed-size segments, allowing one VD to be served by multiple BlockServers. Also, with segmentation, failover is no longer at the granularity of the whole VD but a segment—BlockManager migrates the impacted segment to another BlockServer. In addition, the BlockManager directly uses Pangu distributed lock service instead of Paxos for leader election.

As a result, we modified the I/O procedures as follows. After receiving a VD's request, BlockClient first retrieves the segment's address from BlockManager (① in Figure 3, which can be skipped by caching), and then forwards the I/O requests to the target BlockServer (②). BlockServer employs the Log-Structured Block Device (LSBD) Core to convert I/O requests into Pangu APIs and then calls an embedded Pangu client for persisting or fetching data (③). Note that, since `EBS2`, a BlockServer and a Pangu's ChunkServer, while co-located on the same physical server, are logically independent processes and rely on backend network for transferring data (i.e., not enforcing locality).

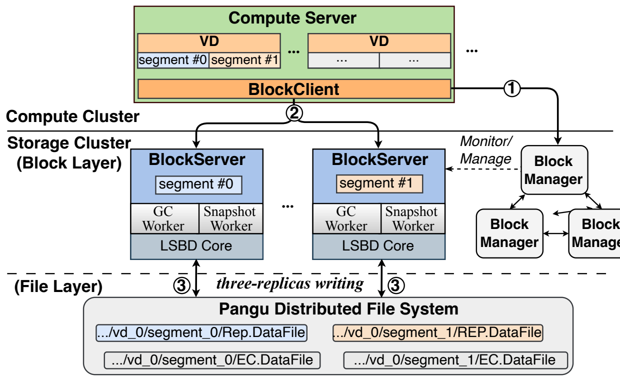


Figure 3: The overview of `EBS2`. ***LSBD***: *Log-Structured Block Device.* ***REP.DataFile***: *DataFile with three-way replication.* ***EC.DataFile***: *DataFile with EC(8,3) encoding.*

Disk segmentation. Figure 4 illustrates how `EBS2` partitions the VD's LBA into several 128 GiB segment groups each of which further comprises multiple 32 GiB segments. BlockServers in `EBS2` operate at the granularity of segments. Further, `EBS2` organizes the segment group as a series of data sectors and allocates them to the segments in a round-robin fashion. Finally, `EBS2` associates one segment with multiple *DataFiles* (512 MiB by default) to support concurrent writes. DataFile is essentially a Pangu file designed to persist a portion of a segment's data. These different levels of parallelism help `EBS2` alleviate the hotspot accessing in VDs.

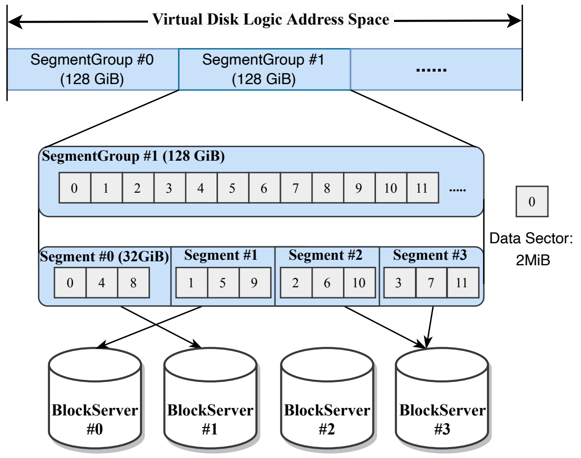


Figure 4: The Disk Segmentation Design of EBS2 (§2.2).

Log-structured Block Device. In EBS2, we developed a LSBD Core to support the append-only semantics of the underlying Pangu and thus split traffic into frontend (i.e., client I/Os) and backend (i.e., GC and compression). Figure 5 shows the frontend I/O flow including persisting users' data as a series of 4 KiB blocks and 64B metadata pairs (①), responding to users (②), recording updates in the transaction file (③), and updating the in-memory index map (④). The index map is essentially a log-structured merge tree (LSM-tree) to speed up the locating process by storing a mapping from VD's LBA to the corresponding DataFile ID, offset and length. The TxnFile accelerates the index map rebuild upon segment migrations. EBS can recover the in-memory index map by reading the latest I/Os' LBA-to-DataFile mappings from the TxnFile, without the need of tail scanning the data blocks in the DataFiles. Note that DataFile, TxnFile and the SSTables in the LSM-tree are all Pangu files.

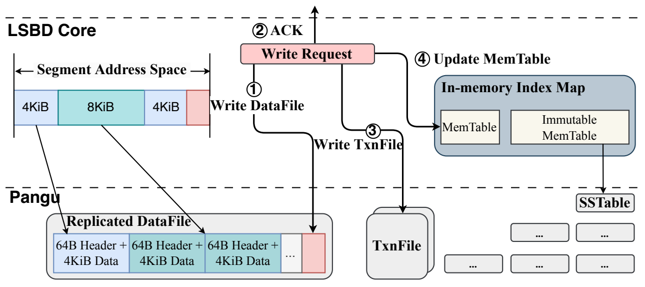


Figure 5: The data organization and persistence format of LSBD. *TxnFile: TransactionFile.*

GC with EC/Compression. EBS2 runs GC at the granularity of *DataFile* (see Figure 6). When stale data within DataFiles reaches the threshold, EBS2 initiates performing GC by collecting valid data from the dirtiest DataFiles under the same segment and combining them as new DataFiles. EBS2 finishes GC by updating the TxnFile and the in-memory index map. During GC, we also convert the replicated "REP.DataFiles" to space-efficient "EC.DataFiles" with (8,3) EC and LZ4 compression.

Given that compression can alter the size of data, we structured the EC.DataFile into three main components: a DataFile Header, a series of CompressedBlocks, and an Offset Table. The Compressed DataFile Header includes a magic number (marking the start of the DataFile), version and checksum. Each CompressedBlock contains CompressionHeader (CmpHdr) and CompressionBody (CmpBdy). The CmpHdr records the timestamp, the compression algorithm (LZ4 by default), the size of CmpBdy, and the checksum. CmpBdy contains the compressed data and metadata (i.e., 4 KiB + 64B before compression). At the end of the Compressed DataFile, we enclose the mapping between the original LBA in VD and the location in the Compressed DataFile as OffsetTable. When reading the compressed data, EBS2 first locates data by querying OffsetTable, then reads and decompresses the data.

We leveraged the opportunity of GC to perform the transformation of erasure coding and compression. If needed, EBS2 can schedule special types of GC tasks that preferably select "Rep.DataFiles" (replicated, non-compressed data) over existing "EC.DataFiles" (erasure-coded, compressed data), in order to make up more storage space for incoming writes.

The garbage percentage thresholds used to trigger GC in production vary, depending on the cluster storage usage and the workload. We deployed a set of optimizations for improving GC efficiency (e.g., placement based on inferring the block invalidation time [39]). In production, for the most stressed clusters, the write amplification due to GC (i.e., the number of bytes written by BlockServer and GCWorker over the number of bytes written by BlockServer) is less than 1.5.

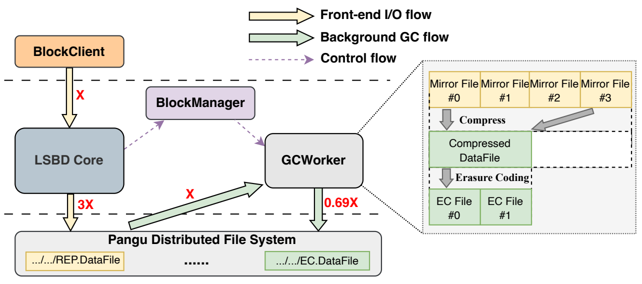


Figure 6: The Garbage Collection in EBS2.

BlockManager with higher availability. The integration of Pangu enhances the availability of EBS2's control plane. First, through the Pangu lock service, the BlockManager can continue serving clients even in the face of two out of three node failures. Second, EBS2 now stores the VDs' metadata in a persistent and replicated key-value store as Pangu files, while EBS1 stores the VDs' metadata in local disks where data loss could lead to an extended repair time.

Network. EBS2 uses a similar network setup as EBS1 except for two fundamental differences. First, for the frontend network, we replace the kernel TCP with our user-space TCP implementation (called Luna [46]) over a 2×25 Gbps network. Luna achieves high performance (up to $3.5\times$ throughput improvement and 53% latency reduction) by leveraging a run-to-completion thread model and a zero-copy memory model. Second, for the backend network, we use a 2×25 Gbps RDMA network to meet the demanding SLAs [29]. Note that the two changes above only affect the data path. For the control path, we still use the kernel TCP (e.g., RPCs between BlockManagers and BlockServers).

Snapshot. The architectural changes in EBS2 also facilitate creating snapshots for VDs. With the out-of-place update, creating a snapshot in EBS2 no longer blocks foreground traffic. Instead, when receiving a snapshot request, BlockManager simply records a timestamp and asks the snapshot workers in BlockServers to upload the updates between the last snapshot timestamp and the latest one. As a result, generating a snapshot for 20GB of new data only takes around 30 seconds in EBS2, much less than the average 33 minutes in EBS1.

Other background I/O. Apart from GC, one important background task is data scrubbing, which performs periodical scanning to detect anomalies such as disk corruption and CPU silent data errors. To minimize the performance impacts, we have separated the scrubbing traffic from the GC tasks, capped the scrubbing traffic at 10 MiB/s (i.e., scanning all DataFiles every 15 days), and leveraged the heartbeat mechanism for monitoring the scrubbing progress.

Deployment. We released EBS2 in 2015 and subsequently scaled to more than 500 clusters for 2 million VDs. EBS2 could provide a virtual disk with an average write latency of 100 μs ($12\times$ reduction than EBS1), a maximum IOPS of 1 M ($50\times$ increase), and a maximum throughput of 4,000 MiB/s ($13\times$ increase) for the guest OS. In the field, the compression ratio of the LZ4 algorithm is between 43.3% $\sim$ 54.7%, and the average compression ratio is 50.1%. With Compress-EC in GC, EBS2 can reduce space usage from 3 to 0.69 replicas. Since un-GCed DataFiles are still stored with three-way replication, the average number of replicas in EBS2 is 1.29 in the field on average. For better management, we also reduced the cluster size from 700 servers in EBS1 to around 100 in EBS2.

Limitations. EBS2 successfully improves the space efficiency but incurred heavy traffic amplification. Compared with EBS1, EBS2 increases the overall traffic from 3 (i.e., from three-way replication) to 4.69 (i.e., 3 from the frontend plus 1.69 from the backend GC), yielding only 15.5% of the network bandwidth for serving the VDs' requests. To alleviate this issue, one promising solution is to adopt online EC/Compression, which means to directly store users' data in erasure-coded and compressed format.

The challenges are twofold. First, erasure coding requires the raw data blocks to be at least 16 KiB to achieve high compression and encoding efficiency. However, in the field, nearly 70% of write requests are smaller than 16 KiB. Moreover, EBS aims to deliver 100 μs write latency, and accumulating enough data in such a short interval can be difficult. For example, in order to perform compression and erasure coding for all user writes (i.e., accumulating 16 KiB data blocks within each 100 μs interval), a segment needs to have a write throughput over 160 MiB/s—surpassing 90% of segments in production. Simply padding zeroes can result in an even higher traffic/space amplification. Second, even with the latency-optimized LZ4 algorithm, compressing a 16 KiB-sized data block still requires 25 μs for CPUs, and such overhead escalates significantly for larger ones, rendering an unacceptable performance penalty for our service.

2.3 EBS3: Foreground EC/Compression

Overview. EBS3 achieves online EC/Compression by utilizing a Fusion Write Engine (FWE) to merge small writes and adopt an FPGA to offload the compression computations. Specifically, EBS3 first leverages FWE to accumulate writes from different segments of different VDs (i.e., step ① in Figure 7). FWE then combines these incoming writes as DataBlocks and sends them to the FPGA-based accelerator for data compression (②). EBS3 then calls Pangu to persist the compressed DataBlocks as JournalFiles with EC(4,2), namely ③. After persisting JournalFiles, EBS3 sends acks to the VD indicating the I/O completion (step ④). Then EBS3 copies the uncompressed data and preserves them within the BlockServer's memory by segment (i.e., SegmentCaches, step ⑤). When the data in a SegmentCache reaches the threshold (512 KiB by default), EBS3 compresses the data via the host CPU, appends them to the DataFile with EC(8,3), and updates the TxnFile and in-memory index map (step ⑥).

For read requests, EBS3 first queries the SegmentCaches in the BlockServers as they have the latest data. If not found, EBS3 would further read the in-memory index map (i.e., the LSM-tree). Note that JournalFiles are EC-ed with compressed data from various segments of different VDs. Directly fetching data from JournalFiles can result in severe read amplification (i.e., decompressing with heavy scanning). Therefore, JournalFile is write-only during runtime and only readable during failover to recover yet-to-be-dumped data upon crash.

Fusion Write Engine. Usually, when receiving a batch of small write requests, FWE waits until the total amount of data reaches a threshold (16 KiB by default) and then forms a DataBlock before sending it to the FPGA for compression. We set the waiting timeout as 8μs (i.e., the interval between NIC pollings). Moreover, we discover that insisting on merging smaller writes (e.g., 4 KiB) can result in higher 99th tail latency (i.e., 220%). Therefore, for clusters that have infrequent small writes (e.g., certain clusters, on average, with only 3.72% of the workload are 4 KiB writes), we do not aggregate 4 KiB writes but directly append them with the traditional three-way replication.

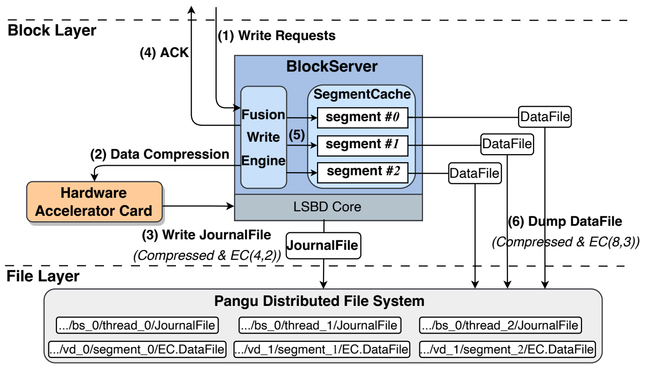


Figure 7: The architecture and I/O flow of EBS3.

FPGA-based compression offloading. We employ a customized FPGA to accelerate (de)compression, which includes a submission queue to buffer newly-formed DataBlocks and a completion queue to poll the results of hardware compression. We implement a scheduler inside the FPGA to split the DataBlocks as fixed-sized (e.g., 4 KiB) slices and employ multiple execution units to perform (de)compression tasks on these slices in parallel. To ensure data integrity, we place an end-to-end CRC check within the driver. After FPGA returns compressed data, EBS3 would immediately decompress the data and verify data integrity via CRC checking. Note this is a necessary overhead as, during failover, JournalFiles are the only data source.

Figure 8 shows the latency and maximum throughput of FPGA-offloading and CPU-only compression across different data block sizes, based on Silesia Compression Corpus [27]. The latency distribution of FPGA-offloading ranges from 29∼65 μs. Notably, when the data block size is 16 KiB, the latency of FPGA-offloading reduces by 78% compared to CPU-only. Further, FPGA-offloading achieves a maximum throughput of 7.3 GiB/s, whereas CPU-only compression is only 3.5 GiB/s. As data block size increases, the FPGA-offloading solution leads to larger performance gains.

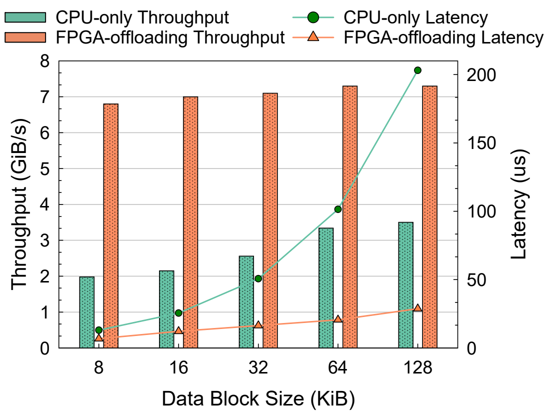
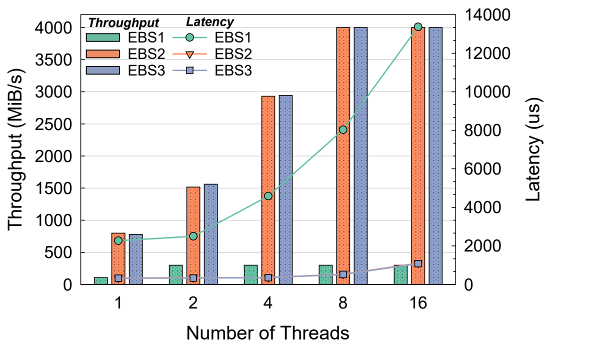


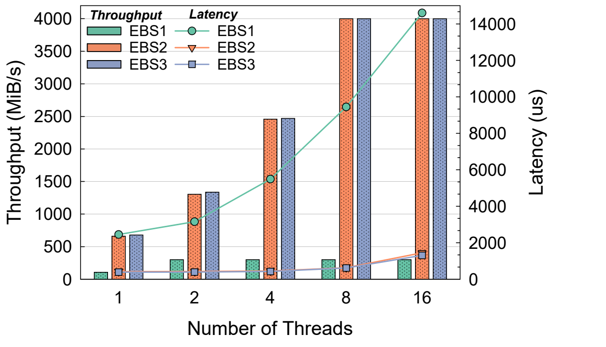
Figure 8: The compression performance comparison of FPGA-offloading and CPU-only with 8 cores compression based on Silesia Compression Corpus.

Network. EBS3 adopts higher linkspeed (i.e., 2 × 100 Gbps) network for both frontend and backend. In addition, we further developed Solar [36], a UDP-based transmission protocol. By leveraging the hardware offloading on our Data Processing Units (DPUs), Solar can pack each storage data block as a network packet, thereby achieving CPU/PCIe bypassing, easy receive-side buffer management and fast multi-path recovery.

Deployment. EBS3 has been deployed in over 100 storage clusters, serving more than 500K virtual disks since released in 2019. EBS3 offers comparable performance to EBS2. The incorporation of foreground EC/Compression in EBS3 enables all data to be stored immediately with high storage efficiency except in a few corner cases. As a result, the space efficiency (i.e., replica per data) in the field further drops from 1.29 in EBS2 to 0.77 in EBS3. In addition, the FPGA-based compression offloading can achieve 7.3 GiB/s throughput per card and the overhead ranges from from 29∼65 μs. The overall traffic amplification drops from 4.69 in EBS2 to 1.59 in EBS3 (based on field statistics and numbers may slightly vary due to compression ratio differences across workloads).



(a) Throughput and Latency of Random **Write** on Thread-to-core Pinning



(b) Throughput and Latency of Random **Read** on Thread-to-core Pinning

Figure 9: Random Write/Read Latency of Each Generation EBS under Multiple Threads and 4 KiB-sized I/O. Thread-to-core pinning means that each thread occupies one CPU core exclusively.

2.4 Evaluation

To quantitatively demonstrate the improvement led by the architectural evolutions, we extensively evaluate the performance of EBS1, EBS2 and EBS3 with microbenchmark (by stressing the VDs using FIO [21]) and application-based macrobenchmark (via RocksDB 6.2 with YCSB [26] and MySQL 8.0 with Sysbench [33]).

We evaluate the throughput and IOPS of the candidates by stressing random 4 KiB write and read. We also increase the number of threads (i.e., FIO jobs) from 1 to 16. Figure 9 shows the overall results. We can see that the throughput (i.e., bars) of EBS2 and EBS3 increases almost linearly before hitting the peak with 8 threads. With 8 threads, EBS2 and EBS3 can deliver 4,000 MiB/s throughput per VD (i.e., 1 M IOPS), which is 13× and 50× higher than the throughput and IOPS of EBS1. Figure 9 further depicts the latency variation with scaling of FIO jobs. We observe that the latency of EBS2 and EBS3 is similar and remains the same from 1 to 8 threads. Their latencies only experience a slight increase with 16 jobs due to delays caused by contention between threads after hitting throughput bottlenecks.

We use RocksDB with the default configuration. For MySQL, we use InnoDB and configure the user buffer size as 1 GiB, the page size as 16 KiB, the flushing method as direct I/O, the ramp-up time as 180 seconds and the execution time as 20 minutes. Figure 10 shows the results. Compared to EBS1, we observe 550% and 573% gains in throughput for insert and update—two write-dominated workloads—in YCSB, respectively. For read-dominated workloads, the throughput experiences an approximately 470% increase. Under oltp_insert workload in Sysbench, the throughput of EBS2 and EBS3 increase by 389% compared to EBS1. For the rest, the throughput increases by an average of 350%.

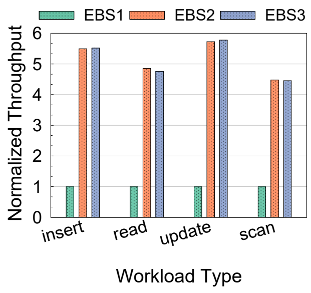
3 Elasticity: A Tale of Four Metrics

The capabilities (e.g., capacity and throughput) of a common block device (e.g., HDD and SSD) are usually bounded by the physical properties, such as encapsulation or interface. Backed by the cloud, EBS can provide VDs with much higher flexibility. In this section, we will share our experience of obtaining high elasticity, including pushing the upper bounds and achieving fine granularity.

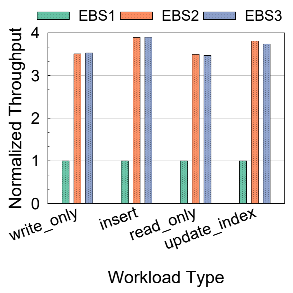
3.1 Latency

The latency of a VD is determined by the architecture, namely the path a request has traveled. For example, the latency of an EBS2-backed VD is bounded by the latency of the two-hop network (from BlockClient to BlockServer and then to ChunkServer), the software stack processing (i.e., BlockClient, BlockServer and Pangu) and the SSD I/O. Hence, the elasticity of latency is inherently coarse-grained, namely the different levels of time overhead under various architectures (e.g., EBS2 and EBS3). Next, from the perspectives of average and tail latency, we further analyze the status quo.

Average latency. In Figure 11a, we measure the 8 KiB random read/write average latency breakdowns across different generations of EBS in their corresponding top 10% busiest production clusters. We choose to not include EBS1 in the comparison as it is no longer deployed and many of its hardware (e.g., HDD and 10 Gbps network) are obsolete. From the comparison, we first observe that the hardware processing—including 1st/2nd hop network (marked as orange and pink) and disk I/O (yellow)—accounts for the majority of the total latency in both EBS2 and EBS3. In addition, while EBS3 requires more time to process the data due to the frontend EC/compression, the reduced data volume in return spends less time traveling the network (i.e., lower 2nd hop latency in EBS3), yielding similar overall latency between EBS2 and EBS3. Third, the major difference between read and write lies in the disk I/O latency. Note that EBS2 is backed by TLC-based SSDs while EBS3 is backed by QLC-based SSDs.

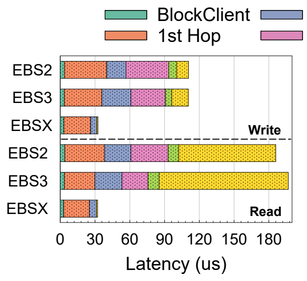


(a) Throughput Comparison of each EBS with **YCSB**

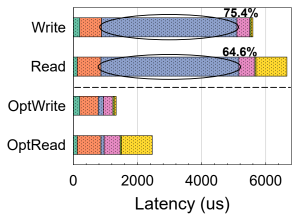


(b) Throughput Comparison of each EBS with **Sysbench**

Figure 10: Throughput Comparison (Normalized with EBS1).



(a) Average Latency Breakdown of EBS2, EBS3 and EBSX



(b) 99.999th Tail Latency Breakdown of EBS3

Figure 11: 8 KiB-Sized Avg. and Tail Latency Breakdown of EBS. ***1st hop:*** *network latency from compute to storage end.* ***2nd hop:*** *network latency from BlockServer to Pangu.*

Clearly, the key to improving average latency is reducing the hardware processing overhead. Therefore, we built EBSX, targeting latency-sensitive scenarios. EBSX installs the persistent memory (PMem) inside the BlockServers and directly stores the data in PMem with three-way replication. Compared to EBS2 and EBS3, EBSX skips the 2nd hop and drastically speeds up the disk I/O with PMem. Figure 11a shows that EBSX achieves 30 μs-level latency on both read and write. Note that, for space efficiency, data in PMem would be eventually flushed to Pangu and read statistics in Figure 11a is performance under cache hit (i.e., data in PMem).

Tail Latency. A user request may not be always served in time due to hardware failures [43,44], misconfigurations [29,36], software bugs [23,32] or simply resource contentions [20]. Figure 11b presents the breakdowns of 99.999th percentile latency, a common threshold for defining the tail latency [28], among EBS3 clusters. We have collected millions of slow requests and calculated the average latency of each procedure. We observe that the BlockServer processing, such as non-IO RPC destruction in the IO thread, background periodic scrubbing and index compaction, accounts for the majority (75.4% for write and 64.6% for read) of the tail latency.

This observation may sound rather counter-intuitive as the hardware-related issues are usually blamed as the culprits [42,45]. However, in EBS2 and EBS3, we have already incorporated a series of simple techniques to improve the quality of service of hardware processing. For example, network multi-path transport allows user requests to automatically shift traffic to other paths to avoid slow IO when suffering network abnormalities [36]. As another example, we employ a backup read/write strategy to trigger a retry to another location if the I/O takes longer than 99.9th percentile latency.

Our analysis indicates that the principal driver of the tail latency is the contention between the IO and the background tasks (e.g., segment status statistics and index compaction) in the BlockServers. In EBS2 and EBS3, the IO and the background tasks are executed on the same thread, leading to the IO hang-ups when the background tasks are triggered. To address this, we segregate the IO flow from other tasks and execute it on independent threads. With these enhancements, the 99.999th percentile write latency of EBS3 has been reduced to 1 ms, and the read latency reduced to 2.5 ms (i.e., OptWrite/Read in Figure 11b).

Summary. First, the elasticity of latency is coarse-grained—defined by the architectures, along with the hardware used (e.g., from EBS2 to EBSX). Second, optimizing hardware-induced latency is often straightforward. One can shorten the path (e.g., skip a network hop), use faster devices (e.g., PMem) or simply offset the risks with multi-path or retries. Third, tail latency by software stack has not received enough attention and may be regarded as noise. Our analysis suggests that under the high-speed network and fast SSDs, software-induced tail latency can be the dominant factor.

3.2 Throughput and IOPS

In the context of EBS, we often discuss the elasticity of throughput and IOPS together because the two metrics are often constrained by the same set of mechanisms. EBS achieves high elasticity in throughput/IOPS by optimizing two components on the key data path, BlockClient and BlockServer.

BlockClient. Every IO issued from the VD is first touched by the BlockClient. Therefore, the throughput and IOPS are bounded by the BlockClient's processing and forwarding capability. In EBS1, the BlockClient is implemented as a kernel module, and all IO requests are processed by the CPU. In EBS2, we move the IO processing to the user space by introducing a user-space TCP stack to handle the IO requests [46]. In EBS3, we further offload the IO processing to the hardware where a general-purpose FPGA, completely bypassing CPU, performs direct data move from VMs, data block CRC calculation, and packet transmission [36].

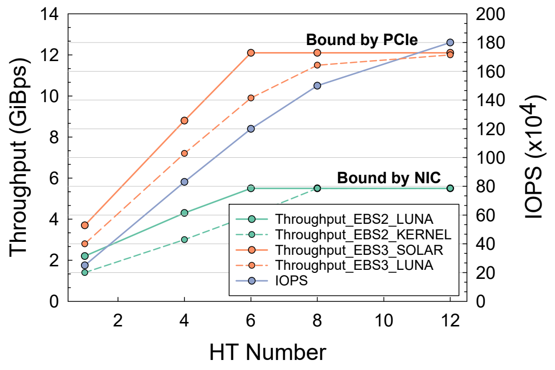


Figure 12: The maximum throughput and IOPS changes of BlockClient with different HT numbers.

In Figure 12, we measure the maximum throughput and IOPS of BlockClient under different optimizations. We observe that EBS2 with the 2×25 Gbps network, throughput is constrained by network capabilities. For EBS3 with the 2x100G network, the bottleneck shifts to the PCIe bandwidth. Furthermore, as long as network bandwidth is available, IOPS increases with the number of hyper-threads (HTs).

BlockServer. Unlike BlockClient, once the requests reach the BlockServer, the throughput and IOPS are constrained by the levels of parallelism. Obviously, the more a request can be divided and served in parallel, the higher the throughput and the IOPS. Since EBS2, we have introduced three levels (i.e., SegmentGroup, Segment and Data Sector) of parallelism to enhance virtual disk performance.

Recall that the Data Sector size (initially 2 MiB) is configurable in the segmentation design. Reducing the Data Sector size allows virtual disks to scale one SegmentGroup across more BlockServers, thereby obtaining higher throughput/IOPS. In the field, we further decrease Data Sector size to 128 KiB and EBS2 (and EBS3) are able to deliver 1,000 IOPS for every GiB subscribed. Note that configuring an even smaller Data Sector size may backfire as the write requests can be too fragmented, thereby leading excessive number of sub-I/Os even for small writes and placing prohibitively high pressure on the first-hop network.

Base+Burst allocation. With high throughput/IOPS enabled by BlockClient and BlockServer optimizations, efficiently allocating throughput/IOPS to VDs is the next step. Unlike the coarse-grained elasticity in latency, users can subscribe throughput and IOPS of VD on demand without altering the capacity, which is called auto performance level (AutoPL). However, we discover that the throughput/IOPS in practice is often over-provisioned by users to handle the sporadic workload bursts. For better resource efficiency, we have proposed the *Base+Burst* strategy based on the following techniques.

- *Priority-based congestion control.* We categorize IOs into baseIO and burstIO. BaseIO is pre-defined during virtual disk creation, while burstIO is allocated based on the available capability (i.e., not guaranteed). When a BlockServer is unable to meet all IO demands, it prioritizes processing the baseIO to ensure consistent latency. Currently, the maximum baseIO capacity of a VD is 50,000 IOPS, and the maximum burstIO capacity is 1 million IOPS.
- *Server-wise dynamic resource allocation.* Burst workloads can also place a heavy burden on BlockServer processing ability. Therefore, since EBS2, we devise the dynamic resource allocation to allow BlockServer to preempt resources (bandwidth, CPU cores and memory) from background tasks (e.g., GCWorker) to handle workload spikes.
- *Cluster-wise hot-spot mitigation.* To ensure enough headroom for burstIO, especially under concurrent bursts onto the same BlockServer, we use cluster-wise load-balancing to remove hot spots. Under such scenarios, BlockManager would aggressively check the traffic status and migrate segments more frequently between BlockServers.

Summary. The first lesson here is that the upper bound of a VD's throughput/IOPS is determined by the client (i.e., processing/forwarding ability) as the backend can easily scale with parallelism. In addition, the high throughput/IOPS is often desired but not always needed. Therefore, using a *Base+Burst* strategy to cope with workload spikes can be more economically beneficial for both users and vendors.

3.3 Capacity

Achieving elasticity in capacity is a fundamental requirement of a cloud block store service. In EBS, we have further included the following features.

- *Flexible space resizing.* The segmentation design enables EBS with seamless support for VD resizing (i.e., adding or removing SegmentGroups). EBS currently supports virtual disk sizes ranging from 1 GiB to 64 TiB.
- *Fast VD cloning.* One outstanding characteristic of serverless applications is a large volume of resources (e.g., VDs) needs to be allocated in a short time. To support this, EBS uses the *Hard Link* of Pangu files, which allows the cloning of multiple disks within a storage cluster via downloading a single snapshot. As a result, EBS2 enables the creation of up to 10,000 virtual disks (each 40 GiB) in 1 minute.

4 Availability: The Dark Side of Scaling

Availability has always been a priority of cloud services. In EBS, we especially focus on the *blast radius*—defined as the number of VDs experiencing unavailable services upon failures. Here, we categorize the blast radius as follows.

- *Global.* In the face of such an event, the service availability of an entire cluster is impacted. A simple example is an abnormally operating BlockManager causing the entire cluster to perform in an undesired fashion (e.g., a misconfiguration causing network congestion and subsequent retry storms). Note that EBS service runs on a per-cluster basis and we do not discuss datacenter-level failures here.
- *Regional.* For a regional event, we define it as a failure that incurs the component(s) to deny service for several VDs. For example, when a BlockServer crashes, the hosted VDs would experience an outage until the corresponding segments are migrated or all incoming I/Os are forwarded.
- *Individual.* When an individual event occurs, only one VD is influenced. Representative examples include an uncorrectable error inside the disk (and subsequently a read retry) and a software bug that leads to an unsuccessful and redirected write.

A straightforward solution to minimize the blast radius is setting smaller clusters. In EBS2 and EBS3, we have reduced the cluster size from 700 nodes (in EBS1) to around 100. The benefit is obvious since there are much fewer VDs influenced by a global event now. However, this approach, while straightforward and effective, would not alleviate regional and individual failure events.

Meanwhile, the *regional* events are likely to be more severe due to two trends. First, EBS2 introduces the segmentation to split one VD into 32 GiB segments and hence a VD in EBS2 (or EBS3) is supported by multiple BlockServers instead of one in EBS1. Moreover, the average capacity of VDs only slightly increases from EBS1 to EBS2 and EBS3 (e.g., 197 GiB to 220 GiB). Therefore, we can conclude that in EBS2 and EBS3, a BlockServer hosts much more VDs than EBS1. Consequently, when a BlockServer crashes, more VDs are going to be influenced.

Moreover, *individual* events can cascade into *regional* failures as the segments can be migrated since EBS2. For example, an internal incident occurred as tens of BlockServers in an EBS2 cluster kept crashing and rebooting, degrading the total cluster capacity and the I/O quality of thousands of VDs. In the beginning, a faulty segment crashes its BlockServer because of a buggy code logic—an individual event. Then, the control plane tries to migrate the segment to other BlockServers. However, as the client keeps retrying the failed requests, every BlockServer that loads the segment crashes as well, turning the individual event into a regional failure. This failure can easily grow to a cluster-wise outage in a short period if not manually intervened. Note that the cascading failures are not unique to EBS (e.g., cases in HBase [4–6]).

To adapt to the trends in regional and individual events, we further come up with techniques in both the control and data plane to improve the availability in EBS.

4.1 Control Plane: Federated BlockManager

In each cluster, the control plane of EBS2 (referred to as BlockManager in §2) initially consists of a group of three nodes that leverages the distributed lock service provided by Pangu for leader elections. The leader node in the group serves all the control plane requests to the corresponding cluster and persists any state changes related to VDs to a single metadata table stored in Pangu.

This setup presents two challenges. First, the single leader serves all the VDs in the cluster with one single server. As the VD's density grows, the chances and scale of its service disruption get higher. Second, the single metadata table hosts the metadata of VDs in the cluster. Once a part of the metadata table becomes corrupted, the BlockManager may not be able to load the metadata in memory, and cannot serve the VDs until the metadata table is repaired. In production, we have seen an increasing VD-per-cluster density over the past several years, urging us to solve both issues to provide high availability. Specifically, since the initial attempt to deploy a 100-node cluster in EBS2, the average number of VDs has increased from 20,000 to 100,000 per cluster. Correspondingly, the CPU and the memory consumption have increased by 4.1× and 4.5×, respectively.

Figure 13 illustrates the architecture of Federated BlockManager—our solution to the availability issue in the control plane. Each cluster now has multiple BlockManagers, with a CentralManager dedicated to managing the BlockManagers. Each BlockManager further manages hundreds of VD-level *partitions*, each of which corresponds to a metadata table that only stores the metadata of a small subset of VDs. The mappings from VDs to partitions are static; given a VD, we use hashing algorithms to compute its corresponding partition. The Client is not aware of the partition concept. For a given VD, it can query all the BlockManagers in the cluster to find out the BlockManager in charge, then send all its control plane requests to the BlockManager.

Note that instead of having three nodes (one leader and two standby), we now have only one node in each BlockManager. Upon the failure of a BlockManager, CentralManager redistributes its partitions to other BlockManagers, which then load the metadata tables of the partitions from Pangu into memory, and start to serve the VDs. Since the number of VDs in a partition is relatively small, the loading time is only several hundred milliseconds, without the need to have standby nodes continuously fetching the latest metadata updates for a fast leader switch. To ensure the availability of partition scheduling, the CentralManager consists of three nodes based on the lock service of Pangu.

Having multiple BlockManagers effectively reduces the blast radius of single leader service disruptions, since now each BlockManager only processes the requests of the subset of VDs in the partitions it manages. For the single table issue, instead of creating more BlockManagers, we adopt the partition design for two reasons. First, with partitions, we can make the number of VDs in a single metadata table small. For 100,000 VDs in a cluster, we need to have 1,000 tables to make the blast radius of metadata table failures smaller than 100, while it is not resource-efficient to create 1,000 BlockManagers in 100-node clusters. Second, the concept of multiple BlockManagers is mainly for distributing the workloads to multiple servers, so as to reduce the chances of service disruptions due to CPU and memory resource limits. Note that the CentralManager only manages BlockManager registrations and partition scheduling, and thus is less relevant to system availability.

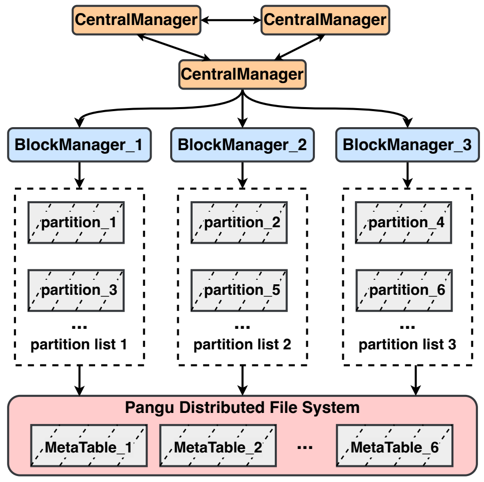


Figure 13: The architecture of Federated BlockManager.

Similar designs for blast radius reduction in the control plane of distributed storage systems can be found in the industry. HDFS Federation [1] is similar to Federated BlockManager, while it does not consider the metadata table failure mode. Each set of NameNode in HDFS Federation persists the metadata to its local disk, and all the requests to a set of NameNodes become unavailable when the data in the local disk is corrupted. AWS Physalia [24] deploys small units called cells, each of which consists of seven nodes deploying Paxos algorithms and serves a group of VDs. Differently, Federated BlockManager adopts a two-level VD management scheme, with each level emphasizing the blast radius of a single node and single table failures, respectively.

4.2 Data Plane: Logical Failure Domain

The data plane of EBS2 constitutes multiple BlockServers, each of which hosts thousands of segments and handles I/O requests of the segments. When a BlockServer crashes, the control plane migrates the segments to other BlockServers in the cluster, such that the I/O services can resume in other BlockServers. However, this mechanism indicates that failures can be cascading among BlockServers. Specifically, if the crash is caused by an error segment (e.g., recall the buggy code case in our production earlier), after the migration, the BlockServer shall resume the requests and crash again. Ironically, optimizing segment scheduling for faster recovery in this case can make more BlockServer crashes, even possibly leading to cluster-wise outages.

We have got several observations based on our deployment experiences. First, the failure typically originates from requests of a single VD or segment, and thus we need to monitor the status of segments instead of BlockServers. Second, the root causes of the failures are mostly due to software errors (e.g., bugs or misconfigurations) as hardware or mechanical failures are unlikely to travel along with the migrated segments. Software-induced failures can be time-consuming to pinpoint the culprit, let alone build automatic tools for recovering. Instead, a more practical way is to proactively reduce the impacts. Third, the cascading failures, if not intervened, can propagate quickly to the whole cluster. As a distributed service, it is acceptable to experience a few BlockServers shutdowns but a not cluster-wise outage.

Based on these observations, we design the logical failure domain. The core idea is to isolate any suspicious segments by grouping them into a small set of BlockServers, to avoid other BlockServers or even the entire cluster being impacted. We first associate each segment with a unique token bucket with a maximum capacity of three. Each migration to a new BlockServer consumes one token, and the token is refilled every 30 minutes. When the token bucket is empty, any subsequent migrations of the segment can only be selected among the three pre-designated BlockServers, referred to as its logical failure domain. The token bucket design does not limit the number of migrations but the range of migrations. When the segment is successfully loaded in a BlockServer and can readily serve I/O requests for several minutes, we lift the failure domain constraints.

The segment-level failure domain can effectively isolate an error segment. However, if there are multiple error segments (e.g., from one VD) or even VDs, the segment-level failure domain is not sufficient to prevent multiple cascading failures from happening at the same time. Our solution is to merge the failure domains into one. Specifically, when there exists more than one segment forming a failure domain, we pick the first failure domain as the global failure domain, so as to ensure there are at most three BlockServers isolated for the cascading failures, without degrading the cluster capacity. Any subsequent segment with an empty token bucket will share the same global failure domain. Recall that we associate each segment with three tokens. As a result, the chances of a normal segment accidentally sharing the same migration path with an error segment is small (note that a cluster is of 100-node size), which effectively reduces false negatives. As such, we can efficiently identify any error segments with small impacts on other VDs.

After deploying the logical failure domain, we have successfully defended our system against several potential outages due to migration-induced cascading failures. Evidence shows that some open-source storage systems also suffer from the same issue from time to time. For example, the HBase community reports several bugs [4–6] that show similar symptoms and lead to system outages. However, their treatments focus on solving the bugs and reducing the blast radius with physical isolation (similar to our first attempts), while the logical failure domain prevents the cascading failure from causing a cluster-wise outage once and for all.

4.3 Lessons Learned

First, with denser SSDs (e.g., QLC NANDs) becoming readily available and the boosting processing ability of CPU or other customized hardware (e.g., DPU), one can expect that a crashed node in a distributed storage system—not just EBS—can impact increasingly more users. As a result, regional failure events would be more frequent and/or severe. The key benefit of Federated Managers in this case is that it enjoys a smaller blast radius without losing the flexibility of a large-scale cluster.

Second, owning a forwarding layer between the users and the underlying service is popular among distributed services, for example, distributed key-value store [2] and cloud storage services [22, 25]. However, this also means the request or certain data structures can be redirected to other destinations upon failures, leading an individual event (e.g., bug or misconfiguration) to become regional or even global. We do not aim at proactively recovering or avoiding these failures because such events, like bugs or human errors, can be unpredictable. Instead, the logical failure domain works in a reactive fashion by confining the suspects among a few controlled servers and does not rely on manual intervention.

5 To Whom the EBS Offloads

Offloading the software stack to specialized hardware for better performance or achieving certain features (e.g., baremetal servers) has been gaining momentum from both the cloud [10,11] and hardware vendors [8,12,13]. Along the evolution of EBS, we have also leveraged FPGA for both frontend BlockClient (i.e., running the customized UDP-based protocol, Solar [36]) and backend BlockServer for accelerating compression/EC in EBS3. In the following subsections, we first demonstrate the motivations behind the two offloading. More importantly, we discuss why the two eventually both dropped FPGA-based solutions but went on different paths—ASIC for BlockClient and ARM CPU for BlockServer.

5.1 Offloading BlockClient

Since EBS2, the frontend BlockClient has become a bottleneck as the backend BlockServers can utilize segmentation for high throughput/IOPS. The BlockClient has been bounded by CPU-heavy tasks, including calculating CRC, encryption and performing per-I/O table lookups. Our stress test reveals it takes 4 CPU cores to saturate a 2×25 Gbps NIC in BlockClient. The newly emerged high-speed network would require a doubled or quadrupled number of cores. More importantly, Elastic Computing Service (ECS, our VM service)—co-located with BlockClient—requires the servers to be bare-metal-ready (i.e., all CPU cores would be allocated to users). Therefore, offloading BlockClient processing to customized hardware is not only recommended but a must.

We initially built an FPGA-based solution for BlockClient and later decided to directly deploy this version in our production systems. This is because, at the time, directly applying the ASIC-based solution requires much longer development cycles. On the other hand, utilizing the ASIC-based approach is also not desirable due to higher power consumption and increased system complexity.

However, after several years of running in production, we discovered that FPGA is actually not the ideal candidate for BlockClient offloading. The major drawback is the instability. Specifically, 37% of data corruption incidents, as identified by CRC mismatches, are directly caused by FPGA-induced errors such as overheating, signal interference, and timing issues. This is because FPGAs are sensitive to environmental conditions and require precise timing, which can be disrupted by various factors like temperature fluctuations and electrical noise. Moreover, FPGA-related issues account for 22% of BlockClient's operational downtime. Typical reasons include hardware malfunctions, software bugs in the FPGA logic, and incompatibility with updated system components. Apart from reliability concerns, the frequency of FPGA is rather limited (e.g., around 200 MHz to 500 MHz), thereby limiting its potential for adapting to high-speed networks.

Therefore, we later move on to adopt the ASIC-based solution. The preliminary deployment of FPGA-based BlockClient considerably saves our time and effort for transitioning to ASIC (i.e., around 12 months between the release of FPGA and ASIC solutions). Compared to FPGA, ASIC-based offloading incurs approximately 5% of the CapEx and around 1/3 of the power consumption. Also, ASICs are optimized at the hardware level for specific tasks, allowing for more efficient use of resources and higher clock cycles. Another key enabler is that the main functionalities of BlockClient, including data movement, data calculation (e.g., CRC and encryption) and network packet processing, are usually stable. Hence, we do not need to periodically redesign the chips. After deployment, field statistics indicate that the failure rate of ASICs is an order of magnitude lower than that of FPGAs.

5.2 Offloading BlockServer

The goal of offloading BlockServer is to reduce costs while maintaining performance. EBS3 introduces data compression in the foreground to reduce traffic and space amplification. However, even with the latency-optimized LZ4 compression algorithm, the compression latency for 16 KiB-sized data blocks remains elevated at 25 μs (25.6% of total write latency) for software-based, and it escalates significantly with larger data blocks. Moreover, to achieve 4,000 MiB/s throughput, at least 8 CPU cores are required, leading to heightened resource contention and diminished performance. Therefore, offloading is necessary to avoid the penalty incurred by software-based compression.

While FPGA-based offloading demonstrates superior performance metrics (see Figure 8), the field deployment has exposed its limitations in terms of sustainability and cost-effectiveness. First, BlockServer faces similar instability FPGA issues. Over the past year, out of every 10,000 deployed production BlockServers, we have documented on average around 150 instances of compression offload failures by FPGA exceptions. Second, we are exploring optimizing compression algorithms tailored to data blocks with varying temperature profiles to minimize storage overhead. For example, using the ZSTD algorithm for separated cold data blocks can further achieve an average 17% space reduction. However, the resource constraints inherent to FPGAs preclude the dynamic adaptation to various compression algorithms. Finally, compared to the scale of BlockClients, the scale of BlockServers is still not large enough to amortize the escalating costs associated with FPGA development.

In light of these considerations, we are reorienting the target of offloading towards server CPUs. This shift is motivated by the advent of multi-core CPUs and specialized computational units integrated within them, which offer superior cost-efficiency while maintaining comparable performance metrics. Noteworthy examples include *Kunpeng 920* ARM CPU [41], and *Yitian 710* ARM CPU [9], all of which are equipped with dedicated units for compression acceleration. The test results show that the average LZ4 compression latency of *Yitian 710* is marginally higher by 1.3 μs in comparison to FPGA-based offload, while 16 ARM cores attain equivalent compression throughput.

Unlike BlockClient, we chose not to use ASIC for BlockServers because of two reasons. First, with no bare-metal requirements, there are no limitations on using CPU cores in BlockServers. Moreover, the BlockServer functionalities (a.k.a., operators) are in an ever-changing fashion. For example, the introduction of new compression and garbage collection algorithms. In this case, applying ASICs may require a complete overhaul from time to time, which can be prohibitively expensive even for the cloud-level scale.

5.3 Field Experience & Lessons

First, FPGA is undoubtedly the first choice for many hardware offloading scenarios due to its high flexibility and competitive performance. We, too, adopted FPGA in both BlockClient and BlockServer as proof-of-concept. However, the frequent errors and high CapEx made us realize that FPGA might not be an ideal acceleration option for large scale storage systems.

Second, ASIC and ARM are both suitable for the compute-to-storage architecture but in a different way. Compute end is cost-sensitive due to its massive scale and has stable operators, such as processing (e.g., encryption) and forwarding, matching the characteristics of ASIC. Storage backend can have frequent upgrades (e.g., improving GC algorithms and optimizing host FTL for ZNS SSDs) and prioritize low interference between tasks. Therefore, the many-core ARM CPU becomes a proper choice.

6 What If?

Evolving EBS has been a long journey. Along the way, it is not surprising we have conceived or even tried out promising ideas that are only later to be proved as impractical. Here, we summarize such discussions as a series of "What If?".

Q1: What if the log-structured design was never adopted?

When developing EBS2, we initially tried to extend the EBS1 with segmentation but later dropped the idea due to high engineering effort. Note that this idea (i.e., in-place update with segmentation), if developed, can meet our requirements, including high performance and space efficiency, for EBS2.

However, this idea still falls short for EBS3. Foreground EC/compression necessitates that a storage system aggregates a sufficient amount of data before the persistence to achieve efficient data reductions. For example, EC(8,3) needs 32 KiB data for one stripe with 4 KiB stripe unit size, and 16 KiB compression units yield higher compression ratios. In §2.2 we have shown the domination of small writes (less than 16 KiB) combined with the need for low write latency prevent us from aggregating 16 KiB data for a single segment. The log-structured design can easily tackle this problem by allowing segments from different VDs to be merged together and flushed to a journal log.

We believe that Ceph [3] faces a similar issue due to the lack of a log-structured layer like BlockServer. To utilize EC in Ceph Block Device and CephFS, with FileStore, users need to set up a cache tier with write-back mode before an erasure-coded pool. With BlueStore, Ceph performs partial writes (i.e., read-modify-write) to an existing stripe without aggregating data, yielding additional network overhead, while EBS3 always writes full EC stripes to Pangu.

Q2: What if we built EBS with open-source software?

At first glance, one might assume that a cloud-level block store could be constructed by using a series of open-source softwares. For example, we can use HBase [2] (i.e., a log-structured distributed key-value store) and HDFS [7] (i.e., a distributed file system) to replace the block layer (i.e., BlockServer and BlockManager) and the file layer (i.e., Pangu).

However, the two designs are markedly different. EBS is a co-design of the block interface, the software, and the hardware. In the block layer, indexing is specifically tailored for the block service. For each segment, the key space is deterministic. Specifically, each segment represents an address space of a 32 GiB segment consisting of 4 KiB blocks, and thus the key space only includes 8 million numbers. This deterministic feature allows for efficient memory allocation for the indexing structure. To achieve low I/O average and tail latency, EBS deploys hardware offloading (i.e., offload compression algorithms to FPGA and ARM CPU) and customized network protocols [36], and decouples non-I/O activities from the critical I/O path, e.g., using individual GCWorker to perform garbage collection and moving index compaction procedures to background threads. In the file layer, Pangu is a high-performance storage system that builds dedicated user-space file systems for high-speed storage devices (i.e., NVMe SSDs), deploys high-speed networks (i.e., RDMA), and incorporates various hardware-offloading technologies [35].

Q3: What if Pangu and EBS were never separated?

The short answer is that such integration would have significantly hindered the development of EBS. Recall that in EBS1, the BlockManagers and BlockServers are integrated with the persistence layer (i.e., ChunkServers and ChunkManagers). This organization becomes increasingly unacceptable with the scaling of our engineering team. First, the interfaces (between the block and chunk layers) grew exceedingly complex—at one point there were nearly 10 sets of persistence APIs in EBS1—in order to support various functionalities and performance optimizations. Maintaining such a complex codebase undoubtedly slows down the development schedule. As an example, around that time, it could take up to 10 months for EBS to release a major upgrade due to delays by software bugs or incompatibility between components.

Decoupling the underlying persistence layer (i.e., Pangu) from EBS and adopting a unified log-structured interface have clearly facilitated the development and ease of communication. In addition, the independent block layer enables rapid segment creation and migration across multiple storage nodes, irrespective of data block locations. The separated architecture also localizes the impact radius of single-point failure to each respective layer. Moreover, this disaggregation also allows us to integrate emerging technologies (e.g., FPGA-based accelerator) and extend Pangu as a general-purpose DFS for other services (e.g., object store [17] and file store [15]).

7 Related Work & Conclusion

Cloud block store is a popular service provided by most cloud vendors [14, 18, 19, 25, 30]. In addition, academia has made great efforts in building and optimizing such a system, such as Salus [40], Ursa [34], Blizzard [37], and LSVD [31]. This paper differs from the above as it not only chronologically revisits the evolutions behind our EBS designs, but also provides a comprehensive summary of lessons we have obtained along the road, including on elasticity, availability, hardware offloads and the failed/alternative attempts.

Acknowledgments

The authors would like to thank our shepherd Prof. George Amvrosiadis and anonymous reviewers for their meticulous reviews, the Pangu and Kuafu team for their tremendous support and Yikang Xu for his contribution in developing EBS2. This research was partly supported by Alibaba ARF program and NSFC(62102424).

References

[1] Apache Hadoop 3.3.6 - HDFS federation. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html.

[2] Apache Hbase. https://hbase.apache.org/.

[3] Erasure code – ceph documentation. https://docs.ceph.com/en/latest/rados/operations/erasure-code.

[4] HBASE-14598. https://issues.apache.org/jira/browse/HBASE-14598.

[5] HBASE-22072. https://issues.apache.org/jira/browse/HBASE-22072.

[6] HBASE-22862. https://issues.apache.org/jira/browse/HBASE-22862.

[7] HDFS architecture guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[8] The fungible DPU™: A new category of microprocessor for the data-centric era : Hot chips 2020. In *Proc. of IEEE HCS*, 2020.

[9] Alibaba Cloud unveils new server chips to optimize cloud computing services. https://www.alibabacloud.com/blog/598159?spm=a3c0i.23458820.2359477120.113.66117d3fm03t9b, 2021.

[10] AWS Nitro system. https://aws.amazon.com/ec2/nitro/, 2021.

[11] A detailed explanation about Alibaba Cloud CIPU. https://www.alibabacloud.com/blog/599183?spm=a3c0i.23458820.2359477120.3.76806e9bESi3SD, 2021.

[12] Intel Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html, 2021.

[13] NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2021.

[14] About Google Persistent Disk. https://cloud.google.com/compute/docs/disks, 2023.

[15] Alibaba Cloud Apsara File Storage NAS. https://www.alibabacloud.com/help/en/nas, 2023.

[16] Alibaba Cloud Elastic Block Storage devices. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/block-storage-overview-elastic-block-storage-devices, 2023.

[17] Alibaba Cloud Object Storage Service. https://www.alibabacloud.com/help/en/object-storage-service, 2023.

[18] Amazon Elastic Block Store. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html, 2023.

[19] Introduction to Azure managed disks. https://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview, 2023.

[20] M. Ajdari, W. Lee, P. Park, J. Kim, and J. Kim. FIDR: A scalable storage system for fine-grain inline data reduction with efficient memory handling. In *Proc. of IEEE/ACM MICRO*, 2019.

[21] J. Axboe. Flexible i/o tester. https://github.com/axboe/fio, 2017.

[22] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proc. of USENIX OSDI*, 2010.

[23] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, et al. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proc. of ACM SOSP*, 2021.

[24] M. Brooker, T. Chen, and F. Ping. Millions of tiny databases. In *Proc. of USENIX NSDI*, 2020.

[25] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proc. of ACM SOSP*, 2011.

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[27] S. Deorowicz. Silesia compression corpus. https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia, 2014.

[28] P. DeSantis. Peter desantis keynote recap - AWS re:Invent 2022. https://caylent.com/blog/peter-de-santis-keynote-recap-aws-re-invent-2022, 2022.

[29] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, et al. When cloud storage meets RDMA. In *Proc. of USENIX NSDI*, 2021.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of ACM SOSP*, 2003.

[31] M. H. Hajkazemi, V. Aschenbrenner, M. Abdi, E. U. Kaynar, A. Mossayebzadeh, O. Krieger, and P. Desnoyers. Beating the I/O bottleneck: a case for log-structured virtual disks. In *Proc. of ACM EuroSys*, 2022.

[32] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *Proc. of USENIX OSDI*, 2022.

[33] A. Kopytov. Sysbench. https://github.com/akopytov/sysbench, 2021.

[34] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proc. of ACM EuroSys*, 2019.

[35] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, et al. More than capacity: Performance-oriented evolution of Pangu in Alibaba. In *Proc. of USENIX FAST*, 2023.

[36] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proc. of ACM SIGCOMM*, 2022.

[37] J. Mickens, E. B. Nightingale, J. Elson, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, O. Khan, and K. Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *Proc. of USENIX NSDI*, 2014.

[38] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. of ACM SOSP*, 1991.

[39] Q. Wang, J. Li, P. P. C. Lee, T. Ouyang, C. Shi, and L. Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *Proc. of USENIX FAST*, 2022.

[40] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the salus scalable block store. In *Proc. of USENIX NSDI*, 2013.

[41] J. Xia, C. Cheng, X. Zhou, Y. Hu, and P. Chun. Kunpeng 920: The first 7-nm chiplet-based 64-core ARM SoC for cloud services. *Proc. of IEEE Micro*, 2021.

[42] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, et al. FlashShare: Punching through server storage stack from kernel to firmware for Ultra-Low latency SSDs. In *Proc. of USENIX OSDI*, 2018.

[43] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, et al. FPGA-accelerated compactions for LSM-based key-value store. In *Proc. of USENIX FAST*, 2020.

[44] X. Zhang, X. Zheng, Z. Wang, H. Yang, Y. Shen, and X. Long. High-density multi-tenant bare-metal cloud. In *Proc. of ACM ASPLOS*, 2020.

[45] B. Zhu, Y. Chen, Q. Wang, Y. Lu, and J. Shu. Octopus+: An RDMA-enabled distributed persistent memory file system. *ACM Trans. on Storage*, 17(3):1–25, 2021.

[46] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, et al. Deploying user-space TCP at cloud scale with LUNA. In *Proc. of USENIX ATC*, 2023.