CAPI-Flash Accelerated Persistent Read Cache for Apache Cassandra

Bedri Sendir*†, Madhusudhan Govindaraju*, Rei Odaira†, Peter Hofstee†‡
SUNY Binghamton*, IBM Research†, TU Delft‡
{bsendir1, mgovinda}@binghamton.edu, {rodaira, hofstee}@us.ibm.com

Abstract—**In real-world NoSQL deployments, users have to trade off CPU, memory, I/O bandwidth and storage space to achieve the required performance and efficiency goals. Data compression is a vital component to improve storage space efficiency, but reading compressed data increases response time. Therefore, compressed data stores rely heavily on using the memory as a cache to speed up read operations. However, as large DRAM capacity is expensive, NoSQL databases have become costly to deploy and hard to scale. In our work, we present a persistent caching mechanism for Apache Cassandra on a high-throughput, low-latency FPGA-based NVMe Flash accelerator (CAPI-Flash), replacing Cassandra's in-memory cache. Because flash is dramatically less expensive per byte than DRAM, our caching mechanism provides Apache Cassandra with access to a large caching layer at lower cost. The experimental results show that for read-intensive workloads, our caching layer provides up to 85% improved throughput and also reduces CPU usage by 25% compared to default Cassandra.**

I. Introduction

NoSQL data stores are designed to handle large volumes of data coming in at high velocity. They are increasingly used in Big Data applications because they do not require data to be stored in a structured format and also provide high throughput with low response latency. Apache Cassandra is a popular example of NoSQL technology that has gained significant traction in the industry and academia [10]. It is used in production in hundreds of companies including Apple, Netflix and Baidu [2].

Data compression is one of the techniques used to maximize the storage capacity by reducing the volume of data on the storage and is crucial for storage efficiency. However, reading data from storage and decompressing it causes long response times as well as high CPU consumption [22] [24]. Therefore, NoSQL databases rely on data cached in DRAM to keep response latencies low [33]. Cassandra typically uses cached data in DRAM in compressed or uncompressed format to improve the performance. However, if the working set size grows larger than the DRAM capacity, Cassandra becomes costly to deploy and hard to scale due to the DRAM cost. It is critical to address this scaling bottleneck while keeping the deployment cost low and yet achieve the performance objectives.

Emerging technologies in the storage ecosystem such as Intel's SPDK [13] and IBM's CAPI-Flash [8] bypass the operating system I/O stack. They allow access to flash with low CPU overhead and low latency. For example, CAPI-Flash [8] is an FPGA-based NVMe Flash accelerator. It is based on the POWER8 Coherent Accelerator Processor Interface (CAPI), which enables cache-coherent hardware accelerators with address translation capability. CAPI-Flash enables user-level access to flash without operating system overhead, eliminating 97% of the code path.

Our goal in this work is to leverage the CAPI-Flash system to design and implement a high speed persistent caching layer for Cassandra, that accesses flash with low latency and low CPU overhead. Our caching mechanism replaces Cassandra's default Chunk Cache mechanism and utilizes CAPI-Flash to cache uncompressed chunks of the Cassandra's data files.

The contributions of this paper are as follows:

- We evaluate and characterize the read performance and resource utilization of CAPI-Flash and NVMe SSD.
- We present the design and implementation of the CAPI-Flash based persistent caching layer.
- We present a comprehensive evaluation of our caching layer compared to the default mechanism in Cassandra and show results for various use cases.

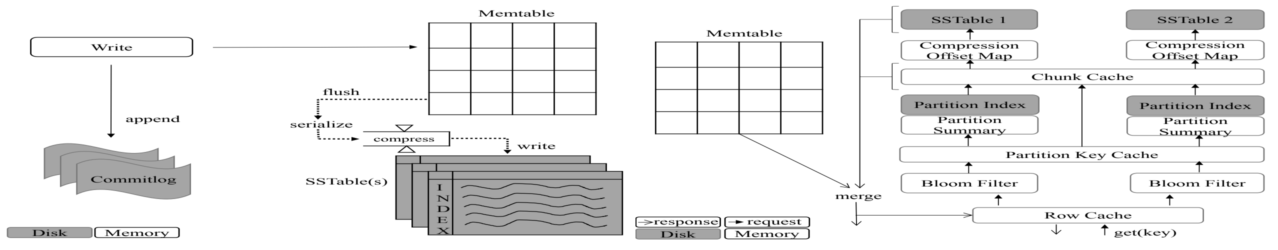
II. Background

A. Apache Cassandra

Apache Cassandra [25] is an open source distributed NoSQL database initially developed at Facebook. It is designed to manage large volumes of data while providing high availability with no single points of failure.

Even though Cassandra's architecture has significantly evolved after its inception, it was initially based on Bigtable's data model [17]. In this data model, a column is the smallest component and it consists of the key or column name, value, and a time stamp for conflict resolution. Columns that share the same key form a row. Rows are sorted in key order and organized into column families.

As a storage engine, Cassandra uses a structure similar to Log Structured Merge (LSM) Trees [28]. LSM-trees are used in many popular NoSQL systems such as HBase, RocksDB, MongoDB (WiredTiger) and BigTable. LSM-trees keep the data sorted in files and avoid in-place updates. Cassandra's storage engine consists of three main components: – (1) *Memtable*: an in-memory write-back cache that maps the row by its key and is maintained on a per column-family basis; (2) Sorted String Table (*SSTable*): persistent, immutable files that keep key-value pairs in sorted order; and (3) *Commitlog*: append-only files that maintain a history of changes in the database to provide fault tolerance.



(a) Write Path (b) Read Path

Fig. 1: **Figure** 1a shows path of a write operation and **Figure** 1b shows the path of a read operation in Cassandra.

1) Write and Read Path: **Figure** 1a shows the write path of Apache Cassandra. When a write or update operation occurs, data is applied to the in-memory data structure (Memtable), which acts as a write-back cache. In order to provide durability, data is simultaneously appended to on-disk Commitlog files. At a configurable threshold, Memtables are flushed to the disk. At the time of flush, data is first serialized and then sequentially written as SSTable files. Contents of SSTable files are immutable and not modified after the Memtable is flushed. Additionally, a partition index file is generated which maps the location of each key to its offset in the SSTable file. As Cassandra's data files are immutable, data that is overwritten or deleted will continue to reside in the SSTable files. Cassandra periodically runs compaction to merge rows and columns, evict deleted rows and consolidate SSTables.

Compression is used in almost all databases in order to maximize the storage capacity, reduce the volume of the disk I/O, and improve the page cache residency. If compression is enabled, Cassandra buffers serialized data into fixed-size chunks, compresses them, and writes them into the disk as SSTable files. In the compression-enabled mode, another in-memory data structure *Compression Offset Map* is generated that maps the offsets stored in the partition index file to the locations of the compressed chunks. Compression reduces the write footprint to the disk at the cost of CPU cycles.

On the read path (**Figure** 1b), reading the requested row data involves merging results from a Memtable and possibly from multiple SSTables. Cassandra maintains an in-memory cache (Row Cache) to serve frequently accessed data. The Row Cache holds the combined row data and is not *write through*. If the row is modified, the corresponding entry is invalidated. As Cassandra does not update data in-place, portions of the row might reside in multiple SSTables. On a Row Cache miss, to accelerate the key lookup, Cassandra checks multiple on-disk and in-memory data structures to avoid costly disk seek operations to multiple partition index files. The Cassandra indexing mechanism is not aware of whether the SSTable data is compressed or not. The partition index maps the SSTable files as if they are uncompressed. After getting an estimate from the partition summary, index lookup can be performed by seeking to a location, followed by a sequential read to find the offset of the queried row in SSTable.

If compression is enabled, Cassandra first searches the Chunk Cache for the uncompressed chunk data. We explain the Chunk Cache mechanism in detail in **Section** II-A2. If there is a miss in the Chunk Cache, data is read from SSTable. If the data is compressed, Cassandra decompresses the entire chunk and populates the Chunk Cache.

2) Existing Caching Mechanism for Read Operations: Other than the Row Cache explained in the previous section, Cassandra mainly relies on the OS page cache and its in-memory Chunk Cache mechanism to speed up read operations. Cassandra utilizes the Chunk Cache mechanism to store an uncompressed version of the frequently accessed compressed chunks. It is a map from *(file name + offset)* to the corresponding chunk. Compression is used to improve page cache residency of the data at the expense of decompressing data at the time of read and compaction. It reduces the size of the data on disk and also typically reduces the amount of I/O required to fetch the data. To avoid processing overhead when the data is frequently accessed, Cassandra utilizes the Chunk Cache mechanism to store uncompressed version of the frequently accessed chunks. Also, for the uncompressed files accessed through the *read()* system call and its variants (non memory-mapped I/O), the Chunk Cache is used to avoid extra copying from kernel to user space. A request to the Chunk Cache layer consists of the following information:

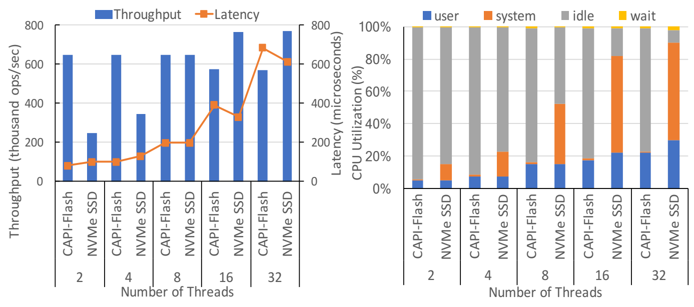
(1) File name of the SSTable and the start offset of the requested row to read within the file; (2) Chunk size of the file: Compression is configured by column family basis and the SSTable file inherits the compression properties of the associated column family. (3) Estimated row size: for each generated SSTable file, Cassandra maintains an estimated row size. In the Chunk Cache, this information is used to set the chunk size of the uncompressed files. However, the record size information has no effect in the default mode when the Chunk Cache is used for caching compressed files. A query to chunk cache returns the entire chunk as a result. A row might extend across multiple chunks and therefore the Chunk Cache can be queried multiple times to satisfy a read request.

B. Coherent Accelerator Processor Interface

The Coherent Accelerator Processor Interface(CAPI) [32] provides a low-latency, memory-coherent path between the POWER8 core and hardware accelerators. CAPI handles virtual to physical memory translations and allows a CAPI-attached device to perform Direct Memory Access (DMA) to application memory without requiring calls to a device driver or underlying operating system kernel. A CAPI adapter attaches to PCIe slots and uses PCIe Gen3 as the underlying transport.

TABLE I: Hardware/Software Stack

Hardware Configuration	**CPU/Memory:** POWER8 822LC, 20 cores (8SMT), 3491MHz, 512GB RAM, CPU governor `performance` **Storage:** 1.5 TB HGST Ultrastar SN100 NVMe SSD (PCIe3 x 8), readahead set to 0 **CAPI-Flash:** 1.92 TB CAPI NVMe Flash Accelerator Adapter (PCIe3 x 8) using Samsung PM963 M.2 NVMe SSD as a backend flash device
OS&Kernel	Ubuntu 17.04 ppc64le (4.10.0-38) **Kernel Page Size:** 4KB, 64KB
Cassandra	Apache Cassandra 3.11 patched with [CASSANDRA-13897] lz4-java 1.5.0 (custom build)
Docker	Docker version 17.09.0-ce
Java	OpenJDK 1.8.0-internal
YCSB	YCSB 0.12.0



(a) Throughput vs. Latency

(b) CPU Utilization

Fig. 2: Micro benchmark results showing 60 seconds of 4KB random read operations on CAPI-Flash and NVMe SSD.

CAPI-Flash: CAPI-Flash [8] provides a software and hardware stack to exploit the high-throughput and low-latency path between the POWER8 processor and the flash storage. Each CAPI-NVMe Flash adapter contains an FPGA interfacing to CAPI and two 960GB flash drives. The CAPI-Flash software environment provides the Capiblock API, which allows user space applications to directly perform I/O to the flash by offloading the kernel I/O driver functionality to the accelerator hardware residing on the adapter. Through the Capiblock API, applications can issue synchronous or asynchronous read/write requests to the specific logical blocks on the flash with block granularity (multiples of 4KB).

In **Figure** 2, we show performance comparison of 4KB random read operations on CAPI-Flash and NVMe SSD accessed through traditional I/O. For these micro benchmarks, we use the hardware configuration defined in **Table** I. For both NVMe SSD and CAPI-Flash, we set I/O depth to 64 and run benchmarks on two cores. We measure the performance (**Figure** 2a) by changing the number of threads issuing asynchronous I/O and collect the CPU utilization (**Figure** 2b) on two cores. For benchmarking NVMe SSD, we use `fio` [7] benchmarking suite and perform direct I/O on the raw device with kernel asynchronous I/O library. We use `fio` counterpart (`blockio`) for benchmarking CAPI-Flash and distribute threads evenly on CAPI-Flash's flash drives. The maximum read bandwidth of the flash drives of CAPI-Flash is around 2.5 GB/sec and the NVMe SSD is around 3.1 GB/sec. Because the underlying NVMe flash drives are different, CAPI-Flash has lower maximum bandwidth and IOPS. However, CAPI-Flash provides lower CPU utilization/latency at the parity IOPS.

Figures 2a and 2b show that CAPI-Flash can saturate its maximum bandwidth with a lower number of threads while consuming only 5% of the 2 assigned cores. We also observe that with larger number of threads, the CAPI-Flash performance degrades. This means the concurrency on the CAPI-Flash needs to be controlled in order to not overwhelm the device. Using 8 threads, NVMe SSD shows similar performance to CAPI-Flash (650k ops/sec) but consumes 10x more CPU compared to a 2-thread CAPI-Flash setting. NVMe SSD reaches its maximum bandwidth with 16 threads using 82% of the assigned cores.

III. Design of Accelerated Persistent Read Cache

As described in **Section** II-B, CAPI-Flash provides a raw block I/O interface for reading and writing to the physical address space on flash. We designed and implemented a CAPI-Flash-based read cache for Apache Cassandra, called CAPI Chunk Cache, by taking advantage of its high-throughput, low-latency path between flash and the processor.

Figure 3 shows the components of our caching mechanism. The CAPI Chunk Cache stores uncompressed chunks of SSTable files. It consists of management components in memory and chunks in CAPI-Flash. In memory, keys are generated by the file name and the start offset of the chunks. Values simply act as pointers to the physical block addresses on the CAPI-Flash drives. A value also contains a version-based lock and a temporary reference to a buffer object, which are not depicted in the figure. The keys (file name and offset) and the pointers are periodically written to CAPI-Flash with a checkpoint operation to provide persistence. When a request misses in the CAPI Chunk Cache, the data is read from SSTables and decompressed if necessary. In order to avoid increasing the read latency, the data retrieved from SSTables is immediately returned to the requester and is also added to a queue. Then, the contents of the queue are written to CAPI-Flash in the background, and the CAPI Chunk Cache is populated. In order to support a wide range of use cases, we also implemented a memory-based caching layer to cache reads from CAPI Chunk Cache, but we omit its details because it was not used in our experiments. We explain the `put` and `get` operations in detail in the **Section** III-A.

A. Internals

We use a high-performance in-memory caching library called Caffeine [1] in the in-memory part of the CAPI Chunk Cache. It is used in production in many NoSQL databases such as Cassandra and Neo4J [11]. Caffeine is similar to a map data structure. However, the main difference is that Caffeine evicts entries automatically in order to constrain the capacity to a user-defined value. Caffeine evicts the entries that have not been used recently or very often using the built-in *Window TinyLfu* [23] algorithm.

After the data is retrieved from SSTables, it is placed in a two-stage queue for further processing. In the first stage, the data is dequeued, and then a pointer object is generated

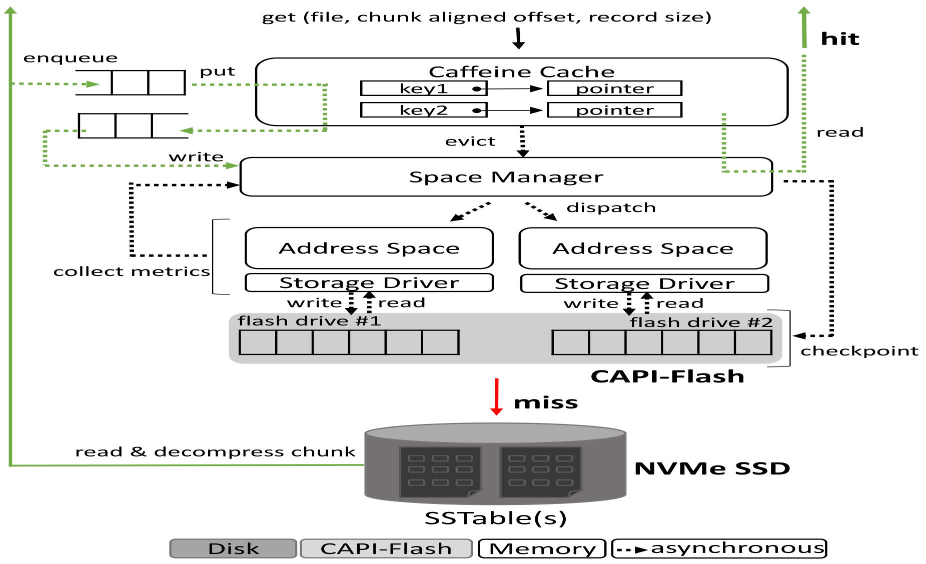


Fig. 3: Design of CAPI-Flash backed caching mechanism

that encapsulates the buffer to be written to CAPI-Flash. This pointer object is inserted into the Caffeine cache, and a reference to this object is added to the second queue. The entries placed in the second queue are present both in cache and the queue until they are written to the flash. When the reference is dequeued, the buffer it points to is written to flash. This two-stage mechanism is designed to deal with possible delays in writing data to the flash. Such delays can occur during allocation in CAPI-Flash when it runs of out of space or can be caused by a heavy read/write load. Placing entries in the Caffeine cache before writing to the flash allows serving these entries from memory while space allocation and writing to the flash are in process. It also allows the eviction algorithm to warm up for these entries. In order to avoid out-of-memory situations, we limit the total buffer memory used by the entries stored the queues. If the keys of temporarily placed buffers are not popular, they can be removed by the eviction before they are written to flash. In this case, we simply free the buffer without writing it to the flash. As writing to flash degrades the read performance, filtering unpopular keys by delaying them in the queue improves the performance and the flash life time as flash has limited write durability.

We split the management of the physical address space, load distribution, maintenance, and storage operations into separate modules. As mentioned in **Section** II-B, our CAPI-Flash device contains two separate flash drives. A machine can have multiple CAPI-Flash devices, and each flash drive contained in the device has its own physical address space. The throughput is maximized when the read/write load is distributed among the flash drives. Therefore, our design considers multiple physical address spaces.

For each flash drive, the *Space Manager* initializes *Address Space* instances with a user defined capacity. Therefore, it limits the amount of CAPI-Flash space that will be used by the application. *Address Space* instances are responsible for free/used space management and collecting metrics regarding the read/write operations that are performed. Each *Address Space* maintains two data structures: (1) a main free list that stores the free blocks that are ready to be allocated and (2) a temporary free list that stores the blocks of evicted entries in between checkpoints. We return the free blocks in the temporary free list to the main free list only after a checkpoint operation finishes. We explain the purpose of these data structures in **Section** III-D.

The *Space Manager* constructs a single, contiguous address space based on a user defined configuration. The *Space Manager* is responsible for dispatching read/write requests or eviction events to the corresponding address spaces. It periodically collects total used space and one minute moving average of reads performed on each *Address Space*. The *Space Manager* is also responsible for load balancing among the *Address Space*s. Depending on the popularity of the chunks stored in *Address Space*s, the read/write load might vary among the *Address Space*s. Therefore, load balancing is required to get optimal performance from CAPI-Flash. For load balancing, the *Space Manager* first tries to dispatch entries according to the read/write metrics collected from the *Address Space*s. If the read/write load is evenly distributed, the *Space Manager* dispatches the entry according to the available space on the *Address Space*s. However, there are cases when a significant imbalance happens between these *Address Space*s. For example, this can occur if the popularity of a chunk changes or if there is a bulk deletion operation. In these cases, the load can shift from one flash drive to another. In order to balance the load, the *Space Manager* periodically spawns a maintenance task to move hottest chunks from the heavily loaded *Address Space* to the lightly loaded *Address Space*.

B. Put and Get operations

On put operations, the *Space Manager* first dispatches the request to an *Address Space*. After dispatching, the corresponding *Address Space* allocates blocks on the physical address space. If there is not enough space on the target *Address Space*, the *Space Manager* blocks until there is enough space for allocation and then retries dispatching. After a successful allocation, data is sent to the corresponding *Storage Driver*. We implement a lightweight storage driver that performs asynchronous read/write to CAPI-Flash and is maintained per *Address Space*. *Storage Driver*s process the results of the asynchronous requests and controls parallelism on the CAPI-Flash layer. Upon successful completion of the write request, the corresponding *Storage Driver* updates the pointer with the start offset of the record on CAPI-Flash and frees the temporary buffer residing in the pointer object.

On a get operation, the CAPI Chunk Cache aligns the offset to the nearest chunk's offset and then searches the Caffeine cache. Cassandra's SSTables are sequentially written and there is no padding in between the records to align them to block or chunk start offsets. Therefore, a record can overflow to other chunks. Returning an incomplete record will cause Cassandra to query the SSTables, and therefore the Chunk Cache layer, repetitively. This will increase the read latency. If the record overflows to the next chunk, we tackle this issue by internally querying the CAPI Chunk Cache for the next chunk. If the next chunk is found in the CAPI Chunk Cache, the *Space Manager* takes its position and the record size parameters into consideration and generate a large buffer. To fill up the buffer from one or more chunks the *Space Manager* issues two asynchronous read commands to fill portions of the buffer. Therefore, we minimize the number of queries to the Chunk Cache and buffer allocation overhead. For each operation that reads the pointer object, the CAPI Chunk Cache applies lightweight version-based locking. During reads, if the entry gets evicted or its mapping changes due to load balancing, we revert to reading from SSTables.

C. Eviction

Eviction from the CAPI Chunk Cache happens under two conditions: (1) Explicit eviction, when an SSTable is deleted due to compaction or if the column family that maintains that particular SSTable has been removed. In these cases we explicitly call eviction on those particular SSTable's entries. The physical blocks used by the evicted cache entries are directly appended to the free list. As a result, tasks that are waiting for allocation can proceed; (2) weight based eviction, if the predefined cache size has been exceeded. In this case, the Caffeine cache determines victim entries. As we protect mappings for each entry with a version-based lock, during eviction we acquire a write lock on the entry that changes the version number. Therefore, if there are any requests pending to read from this entry, it will gracefully fall back to reading from SSTables. Freed physical blocks from weight-based eviction are placed into a temporary free list data structure and later returned to the main free list for further reuse. Note that the eviction process manipulates only in-memory data structures and does not modify any blocks on CAPI-Flash.

D. Checkpoint and Recovery

As warming up cache contents from scratch can be an expensive operation, we implement a periodic checkpointing operation to keep the cache contents valid even after Cassandra shuts down and restarts. It persists the contents of our cache to a designated space on CAPI-Flash. Checkpoints run in the background and provide a consistent view of cache entries. As discussed in Section III-A, a major portion of our caching mechanism is already stored in non-volatile flash. For checkpoints, we only need to store the in-memory portion, specifically, the start block address of the chunk, SSTable *id*, and its corresponding offset in the SSTable. However, as the eviction process only updates in-memory data structures, we must make sure to provide correct mappings for the cache entries. If an entry gets evicted in between checkpoints and if the start block address of the evicted entry is reused to store another chunk, then the entry pointing to that start address from the last checkpoint will become corrupted. We tackle this issue by delaying reusing the start block addresses of the entries evicted by the weight-based eviction until the next checkpoint. We use the temporary free list for this purpose. Note that the explicit eviction does not need to use the temporary free list. As the explicit eviction happens only when the SSTables are deleted, the checkpoint recovery process can validate entries by checking whether the corresponding SSTables exist or not.

At the start of each checkpoint, the *Space Manager* first renames the current temporary free list. We refer to this list as a *retired* list. Then, the *Space Manager* creates a fresh temporary free list. If the cache locality is poor, weight-based cache evictions will happen frequently. Therefore if the main free list runs out of space, new writes to the cache will be rejected until the next checkpoint finishes.

During checkpoint, the *Space Manager* serializes contents of the Caffeine Cache into a buffer that is written to a designated checkpoint space in the CAPI-Flash address spaces. In our design, we currently consider fixed sized chunks throughout the database. Checkpoints traverse an in-memory cache data structure, serialize cache contents into compact format and write it to CAPI-Flash. In each *Address Space*, we use a designated area for checkpoints and each checkpoint writes to a different designated area. After we successfully write the checkpoint data, the *Space Manager* moves the contents of the *retired* list to the main free list. This approach allows us to have a consistent view of the SSTables.

While reconstructing the cache after a graceful or ungraceful shutdown, the *Space Manager* first compares timestamps from the *Address Space*s and determine the most recent checkpoint data. The *Space Manager* sequentially reads the serialized cache contents and populate the Caffeine Cache. The checkpoints are lightweight as they only have a 12 byte overhead per chunk.

IV. Performance Results

A. Experimental Setup

In this section we evaluate the performance of the CAPI Chunk Cache with various workloads using YCSB [18].

TABLE II: Workload/Database Parameters

Dataset	**Total Number of Records:** 5 million **Record:** 10 fields - 6400 bytes each **Compression Chunk Length:** 64KB **Total Size(approx.):** 320GB(170G compressed)
DB Cores	5 cores (8 SMT)
DB Memory	**Cassandra JVM Heap Size:** 16GB **Docker Memory Size:** 32GB, 64GB
DB Parallelism	**concurrent_{readers | writers}:** 128
YCSB Parallelism	**Total YCSB Instances:** 10 **Client Threads per Instance:** 10
YCSB Read/Update Mix	100% read , 80% read - 20% update
YCSB Workload Duration	**Warm-up:** 600 seconds **Execution:** 600 seconds
YCSB Key Selection	**Request Distribution:** hotspot **Hot Set Fraction:** 30% (approximately 96GB) **Hot Set Operation Fraction:** 80%

Tables I and II show the hardware/software stack and workload parameters used in our experiments.[1] We ran the Cassandra server and the benchmark clients on the same machine. We ran Cassandra on five dedicated cores in a separate NUMA node, using a Docker container with `cgroups` to control memory and to provide isolation from the benchmark processes. Container memory size was limited to 32GB and 64GB. In order to avoid network or file system overhead, we mapped Cassandra's data folders to the `XFS` filesystem running on the host and configured the container to use the host's network stack. We allocated 16GB of JVM heap to Cassandra and disabled the Commitlog. As we configure Cassandra to store its Memtables on the JVM heap, approximately the remainder of the memory available in the container is used for Linux page cache.

The Yahoo! Cloud Serving Benchmark(YCSB) [18] is an open source benchmarking suite for evaluating performance of key-value and cloud serving stores. It supports many popular database systems like MongoDB, HBase and Cassandra. A YCSB workload consists of load and transactional phases.

The load phase inserts generated data into the database, and defines the dataset properties such as the number of fields, field length, and number of records. We generated a dataset larger than available memory sizes and select a 64KB record size for fair comparison. A smaller record size can favor CAPI Chunk Cache too much because reads from SSTable files waste the I/O bandwidth. YCSB by default generates random-number fields, which are not compressible and are unrealistic. We modified YCSB to generate compressible fields. Using `lz4` compression with Cassandra's default chunk size (64KB), we achieved 47% reduction in size. For reproducibility, we compacted the dataset into one SSTable. We ran the load phase just once and kept using the same dataset. In each run, we start with a pre-populated SSTable and drop the page cache.

The transactional phase is a mixture of read, insert, update, and scan operations. YCSB allows users to change the request distribution used to select keys to operate on in the transactional phase. As our goal is to evaluate our caching mechanism, there is a need to define the hot set size and the access frequency to the hot set. For this reason, we changed the request distribution property to hotspot distribution, to define the hot set. We note that in 64GB memory setting, a huge portion of the hot set can fit in memory when compressed.

We observe that running a single YCSB client instance with many threads increases the average latency and slows down the benchmarking process. Therefore, we ran multiple instances in parallel on 10 cores. After all instances are completed, we accumulate the results from all of them. In real world NoSQL deployments, users typically have certain performance requirements and try to optimize the resource efficiency. Resource efficiency [22] is optimized under latency and throughput constraints. Therefore, we configured YCSB to throttle the target number of operations per second and measured the average read/update latency for various settings.

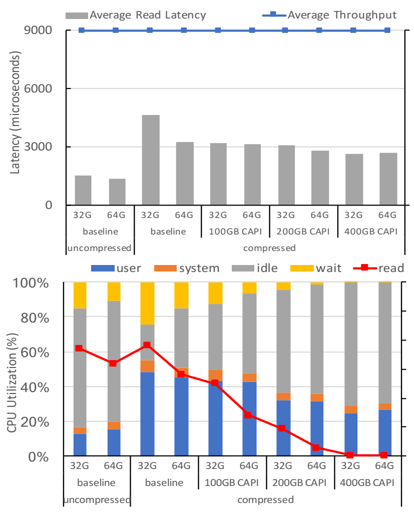
Unless the DIRECT_IO flag is enabled, file I/O operations are performed through the Linux page cache. Therefore, the kernel page size defines the granularity of storage device operations. By default, POWER8 (ppc64le) systems use a 64KB page size and a typical x86 based system uses a 4KB page size. We use a configurable kernel page feature on POWER8 and use both 64KB and 4KB kernel page sizes for our evaluation. In our workload, the uncompressed version of the hot set data is too large to benefit from Cassandra's default Chunk Cache, so we disabled it. As a result, the OS page cache was the main mechanism to cache records in memory in our experiments. By varying the memory size of the Docker container, we controlled the approximate size of the page cache used by the application.

B. Read/Write Mix Workloads

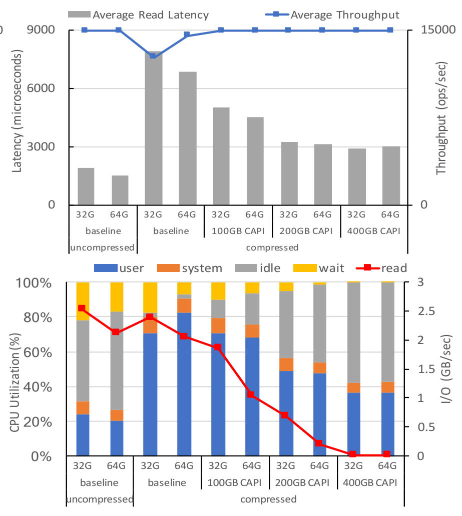
In **Figures** 4, 5, 6 and 7, we show the performance and resource utilization in various settings. We first ran a warm-up phase with **100% read** workload. Then, we started the execution phase by setting a target throughput among the YCSB instances and collected measurements during the execution phase. In the `baseline` settings, we used both compressed and uncompressed datasets. The uncompressed baseline (`baseline-uncompressed`) increases storage demand, so it is not typical in deployment, but we show its results for reference. For the CAPI Chunk Cache, we used the compressed dataset and varied the cache size (100GB, 200GB, 400GB). In the first row of each figure, we show the average latency (first Y-axis) and the average throughput (second Y-axis) for varying target throughput rates per second. In the second row of figures, we show CPU utilization (first Y-axis) and read bandwidth usage of NVMe SSD (second Y-axis).

Overall, the results show that in most cases instead of increasing memory from 32GB to 64GB, users can use 100GB caching space on the CAPI Chunk Cache to get comparable or better performance. When comparing the latency and throughput for the 64GB memory compressed baseline to 100GB CAPI Chunk Cache with 32GB memory, we see improvement in nearly all cases. For example, we show 27% and 48%

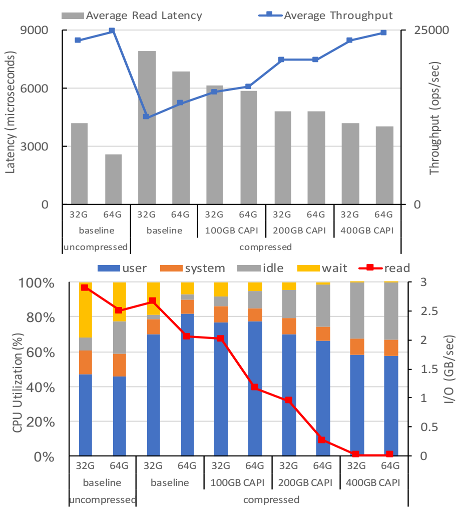
[1]Source code of CAPI Chunk Cache and test scripts are available at https://github.com/bedrisendir/capi-chunkcache



(a) `target=10000 ops/sec`

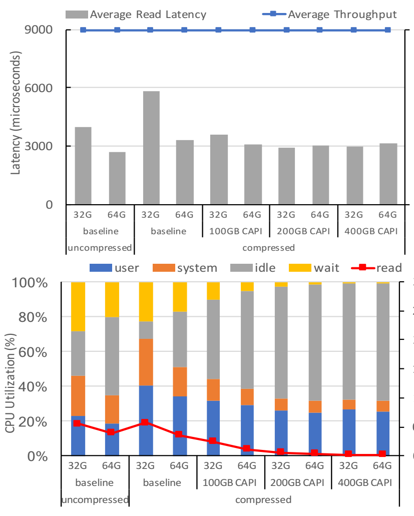


(b) `target=15000 ops/sec`

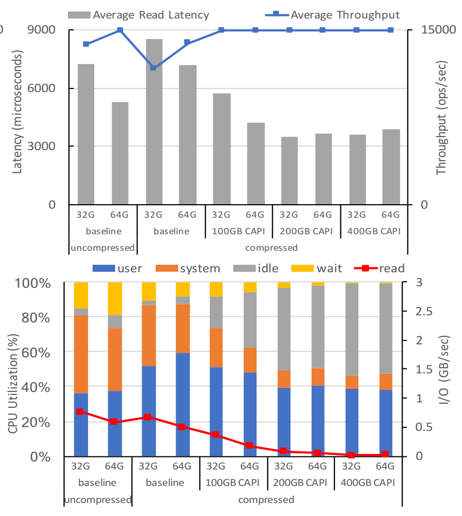


(c) `target=25000 ops/sec`

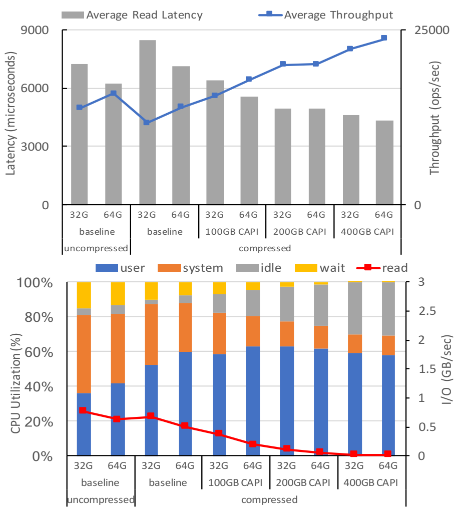
Fig. 4: Running **100% read** workload using the parameters defined in **Table** II and **64KB** kernel pages.



(a) `target=10000 ops/sec`



(b) `target=15000 ops/sec`



(c) `target=25000 ops/sec`

Fig. 5: Running **100% read** workload using the parameters defined in **Table** II and **4KB** kernel pages.

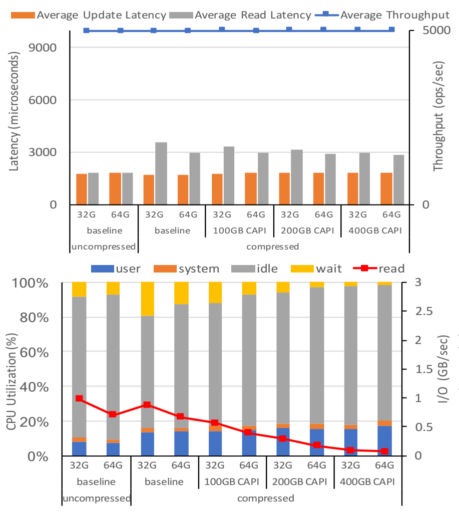
smaller read latency in **Figures** 4b and 7b, respectively. The only exceptions are in **Figures** 5a and 6a, where the target throughput is low, and even there the difference is modest. As we increase the caching space to 200GB and 400GB, CAPI Chunk Cache keeps improving the performance while reducing the CPU utilization.

In **Figures** 5 and 7, because of the increased number of I/O operations with 4KB pages, system CPU cycles are significantly increased. Because of this CPU bottleneck Cassandra cannot drive enough I/O operations on NVMe SSD. However, we were able to resolve the bottleneck by adding the CAPI Chunk Cache.

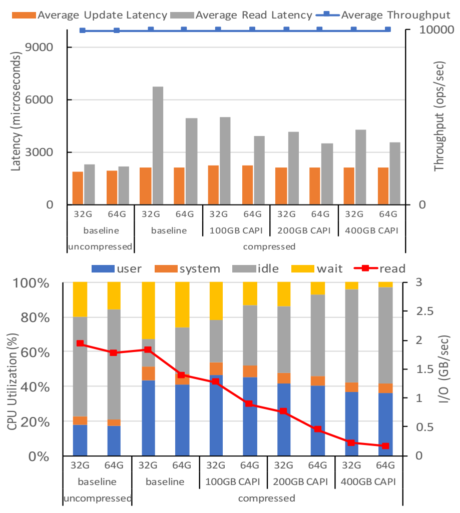
With large target throughput requirements, the 64GB compressed baseline could not reach the required throughput in all cases. By adding 100GB of CAPI Chunk Cache, we can improve throughput nearly in all cases. Increasing the CAPI Chunk Cache size provides more performance and resource utilization also benefits. For example, when we increase the size to 400GB and reduce 32GB of memory, we get up to 85% improvement in throughput while reducing the CPU utilization by 25% (**Figure** 4c).

In **Figures** 6 and 7, additional SSTables were generated as a result of update operations. As writing data files also utilize page cache to buffer the writes, it will cause page cache thrashing. Therefore, the `Baseline` settings aren't able to benefit from page cache as much as in the **100% read** workloads. We also observe that, as the CAPI Chunk Cache reduced CPU utilization and the pressure on I/O subsystem, as a side effect, Cassandra's compaction operations accelerated, which in turn reduced the read latency.

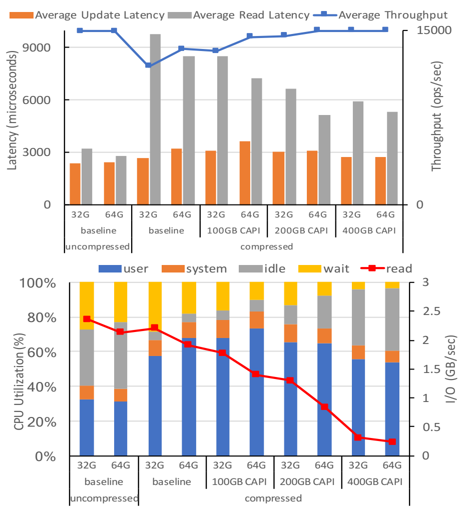
Even though using the `uncompressed` dataset is not comparable with `compressed` dataset, we observe that in **Figures** 5 and 7 `uncompressed` settings suffer overhead in the I/O stack and become bottlenecked on CPU. In this case, we show that our caching mechanism can improve the performance and reduce the CPU utilization over `uncompressed`



(a) `target=5000 ops/sec`

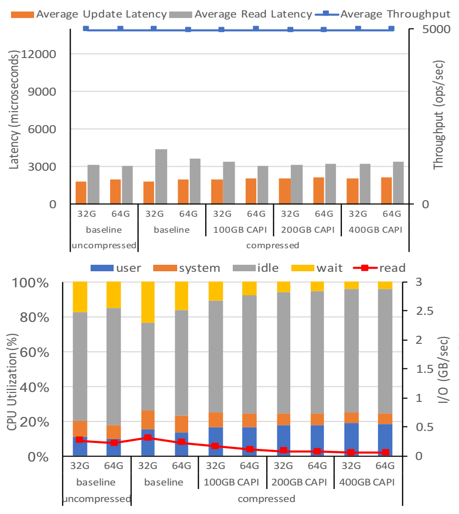


(b) `target=10000 ops/sec`

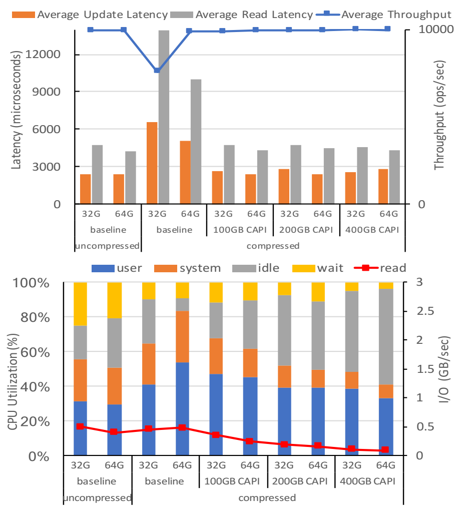


(c) `target=15000 ops/sec`

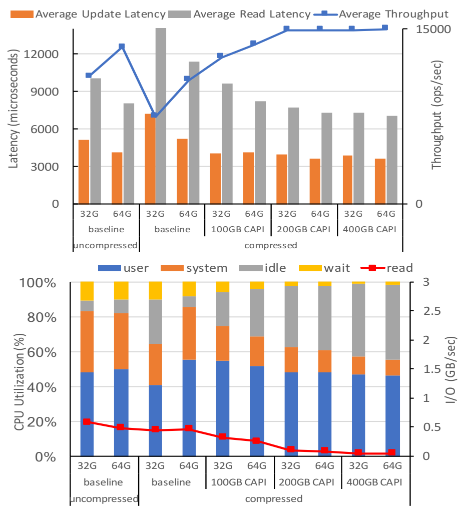
Fig. 6: Running **80% read - 20% update** workload using the parameters defined in **Table** II and **64KB** kernel pages.



(a) `target=5000 ops/sec`



(b) `target=10000 ops/sec`



(c) `target=15000 ops/sec`

Fig. 7: Running **80% read - 20% update** workload using the parameters defined in **Table** II and **4KB** kernel pages.

datasets as well.

V. Related Work

There are several studies on benchmarking and evaluating aspects of NoSQL technologies with different benchmarks [14], [20], [21], [33]. However, their focus is not on optimizing resource utilization or performance.

For optimizing deployment costs, Awasthi et al. [16] and Menon et al. [27] studied the feasibility of combining fast (SSD) and slow (HDD) storage mediums to store different components of LSM-tree based NoSQLs. A recent study from Facebook [22] discusses tunings for RocksDB to optimize storage efficiency while ensuring service level requirements. These studies mainly propose tunings or study feasibility of combining different storage mediums to reduce the deployment cost. In our work, we focus on reducing the deployment costs by replacing costly DRAM with CAPI-Flash.

Another way of introducing SSDs for NoSQL workloads is to use it as a caching layer and an alternative to memory-based caches. There are several modules integrated with the Linux kernel that use SSDs as write-back or write-through cache for spinning disks [3], [6]. FatCache [5], McDipper [9] and FlashStore [19] work as SSD-backed generic key-value cache. CaSSanDra [26] introduces a caching layer by extending Cassandra's Row Cache using consumer grade SSD. Different from our work, the Row Cache stores the entire row and is keyed to the row key. We extended research in this area with the design of a low-latency caching mechanism specialized for Cassandra's chunk based compression mechanism.

There are several solutions that exploit new generation I/O acceleration technologies such as CAPI-Flash and SPDK to accelerate NoSQL workloads. In our previous work [30], we used CAPI-Flash to optimize write-ahead-logging mechanism of Cassandra and provide high-performance durability for write-intensive workloads. Thus, it did not optimize read operations or resource utilization. The Power Systems solution for Redis [29] leverages CAPI-Flash's key-value API and uses flash as a memory expansion for Redis instance. It replaces the key-value backend of Redis, thus does not provide caching for storage components. Similar to CaSSanDra [26], CAPI-RowCache [4] extends Cassandra's row cache on CAPI-Flash to speed-up read intensive workloads.

Neo4J [11] enables CAPI-Flash as storage to store its files instead of the file system. SPDK implements a basic filesystem plug-in called BlobStore that integrates with RockDB's storage backend. TOKVS [31] implements durable, append only storage engine that runs on SPDK. These systems replace existing storage mechanisms with CAPI-Flash or SPDK. Our work presents a caching layer for read operations and does not use any storage components.

VI. CONCLUSION

In this paper, we designed and examined the performance of a CAPI-Flash based persistent caching mechanism for Cassandra. Our caching mechanism caches uncompressed chunks of the data files on user-addressable flash. We conducted a comprehensive performance analysis of our caching mechanism against compressed and uncompressed datasets. We believe that the caching layer that we have presented in this paper can be applied to the NoSQL deployments that use chunk-based compression such as RocksDB, HBase and BigTable. We note that to simplify our experiments, we limited our dataset size and configured a CAPI Chunk Cache up to 400GBs. However, our caching mechanism can cache up to 2TB of data per CAPI-Flash card. In our evaluations, we reduce DRAM allocated for Cassandra and enable CAPI Chunk Cache that has approximately 3x, 6x and 13x larger capacity than the reduced DRAM size. We note that DRAM is typically 6x to 20x the cost per byte of (NAND) flash [12], [15]. Based on the current cost of DRAM and CAPI-Flash, DRAM costs 14-16x more per byte compared to CAPI-Flash card with 2TB of integrated flash. When our caching mechanism is enabled, users can combine stronger compression algorithms to get better compression ratios or use it as a high performance caching layer on slower but cheaper storage devices. We showed that CAPI Chunk Cache is low latency and low overhead caching mechanism. Our caching layer can be utilized to significantly reduce the deployment cost of Cassandra by replacing costly DRAM with a caching space on CAPI-Flash layer and provide high performance.

REFERENCES

[1] A high performance caching library for Java. https://github.com/ben-manes/caffeine. [Online].

[2] Apache cassandra. http://cassandra.apache.org/. [Online].

[3] BCache. https://bcache.evilpiepirate.org/. [Online].

[4] CAPI-RowCache. https://github.com/ppc64le/capi-rowcache/. [Online].

[5] FatCache. https://github.com/twitter/fatcache. [Online].

[6] FlashCache. https://github.com/facebookarchive/flashcache. [Online].

[7] Flexible I/O Benchmark. https://github.com/axboe/fio. [Online].

[8] IBM Data Engine for NoSQL Integrated Flash Edition. http://ibm.biz/capiflash. [Online].

[9] McDipper: A key-value Cache. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/. [Online].

[10] Most popular databases. http://db-engines.com/en/ranking. [Online].

[11] Neo4j. https://neo4j.com. [Online].

[12] REDISCONF 2017: Doing more with Redis. https://redislabs.com/blog/looking-back-redisconf-2017/. [Online].

[13] Spdk. http://www.spdk.io. [Online].

[14] V. Abramova and J. Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13. ACM, 2013.

[15] R. Appuswamy, R. Borovica, G. Graefe, and A. Ailamaki. The five minute rule thirty years later and its impact on the storage hierarchy. *VLDB ADMS*, 2017.

[16] A. Awasthi, A. Nandini, A. Bhattacharya, and P. Sehgal. Hybrid hbase: Leveraging flash ssds to improve cost per throughput of hbase. In *Proceedings of the 18th International Conference on Management of Data*, COMAD '12, pages 68–79, Mumbai, India, India, 2012. Computer Society of India.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[19] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, 2010.

[20] E. Dede, B. Sendir, P. Kuzlu, J. Hartog, and M. Govindaraju. An evaluation of cassandra for hadoop. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 494–501, Washington, DC, USA, 2013. IEEE Computer Society.

[21] E. Dede, B. Sendir, P. Kuzlu, J. Weachock, M. Govindaraju, and L. Ramakrishnan. A processing pipeline for cassandra datasets based on hadoop streaming. In *2014 IEEE International Congress on Big Data, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 168–175, 2014.

[22] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[23] G. Einziger, R. Friedman, and B. Manes. Tinylfu: A highly efficient cache admission policy. *CoRR*, abs/1512.00727, 2015.

[24] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX.

[25] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[26] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. Cassandra: An ssd boosted key-value store. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 1162–1167, March 2014.

[27] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. Optimizing Key-Value Stores for Hybrid Storage Architectures. In *Proceedings of CASCON*, 2014.

[28] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[29] H. Reddy, M. Pandya, S. Mallayya, L. Browning, and B. Phu. IBM Power Systems solution for Redis. Technical report, IBM, 7 2014.

[30] B. Sendir, M. Govindaraju, R. Odaira, and P. Hofstee. Optimized durable commitlog for apache cassandra using capi-flash. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 156–163, June 2016.

[31] S. Seshadri, P. Muench, and L. Chiu. Trillion operations key-value storage engine: Revisiting the mission critical analytics storage software stack. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1865–1872, June 2017.

[32] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, Jan 2015.

[33] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 6:1–6:11, New York, NY, USA, 2015. ACM.