# Optimized Durable Commitlog for Apache Cassandra Using CAPI-Flash

Bedri Sendir*, Madhusudhan Govindaraju†
SUNY Binghamton
Binghamton,NY 13902
{bsendir1*, mgovinda†}@binghamton.edu

Rei Odaira‡, Peter Hofstee§
IBM Research
Austin,TX 78758
{rodaira‡, hofstee§}@us.ibm.com

***Abstract*—High-velocity data imposes high durability overheads on Big Data technology components such as NoSQL data stores. In Apache Cassandra, a widely used NoSQL solution with high scalability and availability, write-ahead logging is used to support Commitlog operations, which in turn provides fault tolerance to applications. However, current write-ahead logging techniques are limited by the excessive overhead in the I/O subsystem. To address this performance gap, we have designed a novel CAPI-Flash based high performance durable Commitlog for Apache Cassandra. We take advantage of the high throughput, low latency path to flash storage provided by the Coherent Accelerator Processor Interface (CAPI) on IBM POWER8 Systems. Our experimental results show that for write-intensive workloads CAPI-Flash logging provides up to 107% improvement in throughput compared to Cassandra's durable alternative. We also provide 77% better throughput in update-mostly workloads.**[1]

## I. INTRODUCTION

The explosive growth of data from various sources requires cloud platforms to store unstructured data for Big Data applications in a scalable and durable manner. NoSQL data stores such as MongoDB [5], HBase [3] and Cassandra [20] are increasingly used in Big Data applications because they do not require data to be stored in a structured format and also provide scalability.

Apache Cassandra is an example of a NoSQL technology that is gaining significant traction in the industry. Cassandra is used in production in hundreds of companies including eBay, GoDaddy, Netflix, and the Weather channel [1].

Write-ahead logging is adopted by NoSQL data stores to provide durability. All modifications to data are first recorded in logs before they are applied. In case of failures, such as server crashes, the logs allow changes to be replayed. Logging is supported via the Commitlog in Cassandra, Journaling in MongoDB, and HLog in HBase. The logging operations are typically inefficient as they require slow I/O to disk-based storage. It is critically important to address this bottleneck.

Performing I/O on flash greatly improves the log performance over disk-based storage, but this approach still suffers from software overhead in the device driver and operating system. The POWER8 CAPI-Flash system provides user-level applications with access to flash without operating system intervention and thus eliminates 97% of the code path length [9]. It is based on the POWER8 Coherent Accelerator Processor Interface (CAPI), which enables cache-coherent hardware accelerators with address translation capability.

Our goal is to use the POWER8 CAPI-Flash system to optimize the performance of Cassandra's Commitlog, and able high-performance durability. Our *CAPIFlash Commitlog* is specifically designed for the block-level read and write interfaces of CAPI-Flash. It fully takes advantage of direct reads and writes to the physical address space on flash.

The contributions of this paper are as follows:

- Durable logging on flash using POWER8 CAPI-Flash.
- Performance evaluation of our logging mechanism vs. Cassandra's built-in logging, on flash and hard disks.
- We analyze the performance of our logging mechanism with various read/write mixes and varying record sizes.

## II. BACKGROUND

### A. Apache Cassandra

Apache Cassandra [20] is an open source, non-relational, horizontally scalable, distributed database, initially developed at Facebook. It is designed to handle large volumes of data while providing high availability with no single points of failure. Cassandra's data distribution mechanism is based on Amazon's Dynamo [14]. Similar to the design of Dynamo, Cassandra applies consistent hashing by placing the data on each node according to the value of the row key and the range that the node is responsible for. This approach decreases the need for shuffling data between the nodes when a new node is added or removed. Cassandra offers tuneable eventual consistency, which provides configurable trade-offs between latency and consistency. Cassandra's data model employs the column oriented architecture of Google's BigTable [10]. In this data model, a column constitutes the smallest component of the database. 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. Like a table in a relational database, rows are organized into column families sorted in a key order. Rows stored in a column family do not share the same schema and each row may contain different set of columns.

Cassandra consists of three main core components – (1) *Memtable*: a write-back cache that maps the row by its key and is maintained per column-family basis; (2) Sorted String Table

[1]This work was supported in part by a grant from IBM.

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.27

(*SSTable*): persistent, immutable files that keep key-value pairs in sorted order; and (3) *Commitlog*: append only files that keep modified data to provide durability.

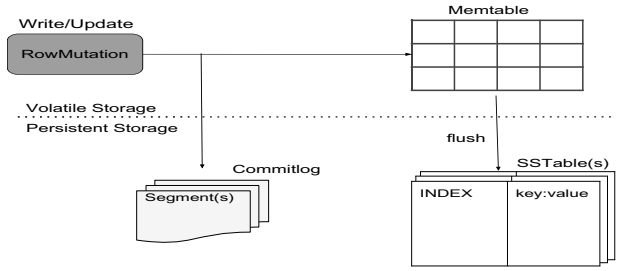



Fig. 1. Path of a write operation. When fault tolerance is required, before applying RowMutations into key-value data structure stored in the memory (Memtable), Cassandra ensures durability by appending the data to Commitlog which is stored in persistent storage. At a configurable threshold, Memtables flush their data into immutable SSTables.

**Figure** 1 shows the path of a write operation. Each insert, update, and delete operation is encapsulated as a *RowMutation* for a particular key. Cassandra first appends the *RowMutation* to the *Commitlog* and then places it into an in-memory data structure, *Memtable*. At a configurable threshold, Cassandra flushes the *Memtable* data to *SSTable* files. A *Memtable* that is scheduled for flush is placed into a flush queue. A flush writer thread dequeues the task from flush queue, serializes the *Memtable* and sequentially writes it as an *SSTable*. If the flush queue is full, incoming write requests are blocked until the next flush succeeds. Once *SSTable* is created, it cannot be modified. As insert or update operations occur, a new entry for that particular key is created in a *Memtable* with a new timestamp. Therefore, portions or different versions of a row might reside in multiple *SSTable*s. For each flush operation, Cassandra holds a *ReplayPosition* on the *Commitlog* that indicates the latest offset at the time of the flush operation. After a successful flush, *RowMutation*s stored below the offset of *ReplayPosition* for that particular column-family are evicted from the *Commitlog*. Cassandra handles the accumulation of *SSTable*s by consolidating them periodically in compaction stage. It marks the key of the stale data and merges multiple *SSTable*s into a single *SSTable*. When a read request is received, Cassandra merges the recent data from *SSTable*s and *Memtable*s into a single record. Cassandra speeds-up read operations by implementing row and key cache properties to reduce the cost of searching a given key in the *SSTable*s.

### B. Durability of Cassandra

As described in **Section** II-A, Cassandra increases the write throughput by delaying and optimizing pending disk write operations in memory. In order to provide fault tolerance, each *RowMutation* gets appended to on-disk Commitlog files before they are applied into in-memory data structure called *Memtable* As all changes on the key are grouped into a single *RowMutation*, *Commitlog* is also used for providing row-level atomicity on a per node basis. *RowMutation*s continue residing in *Commitlog* until the affected *Memtable*s are flushed to persistent storage as an *SSTable*. In order to keep track of the *RowMutation*s that are no longer needed and to avoid fragmentation on the logging drive, Cassandra's default *Commitlog* pre-allocates fixed size segments and reuses the segments after *RowMutation*s in the segment are fully flushed. While managing its segments on disk, it uses the Linux file attributes to determine existing *Commitlog* segments and their order. On a POSIX compliant system, when a write cache is enabled, the operating system places dirty pages into a queue and asynchronously flushes it to persistent storage. However, this approach lacks durability and can potentially cause data loss in case of a power failure. The `fsync()` call can be utilized to force synchronization of modified buffer cache pages to persistent storage and establish a checkpoint for durability.

Hard disk drives suffer from seek delays caused by forced synchronization of data, such as using `fsync()`, if the target blocks are not properly sorted and too far apart from each other. Flash based storage systems, on the other hand, have no moving mechanical parts and do not suffer from such issues. However, a file system mounted on flash still suffers from overhead due to device driver and operating system software.

Cassandra uses memory-mapped I/O for *Commitlog* operations to avoid expensive copying of buffers. *Commitlog* files are pre-allocated as segments to co-locate them on disk. When durability is required, file buffer synchronization is forced by calling the function msync() with the *MS_SYNC* flag which synchronizes memory-mapped data to the underlying storage. This call ultimately results in calling `fsync()` on dirty pages.

Cassandra offers two configurable synchronization strategies to provide durability. By default, Cassandra uses *periodic* synchronization. This strategy queues the mutations and periodically performs synchronization. However, this approach provides weak durability. This is because the client receives acknowledgment right after writing the mutation to the *Memtable*s. So, if all the replicas crash within the synchronization window, it is possible to lose data. The high durable alternative, *batch* synchronization, guarantees persistence before acknowledging to the client by collecting the mutations over a predefined time period window into batches.

There are two cases when *Batch Commitlog* calls `fsync()`: (1) batch window timer expiration and (2) when Cassandra applies limits to the number of clients that are performing write requests with the *concurrent_writes* property. If all of the client threads have arrived into *Commitlog* and formed a batch, *Batch Commitlog* does not wait for the batch window timer to expire and calls `fsync()` immediately. The *Batch Commitlog* is critically important in terms of durability and is a performance bottleneck even if we store the *Commitlog* data on flash.

### C. Coherent Accelerator Processor Interface

The Coherent Accelerator Processor Interface (CAPI) [25] on POWER8 systems allows implementation of low-latency cache-coherent hardware accelerators. The CAPI interface is implemented in two parts: the coherent accelerator processor proxy (CAPP) on the POWER8 processor and the POWER Service Layer (PSL) in the attached device interfacing with

the specific accelerator. The PSL is responsible for address translation and caching and allows the attached device to share memory coherently with the processor. The on-processor CAPP unit maintains a copy of the cache directory and responds to snoop commands without degrading SMP performance. A CAPI-attached device can perform Direct Memory Access (DMA) to application memory, without calls to a device driver or underlying operating system kernel, resulting in a reduction in the code path. This improves performance significantly compared to the traditional I/O model. [23]

### D. CAPI-Flash

CAPI-Flash [9] provides a software and hardware stack to exploit a high throughput and low latency path between the processor and flash storage. An FPGA includes the CAPI PSL and interfaces to fiber channel I/O ports to allow direct access to an IBM Flash System. CAPI-Flash exposes the *Capiblock API*, which provides block level (multiples of 4KB) access to flash. The API provides synchronous and asynchronous read/write requests to the flash. Block level access can be configured as either persistent or transient mode. Our preliminary experiments show that, in terms of 4KB random read IOPS, CAPI-Flash delivers 360K IOPS, while conventional flash delivers 250K IOPS. For 4KB write latency, CAPI-Flash is around 240 usec, while conventional flash (`write()` + `fsync()`) is around 620 usec. Because the operating system is not in the path, jitter is also significantly reduced.

## III. Design

As described in **Section** II-D, the *Capiblock API* provides a raw block I/O interface for reading and writing to the physical address space on flash. By taking advantage of the high-bandwidth, low-latency path between flash and processor, we implemented flash-based durable logging.

### A. Layout on Flash and Memory

The *Capiblock API* limits flash access to block-level read and write operations. **Figure** 2 shows the block layout of the *CAPIFlash Commitlog* on the flash. As shown in **Figure** 2c, in order to be able to manage the address space efficiently and to allow re-usability of the space on flash, we logically divide the address space into fixed size segments.In order to identify the segments that contain unflushed data (live segments) and to determine the order of the segments, we reserve a fixed number of blocks in a bookkeeping area to be equal to the number of segments (**Figure** 2b). The capacity of the *CAPIFlash Commitlog* is defined by number of segments and number of blocks allocated in each segment.

In our design, column family usage patterns and flush frequency should be taken into consideration while deciding on the size of a segment. For example: if the segment size is too large and a column family is rarely updated, Cassandra may not flush the *Memtable* of that particular column family and therefore *Commitlog* segments containing the mutations for that column family will continue staying live. However, smaller segment sizes will cause more frequent access to bookkeeping area, thus will bring segment management overhead.

Each bookkeeping block maps to a segment by its order. In bookkeeping blocks, if the corresponding segment is available, we store a unique identifier that presents the order of the segment and its availability. Otherwise, the bookkeeping block contains zero. Even though the one-block-for-one-identifier approach is space inefficient in the bookkeeping area, it allows the *CAPIFlash Commitlog* to asynchronously update the bookkeeping blocks without requiring synchronous read-modify-write operations.

In memory, we keep three different data items to track the status of the *CAPIFlash Commitlog*. First, for the active segment, we keep the position of the last record *ReplayPosition* to identify the mark for flush operations. It is globally represented and contains a tuple of unique segment identifier of the active segment and number of blocks used in the active segment. In Cassandra's default *Commitlog*, *ReplayPosition* information represented with unique a filename for that particular segment and position in file's buffer. Second, as Cassandra flushes *Memtables* on a column family basis, there is a need for a caching mechanism to identify which segments contain unflushed *RowMutation* data. Each segment could get mutations from various column families and to identify unused segment incurs the cost of seeking the entire segment. Like Cassandra's default *Commitlog*, on each *Commitlog* segment, we keep a cache that contains an entry per column family which consists of the last write position for that particular column family. Third, we keep an in-memory representation of bookkeeping blocks as a free list to efficiently manage segment allocation.

A segment in the *CAPIFlash Commitlog* may contain an arbitrary number of records due to the size of the data stored in each record. Each record is aligned by 4KB blocks. **Figure** 2e shows the smallest component of a *CAPIFlash Commitlog* record, which represents a *RowMutation* in Cassandra. The first field stores a unique identifier that shows which segment that record belongs to. Since we are not clearing the data on physical address space, while reading records, we may encounter a record that has already been flushed. By matching identifiers with active segments on bookkeeping blocks, we determine valid records. The second field represents the size of the serialized data stored in the record. The fourth field consists of a serialized *RowMutation* object. Similar to Cassandra's default *Commitlog*, we compute a cyclic redundancy check (CRC) to ensure integrity of the record during the recovery phase. The first CRC is computed using the first two fields and the second CRC is computed with the DATA field.

### B. Interface and Semantics

On a write operation, if the active segment has enough capacity, we serialize *RowMutation* and generate a record. Cassandra's default *Commitlog* handles the *Commitlog* operations through a single threaded executor. However, the *CAPIFlash Commitlog* handles write operations in parallel. The prepared record gets appended to the active segment. If we are out of capacity for the active segment, we activate

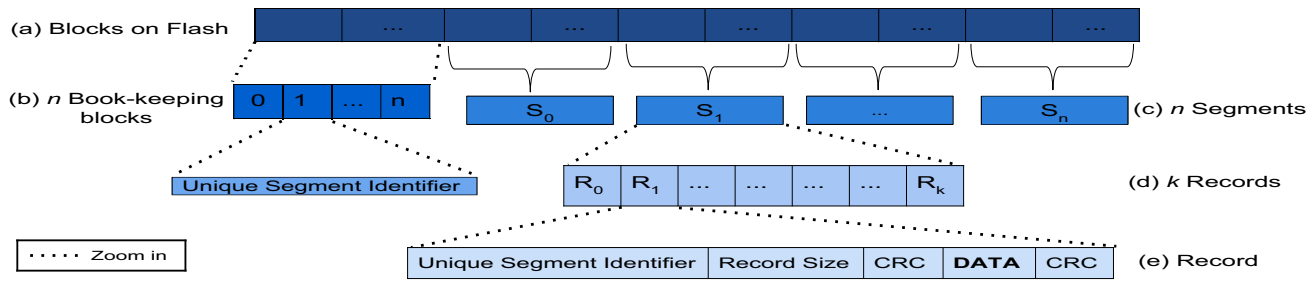



Fig. 2. Block level layout of *CAPIFlash Commitlog* on flash. **(a)** shows block-addressable layout on flash. First *n* blocks are reserved for bookkeeping space and the rest used for storing the records. **(b)** shows reserved bookkeeping space. Each bookkeeping block stores a unique segment identifier. **(c)** shows segments which are formed by group of blocks. **(d)** a segment can store arbitrary number of records. Minimum record size is a block(4KB) and each record is aligned by the size of a block. **(e)** Represents smallest component of *CAPIFlash Commitlog*. Each record contains serialized representation of a *RowMutation* and other fields to ensure correctness of a record during recovery.

a new segment and write its unique identifier to the relevant bookkeeping block. The unique identifier is generated by using milliseconds since UNIX epoch and each time we activate a new *CAPIFlash Commitlog* segment we atomically increment the identifier by one to provide uniqueness. For each write operation, the cache residing in the active segment is updated with the column families that are affected by the mutation and the current offset of the segment.

Since a flush operation relies on *ReplayPosition* for identifying the cut for performing the flush, we follow the same approach as Cassandra's default *Commitlog* for pre-flush and post-flush operations. At the time of flush, we retrieve the global *ReplayPosition* atomically from the *Commitlog*. After completion, we iterate through the live segments and identify the segments that are fully flushed. Cassandra's default *Commitlog* recycles flushed *Commitlog* segments by simply renaming the corresponding segment files. However, since the *CAPIFlash Commitlog* operates at the block-level, we first mark relevant bookkeeping block for that segment as zero, and then return the segment back to the in-memory free list.

For the workloads where *Memtable*s are updated in place, infrequent flush activity causes the *Commitlog* to grow larger. Cassandra's default *Commitlog* keeps a threshold for *Commitlog* size but it does not strictly enforce it. When the capacity is exceeded, flush is initiated on the oldest keyspaces that are active in the *Commitlog* but it does not block the new segment activation process. In the *CAPIFlash Commitlog*, the number of segments and blocks in each segment are statically initialized on startup. Since we allocate fixed space, the *CAPIFlash Commitlog* can run out of space in some workloads.(e.g., update-heavy, update-mostly workloads). An emergency flush mechanism addresses this case. We define a threshold on the free segments list and when the *CAPIFlash Commitlog* space gets below this threshold we start scheduling a flush on the oldest keyspaces. In order to prevent accumulation of flush requests, we make scheduling decisions according to the current state of the flush writer queue. If the *CAPIFlash Commitlog* is out of space, all the client and internal write requests are blocked until all scheduled flush operations have completed.

### C. Recovery Mechanism

For recovery, the *CAPIFlash Commitlog* follows an approach similar to the default Cassandra implementation. When a Cassandra instance boots up, before accepting any writes, its recovery mechanism starts. We first iterate through bookkeeping blocks and determine segments that are not completely flushed. At the time of flush, Cassandra creates a metadata file which contains information such as the minimum and maximum timestamp of the keys, the partitioner used to create this *SSTable*, the lowest *ReplayPosition* offset, and a histogram of column and row sizes. We scan the *SSTable* metadata and identify the lowest replay offset. In order to avoid multiple sequential read requests to flash, we load large portions of the segment data into off-heap memory and then merge it as a whole segment. While reading records from segment data, we confirm that the position of the entry is bigger than the most recent *ReplayPosition* that was retrieved from *SSTable* metadata. For redundancy, we implement a three-step validation to ensure that the record that has been read is accurate. First, to avoid replaying a stale record we check the segment identifier of the record with the relevant bookkeeping block. This is because when a segment is recycled, we only nullify the bookkeeping area to avoid unnecessary writes on flash. For this reason, on physical address space, it is possible that we hit a block that is valid but belongs to a recycled segment. Second, we calculate CRC of the segment id and size and cross check it with the stored CRC. Finally, we read the serialized *RowMutation* from off-heap buffer and validate its CRC. Once the data is validated, it is deserialized into a *RowMutation* and replayed into the database. We confirmed operation of our implementation by verifying the number of records replayed on the Cassandra server after the failure.

## IV. Performance Results

In this section we first evaluate performance of Cassandra with the *CAPIFlash Commitlog* using the YCSB write-only workload and various read-write-mix workloads.

We ran our experiments on the following HW/SW stack:

- POWER8 822L(20core, SMT8, 3690MHz, 256GB RAM)
- Ubuntu 15.04 ppc64le (4.1.0)
- 1 x Emulex LPe16000 FibreChannel card, 2x 8Gbps ports to RAID5 flash on IBM FlashSystem 840

| Setting | Commitlog Sync | Commitlog Location | SSTable Location |
|---|---|---|---|
| S1 | disabled | none | Flash |
| S2 | disabled | none | HDD |
| S3 | capi | CAPI-Flash | Flash |
| S4 | capi | CAPI-Flash | HDD |
| S5 | batch | Flash | Flash |
| S6 | batch | Flash | HDD |
| S7 | batch | HDD | Flash |
| S8 | batch | HDD | HDD |
| S9 | periodic | Flash | Flash |
| S10 | periodic | Flash | HDD |
| S11 | periodic | HDD | Flash |
| S12 | periodic | HDD | HDD |

TABLE I
SETTINGS TABLE

- CAPI-Flash card connected to IBM FlashSystem 840(Surelock GA2) with 1x 8Gbps FiberChannel
- Local Hard Disk, RAID0 300GB 10000 RPM SAS
- Apache Cassandra version 2.0.16
- JDK Version, OpenJDK 1.8.045-internal

In our experiments we set the CPU governor to "performance". We run Cassandra and the YCSB client on the same machine. The Cassandra process uses all of the 20 cores, but the YCSB client is bound to 10 cores to not disturb the Cassandra process. The Cassandra heap size is set to 64GB. Both the *concurrent_writes* and *concurrent_reads* parameters are set to 64. Since we are using a large Java heap, to minimize the congestion on the flush queue, the number of flush writers is increased to 4. On the *CAPIFlash Commitlog*, we set blocks per segment to 32000 and the number of segments to 512.

**The Yahoo! Cloud Serving Benchmark** [12] is an open source benchmarking suite for evaluating performance of key-value and cloud serving stores. YCSB supports many popular database systems like MongoDB [5], HBase [3], Redis [6] and Cassandra. A YCSB workload consists of load and transactional phases. The load phase inserts records, and is write-only. The transactional phase is a mixture of read, insert, update, and scan operations. YCSB lets users to define dataset properties: number of fields, field length, and number of records.

### A. Write-only Workloads

As described in **Section** II-A, Cassandra is highly optimized for write operations and thus is used for write-heavy workloads [4]. Therefore, we first focus on the load phase of YCSB.

Synchronization options *batch*(durable) and *periodic*(non-durable) use Cassandra's built-in *Commitlog* with the default settings defined in Cassandra version 2.0.16. For durable *batch* synchronization, we use 2 milliseconds as synchronization interval. For non-durable *periodic* synchronization, we use 10 seconds as the window size for performing synchronization. Option *capi* uses *CAPIFlash Commitlog* and synchronizing each *RowMutation* immediately. **Table I** shows persistent storage and synchronization settings used in our experiments.

**Figure** 3 shows the software and hardware components used in our experiments. Both CAPI Card and Fibre Channel Card connect to POWER8 core through PCIe Gen3 bus. As persistent storage location we use 3 different configurations:

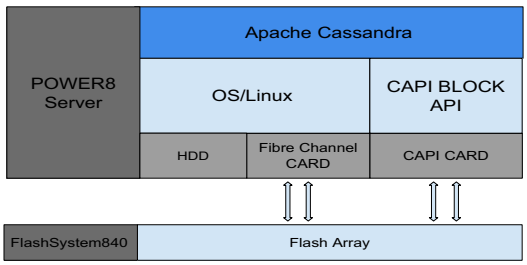



Fig. 3. Layout of software and hardware components of Apache Cassandra on POWER8.

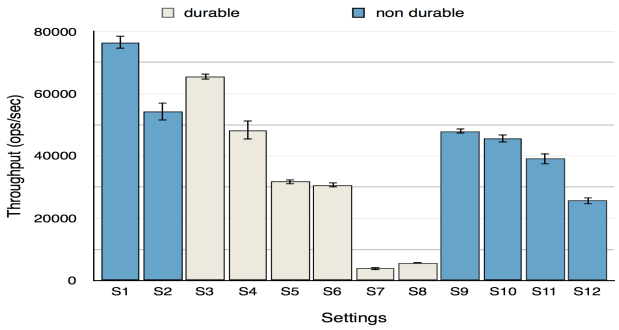


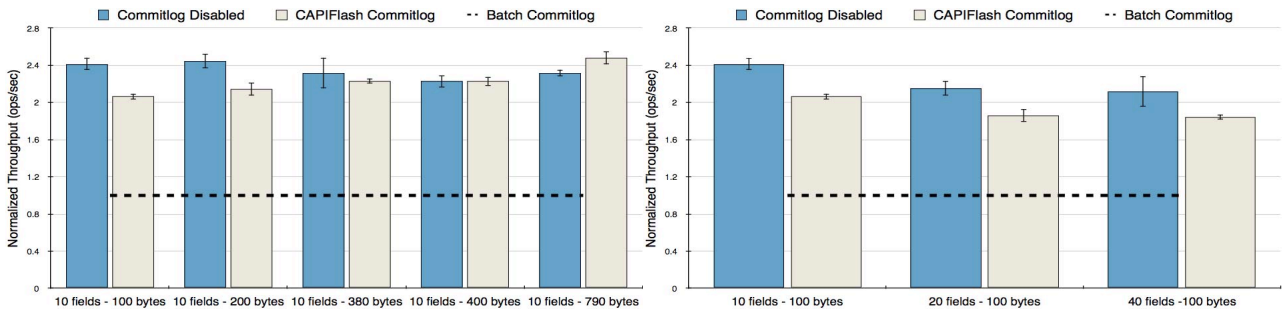

Fig. 4. Inserting 10M records using combination of commitlog settings and storage mechanisms defined in **Table I**. In this experiment, a record consists of 10 fields each storing 100 bytes of randomly generated data. Error bars show the 95% confidence intervals.

- *HDD*, ext4 file system on local hard disks
- *Flash*, ext4 file system on a 1 TB volume in IBM FlashSystem 840 connected with FibreChannel card
- *CAPI-Flash*, 1.5 TB volume on IBM FlashSystem 840 attached to CAPI-Flash card using a FibreChannel link

**Figure** 4 shows the average throughput for write-only workloads with 10 million records using YCSB's default settings (10 fields, each 100 bytes). A larger number of records did not change the throughput and hence we omitted those results. *S1* and *S2* bypass logging operation and directly add data to *Memtable*s. *S1* shows maximum possible performance (no durability). As described in **Section** II-A, while *Memtable* flush operation is optimized for spinning disks and only consists of sequential write operations, we observe that slow I/O to *HDD* causes accumulation of flush tasks under heavy write workload. Due to many queued flush tasks, incoming write requests are blocked. Therefore, *S2* performs 29% slower in compare to *S1* when we use *HDD* as the *SSTable* location.

Settings *S3* through *S8* provide high durability by guaranteeing persistence of the data before acknowledging the client. From *S1* to *S3*, performance decreases by 14%, this behavior is expected. For settings *S3* and *S4*, using HDD as *SSTable* location decreases throughput by 26%. With durable writes, *CAPIFlash Commitlog* (*S3*) provides 107% better throughput than Cassandra's durable *Batch Commitlog* (*S5*) using flash as both *SSTable* and *Commitlog* storage.

For settings *S5* and *S6*, the *SSTable* location does not change Cassandra write throughput. Synchronization overhead in the I/O subcomponent of Cassandra hides the performance gain from writing *SSTable*s on flash. Since synchronizing to




(a) Field Length (b) Field Count

Fig. 5. Normalized performance for running 10 million record insert-only workload with varying record widths. The baseline is performance of Cassandra's default durable setting *Batch Commitlog*. In **(a)**, we show inserting records with fixed number of fields and changing field lengths. After serializing a record, on *CAPIFlash Commitlog*, field sizes 400 and 790 allocate two blocks on flash for each record. In **(b)**, we show the performance with varying field counts. In this experiment, field setting of 40 allocates two blocks on flash while using *CAPIFlash Commitlog*. Error bars show the 95% confidence intervals.

rotational media is too expensive, the target location for the *Commitlog* dramatically affects the performance. Settings *S7* and *S8* show a decrease in write performance of +/- 82% compared to *S5* and *S6*. Synchronizing HDD creates a severe bottleneck on the write path regardless of *SSTable* location.

Settings *S9* through *S12* show performance with *periodic* synchronization where clients do not wait for acknowledgment whether their request persisted on the Cassandra server or not. Settings *S9* and *S10* display similar performance characteristics and show around 38% decrease in performance compared to the top performing non-durable setting, *S1*. Even though there is no blocking behavior on client requests for persisting logs, *periodic* synchronization still causes a bottleneck on the write path. *S11* suffers slow I/O on HDD and performs 18% slower in comparison to *S9*. Setting *S12* shows the worst performance among all *Periodic Commitlog* experiments. This is because heavy I/O to the same HDD from *SSTable* creation and *Commitlog* operations decreases the overall performance.

**Figure** 5 shows performance of inserting 10 million records with various record sizes. As mentioned in **Section** III-A, write granularity on flash is aligned by 4KB blocks. In this experiment, performance improves by fully utilizing block writes and with more blocks written per write request. Record sizes of 380 bytes and 790 bytes fully use one and two blocks respectively. When serialized, a record size of 400 bytes overflows to the second block. So, in this case, around $\frac{3}{4}th$ of the second block is not used but written to the flash. **Figure** 5a shows that with increasing field length, the performance gap between *CAPIFlash Commitlog* and *Batch Commitlog* continuously increases. Thus *CAPIFlash Commitlog* improves most vs. *Batch Commitlog* when the block buffer is fully utilized. In (10 fields-790 bytes) experiments, *Commitlog Disabled* loses its advantage over the *CAPIFlash Commitlog*. *Commitlog Disabled* garbage collection compared to *CAPIFlash Commitlog* and *Batch Commitlog* takes 2.5 times longer.

In **Figure** 5b, we show the performance of introducing fixed-length fields. In this experiment, for *Commitlog Disabled* and *CAPIFlash Commitlog* we measure blocking for client write requests due to contention in the flush queue. As described in **Section** II-A, each column in Cassandra contains a value, timestamp and a name. Increasing number of fields inserted causes the number of inserted columns to increases. Increased field count causes serialization overhead and time spent on each flush task goes up.Since write requests bottleneck on *Batch Commitlog*, all flush requests are smoothly scheduled and no blocking behavior is observed. As a result, with increasing size of fields, *CAPIFlash Commitlog* performs just 1.8 times better than *Batch Commitlog*.

| **Setting** | Average Throughput (ops/sec) | Average Latency (microseconds) |
|---|---|---|
| 0 ms batch | 1393 | 92115 |
| 1 ms batch | 30302 | 4188 |
| 2 ms batch | 31111 | 4036 |
| 5 ms batch | 30975 | 4101 |
| CAPIFlash Commitlog | 65144 | 1914 |

TABLE II
COMPARING AVERAGE LATENCY AGAINST AVERAGE THROUGHPUT WITH VARIOUS BATCH SYNCHRONIZATION WINDOW SETTINGS.

In **Table II**, we show performance of writing 10 million records with different batch window sizes. A 0ms batch ignores the window size property and synchronizes mutations one by one. As expected, this configuration performs dramatically slower. Each client request results in a separate call to `fysnc()`. 1ms, 2ms and 5ms settings show similar performance because heavy write load causes fixed number of client threads to form the batch before the defined window time. So, window size only defines an upper limit for client requests to wait before forming a batch. On the other hand, *CAPIFlash Commitlog* does not group the requests to perform synchronization. In this experiment, *CAPIFlash Commitlog* shows the smallest latency. For each write request, *CAPIFlash Commitlog* issues a synchronous write call through its own chunk on flash. With these results, we confirm that no matter how we tune the batch window size *CAPIFlash Commitlog* has a performance advantage over the *Batch Commitlog*.

As mentioned in **Section** II-A, creation and maintenance of *SSTable*s causes CPU and I/O intensive operations running in the background of the Cassandra server. In **Figure** 6, we modify Cassandra's flush operations and increase the value of configurables to prevent flushing *Memtable*s. We increase the JVM Heap size to 160GB, *Memtable* space to

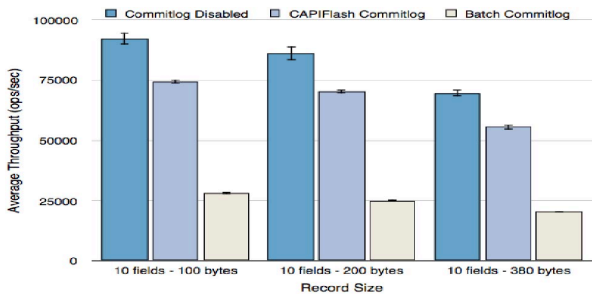



Fig. 6. Inserting 10 million write-only records while forcing Cassandra to keep its data in memory and use large *Memtable*s. For comparison, we use Cassandra's durable *Batch Commitlog* and flash storage as *Commitlog* location. Error bars show the 95% confidence intervals.

128GB, and *Commitlog* total space to 96GB. On *CAPIFlash Commitlog*, the capacity is increased to 768 segments and flush threshold decreased to 0.1. In this experiment, in order to reduce garbage collection pauses, we tune Cassandra's default garbage collector parameter (Concurrent Mark Sweep) to Garbage First Garbage Collector (G1GC). Since flush is disabled, in this experiment, we show pure logging performance with no interference from *SSTable* related operations. In this case, *Commitlog Disabled* setting shows Cassandra's insert performance when using an in-memory data structure. For 1KB records, the *CAPIFlash Commitlog* shows 162% better performance compared to *Batch Commitlog*. While using *CAPIFlash Commitlog*, making writes durable results in 19% decrease in throughput compared to *Commitlog Disabled*. Performance slightly decreases when we increase the record width. However, when we double the record size, the performance gap between *Batch Commitlog* and *CAPIFlash Commitlog* increases by 20%. This is simply because *CAPIFlash Commitlog* fills up its 4KB block buffer more efficiently.

### B. Read-Write Workloads

In this section, we show performance of running various read-write-mix workloads using the transactional phase of YCSB across a wide range of use cases. In **Figure** 7 we examine performance by presenting throughput and average latencies for read/update portions of the workloads separately. In **Figure** 7c we observe that the *CAPIFlash Commitlog* shows around 65% less update latency in (5% read - 95% update) and (15% read - 85% update) compared to *Batch Commitlog*. This gap drops to 60% in the (50% read - 50% update) workload

In **Figure** 7b we observe that Cassandra, with *Batch Commitlog*, performs read operations better than with *CAPIFlash Commitlog* and *Commitlog Disabled*. This is counterintuitive because the read path of Cassandra does not include any Commitlog operations. We postulate that in the Batch Commitlog, writer threads are placed into idle state while waiting on synchronization. In the meantime, reader threads get more execution time. As a result, *Batch Commitlog* outperforms *Commitlog Disabled* and *CAPIFlash Commitlog* in terms of the read latency. In terms of overall throughput, the *CAPIFlash Commitlog* performs 77% faster in 5% read - 95% update workloads and 38% faster in 15% read - 85% update workloads vs. *Batch Commitlog*. However, in rest of the workloads *CAPIFlash Commitlog* shows similar performance as *Batch Commitlog*. We note that Cassandra is mainly used for write-heavy loads, and users are typically aware of the read/write mix of their workloads, and so they can selectively enable *CAPIFlash Commitlog* when their workloads are write-heavy.

## V. Related Work

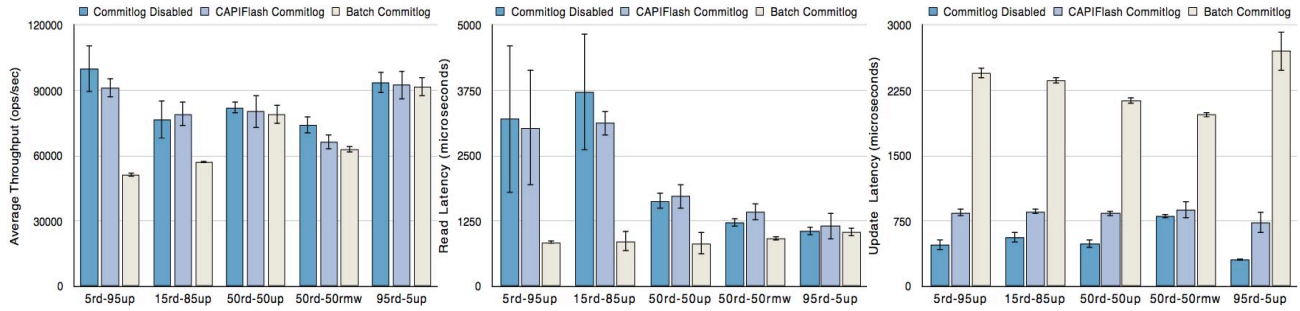
There are various studies on benchmarking and evaluating aspects of NoSQL technologies with different benchmarks, but their focus is not on optimizing Commitlog based durability [7], [12], [15], [16], [18]. Menon et al. [21] evaluated performance of Cassandra with read-mostly workloads using HDD and consumer grade SSD as storage mechanisms and presented secondary row cache by taking advantage of SSD storage, which achieves up to 86 percent improvement in read-mostly workloads. Awasthi et al. [8] analyze feasibility of using SSDs for HBase [3], which also uses the same storage concepts as BigTable [10]. In this work, the authors assess the effects on YCSB of placing core components of HBase on different storage types. Currently, hybrid storage architecture for HBase achieves 33% better throughput compared to an architecture that uses HDD for all components. Power Systems solution for Redis [22] leverages CAPI-Flash through key-value API provided by IBM Data Engine for NoSQL [9] and uses 56TB of flash as memory extension for a Redis instance.

The benefits of exploiting non-volatile memory to improve logging performance for NoSQL and relational databases have been shown in many studies [17], [24]. FlashLogging [11], uses inexpensive USB flash drives to improve synchronous logging relational databases. FlashStore [13], a persistent key-value store, uses flash as a non-volatile cache between RAM and spinning disk. It provides durability by periodically flushing in-memory buffer to flash with a configurable timeout. DuraSSD [19], presents durable write cache embedded in flash memory and provides significant performance in the update heavy workloads by reducing synchronization overhead. These existing systems using flash suffer from overhead in the operations system, and our work is the first to study the performance of the Commitlog bypassing such overhead.

## VI. Conclusion

We examined the performance of durable CAPI-Flash based logging implementation with various use cases. We believe that the techniques that we have presented in this paper can be applied to any systems using write-ahead logging.

- By bypassing Linux I/O subsystem the *CAPIFlash Commitlog* shows 107% better throughput in write-only workloads compared to Cassandra's durable alternative.
- On write-only workloads, when width of a record is increased to fit in multiples of 4KB blocks, the performance gap between *CAPIFlash Commitlog* and Cassandra's durable alternative *Batch Commitlog* widens.




(a) Average Throughput (b) Average Read Latency (c) Average Update Latency

Fig. 7. Running various read-write workloads on 10 million records stored in a single *SSTable*. A record contains 10 fields with randomly generated 100 byte data field and workloads consists of 9 million operations each. All workloads use 128 YCSB threads and keys used in benchmarks selected with zipfian distribution. In order to isolate each run, we disable Cassandra's key caches and drop OS page cache in between each iteration. In **(b)** we show average latency for read requests and in **(c)** we show average latency for update requests. Error bars show the 95% confidence intervals.

- The *CAPIFlash Commitlog* shows 77% better throughput in (5% read - 95% update) workloads and 38% better throughput in (15% read - 85% update) workloads compared to Cassandra's durable alternative. However, with increasing read ratios *CAPIFlash Commitlog* loses its performance advantage over *Batch Commitlog*.

We showed the *CAPIFlash Commitlog* is a high-throughput, low-latency durability mechanism for Cassandra, compared with the existing commitlog mechanisms. *CAPIFlash Commitlog* is available on Github as an open source project for Cassandra versions *2.0.16* and *2.1.18*. [2]

In future work, we plan to investigate discrepancies in read latency of mixed read-write workloads in detail. We will also examine the design and analysis of CAPI-Flash based logging mechanisms on different data models and investigate opportunities to accelerate other components of Cassandra which involve heavy I/O and compute-intensive operations.

## REFERENCES


[1] Apache cassandra. http://cassandra.apache.org/.

[2] Capiflash commitlog source code. https://github.com/odaira/cassandra-capiflash/.

[3] Hbase. http://hbase.apache.org/.

[4] Making switch to cassandra from redis. http://www.planetcassandra.org/blog/interview/facebooks-instagram-making-the-switch-to-cassandra-from-redis-a-75-insta-savings/.

[5] Mongodb. http://www.mongodb.org.

[6] Redis. http://www.redis.io/.

[7] 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, pages 14–22, New York, NY, USA, 2013. ACM.

[8] 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.

[9] B. Brech, J. Rubio, and M. Hollinger. Data Engine for NoSQL - IBM Power Systems Edition. Technical report, IBM, 06 2015.

[10] 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 Association.

[11] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 73–86, New York, NY, USA, 2009. ACM.

[12] 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.

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

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[15] E. Dede, M. Govindaraju, D. Gunter, R. S. Canon, and L. Ramakrishnan. Performance evaluation of a mongodb and hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*, Science Cloud '13, pages 13–20, New York, NY, USA, 2013. ACM.

[16] 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.

[17] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. Pcmlogging: Reducing transaction logging overhead with pcm. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2401–2404, New York, NY, USA, 2011. ACM.

[18] T. Ivanov, R. Niemann, S. Izberovic, M. Rosselli, K. Tolle, and R. Zicari. Performance evaluation of enterprise big data platforms with hibench. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 2, pages 120–127, Aug 2015.

[19] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 529–540, New York, NY, USA, 2014. ACM.

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

[21] 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.

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

[23] 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.

[24] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[25] B. Wile. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. Technical report, IBM, 10 2014.