# The Freeze-Frame File System

Weijia Song[1], Theodoros Gkountouvas[1], Kenneth Birman[1], Qi Chen[2], and Zhen Xiao[2]
[1]*Computer Science, Cornell University* [2]*Computer Science, Peking University*

Submission Type: Research

## Abstract

Many applications perform real-time analysis on data streams. We argue that existing solutions are poorly matched to the need, and introduce our new Freeze-Frame File System. Freeze-Frame FS is able to accept streams of updates while satisfying "temporal reads" on demand. The system is fast and accurate: we keep all update history in a memory-mapped log, cache recently retrieved data for repeat reads, and use a hybrid of a real-time and a logical clock to respond to read requests in a manner that is both temporally precise and causally consistent. When RDMA hardware is available, the write and read throughput of a single client reaches 2.6G Byte/s for writes, 5G Byte/s for reads, close to the limit on the 56Gbps RDMA hardware used in our experiments. Even without RDMA, Freeze Frame FS substantially outperforms existing file system options for our target settings.

## 1 Introduction

Consider an Internet of Things application that captures data in real-time (perhaps onto a cloud-hosted server), runs a machine-learning algorithm, and then initiates actions in the real-world. Such applications are increasingly common: examples include the smart power grid, self-driving cars, smart highways that help vehicles anticipate rapidly changing road conditions, smart home and cities, and so forth.

This computing model creates new challenges, as illustrated in Figure 1. To create these three images we simulated a wave propagating in a fish-tank and generated 100 10x10 image streams, as if each cell in the mesh were monitored by a distinct camera, including a timestamp for each of these tiny images. We streamed the images in a time-synchronized manner to a set of cloud-hosted data collectors over TCP, using a separate connection for each stream (mean RTT was about 10ms). Each data collector simply writes incoming data to files. Finally, to create a movie we extracted data for the time appropriate to each frame *trusting the file system to read the proper data*, fused the data, then repeated. In the figure we see representative output.

The image on the left used HDFS to store the files; we triggered its snapshot feature once every 100 milliseconds, then rendered the data included in that snapshot to create each frame. This version is of low quality: HDFS is oblivious to timestamps and hence often mixes frames from different times. In the middle, we used our Freeze Frame FS (in the remainder of this paper, FFFS[1]), configured to assume that each update occurred at the time the data reached the data-storage node. On the right, we again used FFFS, but this time configured it to extract time directly from the original image by providing a datatype-specific plug-in.

If a file system can't support making a movie, it clearly couldn't support other time-sensitive computations. Today, the only option is to build applications that understand time signals in the data, but this pushes a non-trivial task to developers and makes it hard to leverage the huge base of existing cloud-computing analytic tools that just run on files. With FFFS the issue is eliminated.

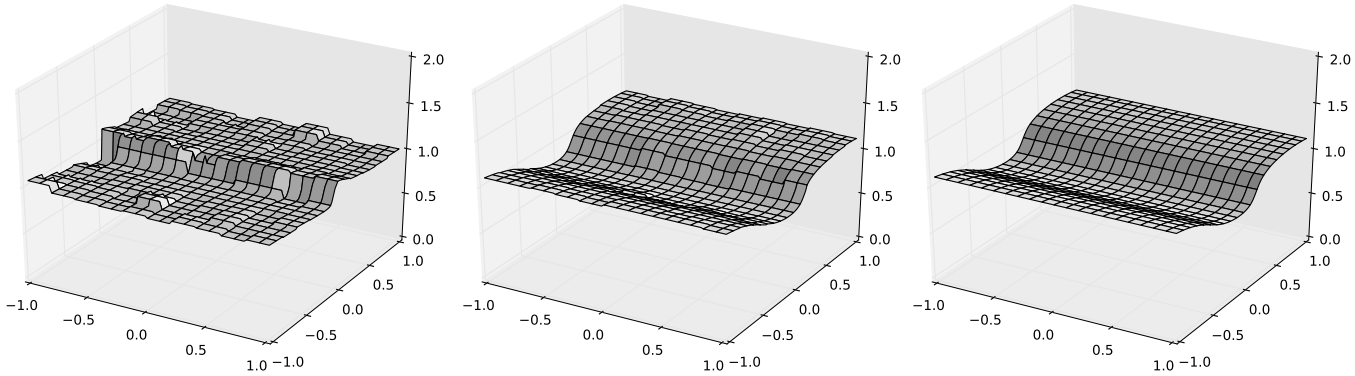
Our principle objectives are as follows:

1. High speed, crash-fault tolerance and scalability. When available, we wish to leverage remote direct-memory access: RDMA.
2. Support for temporal reads.
3. Determinism (repeated reads yield the same result), temporal accuracy and logical consistency [5].
4. Ease of use. POSIX programs use a file naming convention to issue temporal reads; time-aware programs use a new API that makes time explicit.

## 2 Background

Among existing file systems, snapshot capabilities are common, but few permit large numbers of real-time states to be captured. HDFS [32] limits snapshots to append-only files, requires that snapshots be preplanned, and creates them when the snapshot request reaches the HDFS

[1] Note that the Flash-Friendly File System is also abbreviated FFFS, but is completely unrelated to our work.



(a) HDFS (b) FFFS (server time) (c) FFFS (user time)

Figure 1: Reconstructed Wave. Click this caption to see the full animations.

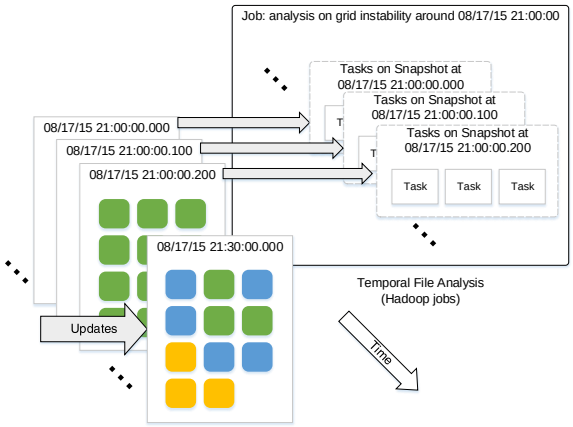



Figure 2: Temporal Parallelism with Hadoop

server, introducing inaccuracy. Ceph pauses I/O during snapshot creation, and snapshots are costly to create [35]. Indeed, while file-system snapshot features have existed for many years, such limitations are surprisingly common [13, 14, 26, 28, 30]. In contrast, FFFS offers a POSIX-compliant API with optional extensions for temporally-aware applications, uses a novel memory-mapped data structure to support high speed access, has a particularly efficient way of employing SSD for persistent storage, and is both temporally and causally consistent. Moreover, there is no need to preplan snapshot times: FFFS rapidly materializes data as needed.

## 2.1 Hadoop File System and Snapshot

Our work builds on the open-source HDFS [32] release: we retained its outer layers, but largely replaced the internals. HDFS has two subsystems: the *NameNode*, which implements the file system name-space, and a collection of *DataNode* servers. Individual files are represented as sets of data blocks, replicated across DataNodes so as to balance loads and achieve fault-tolerance. The NameNode manages file meta data along with the mappings from files to block lists. To access a file, a client contacts the NameNode to locate the block(s), then connects to the DataNodes, which handle read and write requests. The NameNode participates in operations that open, create, delete, or extend a file.

The existing HDFS system offers a restrictive form of snapshots at the directory level, but it was inadequate for our needs; we'll briefly explain the issue. Consider a client that requests a snapshot on directory "foo". The HDFS NameNode will create a new *diff* structure for the meta data of "foo". Initially this folder is empty, but on future file creation, extension or deletion events for a file in the directory, the event is logged into the *diff*. The client accesses the snapshot via a read-only virtual directory "foo/.snapshot/s0'.'. Snapshot creation costs are constant and low: in our testbed, 10ms for an HDFS cluster running on commodity servers. However, the approach does increase HDFS costs when reading files that belong to a snapshot and have changed since it was created: such actions must touch both the original file and the *diff*. On the other hand, because HDFS is limited to appends, the *diff* representation is simple.

We rejected the existing HDFS snapshot mechanism for several reasons. One is the append-only limitation: we believe that our users will require a standard file update API with arbitrary in-place file writes. A second concern is that HDFS treats the entire "session" from file open to close as a single atomic update, which it tracks as if the update had occurred *when the file is closed*. To compensate for this, the HDFS user would need to issue frequent file close and re-open operations, a costly overhead. Additional issues arise because HDFS snapshots are determined by the moment when the system

call was processed, and dependent on the delay between when the system call is issued and when the NameNode processes the request. In particular, if no snapshot was anticipated for a given time, that system state cannot later be accessed.

### 2.2 Hybrid Logical Clock

FFFS also builds on prior work to create clocks combining logical and real-time guarantees. This research dates back to Lamport's widely-cited 1978 paper [17], defining what he called *Logical Clocks* (LCs) and discussing their relationship with *Real-Time Clocks* (RTCs). For brevity we assume general familiarity Lamport's logical clocks, and instead focus on the *Hybrid Logical Clocks* (HLCs) used in our work. The idea was introduced by Kulkarn i [16] with the goal of using an RTC to timestamp events, but adjusting the RTC in such a manner that it would also possess the logical guarantees of a logical clock. An HLC starts with a synchronized RTC, but pairs the RTC with an LC in such a manner that if two events are more that $\epsilon$ time difference apart in real-time, they are ordered by RTC, and if not, the LC is used to break the tie (the actual rule is slightly more complicated and we refer readers to [16] for details). To synchronize clocks, we use the *Network Time Protocol* (NTP) [23] or the *Precision Time Protocol* (PTP) [18]. Nodes with malfunctioning clocks shut down.

## 3 Architecture

As shown in Figure 3, FFFS retains the basic architecture and API of HDFS, and is fully compatible with existing HDFS applications. However, the core of FFFS replaces the HDFS snapshot algorithm with our own mechanism.

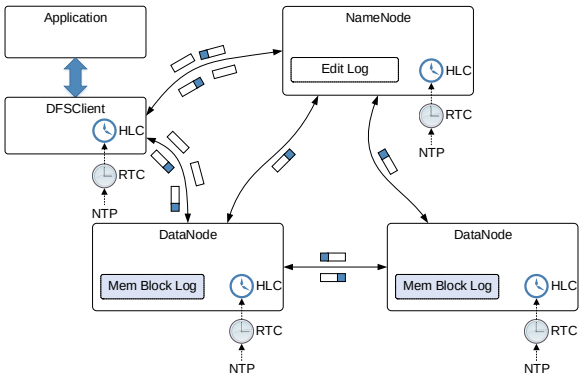



Figure 3: FFFS Architecture

FFFS has three types of nodes: clients, the NameNode, and the DataNodes. The NameNode and DataNodes each possess an RTC and HLC clock. The FFFS *Snapshot Agent* runs as a module within the NameNode and the DataNode. Snapshot requests are supported, but here we depart from the HDFS approach. In HDFS, the *time* of a snapshot is simply the time when the request reaches the NameNode and is processed. This is problematic, because there may be a significant delay between when the request is issued and when the snapshot is formed. Further, as we saw in Figure 1, an HDFS snapshot can mix contents from distinct states, or exhibit gaps in the causal message order.

In FFFS, we treat the snapshot as a kind of *dynamically materialized temporal query*: a view of the data in the system as of the time that the analysis program wishes to read it. In support of this new model, we changed the DataNode block storage structure into what the figure shows as the *Mem Block Log* (see Figure 3; details appear in Section 5). This log-structure is highly efficient for writes [25,29] and can support writes at arbitrary locations within files. We persist data to SSD storage, caching active data and our index structure in memory.

The NameNode *EditLog* is a data structure used by HDFS to recover non-persistent updates in the event of a system crash. In our design, we extend the EditLog to include the history of past NameNode states. Details and other design choices are discussed in Section 6.

## 4 The Snapshot Mechanism

An FFFS Snapshot materializes file system data at time $t$ from a collection of states of the NameNode and DataNodes. A snapshot should be temporally precise and closed under causality: if some event $X$ *happened before* some event $Y$ included in the snapshot, than $X$ must also be included. For example, suppose that event $X$ writes block $B$ of file $A$ at time $t$. If a snapshot for time $t$ includes data associated with $B$, but omits the creation of $B$ at the NameNode, the data for $B$ would be inaccessible even though it should logically be part of the time-$t$ snapshot. Conversely, if the snapshot includes $B$ but omits writes to a predecessor block (perhaps at some other DataNode), file $A$ will be left with gap or might exhibit a mashup of data from different points in time, as we saw in the reconstructed HDFS image in Figure 1. Thus, we need a *consistent snapshot* [5].

To construct such a snapshot, FFFS makes use of the HLC timestamps. Let's make the term *event* more precise: it will denote any operation that changes node status, including the sending or receiving of messages that will cause such an operation to occur. FFFS tags all such events with an HLC timestamp $(r, l)$, as detailed in Section 4.1. Further, each node logs all events that cause local state changes.

The Namenode and Datanode maintain mappings from real time $t$ to the latest event that *happens before* the

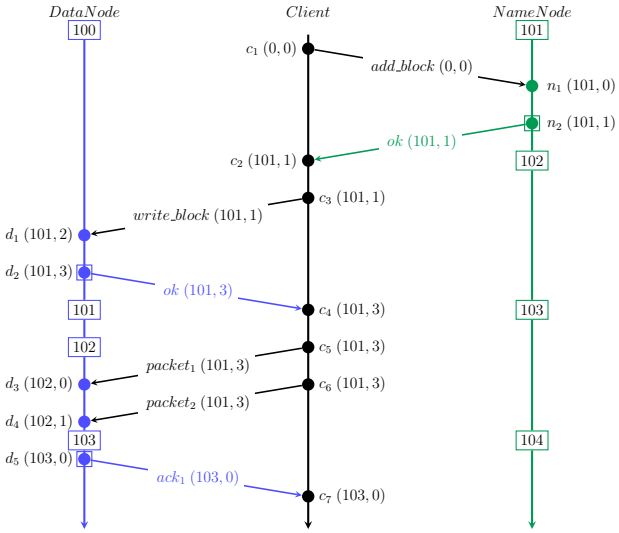



Figure 4: Events and Messages

event at HLC clock $t, 0$, inclusively. These mapping are computed on demand when the client request for the state at given point of time. Further, as noted earlier, FFFS does not require an explicit $createSnapshot$ API. Instead, the internal data fetch mechanism always uses time to satisfy reads, fetching the data applicable at any time point in the past. For a read at time $t$, our algorithm finds the logged event marked with value $r$ closest to, but not exceeding, $t$ in its log. We do maintain a form of secondary index to simplify accessing the blocks associated with a given time, and we call this a *snapshot data structure,* but it is a form of soft-state: a cached copy of information that can be recomputed whenever needed.

When a set of reads are issued at time $t$, FFFS snapshots will be both temporally accurate and also closed under the causal order. As shown in Kulkarni's work [16], the HLC-timestamped events closest to a given RTC value form a set having $\epsilon$-temporal precision and logical consistency: If real-time clocks are synchronized with maximum skew $\epsilon$) the $RTC$ value of all the processes for the snapshot request at time $t$ falls within $[t - \epsilon, t]$. Further, the method is deterministic: given the same NameNode and DataNode logs, the same result will always be computed for a given value of $t$. This is helpful when reconstructing FFFS state after a complete shutdown, and gives the system freedom to discard cached results.

## 4.1 Events and Messages

Our protocol extends the HDFS communication protocols to ensure that that requests and responses piggyback HLC timestamps. Consider the example of a block write shown in Figure 4. Notice that the client does not tick the clock, although it participates in propagating the maximum received clock value from other nodes (a write-only data source could thus just send its RTC and an LC of 0). The client first issues an *add_block* RPC call to the NameNode to allocate a new block. The HLC timestamp for the event and the message is initialized to $(0, 0)$. On receiving the RPC call, the NameNode local RTC clock is 101. The HLC for $n_1$ will have value $(101, 0)$. The NameNode next creates the metadata for the new block, then responds to the client($n_2$). In the example, this occurs quickly enough so that the RTC has not ticked, but we do tick the logical clock, as seen in the event diagram. Now, the client connects to the corresponding DataNode, issues a *write_block* RPC call and writes data to the new block.

In the above example, we only need to log three events: $n_2$, $d_2$, and $d_5$, because they represent the points where data is changed: on event $n_2$, the block metadata is created; on event $d_2$ the block is created in DataNode; and on event $d_5$ the first packet of data is written to the block. Events internal to the client are not given HLC timestamps because they are irrelevant to the filesystem states, and not every event occurring in the NameNode or DataNode causes data to change. For example, since read operations do not change file system states, we neither count read operations as events nor tick the HLC clock for them. In this way, we keep track of all state changes and minimize the number of events in our log: the shorter the log, the faster the state reconstruction.

In contrast, had we used the RTC when creating the snapshot, event $d_2$ would have been assigned timestamp 100, while event $n_2$ (which casually precedes event $d_2$) would have been assigned timestamp 101. This would have resulted the inclusion of data from a block present in the DataNode, but not the corresponding metadata event in the NameNode: one of the two examples of inconsistency mentioned above. It is equally easy to construct examples in which naive use of real-time clocks results in mashed up snapshots containing data from two or more distinct states.

## 4.2 The Quality of the Snapshots

In many real-time settings, a precise estimate of the snapshot quality is used to parameterize computation on the snapshot data. Our method lends itself to such an analysis. For example, if a system were to use Google's *TrueTime* clock synchronization method, as implemented in Spanner [7], it would be possible to constrain time skew between the NTP master and slave to less than 5ms with 99.9% confidence level or 1ms with 99% confidence level. The maximum skew between two slaves will be 10ms at 99.9%, or 2ms at 99%. The FFFS snapshot will then have 10ms temporal precision with 99.9%, and 2ms precision with 99% confidence.

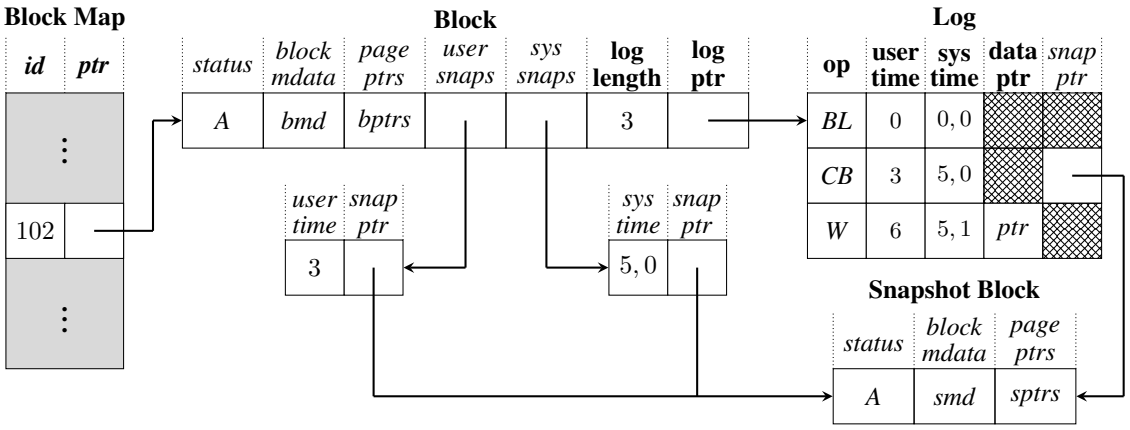



Figure 5: Multi-Log Structure inside DataNode

# 5 Multi-Log Structure

FFFS needs the history of each file block to reconstruct past versions of data blocks. To this end, each block is represented by a separate log within the FFFS *Multi-Log* structure. The Multi-Log is stored in memory and asynchronously flushed to non-volatile storage. We use it to implement both normal POSIX file system operations and temporal reads and writes.

The Multi-Log is shown in Figure 5. Portions of the state that we persist to SSD are highlighted in bold; the remainder is cached state that can be reconstructed as needed. Each DataNode maintains a set of active *Blocks* along with previously deleted ones (for snapshot reads). The *Block Map* is, as the name suggests, a map from *Block IDs* to Blocks. Every Block keeps a *Log*, which maintains all the updates in the Block, as well as the *Log length*. Each log entry has four fields: a) the *type* of an operation (*Beginning of Log*, *Create Block*, *Write*, *Delete Block*), b) the User timestamp (if it exists) derived from the actual data being written, c) the HLC timestamp corresponding to the operation, and d) in case of a write operation, metadata (offset and length of the write, etc.) along with pointers to the data.

We keep the block contents in separate *FFFS pages* which are stored on SSD, paged into memory on demand, and then cached for quick access on repeated reads. Every write operation consumes some integral number of FFFS pages; this reduces memory fragmentation and allows efficient management of data, but incurs space overhead, since even a small write requires a full page. By default, we use the operating system page size, which yields high read/write performance, but FFFS can be configured to use other values.

## 5.1 Block Operations

FFFS must append every update operation as a Log entry to the corresponding Log, including the user-specified timestamp (if any), and the server's local timestamp. Since a write operation can partially edit a page, a backward search seemingly required, to find the previous version of the page, but if naively implemented, such a search would have cost O(Log Length). To avoid this overhead, FFFS maintains a number of hints and secondary indices. The Block structure contains a *status* fields that takes (A)ctive and (D)eleted as values, as well as some *block metadata* that includes the block length, last update timestamps, etc. Additionally, it maintains an array of page pointers where for each page, the corresponding entry points to the latest updated data. Notice that there is no data duplication, since these are soft links (pointers) to the actual data. Update operations keep these fields current. Using them, we are able to reduce the update cost to a constant delay.

A similar problem arises when satisfying temporal reads. Suppose a read needs data from a Block at a specific point $t$. Here, FFFS must find the most recent update to that block, and if that was the result of a series of partial updates to other preexisting blocks, must also locate those. To facilitate this operation, FFFS caches all the needed information needed in *Snapshot Blocks*. In case of a temporal read, FFFS first checks the snapshot index. If there is a snapshot associated with the last entry, FFFS updates the snapshot entry in the Block Structure and performs the operation from the corresponding Snapshot Block (avoiding the major overhead of a backward search). If the block is not in cache, FFFS carries out a binary search to find the last Log entry in this snapshot. Notice that reads at different time will often be satisfied from a single snapshot, and reads from different snapshots will often be satisfied from the same cached block, because blocks change only when

updated. For example, in Figure 5, user time 3 and system time $5, 0$ lead to the same snapshot.

### 5.2 Data Persistence

FFFS stores data to a non-volatile medium for fault-tolerant persistence. Additionally, the persistent store allows FFFS to manage much larger amounts of data than can fit in DRAM. Our eviction policy uses a simple FIFO rule: younger pages are retained, and older pages are evicted in age order. Later, if an evicted page is reaccessed and not available in the block cache, FFFS will reload it from persistent storage.

It is not completely obvious how to handle I/O to the persistent store. A first idea was to use memory-mapped files for the FFFS page files and logs, employing a background thread to flush data periodically to the non-volatile storage. However, while an occasional DMA write to SSD might seem like a minor cost, we found it difficult to control the time and order when data to be flushed to disk[2]. Further, cases arose in which our page cache flusher needed to lock a page but by doing so, prevented other writes from creating new log entries, a highly disruptive situation.

Accordingly, we shifted to a different approach. FFFS utilizes semaphore-guarded event queues to synchronize the data persistence thread (the consumer) and writers (the producers). After a writer finishes writing a block, it posts an event to the corresponding queue (selection with the block ID), which contains 1) a pointer to the block entry being written and 2) the number of log entries in its log. The data-persistence thread reads from the queue and flushs the log entries in FIFO order. For each log entry, its *FFFS pages* are flushed first, so that a persistent entry will always be complete. By shifting persistence off the critical path, we gain substantial speedup. The approach generates very random IOs, and in our experience performs poorly with rotational disks; our evaluations focus on SSD, where this effect is not observed (good performance would also be seen with new technologies like PCM [36] and ReRAM [4]). A further optimization is to use multiple queues and persistence-threads to maximize IO parallelism.

Upon recovery from a failure, a DataNode reconstructs its memory structures from persisted data. Notice that (as with other file systems), data that has not yet been persisted could be lost in a failure. Accordingly, if an application needs to be certain that its state is safely persisted, a *fsync* operation should be issued to flush any pending writes.

[2]In Linux, the behavior of *pdflush* kernel threads are controlled by six thresholds based on the volume and age of page cache [1]. Those kernel actions can be triggered by other applications.

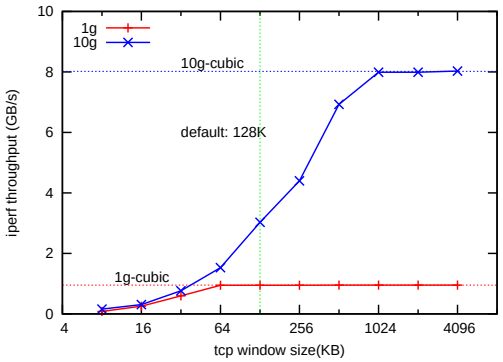



Figure 6: TCP Window Size and Throughput

## 6 Additional Design Choices

### 6.1 Random Write Support

Whereas HDFS only permits appends, FFFS allows updates at arbitrary offsets within files. To support this, we enable use of the *seek* system call, which the HDFS API includes but does not actually support (it returns an error unless the seek was to the end of the file). On each seek, the client library checks whether the request moves the file pointer to a new block or not. If the pointer is in the same block, we just update the pointer reuse the data transfer connection to the datanode. Otherwise, we close the current DataNode connection and start a new connection to a DataNode that contains the target block. Note that we currently do not allow seeking beyond the end of file, which leaves a hole in the file.

HDFS uses a fixed 128KB TCP window size for data transfer between a client and a datanode, or two data nodes, selected to maximize throughput in Gigabit networks. However, a much larger window size is needed to saturate a 10 Gigabit NIC. We used the iperf benchmark [34] to test throughput between two servers with both Gigabit and 10 Gigabit NICs. As shown in Figure 6, the Gigabit NIC saturates when the TCP window size reaches 64KB, whereas the 10 Gigabit NIC doesn't saturate until the TCP window size reaches 1MB. To improve throughput, instead of using a fixed TCP window size, we choose to allow the TCP congestion control algorithm to adapt to available bandwidth. CUBIC [11], the default algorithm in Linux, finds this saturation point quickly.

### 6.2 Throughput and Latency Optimization

The HDFS DataNode uses a single thread to handle each write operation. It first reads a data packet from the network, writes to disk, and then reads the next data packet. To improve performance, FFFS uses two threads for this purpose: One reads network data, writing it to a

circular buffer, and the other reads from the buffer and writes to the blog. In our evaluation, we refer to this as the *write pipeline*.

For embedded real-time applications, low latency is sometimes more important than throughput. Accordingly, we introduced a *latency priority* mode to FFFS. When *latency priority* is requested, we disable Nagle's algorithm in TCP [24]. We also disable buffering in the client library so that the written data is sent as soon as it is written. These changes reduce throughput because less pipelining occurs, but also reduce delay.

Copying data, particularly for large files, introduces huge overheads. To avoid these high costs, HDFS uses the zero-copy *sendfile* system call to accelerate reads. Once a block file is cached, *sendfile* can send data from cache to NIC driver directly, avoiding data copying between kernel and user spaces. To leverage this API, we create a file in *tmpfs* [33] and map it in the address space of FFFS datanode process. This enables *sendfile* for reads from remote clients. With this approach, data can be accessed both as a file, and as a memory-data structure, without any need to duplicate data and with no loss of performance when bulk transfers occur.

### 6.3 RDMA Transport Layer

Cloud computing systems such as Azure increasingly offer RDMA [6, 12, 20, 27] data transfer via RoCE or InfiniBand network switches and NICs, and we also wished to leverage this capability, when available. Accordingly, FFFS interrogates the network interfaces and, if RDMA is enabled, uses RDMA instead of TCP/IP to move data. Our experience shows that, even when jumbo frames and the TCP offload Engine are both enabled, a TCP/IP session will only achieve 40Gbps if running over a network that uses Ethernet standards (for example, on a 100Gbps Ethernet, our experiments showed that TCP over IP peaked at 40Gbps). In contrast, using RoCE, we can easily achieve a read/write throughput at 97Gbps, very close to the hardware limit. The FFFS RDMA transport layer moves data at RDMA speeds whenever the hardware is available on both the client and server platforms, reverting to TCP if one or both lacks RDMA capable NICs.

RDMA supports two modes of operation: the so-called one-sided and two-sided cases. Both involve some set-up. With one-sided RDMA, one node grants the other permission to read and write in some memory region; two-sided RDMA is more like TCP, with a connection to which one side writes, and that the other reads. The actual operations are posted asynchronously and carried out using reliable zero-copy hardware DMA transfers. Our design uses the one-sided mode, with the FFFS DataNodes initiating all RDMA operations.

### 6.4 External time sources

Notice that when capturing data from a sensor, the resulting update may contain temporal data of higher quality than anything accessible within our system. For demanding real-time applications, it would be of interest to extract these timestamps. We introduce a per-datatype plug-in to FFFS, which parses and extracts the user timestamps from data records. On processing the data stream, each record is tagged with both Kulkarni HLC timestamp and the user timestamp, shown as 'sys time' and 'user time' in Figure 5. Both TCP and RDMA preserve FIFO order, hence timestamps of legitimate records will increase monotonically. FFFS protects itself against faulty writers by rejecting any record carrying a user timestamp lower than a previous one. FFFS also tracks the platform time at which records were received and written. Note that because FFFS tracks both user time and system time, a temporal read can specify which is preferred.

## 7 Evaluation

We deployed FFFS in a variety of environments to confirm that our solution is portable and to evaluate performance. Our microbenchmarks were conducted on a 19-node cluster called Fractus deployed in the Department of Computer at Cornell. Each Fractus server is equipped with one 10 Gigabit NIC connected to a 10 Gigabit Ethernet environment, has 96GB RAM and two Intel E5-2690 CPUs(8 cores each), and runs Linux 3.13. For persistent storage, our servers are equipped with a 240GB PCIe SSD card containing four drives, each 60GB in size. The peak throughput of any single drive is 450MB/s. We configured one server as an NTP server, and synchronized the others against it. Time offset between the NTP server and the others was monitored using the NTP query tool(`ntpq`); we found that our nodes were synchronized with a time skew of no more than 1ms.

Cloud platforms such as Microsoft Azure are now offering RDMA over both Infiniband and ROCE (fast Ethernet). To evaluate FFFS performance over RDMA, we equipped Fractus with Mellanox 100 Gbps RDMA dual-capable NICs, and installed two 100Gbps Mellanox switches, one for ROCE and one for Infiniband. For the work reported here, both yielded identical performance.

We also include data from some experiments conducted using the U. Texas Stampede system, an HPC cluster with thousands of nodes. Each Stampede node has 32GB memory and two 8-core CPUs and SSD storage units. The nodes are equipped with a 56 Gigabit InfiniBand adapter with RDMA support. Finally, we tested FFFS in a private Cloud service at Cornell, which is configured to mirror the Amazon EC2 environment. This private Cloud runs

virtual machines on KVM hypervisor.

Unless otherwise specified, FFFS and HDFS both used 64MB blocks. Data replication and checksums are disabled in HDFS. The FFFS page size is 4096 Byte.

## 7.1 Snapshot Quality

Our first experiment was the one used to create Figure 1. Here we used a water wave simulator [2] to generate a dataset representing how the height of surface will change during a wave propagating in a square fish-tank. The dataset contains surface height values that we translated into a 100x100 mesh of small 10x10 images, each containing the exact simulation time, at a rate of 20 samples per second.

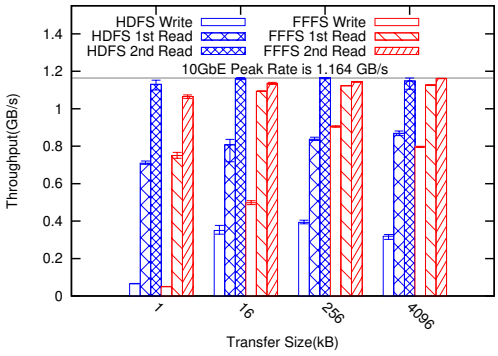
We ran this experiment in our private Cloud. In both the HDFS and FFFS deployment, one NameNode and six DataNodes exclusively run in seven high performance virtual machines, each of which has 4 CPU cores and 16 GB memory, and with clocks synchronized using NTP. The 100 client applications run in 10 virtual machines, each of which has 1 CPU and 4GB memory (had the data been captured by IoT sensors, network delays would have created even greater latency variations). For HDFS, we run an additional thread to create a snapshot every 100ms. In this case, each time we write a data record to HDFS, we close and re-open the file to force an update to the file metadata in the NameNode.

Then we reconstruct three movies containing 100 frames each: a) HDFS snapshots, b) FFFS state by 100ms interval according to HLC timestamp, and c) FFFS state by 100ms interval according to user timestamp. respectively. Figure 1 shows the 37-th frame from each of the three movies; by clicking the caption, the reader can see the full animations. With HDFS updates to the HDFS NameNode metadata are too slow and data piles up, causing bursty writes, and the snapshot operations suffer delays. We obtain snapshots that mix data from different points in time and might not be closed under the causal *happens-before* relation.

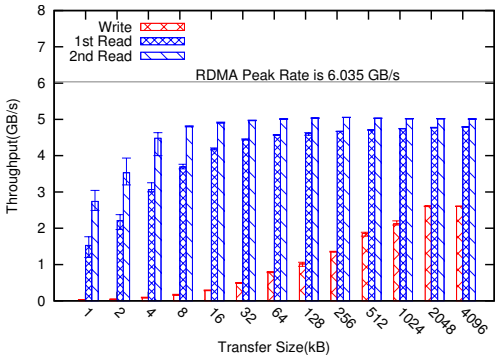
FFFS does far better: by relying on its temporal read feature, it extracts precisely the data written. Here we see both the platform-time case (with updates timestamped using the DataNode server clocks) and user-time (with update times extracted from the records themselves). The small distortions in Figure 1b arise from network latency and scheduling skew. Figure 1c is perfectly accurate.

## 7.2 Read and Write Performance

Next we evaluated the FFFS read/write performance, comparing this with that of HDFS. All persistent file data is written to SSD. For this experiment, we limited the system to network IO over our 10 Gigabit Ethernet, and




(a) TCP/IP on 10GbE



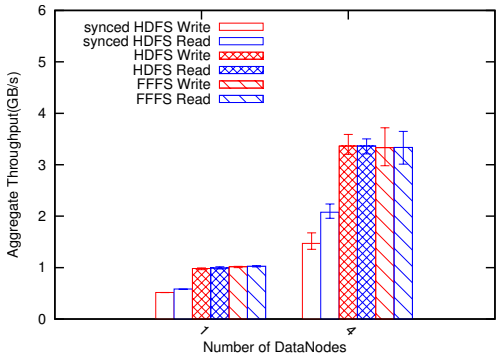

(b) RDMA on 56Gb IB

Figure 7: Throughput Perfromance

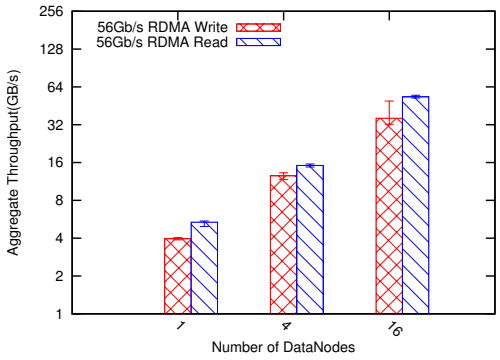
disabled replication and data checksum in HDFS. In this mode, both HDFS and FFFS transfer data over TCP.

Our experiment launches a single client that writes at full speed for ten seconds. Then some other client reads the file. The peak read and write throughput is calculated by dividing the file size by the time spent reading or writing. Figure 7a shows how throughput changes as a function of data packet size. When the data packet size is small (1KB), overheads of our communication layer dominate. Once the data packet size grows larger, the bottleneck moves to the IO system. The HDFS DataNode invokes *write* system calls to persist data. Although HDFS does not sync its writes immediately, the kernel will soon do so, and we see that write performance converges to the SSD I/O rate. In contrast, the FFFS DataNode has a separate persistent thread that flushes data into disk, without blocking the data-receiving path. Therefore, FFFS hits a different and higher bottleneck: the network. This explains why, when data packet size is 256KBytes, the FFFS write achieves 900 MB/s while the HDFS write is only 393 MB/s.

Both HDFS and FFFS benefit from caching on re-reads. To quantify this effect, we read each file twice with the data packet size fixed at 128KB, which is the optimal size for a 10G network. The first HDFS read is slower




(a) TCP/IP on 10GbE




(b) RDMA on 56Gb IB

Figure 8: Bursting Scalability

because the data is not yet fully cached. FFFS has a further overhead: although the data blocks will still be in memory, when these first reads occur, FFFS will not yet have constructed its snapshot data structures. On the other hand, with larger writes the log will have a small number of records, and hence the first FFFS read performance improves as the write transfer size grows. The overhead of constructing the index is acceptable: for writes larger than 16KB, it slows down the read by less than 4%. Once all data is cached in HDFS, and in the case of FFFS we have also constructed the indices, the read performance(second read) rises, reaching the limit of a 10 Gigabit NIC, which is about 1.164GB/s (measured by *iperf*).

Next, we looked at the benefit of RDMA. For this, we ran our experiments on the U. Texas Stampede cluster. Recall that Stampede has smaller per-server memory limits, hence we reduce the write duration from ten seconds to three seconds. This amount of data will fit in memory, hence we are not rate-limited by I/O to the SSD storage devices, but on the other hand, writes still involve more work than reads. We configured our RDMA layer to transfer data in pages of 4KB, and used scatter/gather to batch as many as 16 pages per RDMA read/write. Figure 7b shows that the FFFS read and write throughput grows when the data packet size used in write increases, similar to what was seen on Fractus. When our clients write 4MBytes per update, FFFS write throughput reaches 2.6 GB/s, which is roughly half the RDMA hardware limit of 6 GB/s (measured with *ib_send_bw*). FFFS read throughput reaches 5.0GB/s.

We tested the aggregate throughput by using multiple clients to saturate the two file systems. Here, we used six times as many clients as DataNodes. We run 3 client instances per node, in the same cluster with the same hardware configuration as for the DataNode servers. For the write test, each client writes 1GB data to a file at full speed. We start all clients at the same time and wait till the last client finishes. The aggregate write throughput is calculated as 6GB times the number of DataNodes and divided by the test time span. The read throughput is computed in a similar manner, except that the page cache is evicted prior to the first read to ensure a cold-start. We ran this experiment at different scales in both Fractus and the Stampede HPC cluster. The experiments scales up to a 4-DataNode setup in Fractus, consisting of 1 NameNode server, 4 DataNode servers, and 8 client servers. On Stampede. we scaled up to a 16-DataNode setup consisting of 1 NameNode, 16 DataNode servers, and 32 client servers. We compared FFFS with two configurations of HDFS: one in which we used HDFS in its normal manner. Here, HDFS will not immediately flush data to disk even on file close. In the second case we illustrate a "synced" HDFS, which flushes data to disk on the fly, and then removes pages from its cache to maximize available memory for new incoming data.

Figure 8a shows the Fractus experiment results. The aggregate throughput grows in proportion to the number of DataNodes. FFFS and HDFS both achieve nearly the full 10G network throughput. In contrast, synced HDFS is half as fast: clients must wait for data to be flushed to disk, causing the SSD I/O to emerge as a bottleneck. Figure 8b shows the HPC results for FFFS (HDFS doesn't have native support for RDMA). Notice that the Y axis is log scaled. Again, we can see that file system throughput grows in proportion to the number of DataNodes, confirming that FFFS scales well.

## 7.3 Data Persistence

Next, we ran a single client writing to FFFS on Fractus over its 10 Gigabit Ethernet, to show the influence of data persistence on write performance. The FFFS setup is configured with one DataNode. We tested FFFS write throughput in two scenarios: a) SSD Persistence: data is flushed to SSD; b) No Persistence: data is not flushed to persistent storage at all. We started a single client writing a file at full speed for ten seconds. We run the test for five times for each scenario and calculate the average and error bars. Figure 9 shows how the write throughputs

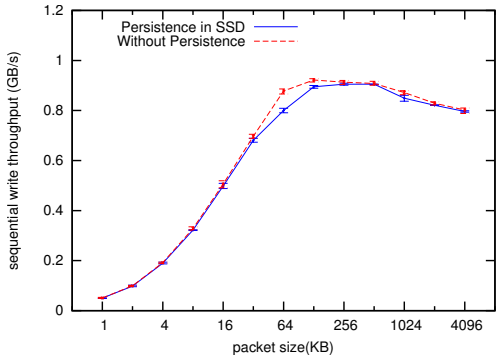
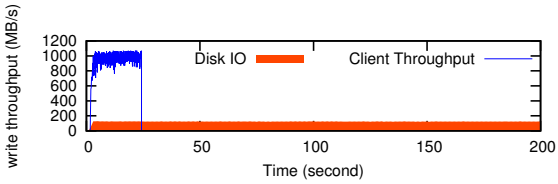
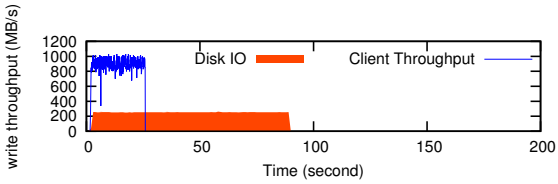


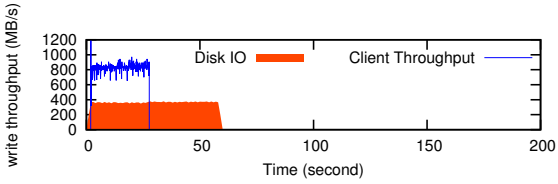

Figure 9: Persistence Overhead




(a) Packet size: 128KB




(b) Packet size: 512KB




(c) Packet size: 2MB

Figure 10: DISK I/O Pattern

change with transfer data size. We find no significant throughput differences with or without data persistence. This demonstrates that critical path latency is not affected by speed of the slower persistence thread.

We monitored the disk IO pattern to understand how data is flushed to disk. Figure 10 shows the results when data packet size is 128KB, 512KB, and 2MB, where X is the time axis and the Y-axis shows the throughput seen by the client (red), side by side with that measured by *io_stat* (blue), a disk IO profiling tool we launched on the DataNode. Although the client always sees optimal end-to-end throughput, the background disk I/O rates tell a very different story. The larger the transfer packet size is, the higher disk I/O rate it can achieve. This is because the transfer packet size determines the data

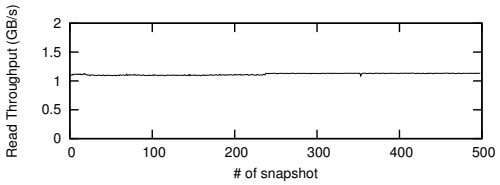



Figure 11: Read From Snapshot

size of a log entry, which in turn determines the amount of data simultaneously flushed [3]. Flush size smaller than 2MB cannot saturate the I/O bandwidth of our SSD device, causing the slow disk I/O rates in figure 10a and figure 10b. We believe that a a batched flush would improve the disk I/O rate and plan to add this feature in the future.

## 7.4 Block Log Overhead

FFFS snapshots have no particular overhead, but our algorithm does construct a soft-state snapshot index structure via log traversal on demand. To evaluate the snapshot construction overhead, we started a client that issues 500,000 randomly located writes to a 64MB file. Each random write overwrites a 32KB file region. Here we deployed FFFS on Fractus with 10 Gigabit Ethernet. Notice that the block size is 64MB so that the file is exactly in one block and all writes go to the same block log. The page size is 4KB so each write will create a log with 8∼9 pages of new data. We then start a client to read 500 file states after every 1000 random writes. The read throughput measurements are shown in figure 11. Snapshot 0 is the earliest snapshot. The read throughput is stable at 1.1 GB/s, which is close to the hardware peak rate (1.164 GB/s). This supports our assertion that snapshot construction is highly efficient.

We should mention one small caveat: we have not yet evaluated the FFFS data eviction scheme. With very large amounts of memory in use, some data pages might need to be loaded from slower persistent storage. Because our eviction algorithm keeps younger data in memory, we expect that this will result in a performance drop for access to very old data, but would have no impact on access to more recent data.

We also use the single DataNode FFFS setup to test the memory overhead of the blog. We write a total of 128MB to a file and measure the total memory use of the entire file system. The packet size is configured to 64KB, the default value used by HDFS. We used random writes with varying sizes as well as sequential writes with different page size configurations, subtracted the initial memory usage, and then normalized the result relative to

[3] We do this to guarantee that, when a DataNode crashes, no log entry is lost if its timestamp is earlier than that of some log entry in SSD.

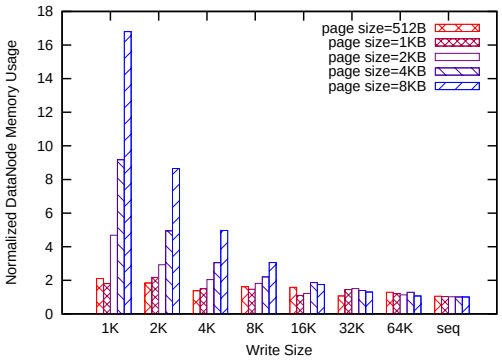



Figure 12: DataNode Memory Overhead

the data size, as shown in Figure 12. The x-axis shows how much data is written at a random location. Write size 'seq' represents sequential write. When the write size is small, the memory usage is high. This is because the blog manages data with page granularity, hence even a small update occupies at least one page. This is also why, when the write size is small, smaller page sizes are more space efficient. On the other hand, small pages require more page pointers in the blog. It turns out that pointer overhead is of minor consequence: 512 byte pages used only 5% more memory, 1KB pages require 2.8% more memory, and 4K pages require just .1% memory overhead.

# 8 Related Work

Our work falls into a richly populated space that includes prior research on distributed clocks, snapshottable file systems and in-memory storage systems.

## 8.1 Distributed Clocks

Lamport [17] introduced logic clocks as a way to reason about causality in distributed systems. However, as discussed in [16], naive merging of LC and RTC values can lead to unbounded drift of the clock from the RTC values under certain circumstances. The HLC algorithm we used, introduced by Kulkarni, avoids this problem. The *Vector Clock* (VC) [22] represents causal order more accurately than a basic LC, but is not needed in our setting: in effect, the HLC timestamps on the blogged events capture all the information needed to achieve an accurate snapshot. Google's Spanner system uses a scheme called *True Time* (TT) [7]. TT requires more accurate clock synchronization than HLC, and Spanner sometimes needs to delay an event if the clock of the sender of a message is ahead from the receiver. Orbe's *Dependency Matrix Clock* (DM-Clock) [9] also suffers from this problem.

## 8.2 Snapshottable File Systems

HDFS originated as an open-source implementation of Google's GFS [10], a fault-tolerant file system that runs on commodity hardware. In consequence, GFS has a design similar to ours and could probably be extended to use our techniques: there is one master server, responsible for metadata management, and many chunk servers storing file blocks/chunks. As currently implemented, GFS supports directory-based snapshots, created as a delta over the directory metadata. In-place file modifications are not permitted for files included in snapshots. The snapshot implementation in HDFS is due to Agarwal etc, and is discussed in more detail in [3].

The Elephant file system [30] enables the user to back up important versions of their files, intended as a way to back out of erroneous file deletions or overwrites. It extends the inode and directory data structures to store version history. Elephant is implemented in FreeBSD kernel and exposes a set of new system calls. In contrast to FFFS, this work is a purely single node solution .

Ext3cow [26] is an extension to the ext3 file system providing functionality similar to the Elephant File system. Unlike elephant file system, ext3cow has better compatibility since its snapshot is managed in user space and preserves the VFS interface. Like elephant, ext3cow is a single node solution. The B-Tree file system(Btrfs) [28] was designed to be Linuxs default filesystem. Btrfs uses B-tree forest to manage all objects in a file system. This facilitates Copy-on-Write(COW) on updates. Avoiding in-place write helps Btrfs achieve atomicity and support snapshots, although the metadata is significantly more complicated. Btrfs supports snapshot on subvolumes. Both of these systems are traditional disk file systems, and the snapshot is limited to append-only operations. Like HDFS, both treat operations between open, flushes, and close as a transaction that is logged at the time of a flush or close. Note that flush isn't totally under user control and can also be triggered by a file IO buffer, output of a newline, or other conditions. An Ext3cow snapshot is read-only, whereas BTRFS actually allows writes to a snapshot: an application can "change the past". A concern is that such an option could be misused to tamper with the historical record of the system.

Other snapshottable file systems include WAFL [13], a product from NetApps. It provides periodic snapshots for recovery purposes. Similar to ext3cow, the WAFL snapshot mechanism is based on Copy-On-Write inodes. WAFL supports only at most 20 snapshots. Coda [31] replicates data on different servers to achieve high availability, and this represents a form of snapshot capability; the client can continue working on cached data

during network disconnection and Coda resynchronizes later. Coda tracks modification in the file versions to detect write-write conflicts, which the user may need to resolve. The Ori file system [21] enables users to manage files on various devices. It captures file history information in a revision control system, and has a flexible snapshot feature similar to ours. However, this solution is focused primarily on small files in a multi-device scenario where offline updates and split-brain issues are common. Our data-center scenario poses very different use cases and needs.

### 8.3 In-Memory Storage

Spark, which uses HDFS but keeps files mapped into memory, is not the first system to explore that approach. Other memory-mapped store systems include RamCloud [25] is an in-memory distributed K/V store, FaRM [8], which is also a K/B store, and Resilient distributed datasets (RDD) [37], which solves speed up distributed computing frameworks by storing the intermediate results in memory. The Tachyon file system [19] is one member of the UC Berkeley stack. It is an in memory file system and implements a HDFS interface. Tachyon does not use replication for fault-tolerance but use lineage to reconstruct lost data, like RDD. In our understanding, Tachyon is more like a memory caching tool than a complete in-memory file system in that it only hold the working set in memory.

### 8.4 Others

Spanner [7] is Googles geographically distributed, transaction capable, semi-relational database storage. Spanner uses high precision clocks (GPS and atomic clocks) to make transaction timestamps. This enable a reader to perform lock-free snapshot transactions and snapshot reads. We did not feel that direct comparison would be appropriate: Spanner emphasizes geo-replication, and FFFS is intended for use in a data center. Kafka [15] is a log-structured storage system and can support many of the same features as FFFS. However, it is a real-time system, and is not optimized for highly parallel access to huge numbers of snapshots and would not perform well in our target environments.

## 9 Conclusions

Our paper presented the design, implementation, and evalutation of the Freeze-Frame File System. FFFS is able to accept streams of updates from real-time data sources while supporting highly parallel read-only access to data from backend computations running on platforms like Spark (the in-memory Hadoop). The FFFS read API supports access to large numbers of consistent and lightweight snapshots, with tens of milliseconds granularity, enabling a new type of temporal analysis that can access data in time even as the computation is spread over large numbers of nodes for speed. Our approach offers both causal consistency and a high degree of temporal precision. We use in-memory log-structured storage, so that snapshots can be created very rapidly; the associated data is materialized on demand with small overhead, and the algorithm is deterministic, so that the same request will return the same result even if our helper data structures are no longer available. Future work will add a comprehensive security architecture (including tamper-resistance), data replication features, and a remote backup capability.

## 10 Acknowledgements

We are grateful to Robbert Van Renesse, Ittay Eyal, Hakim Weatherspoon, Alan J Demers, Zhiming Shen, and David Bindel for their comments and suggestions. Our research was supported in part by grants from the NSF, DARPA and DOE. Mellanox provided access to RDMA hardware. Our experimental work was made possible by access to the U. Texas Stampede large-scale computing cluster, and in ongoing work, to Microsoft Azure's containerization environments.

## References

[1] Flushing out pdflush. `https://lwn.net/Articles/326552/`. Accessed: 2016-05-04.

[2] Shallow water simulator. `https://github.com/dbindel/water`. Accessed: 2016-05-04.

[3] AGARWAL, S., BORTHAKUR, D., AND STOICA, I. Snapshots in hadoop distributed file system. Tech. rep., UC Berkeley Technical Report UCB/EECS, 2011.

[4] BAEK, I., PARK, C., JU, H., SEONG, D., AHN, H., KIM, J., YANG, M., SONG, S., KIM, E., PARK, S., ET AL. Realization of vertical resistive memory (vrram) using cost effective 3d process. In *Electron Devices Meeting (IEDM), 2011 IEEE International* (2011), IEEE, pp. 31–8.

[5] CHANDY, K. M., AND LAMPORT, L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 63–75.

[6] COHEN, D., TALPEY, T., KANEVSKY, A., CUMMINGS, U., KRAUSE, M., RECIO, R., CRUPNICOFF, D., DICKMAN, L., AND GRUN, P. Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options. In *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on* (2009), IEEE, pp. 123–130.

[7] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK,

M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google&Rsquo;s Globally Distributed Database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013), 8:1–8:22.

[8] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 401–414.

[9] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 11:1–11:14.

[10] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[11] HA, S., RHEE, I., AND XU, L. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 64–74.

[12] HILLAND, J., CULLEY, P., PINKERTON, J., AND RECIO, R. RDMA protocol verbs specification. *RDMAC Consortium Draft Specification draft-hilland-iwarp-verbsv1. 0-RDMAC* (2003).

[13] HITZ, D., LAU, J., AND MALCOLM, M. A. File System Design for an NFS File Server Appliance. In *USENIX winter* (1994), vol. 94.

[14] HOWARD, J. H., ET AL. *An Overview of the Andrew File System*. Carnegie Mellon University, Information Technology Center, 1988.

[15] KREPS, J., NARKHEDE, N., RAO, J., ET AL. Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of the NetDB* (2011), pp. 1–7.

[16] KULKARNI, S. S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical Physical Clocks. In *Principles of Distributed Systems*. Springer, 2014, pp. 17–32.

[17] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21*, 7 (July 1978), 558–565.

[18] LEE, K., EIDSON, J. C., WEIBEL, H., AND MOHL, D. IEEE 1588-standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *Conference on IEEE* (2005), vol. 1588, p. 2.

[19] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 6:1–6:15.

[20] LIU, J., WU, J., AND PANDA, D. K. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming 32*, 3 (2004), 167–198.

[21] MASHTIZADEH, A. J., BITTAU, A., HUANG, Y. F., AND MAZIÈRES, D. Replication, History, and Grafting in the Ori File System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 151–166.

[22] MATTERN, F. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms 1*, 23 (1989), 215–226.

[23] MILLS, D. L. Internet Time Synchronization: The Network Time Protocol. *Communications, IEEE Transactions on 39*, 10 (1991), 1482–1493.

[24] MINSHALL, G., SAITO, Y., MOGUL, J. C., AND VERGHESE, B. Application Performance Pitfalls and TCP's Nagle Algorithm. *SIGMETRICS Perform. Eval. Rev. 27*, 4 (Mar. 2000), 36–44.

[25] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev. 43*, 4 (Jan. 2010), 92–105.

[26] PETERSON, Z., AND BURNS, R. Ext3Cow: A Time-shifting File System for Regulatory Compliance. *Trans. Storage 1*, 2 (May 2005), 190–212.

[27] RECIO, R., CULLEY, P., GARCIA, D., HILLAND, J., AND METZLER, B. An RDMA protocol specification. Tech. rep., IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress), 2005.

[28] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage 9*, 3 (Aug. 2013), 9:1–9:32.

[29] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-structured File System. *SIGOPS Oper. Syst. Rev. 25*, 5 (Sept. 1991), 1–15.

[30] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to Forget in the Elephant File System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1999), SOSP '99, ACM, pp. 110–123.

[31] SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. Coda: A highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on 39*, 4 (1990), 447–459.

[32] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (May 2010), pp. 1–10.

[33] SNYDER, P. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference* (1990), pp. 241–248.

[34] TIRUMALA, A., QIN, F., DUGAN, J., FERGUSON, J., AND GIBBS, K. Iperf: The TCP/UDP bandwidth measurement tool. *htt p://dast. nlanr. net/Projects* (2005).

[35] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[36] WONG, H. P., RAOUX, S., KIM, S., LIANG, J., REIFENBERG, J. P., RAJENDRAN, B., ASHEGHI, M., AND GOODSON, K. E. Phase change memory. *Proceedings of the IEEE 98*, 12 (2010), 2201–2227.

[37] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.