Enabling Lightweight Transactions with Precision Time

Pulkit A. Misra
Duke University
pulkit@cs.duke.edu

Jeffrey S. Chase
Duke University
chase@cs.duke.edu

Johannes Gehrke
Microsoft Corporation
johannes@acm.org

Alvin R. Lebeck
Duke University
alvy@cs.duke.edu

Abstract

Distributed transactional storage is an important service in today's data centers. Achieving high performance without high complexity is often a challenge for these systems due to sophisticated consistency protocols and multiple layers of abstraction. In this paper we show how to combine two emerging technologies—Software-Defined Flash (SDF) and precise synchronized clocks—to improve performance and reduce complexity for transactional storage within the data center.

We present a distributed transactional system (called MILANA) as a layer above a durable multi-version key-value store (called SEMEL) for read-heavy workloads within a data center. SEMEL exploits write behavior of SSDs to maintain a time-ordered sequence of versions for each key efficiently and durably. MILANA adds a variant of optimistic concurrency control above SEMEL's API to service read requests from a consistent snapshot and to enable clients to make fast local commit or abort decisions for read-only transactions.

Experiments with the prototype reveal up to 43% lower transaction abort rates using IEEE Precision Time Protocol (PTP) vs. the standard Network Time Protocol (NTP). Under the Retwis benchmark, client-local validation of read-only transactions yields a 35% reduction in latency and 55% increase in transaction throughput.

CCS Concepts **• Information systems $\rightarrow$ Distributed storage**

Keywords clock-synchronization; non-volatile memory; distributed storage systems; strong consistency; transactions

ASPLOS '17, April 08-12, 2017, Xi'an, China
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037722

1. Introduction

Large-scale data centers provide the computational infrastructure that underlies the increasing use of cloud services. Today's data centers exhibit properties of both loosely coupled distributed systems and tightly coupled supercomputers. For example, networking infrastructure now mimics early supercomputers with low latency and high bandwidth per link, high bisection bandwidth Fat-Tree topologies [3, 38], and remote memory operations (e.g., RoCE). We believe that the rapid increase in cloud computing is driving a trend to move further toward supercomputing-like capabilities. Nonetheless, the scale and criticality of today's systems demands a distributed service architecture that is resilient to failures even within the data center, including a replicated storage tier with transactional data consistency.

This paper presents a high-performance, replicated, transactional key-value store that is designed to exploit two emerging data center capabilities: precision time and efficient persistent memory that is based on flash or other NVM/SSD technologies. Our approach to transactional key-value storage is unique in that it is optimized for low-latency access within a single data center, where network and persistent memory latencies are increasingly measured in micro or nano seconds. Synchronized global time is achieved using the IEEE Precision Time Protocol (PTP) [28], which can obtain server clock skew $< 1\mu$s within a data center. Originally developed for distributed control systems, the PTP standard is increasingly supported in commodity Ethernet components. The bounds on clock skew continue to tighten: recent research demonstrates ≈ 150ns skew across a data center [37]. Crucially, these tolerances are within the access latencies of flash-based SSDs, and are approaching DRAM latencies. Flash write latencies are approximately $50 - 100\mu$s; byte-addressable persistent memory can achieve DRAM latencies (≤ 100ns) with emerging non-volatile memory technologies (STTRAM, PCM, etc.), or using battery-backed DRAM. We show that precision time enables a unique timestamp for each update, which can be compared as a basis for consistency.

Our work uses precision time as the foundation for a persistent multi-version key-value store—called SEMEL—and a lightweight transactional system—called MILANA. In SEMEL, each version of a key's value is timestamped using precision time. These timestamps enable a lightweight primary-backup replication protocol that moves update ordering off the critical path (**Contribution 1**). SEMEL and MILANA represent solutions for the region in the data center design space between loosely coupled distributed systems and tightly coupled coherent shared memory systems.

MILANA adds optimistic concurrency control (OCC [32]) to support serializable ACID transactions over SEMEL (**Contribution 2**), adapted to a client/server setting based on techniques pioneered in Thor [1]. Each transaction executes on a single client (e.g., an application server): the client issues read/write requests to SEMEL storage servers, assigns precision timestamps for the start and end of the transaction, and acts as the coordinator to commit or abort the transaction. Thor showed that OCC can improve throughput (concurrency) and latency relative to the pessimistic alternative (two-phase locking or 2PL), which also causes blocking and deadlocks under contention. However, OCC transactions may be forced to abort/rollback under contention due to timestamp ordering conflicts with other transactions, and this risk increases with clock skew [1]. Our results show that in this setting precision time (PTP) achieves a lower rate of spurious aborts due to false conflicts (and therefore higher peak throughput) when compared to clock synchronization using NTP—the current state of the art—and also helps improve fairness among clients. The abort rate is up to 43% lower in high-contention scenarios.

Moreover, timestamp-based concurrency control enables use of multi-version approaches [8] to further improve concurrency and to enable snapshot-isolated read-only transactions with low cost. For example, Google's Spanner [16] uses fine-grained timestamp ordering and multi-version storage primarily to support lock-free read-only transactions on snapshots; Spanner uses 2PL for other transactions and requires a wait time on transactions to tolerate the ms skew possible in TrueTime.

SEMEL leverages the erase-before-write (remap-on-write) behavior of flash SSDs to enable cheap multi-version storage. SEMEL and MILANA are based on an extended SSD Flash Translation Layer (FTL) that writes updated values in a log-structured fashion on physical storage, maps keys directly to values at physical locations, and integrates version management with FTL garbage collection. Our approach builds on the growing use of software-defined flash (SDF) to enhance SSD storage while eliminating unnecessary abstraction boundary crossings [13, 14, 46, 47, 51]. Previous work extended the FTL to integrate support for transactions [14, 47, 48], direct key-value storage [41], and snapshots [52]. *To our knowledge,* MILANA *is the first system to leverage the flash remap-before-write behavior for FTL-integrated multi-version concurrency control* (**Contribution 3**).

SEMEL and MILANA reflect the state of the art in sharded, replicated, transactional key/value storage, but embody a unique set of design tradeoffs for low-latency intra-data center storage with SSDs and precision time. For example, MILANA is similar to TAPIR [57] in that it uses OCC in conjunction with an unordered replication protocol, which has potential for lower latency than ordered consensus (as in Thor and Spanner). Consensus forces all replicas to agree on operations in a total order, which is not necessary to preserve transactional consistency.

However, in contrast to TAPIR, SEMEL uses primary/backup replication. This choice reduces OCC validation costs in MILANA: write validation occurs only on the primary replica for each affected shard, and read-only transactions validate locally at the client (**Contribution 4**). The price of this efficiency is that read-write transactions require an extra round-trip latency (though not extra messages) to sequence through the primary. For intra-data center storage the extra round-trip latency is a small price for cheaper reads, given the prevalance of read-dominated workloads [5, 44]. TAPIR's design—in which all transactions require full consensus—is suitable for a geo-replicated system, where eliminating the primary saves a cross-data center round trip. The cross-data center latency in TAPIR also permits the use of coarse-grained NTP time synchronization; we show that NTP time skew is too high for modern low-latency data centers and that PTP enables use of OCC with low abort rates, even in high-contention scenarios (**Contribution 5**).

Our SEMEL implementation utilizing PTP and the LightNVM Open-Channel SSD emulation framework [11] reveals a 20-45% increase in IOPs and up to 7X lower GET latency on a single machine using unified version and flash management compared to a naive multi-version KV-store implemented using a standard FTL for read heavy workloads (50-100% GET ops). Our experiments running Retwis [35] with MILANA show up to 43% reduction in abort rates using PTP vs. NTP due to tighter clock synchronization. Furthermore, local client validation in MILANA reduces transaction latency by 35% and increases throughput by 55% for read-heavy workloads.

The remainder of this paper is organized as follows. §2 provides background on PTP and SDF. The design of SEMEL and MILANA are described in §3 and §4. We evaluate our prototype systems in §5. §6 discusses related work and we conclude in §7.

2. Enabling Technologies

2.1 Precision Time

The Precision Time Protocol (PTP) was originally defined in the IEEE 1588 standard [28] for use in control systems or for

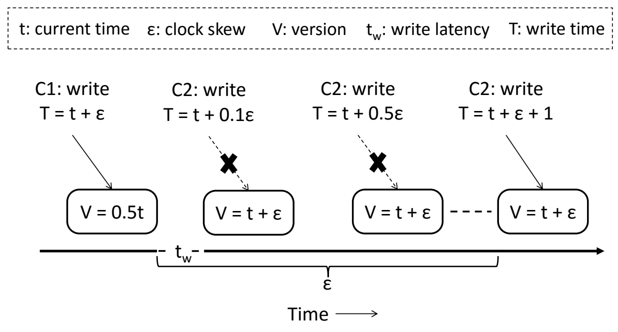

Figure 1: Impact of Clock Skew

precision measurement instrumentation. It achieves $< 1\mu s$ accuracy in synchronizing real time clocks of servers that are connected over a Local Area Network (LAN). The protocol classifies servers into a master or slave hierarchy and defines synchronization messages that are passed between a master and a slave clock. Each synchronization message contains a timestamp of when it was sent or received and through four back-and-forth messages between a master and a slave, a slave can determine the network latency and calculate its offset to the master clock.

Clock synchronization typically occurs every two seconds and the synchronization messages can be timestamped using hardware or software, with hardware providing the lowest skew across servers. PTP is supported in current Linux releases and accessible through user level APIs. *Recent proposals improve synchronization skew beyond PTP to less than* $160ns$ *throughout a data center and less than* $30ns$ *for directly connected servers [37].*

Using synchronized clocks for optimistic concurrency control is a natural step since it can help reduce spurious aborts. Figure 1 shows an example of a shared object updated by two clients C_1 and C_2, ϵ is the clock skew and t_w is the write latency. Since the client with a leading clock (C_1) updates the object first, the lagging client (C_2) has to wait for a duration $> \epsilon$ before it can successfully update the shared object. If $\epsilon >> t_w$ then there are spurious aborts even though the device is capable of satisfying a new write request. Such aborts lead to increased application latencies and lower throughput. The problem is exacerbated for faster devices, where the access latency t_w is on the order of 100s of nanoseconds while the clock skew ϵ may be several milliseconds under clock synchronization protocols like NTP.

SEMEL and MILANA rely on PTP's precise clock synchronization to relax ordering requirements for replication, to perform validation aggressively, and also for garbage collection (explained later).

2.2 Software-Defined Flash

A key service in data centers is reliable persistent storage—historically provided by replicated disk drives. Persistent memory technologies (battery-backed DRAM, emerging

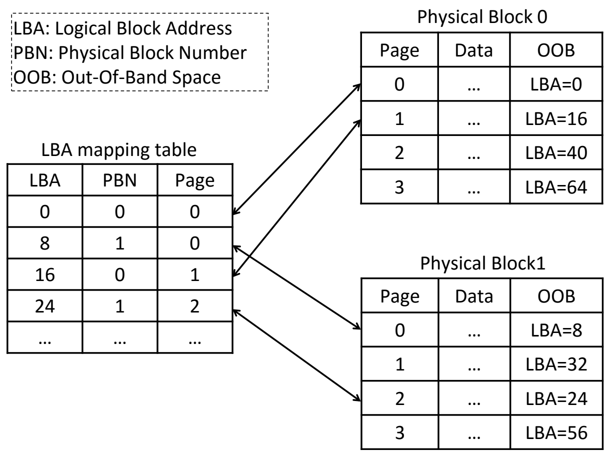

Figure 2: Flash Translation Layer (FTL) Mapping of Logical Blocks to Physical Pages & Blocks

non-volatile memories, and flash-based SSDs) enable newer approaches with dramatically lower latency. These technologies have characteristic access latency between 100ns and $1\mu s$.

A Solid State Drive (SSD) comprises several flash memory chips and a programmable controller that executes the Flash Translation Layer (FTL). SSD capacity is increasing rapidly through the use of vertical stacking. Simultaneously, increased throughput and decreased latency is achieved by using new queue-based PCIe interfaces (e.g., NVMe), similar to those used in high-performance network interfaces (e.g., Infiniband). SSDs can achieve ≈1M IOPs with capacities near 1TB per drive, and latencies of $\approx 50 - 100\mu s$ at less than \$1.00/GB. These advances in SSD design and implementation further improve their ability to handle high-throughput big data processing. Flash continues to see increased use within data centers.

Flash memory is organized as an array of blocks where each block contains some number of pages. Typically pages are 2-16KB in size and each block contains 128-256 pages. The page size is the smallest unit for reads and writes. Without loss of generality, we consider a single-level flash bit cell that can be written in only one direction (0 to 1). A page write operation only sets values to 1, and thus if the data changes from a 1 to 0, the page must first be erased and then the new data written (*erase-before-write*). Unfortunately, erase operations on flash occur only at a block granularity (*block-grained erase*). Moreover, flash has limited endurance: each block can be erased only a certain number of times before the cells wear out.

To accommodate the above characteristics and limitations, the Flash Translation Layer (FTL) of flash-based SSDs provides a dynamic mapping of logical addresses to physical locations. The FTL presents a block device interface to the Operating System (OS) [2] and maps a Logical Block Address (LBA) to a page in flash memory, as shown in Figure 2. This level of indirection allows the FTL to remap a logical block to a new physical page on each write, leaving the old value in place pending garbage collection. The FTL's garbage collector may remap existing (current and valid) pages as needed to free complete blocks to erase. The FTL also implements *wear-leveling*: it distributes writes uniformly across physical locations, so the flash cells wear at the same rate.

Historically, the FTL was implemented entirely within the SSD enclosure and exposed a traditional block abstraction to software. This structure enables tight control over garbage collection and wear-leveling, aspects that can influence warranty guarantees for vendors, and allows close integration with flash read/write circuitry for read/write operations; however, it can limit flexibility.

The recently proposed Software-Defined Flash (SDF) is a technique to separate the FTL functionality and enable host software to participate in flash management [13, 29, 46, 47], with several vendors providing some form of this capability (e.g., CNEX Labs, SanDisk/FusionIO, Radian Memory). This approach enables optimizations across traditional system boundaries, since it eliminates one level of indirection (i.e., the FTL mapping). Furthermore, customized mapping techniques that exploit application or system specific information can provide new functionality and/or improve performance. We exploit this same opportunity to develop our own FTL, as described in §3.

Although our approach generalizes to other storage technologies, it is designed to leverage remap-on-write storage devices that naturally preserve multiple versions of each key as it is updated. We show how to build high-performance, reliable, low-cost, scale-out storage from these technologies.

3. SEMEL: A Replicated Multi-version Key-Value Store

This section presents SEMEL, a replicated multi-version key-value store that exploits precision time and remap-on-write storage. SEMEL provides safety guarantees for ordering of operations to individual keys (coherence). §4 shows how to support transactional atomicity and consistency for operations on multiple keys, layered above SEMEL.

The SEMEL design targets an intra-data center client/server storage model. The persistent memory (SSDs) reside on storage servers. The key space is sharded among the storage servers, and each shard is replicated for availability and fault tolerance. The clients of SEMEL are application servers. Each client has a unique ID and runs a SEMEL library that exposes the key/value storage API and issues read and write operations to the storage servers. The client library coordinates with a global master to map each key to a data shard and to the shard's primary replica using standard techniques (e.g., consistent hashing [31]). The master maintains the shard maps based on its global view of participating servers. The master can be implemented using standard techniques (e.g., Apache Zookeeper [27]).

Values for each key are stored as a sequence of versions timestamped by the client that issued the write. Versions are ordered by the version number, which is a $V = \langle timestamp, clientID \rangle$ tuple. The clientID induces a total order over simultaneous writes from different clients, and also supports linearizability (§3.3). SEMEL uses these timestamps to maintain a coherent view across all clients for each key. We do not expect timestamp wraparound to be an issue if we use 64-bit timestamps. Assuming $\approx$ 100ns resolution for timestamps, a 64-bit timestamp does not overflow for nearly 60 *thousand* years.

The application API to SEMEL client library is defined below. We use $t_{current}$ to denote a client's view of the current time.

- **put(key, value)**: Create a new version for the given key.
- **get(key)** $\rightarrow$ **value**: Return a version with timestamp $\leq t_{current}$.
- **delete(key)**: Delete all versions of the key.

The client library assigns a timestamp $t_{current}$ to all *get* and *put* requests. This timestamp is used for creating a new version $V = \langle t_{current}, clientID \rangle$ of a key on a put request. For a get request, SEMEL uses $t_{current}$ to read the *youngest* version with timestamp $\leq t_{current}$. MILANA (§4) extends the SEMEL client to issue reads for a specific timestamp other than $t_{current}$ as required for the transaction protocol.

3.1 Multi-version FTL with Software-Defined Flash

A standard flash FTL writes each modified page to a freshly erased block and remaps the page (§2.2). Therefore, flash SSDs may naturally provide multiple versions of a given key with little additional overhead [52].

SEMEL leverages the Open-Channel SSD framework [11, 12] to extend the FTL for multi-version key-value storage. A key-value store implemented using traditional SSDs requires two mapping steps: $Key \rightarrow LBA \rightarrow \langle PBN, Page \rangle$. SDF enables modifying the FTL to collapse this two-step translation into a single translation [26, 29, 41, 58], so that it maps a key directly to a physical address with a single map table access.

Mapping table: The mapping table in SEMEL SDF maintains multiple versions of a key as a linked list. Each version is assigned a 64-bit create timestamp; the linked list is sorted in descending order of create timestamps of the versions. SEMEL writes new values in a log-structured fashion on flash. Figure 3 shows the mapping table and data layout in SEMEL. For small key-value pairs, SEMEL packs multiple pairs into a single page. The mapping table maintains the page and the offset within the page where the version is stored.

SEMEL SDF assumes that adequate server DRAM is available to store the entire mapping table in main mem-

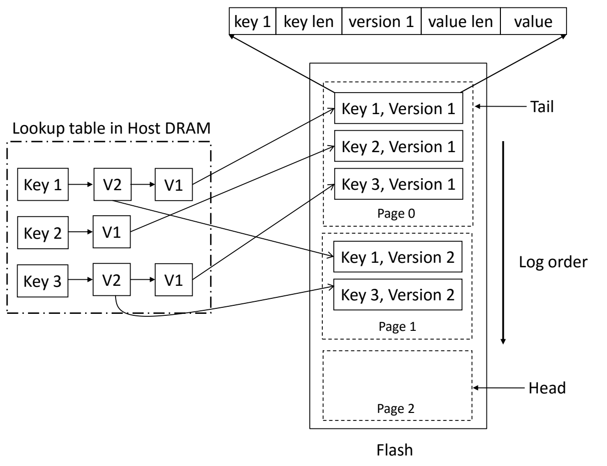

Figure 3: Mapping Table and Data Layout in SEMEL

ory. However, if main memory usage is a concern, it can use a two-level mapping strategy, similar to the one proposed in DFTL [25]. For example, we could extend the design to retain only frequently accessed keys in main memory, destaging cold mappings to a bounded-size second-level table on flash. This extension is future work.

Garbage collection: Keeping versions around longer than necessary on flash-based systems may cause wasteful remapping (moving) during garbage collection. Ideally we want to balance remapping cost with the desire to provide historical versions within a certain threshold (window size), e.g., keep all versions that are less than 5 seconds old. The window size can be tunable to keep older versions as needed, e.g., for read-only analytics workloads. SEMEL utilizes watermarking [20], which establishes a lower bound on the client clocks. Each client periodically broadcasts the timestamp of its last acknowledged operation to all storage servers and the minimum of all these timestamps is the watermark in SEMEL. Since NTP/PTP clocks are monotonic, no client issues a new operation with a timestamp below the watermark. Therefore, the garbage collection algorithm needs to keep only the youngest write with a timestamp less than the watermark; it is safe to discard all prior versions.

3.2 Lightweight Inconsistent Replication

Replication protocols are used in distributed storage systems to provide fault tolerance and availability. Many systems use consensus protocols (state-machine replication), i.e., variants of Viewstamped Replication [45] or Paxos [34]. In consensus protocols, a leader drives operations in sequence to the replicas. Consensus is expensive in part because each operation commits only after a majority of replicas accept it, and the ordering requirement prevents a replica from accepting an operation until it has accepted all prior operations.

SEMEL uses primary/backup replication with a designated primary for each shard, and exploits tightly synchronized clocks to *relax the ordering requirement* and commit each update as soon as a majority of replicas receive (and acknowledge) it. Since the replicated SEMEL operations are timestamped writes to independent versions of independent data items, there is no need to maintain ordering: the ordering is explicit in the version timestamps, which are recovered along with the data.

TAPIR [57] is based on a similar *inconsistent replication* approach that decouples replication from ordering (see §4.6 for a comparison against TAPIR). Inconsistent replication reduces latency because each replica can execute and acknowledge an unordered operation as soon as it receives it, even if it is missing earlier operations. Such an approach does not violate consistency after failover since all acknowledged updates can be recovered if a majority of replicas are available, and a correct ordering can be constructed based on version numbers. We explain recovery in §4.5.

3.3 Linearizability with Global Clocks

SEMEL leverages precise clocks to enable a simple and general timestamping approach to linearizable RPCs on objects. RPC calls on an object execute serially at the primary for the object's shard. The timestamp of a write request persists with the new object version, even across primary failover. The version stamps allow the server to ensure idempotence for retransmitted requests; the client ID distinguishes requests from different clients with the same timestamp. The server executes reads on the named version and rejects writes with timestamps older than the current version, guaranteeing at-most-once semantics. Thus writes execute in a serial timestamp order that is consistent with the real-time ordering. SEMEL also permits snapshot reads in the past, which are not linearizable, but they allow higher concurrency and it is a client's choice to use them.

SEMEL's approach to linearizable RPC is similar in spirit to RIFL [36], which also timestamps requests at the client and persists a *completion record* containing each request's timestamp with the object. The key difference is that SEMEL's request timestamps are *global* and synchronized across the clients. Precise clocks enable us to simplify the ordering protocol. For example, it becomes safe to garbage-collect old versions and their timestamps at any time. If a client replays a completed request after the server discards its version, a simple timestamp comparison blocks it from overwriting an earlier request on the same object: the client receives a rejection for the retransmitted request (not idempotent), but at-most-once semantics are preserved.

SEMEL also guarantees idempotence. The watermarking scheme (§3.1) ensures that the server retains each version at least until the client receives a reply to the write that created it. A client never retransmits a request for which it has received a reply. Therefore, if a client retransmits a write request, then the SEMEL server can always identify it as a duplicate and repeat its earlier response.

When precision timestamps are available, ordering with global clocks is simpler than approaches based on leases and/or causal information [23, 36, 42]. The key tradeoff is that clients with lagging clocks may see their requests rejected under contention, forcing them to retry more often. The SEMEL approach is suitable when the expected clock skew is less than the request cost, which is the case for operations on flash-based SSDs with PTP-based precision timestamps. Note that ordering with global clocks depends on low clock skew only for performance, and not for correctness.

4. MILANA: A Transactional K-V Store

SEMEL provides a Key-Value (K-V) interface to operate on single objects without any support for atomic updates to multiple objects i.e., ACID transactions. This limitation is common to many early K-V stores [15, 19, 33], because transactions were thought to be incompatible with their scalability goals. Recent work has shown that ACID transactions can be a practical extension to these systems, and that many application services benefit from them [6, 16].

This section shows how to use SEMEL's timestamped values to support transactions in a software layer above SEMEL. Our transaction system—called MILANA—supports transactions that update keys in multiple shards atomically using a classical two-phase commit (2PC) protocol. MILANA leverages SEMEL's precision timestamps for Optimistic Concurrency Control (OCC) [32] adapted to a client/server setting [1]. OCC is an alternative to locking (two-phase locking or 2PL): OCC enhances concurrency relative to 2PL, and is not prone to 2PL's blocking and deadlocks. OCC systems *validate* each transaction T before commit by comparing T's timestamped data accesses—T's *read set* and *write set*—to those of other transactions to identify any access conflicts that violate a serializable ordering. Conflicting transactions are aborted and then restarted at the client. Other client/server OCC transaction systems include Thor [1], Centiman [20], and TAPIR [57].

MILANA benefits directly from PTP because low clock skew reduces the incidence of false aborts in client/server OCC [1]. For example, a false abort occurs if a late-arriving transaction (e.g., a commit request from a client with a lagging clock) conflicts with an already-committed transaction with a later timestamp. As explained previously, PTP is particularly important for storage services based on low-latency persistent memory, e.g., flash-based SSDs and emerging NVM technologies. MILANA exploits PTP to improve performance, but it is not required for correctness.

To implement OCC, MILANA assigns precision (PTP) timestamps *begin* (ts_{begin}) and *commit* (ts_{commit}) to each transaction T at the client; all read operations for T are issued at the SEMEL layer with T's *begin* timestamp and all write operations create a new version with T's *commit* timestamp. MILANA also leverages SEMEL's multi-version flash SSD store to support *snapshot reads*: MILANA satisfies T's reads for a key K by returning a version that is current as of T's ts_{begin}, even if a writer has written a new version of K with a later timestamp. This approach reduces false conflicts and further improves concurrency and throughput.

MILANA is optimized for read-heavy workloads, which typically dominate within a data center [5, 44]. In particular, MILANA servers return sufficient version information to enable a client to perform *local validation* for read-only transactions (§4.3). Local validation allows a read-only transaction T to commit if and only if the values in T's read set are from a consistent snapshot: each value for a key K in T's read set is the *youngest committed* version of K with timestamp $\leq ts_{begin}$, and no key K in the read set has a prepared version with timestamp $\leq ts_{begin}$. Local validation ensures a serializable transaction ordering for read-only transactions, but it does not necessarily provide external consistency. MILANA provides both serializability and external consistency for read-write transactions, which validate on the servers.

4.1 Transaction Protocol

MILANA adds an extended transaction API to the SEMEL primary servers, and an enhanced client library to use it. A MILANA primary maintains a *transaction table* recording the status of transactions that have prepared but for which commits have not yet been acknowledged: updates to this table are logged in persistent memory as they occur and are replicated to the backup servers using the SEMEL replication protocol. If the primary fails, a new primary recovers the transaction table before continuing (§4.5).

MILANA uses the version stamps provided by SEMEL, $V = \langle timestamp, clientID \rangle$. This approach provides monotonically increasing timestamps with a total order. We assign each transaction T two timestamps ts_{begin} and ts_{commit} at T's begin and commit time respectively. T's ts_{begin} is assigned to all GET (read) requests and ts_{commit} is assigned to all PUT (write) requests.

In addition, a MILANA primary server also maintains in DRAM a $ts_{latestRead}$, $ts_{prepared}$ and $ts_{latestCommitted}$ timestamp for each active key. $ts_{latestRead}$ is set on a get request if $ts_{get} > ts_{latestRead}$. $ts_{prepared}$ and $ts_{latestCommitted}$ timestamps are set after a successful validation and commit, respectively. None of these values are persisted; §4.5 explains how to recover them.

Here is the application API to the MILANA client library. Timestamp $t_{current}$ represents the client's local view of the current time as given by PTP.

- **beginTransaction()**: Start a new transaction T. Assign a begin timestamp to T $(ts_{begin} = t_{current})$, and initialize an empty read and write set for T.
- **abortTransaction()**: Discard the read and write set maintained for the current transaction and remove all state.
- **commitTransaction()** $\rightarrow$ **Success / Fail**: Assign commit timestamp $ts_{commit} = t_{current}$ for the current transac-

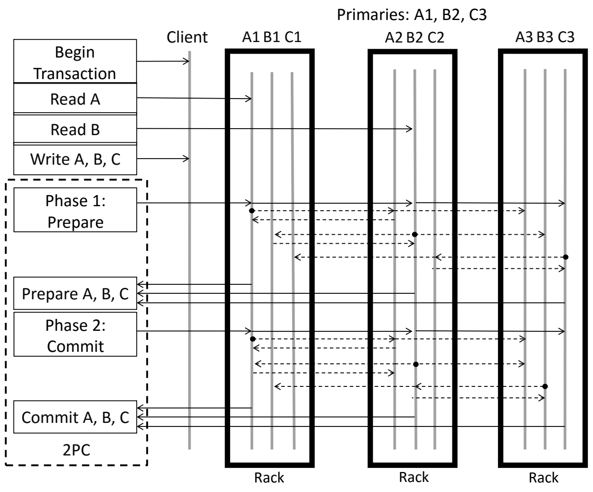

Figure 4: Two Phase Commit

Algorithm 1 MILANA Primary Validation Algorithm

```
 1: procedure VALIDATE(transaction)
 2:     for each (key, version) $\in$ transaction.readSet do
 3:         if key.prepared $\neq$ NONE then
 4:             return ABORT
 5:         else if key.latestCommitted $\neq$ version then
 6:             return ABORT
 7:         end if
 8:     end for
 9:     newVersion = transaction.commitTimestamp
10:     for each (key, version) $\in$ transaction.writeSet do
11:         if key.prepared $\neq$ NONE then
12:             return ABORT
13:         else if key.latestRead $\geq$ newVersion then
14:             return ABORT
15:         else if key.latestCommitted $\geq$ newVersion then
16:             return ABORT
17:         end if
18:     end for
19:     return SUCCESS
20: end procedure
```

tion and initiate the commit protocol, which either succeeds or fails.

- **put(key, value)**: Buffer the key-value pair; add key to the current transaction's write set.
- **get(key) $\rightarrow$ value**: Return a consistent value for a key; add key to the current transaction's read set.

MILANA executes transactions in the usual manner for client/server transaction systems with OCC [20, 36, 57], following Thor [1]. As a transaction executes, it maintains the read and write set in client memory, satisfies reads of values in the write or read set from the cache, and buffers all writes. To commit a read-write transaction, the client chooses the commit timestamp from its local time ($ts_{commit} = t_{current}$), and initiates and coordinates the two-phase commit (2PC) protocol; the 2PC participants include the servers for all keys in either set. The client pushes modified values for keys in the write set to their servers on a commit request, and not before. Each server validates the transaction for serializability before accepting it (§4.2).

MILANA differs from earlier systems in that its multi-version store (SEMEL) allows it to support snapshot read-only transactions with local validation. For any transaction T, the client issues reads for versions that are valid at a point in time: T's begin timestamp ts_{begin}. A MILANA client validates and commits read-only transactions locally if it succeeds in reading a consistent snapshot at time ts_{begin} (§4.3).

4.2 Two-Phase Commit: Write Validation

Figure 4 shows an example transaction with a standard two phase commit (2PC) protocol. On a commit request for a read-write transaction T, the client library initiates 2PC and acts as the coordinator. It first sends a *Prepare()* request to the primary of each participant shard, passing each primary all keys in T's read and write sets for shards that the primary controls. It also passes a list of other affected shards for possible use in recovery (§4.5).

Each primary uses Algorithm 1 to validate T's keys. T fails validation if it has conflicts that violate transactional serializability. It then propagates the validation decision (SUCCESS/ABORT) along with the write set (on successful validation) and shard list to the backup replicas, waits for f (out of $2f$) backups to respond, and then reports the decision as its vote to the client/coordinator. If a primary votes to commit T then T is *prepared* at that primary.

The client accumulates the votes from all primaries and determines the outcome: T commits if and only if all primaries vote to commit, else T aborts. The client reports the outcome to the application and then asynchronously notifies all primaries of the outcome.

4.3 Local validation

As mentioned earlier, a MILANA client performs local validation for read-only transactions. Local validation eliminates *two round trips* at validation time: client to primary and primary to backups.

As a transaction T runs, the client issues a read (get) to the primary server for each key K read by T, and satisfies subsequent reads to K from its cache. The client issues gets with T's begin timestamp ts_{begin}. On a get, the primary returns the youngest committed version of K with a timestamp $K.ts_{commit} \leq T.ts_{begin}$, and a boolean that indicates if there is a prepared version of K with a timestamp $K.ts_{prepared} \leq T.ts_{begin}$. Note that $K.ts_{commit} < K.ts_{prepared} \leq T.ts_{begin}$. The primary also records the read timestamp ts_{begin} in DRAM if it is $> K.ts_{latestRead}$.

Local validation works because a MILANA primary aborts any late-arriving transaction S that attempts to commit a new value for a key K with an earlier timestamp $S.ts_{commit} \leq K.ts_{latestRead}$ (see Algorithm 1). Therefore, if K did not have a prepared version when it was read, then it is guaranteed that there can be no prepared version with a timestamp less than T's begin timestamp (ts_{begin}). Thus the client has all the information needed to locally validate T: it can commit T if and only if none of the keys in T's read set had a prepared version at T's read time (ts_{begin}).

Local validation is aided by both SDF and PTP. SDF provides a lightweight mechanism to maintain multiple versions of a key, thus enabling snapshot reads. PTP's low clock skew makes local validation practical from a *performance* standpoint. For a given clock skew ϵ, if a client with a leading clock reads a key K, then a client with a lagging clock has to wait up to ϵ duration before it can commit a transaction that updates K. For NTP, ϵ is on the order of milliseconds, while PTP reduces it to microseconds.

It is future work to combine local validation with aggressive caching. Our current approach requires the client to notify a primary as each key is added to the read set, reducing the potential benefit from inter-transaction caching of values at the client. In principle, clients can choose between aggressive caching and local validation: any transaction T that is marked as read-write in advance may read from its cache, but then T must validate remotely.

4.4 Version Management

MILANA leverages SEMEL's watermarking-based garbage collection to manage versions and satisfy long-running read-only transactions. Each MILANA client periodically broadcasts the timestamp t_d of its latest decided (committed or aborted) transaction to all primaries. The minimum over the t_ds becomes the watermark t_w. Since PTP time increases monotonically, no client can have a transaction begin time that is less than the watermark. Therefore, the SEMEL garbage collector only needs to keep the youngest version with a timestamp $\leq t_w$ and can discard all prior versions.

Consider an active long-running read-only transaction T with a begin timestamp ts_{begin}. Then the watermark $t_w < ts_{begin}$. Therefore, a MILANA server retains at least the youngest version of any key K with a timestamp $\leq ts_{begin}$, so T can read a version from a consistent snapshot at its ts_{begin}. The watermarking scheme dynamically tunes the number of versions kept for all keys and is a function of the duration of transactions: fewer versions are kept when transactions are short, and the threshold increases as longer transactions are added to the mix.

4.5 Recovery

This section describes what happens if a client or storage server (e.g., a primary) fails during the process of committing a transaction. MILANA assumes fail-stop (non-byzantine) failures.

Client Failure. If the client fails during 2PC, then the participants (primaries) time out waiting for a commit or abort decision for a prepared transaction T. T is blocked until its commit/abort status is known. This situation does not affect any transactions operating on key sets that are disjoint from T's read/write sets. However, the participating primaries are forced to abort any transaction that attempts to read/write any of the keys in T's read or write set, until T's commit/abort status is known. In such a case, one of the participating primaries is designated as a backup coordinator for T. The backup coordinator can use the Cooperative Termination Protocol (CTP) [9] to determine if T should commit. The backup coordinator queries the other participating primaries for the status of T, and takes appropriate action. The states are *Received Commit, Received Abort, Prepared, Sent Commit, Sent Abort* and the actions can be any of the following:

1. If any primary received a commit or abort then T should be committed or aborted since the client made a decision only after receiving a response from all the primaries.
2. If any primary did not receive a prepare request for T, then all primaries can agree to abort T because the client does not commit a transaction until it receives a response from all primaries for its shards.
3. If any primary responded with ABORT to the prepare request, then all primaries abort T.
4. If all primaries responded SUCCESS for the prepare request then the backup coordinator commits T.

Replica Failure / Recovery. If a backup replica of a participant shard fails during 2PC, it does not block any transaction as long as a majority of replicas for a shard are available to store transactions. However if a primary of a participant shard fails then it would block all transactions involving that shard. A new primary must be elected (failover) in order to unblock any running transactions and resume service.

Distributed transactions require a protocol to ensure atomicity and consistency of keys and shards across failures of servers and clients. Many storage systems that provide transactional semantics and fault tolerance use both a transaction protocol and a replication protocol, which enforce a serial ordering in two places: transactions across shards and updates among replicas. This redundancy can add latency and reduce throughput. Since the transaction protocol enforces ordering among the transactions and consequently the updates, the replication protocol does not need to also enforce ordering. This observation was previously exploited to reduce write transaction latency in TAPIR [57] by allowing inconsistent replication.

SEMEL and MILANA replicate using a primary-backup approach: all the updates to a shard flow through the primary. As a result, the MILANA primary has the consistent view (an up-to-date transaction table) needed to validate transactions without involving the backups, reducing validation costs and abort rates. Since the backups play no role in validating or executing transactions, their only purpose is to provide

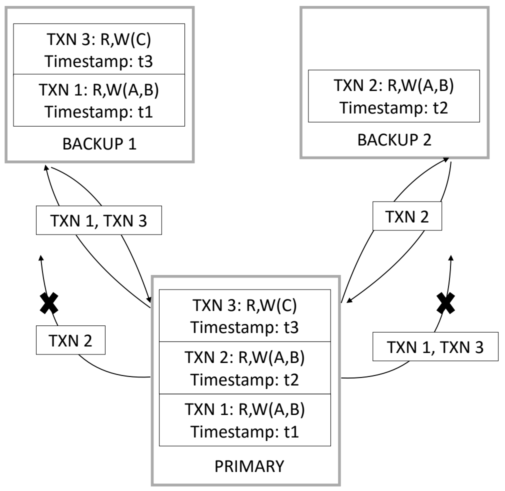

Figure 5: MILANA Relaxed Backup Updates

fault tolerance, as in SEMEL, and not consistency, which is handled by the MILANA code on the primary.

Once the primary validates a transaction, it can propagate updates and prepare records to the replicas in any order, as long as a new primary can rebuild the transaction table during failover. Figure 5 shows how MILANA relaxes backup update ordering and how this can tolerate transient failures. In this example there are three storage servers, a primary and two backups. The primary requires only one of the two backups to acknowledge a prepare and commit of a transaction. In this case, backup 1 acknowledges prepare and commit of transactions 1 and 3, while backup 2 acknowledges prepare and commit of transaction 2. In another scenario, backup 1 acknowledges prepare of transactions 1,2 and 3, and backup 2 acknowledges the commit for these transactions. In both cases, traditional replication would be forced to signal an error, and possibly abort transactions since the backups did not receive updates in sequence order. MILANA eliminates these scenarios by reconstructing the correct overall order during failure recovery.

Since SEMEL does not enforce strict global ordering for all updates during replication, therefore in MILANA the new primary must be brought to a consistent state before it can start servicing transaction requests. The new primary can always reach a consistent state if there are $f + 1$ replicas available (out of $2f + 1$), which are needed for a majority quorum. For $f + 1$ available replicas, there must always be at least one replica that has seen any given transaction committed or prepared by the previous primary. Therefore the new primary has access to all the transactions and can rebuild the data versions and transaction table by merging the updates from all the replicas, as shown in Algorithm 2. If a transaction prepare or commit record is not present on at least one replica at recovery, then the previous primary could not have obtained a majority for the operation, and therefore

Algorithm 2 MILANA Recovery Merge Algorithm

```
 1: procedure MERGELOG(transactions, table = NULL)
 2:     for each T $\in$ transactions do
 3:         if T.status == COMMITTED then
 4:             table.insert(T)
 5:         else if T.status == PREPARED then
 6:             if T.participants == 1 then
 7:                 table.insert(T)
 8:             else
 9:                 decision = queryParticipant(T, participants)
10:                 if $decision \in COMMIT, PREPARED$ then
11:                     table.insert(T)
12:                 end if
13:             end if
14:         end if
15:     end for
16:     return table
17: end procedure
```

could not have acknowledged the request. In this case, 2PC recovery restores the state of these transactions, as detailed above (CTP).

The new primary can then apply all the successfully committed transactions without any validation. It can also apply a successfully prepared transaction that included a single shard because that would have been committed. For a prepared transaction involving multiple shards, the new primary must contact the primary of the other shards to determine whether the transaction committed. The transaction is committed or aborted if any shard responds with a COMMIT or ABORT, respectively. If all participants respond with a prepared status then the transaction is still outstanding and should be committed and a response sent to the client.

After creating the transaction table, the new primary propagates the table to the backups to bring them to a consistent state. It then populates $ts_{prepared}$ and $ts_{latestCommitted}$ values for each key. These values can be inferred from prepare requests obtained from other replicas during recovery and from the version stamps included with each write (see Figure 3), respectively.

The new primary cannot populate $ts_{latestRead}$ for keys as these values are not persisted nor are the backups informed about the latest read timestamp of a key while servicing a get request. Validating new transactions without these values can violate serializability. Consider the following scenario: a read-only transaction T_a issues a get request for key K with a timestamp t_2, a primary returns a version of K with a timestamp t_0 and T_a commits. Now a failover occurs and a new primary allows a read-write transaction T_b to commit that creates a new version of K with timestamp t_1 ($t_0 < t_1 \leq t_2$). This violates serializability because T_a (already committed) should have read T_b's write.

MILANA uses *leases* [24] to avoid this scenario. A primary in MILANA obtains a periodically renewed lease from at least f backups to process any get request with a timestamp $< t_{lease}$. After recovery, the new primary waits for its local clock to advance past t_{lease} before servicing transaction requests for its shard. As an optimization, we can combine this mechanism with leases used for avoiding spurious failovers in primary/backup based replicated state machine protocols [40].

In all failure scenarios (client, primary, backup replica or some combination of the three) a decision can be made on any outstanding transaction and service can be resumed as long as a majority of replicas ($f + 1$) of all shards are available.

4.6 Comparison with TAPIR

There are some similarities between SEMEL/ MILANA and TAPIR [57]. Like SEMEL, TAPIR is based on an inconsistent replication approach that decouples replication from ordering for lowering write latencies. The SEMEL inconsistent replication protocol differs from TAPIR in that SEMEL uses primary/backup replication with a designated primary for each shard, rather than having the client propagate the operation to symmetric replicas, as in TAPIR. Both MILANA and TAPIR build on top of inconsistent replication to provide transactional semantics and use OCC for ordering operations. To reduce read latencies, TAPIR clients read data from the nearest replica during a transaction. This approach also helps balance the read load across replicas. In contrast, all reads in MILANA are serviced by the primary but this requirement can be relaxed for read-write transactions, which can read data from the nearest replica and validate at the primary before commit.

Validation in TAPIR succeeds only after a majority of replicas for each affected shard agree to validate the transaction. TAPIRs approach of eliminating the primary saves a round-trip latency for each prepare. This may be a substantial saving if the primary resides in a different data center, but it requires all replicas to maintain additional state and validate both read-only and read-write transactions. This is less energy-efficient since additional compute and memory resources are needed, and also consumes precious memory / storage bandwidth on all replicas. In contrast, MILANA clients validate read-only transactions locally, which eliminates two round trips. For read-write transactions, once validation on a primary is complete, the updates and prepare records can propagate to backups in any order. Since the backups play no role in validating transactions, their only purpose is to provide fault tolerance: validation imposes no memory or compute cost on the backups.

In summary, TAPIRs design is suitable for a geo-replicated system, where eliminating the primary saves a cross-data center round trip. In contrast, MILANA targets intra-data center storage and its approach reduces total validation costs: every transaction validates on a single node and not on all replicas as in TAPIR. The tradeoff is that all read-write transactions require an extra round trip (from primary to backups), but no extra messages are sent.

5. Evaluation

We present preliminary results for our prototype implementations of SEMEL and MILANA. We use the Open-Channel SSD framework [11] for our SDF implementation. In software-only mode, the framework emulates the internals of a NVMe SSD and supports timing simulation of I/O operations. We extend the timing-only simulation with functional emulation by adding support for storing data values and IOCTLs that provide a get, put and erase functionality for flash blocks. We also added the capability to specify latencies for read page, write page and erase block operations. These operations are simulated by adding a timer interrupt in the kernel and the request is acknowledged after the timer expires.

The performance of SDF suffers due to emulation limitations: crossing the kernel boundary for submitting each I/O request and lack of batched interrupts for request completion. An open-channel compatible NVMe SSD does not suffer from these limitations since the NVMe standard provides user-space queues for submitting requests and also supports batching interrupts to reduce overhead.

Experimental Setup: We use a set of Linux virtual machines from a single ExoGENI [7] site for both clients and servers. The client VMs have 2 cores, clocked at 2.6 GHz and 6 GB of DRAM. The system clocks on client servers are synchronized using PTP software timestamping or NTP. The storage server VMs have 8 cores, clocked at 2.6 GHz and 32 GB of DRAM. The cores of all storage VMs are pinned to allocate from a specific NUMA zone on the host. Each storage VM is configured with an emulated SSD, backed by 12 GB DRAM, with a hardware queue depth of 128. The SSD has a page size of 4KB and there are 32 pages in a block. A page read, write time is 50 μs and 100 μs respectively and it takes 1 ms to erase a block. We use Ubuntu 14.04 with kernel 4.6 on all VMs and our code is compiled using gcc version 4.9 with -O3 flag enabled.

In all our experiments, we set the key size to 16B and a $\langle key, value, version\rangle$ tuple is 512B. Although our implementation supports variable-sized keys and values, we decided to use a fixed size for evaluation since it allows efficient packing of data. As a flash page is 4KB in size, we employ a *packing logic* in the FTL that waits for up to 1 ms (tunable) to pack data of multiple keys into a page (see Figure 3 for data layout). We run each experiment for 15 minutes to ensure that garbage collection is running in the background (for all runs with non-zero put request %).

5.1 SEMEL Evaluation

To elucidate the advantages of implementing multi-versioning within the FTL, we implement a single-version generic FTL and a separate multi-version KV store on top of a generic FTL. This multi-version KV layer implements its own lookup, request handling and garbage collection logic that is *separate* from that of the FTL. The multi-version layer

Table 1: Single SSD Multi-version FTL Performance

Get %	Throughput Kilo Reqs/Sec		Average Latency (μs)			
			Get		Put	
	VFTL	MFTL	VFTL	MFTL	VFTL	MFTL
100	351	456	68.1	59.9		
75	295	430	363.1	62.9	568.5	872.8
50	217	277	516.6	70.3	673.8	859.0
25	215	189	435.6	77.7	659.8	895.8

Table 2: Retwis Configuration

Transaction Type	Num GETs	Num PUTs	Workload %
Add User	1	2	5
Follow User	2	2	10
Post Tweet	3	5	35
Get Timeline	rand (1,10)	0	50

operates at 4KB granularity and uses a log-based approach to write data to the SSD. We refer to the single-version generic FTL as SFTL, the split multi-version layer on top of a generic FTL as VFTL and the unified multi-version FTL as MFTL. To ease garbage collection, SFTL, MFTL and the multi-version KV layer in VFTL reserve 10% of available capacity for remapping data.

We first measure the throughput and latency for KV operations by emulating a single SSD for both MFTL and VFTL. For these experiments, we populate the device with 2 million keys and use a micro-benchmark to issue KV requests for varying get request percentages. Table 1 shows our results. As expected, MFTL delivers up to 45% higher throughput, and up to 7x lower latency compared to VFTL. For 25% get rate, VFTL performs better since it has a lower packing delay. A key-value pair is 512B in size, thus our packing logic waits for up to 1 ms to pack data of multiple keys (puts or remapped keys) into a page. Since VFTL has less available space compared to MFTL (10% capacity reserved at two levels), it performs more garbage collection, has more data (keys) to remap and therefore, incurs less packing delay. For 25% get rate, VFTL remaps 15% more data than MFTL.

We also evaluated the performance of SEMEL in a distributed setup and the results are qualitatively similar to the single SSD numbers.

5.2 MILANA Evaluation

We evaluate MILANA by exploring the impact of multi-versioning and precise time on transaction abort rates. Our first experiment evaluates the impact of multi-versioning vs. using a single version FTL (MFTL vs. SFTL). We use a single VM for this experiment to eliminate clock skew. The VM hosts a storage layer and runs varying number of clients that issue transactions to the storage layer. We populate the storage layer with 2 million keys. The clients run the Retwis

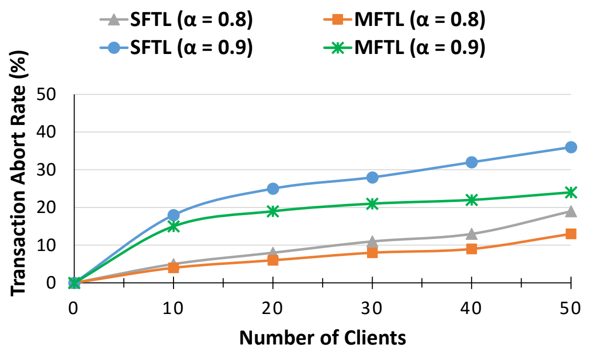

Figure 6: Transaction abort rate for varying number of clients

benchmark [35] with the transaction mix shown in Table 2. Each client has one outstanding transaction and all transactions are executed sequentially. We run this experiment for varying number of clients and to simulate key-sharing, we also vary the Retwis Contention parameter (α).

Figure 6 shows transaction abort rates versus number of clients for a single and multi-version FTL. From the results we see that with increased key contention, a multi-version FTL helps reduce abort rates since tardy read-only transactions are able to read from a consistent snapshot and commit whereas these transactions are aborted on a single-version FTL. Abort rates using VFTL are qualitatively similar to MFTL and omitted for clarity.

To evaluate the impact of clock skew on transaction abort rates, we use 3 storage (1 primary and 2 backups) and 5 client VMs and synchronize clocks on the client VMs using either PTP software timestamping mode or NTP. The NTP daemon is free to choose the best master based on NTP's criterion (lowest jitter). We populate the storage VMs with 2 million keys. Each client VM runs 4 independent instances of the Retwis benchmark (20 instances in total). Each instance executes one transaction at a time and retries an aborted transaction with the same set of keys and without any wait.

Figure 7 shows transaction abort rates versus the amount of contention in the Retwis benchmarks using MILANA with a DRAM backend, VFTL and SEMEL's MFTL. From these results we see that PTP provides superior performance for all storage backends due to the tighter clock synchronization. For NTP the DRAM backend incurs the highest abort rates, as expected, since the faster write time requires lower clock skew across clients. VFTL also incurs slightly higher abort rates compared to MFTL due to lower write latency (see Table 1). NTP shows an average skew of 1.51ms among clients, while software timestamped PTP has average skew of 53.2 μs.

To evaluate throughput and latency, we deploy Milana over 3 shards where each shard has 3 replicas. We populate the system with 6 million keys and ran the Retwis workload

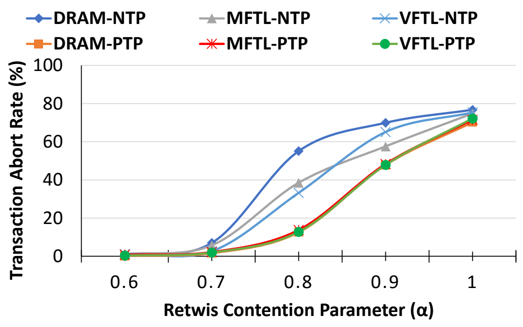

Figure 7: PTP vs. NTP: MILANA Transaction Abort Rates

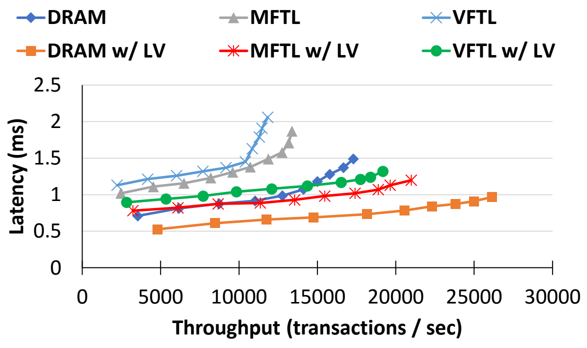

Figure 8: Retwis Transaction Latency vs. Throughput

with 75% read-only transactions (5%, 10%, 10% and 75% breakdown - Table 2) for an increasing number of clients. We perform this evaluation for all 3 storage backends (DRAM, VFTL and MFTL) and also measure the impact of local validation (LV).

Figure 8 shows the average transaction latency vs. throughput for the 3 storage backends. From the results we see that MILANA with local validation is able to achieve up to 55% higher throughput and 35% lower latency. Local validation enables a MILANA client to independently make a commit or abort decision for a read only transaction, without affecting consistency. This saves *two* round-trip times for validation (§ 4.3) and helps reduce transaction latency. These results also show that MFTL achieves 15% higher throughput and 10% lower latency compared to VFTL. VFTL w/ local validation achieves higher throughput than MFTL w/o local validation, showing the importance of local validation.

5.3 Comparison of Local Validation Techniques

This experiment compares Centiman's local validation approach [20] with that of MILANA. Centiman uses a watermark-based technique that allows a client to locally validate a read-only transaction, if it read a consistent snapshot of keys with timestamp $<$ watermark. Otherwise, a Centiman client reverts to remote validation. Centiman's approach works well for low contention scenarios. But under high contention, more key-sharing between transactions increases the probability of a read-only transaction reading a version younger than the watermark and failing the local validation check. Centiman can counteract this by faster dissemination of watermarks, but this increases coordination overhead.

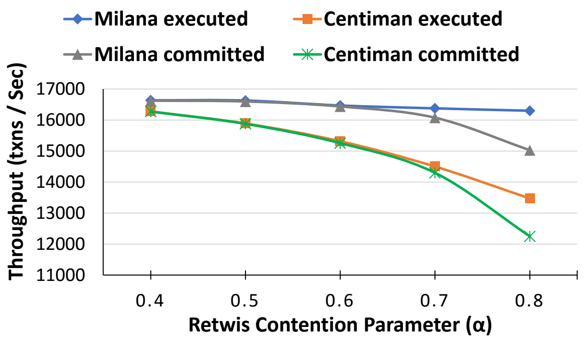

Figure 9: Comparison of Local Validation Techniques

For this experiment, we use 3 storage and 5 client VMs. The storage VMs are used for creating 3 shards, each VM stores data on SSD (MFTL). We populate the shards with 6 million keys. To eliminate impact on throughput, we use the same number of validators (3) with Centiman (one per shard) and these validators run on the storage VMs. We do not use replication in MILANA since Centiman's validators do not replicate. On each client VM, we run 6 independent instances of the Retwis benchmark (30 instances in total) with 75% read-only transaction workload and vary α to simulate key contention. Clients disseminate watermark after every 1,000 transactions. The clocks on client VMs are synchronized using PTP software timestamping.

Figure 9 shows the results. Under low contention ($\alpha = 0.4$), Centiman achieves a similar throughput as MILANA. However, the throughput drops under increasing contention as Centiman's local validation check fails thereby forcing a remote validation. Centiman locally validates 89% of read-only transactions for $\alpha = 0.4$ and this value drops to 25% for $\alpha = 0.8$. One the other hand, MILANA can perform local validation for *all* read-only transactions and therefore achieves 20% higher throughput under high contention settings. Both systems observe similar abort rates.

6. Related Work

There is a long history of work exploring distributed systems, data center services and flash storage systems. This section places our work in context relative to a subset of distributed transactional storage systems and flash-based storage systems.

There are numerous client-server distributed systems, such as key-value services; however, many of these systems lack support for updating multiple objects atomically [15, 19, 33] or restrict partitioning [6] due to the complexity of supporting distributed transactions.

Thor [1] introduced loosely synchronized clocks for OCC and performed validation on the storage servers. Our approach differs by maintaining multiple versions of a key, which helps avoid conflicts between concurrent read and write transactions and performs local validation at the client for all read-only transactions.

TAPIR [57] and Spanner [16] along with some of the differences from our work were discussed previously. A main difference is our focus on intra data center operation vs. inter data center. Centiman [20] uses OCC to support intra-data center distributed transactions but validations are performed on a different set of servers called validators. This helps in a multi-tenant data center where different applications can have their own validators. However this approach involves increased coordination and suffers from limited availability since transactions are made durable on a single client before they are committed. It optimizes for local validation of read only transactions using watermarks but needs remote validation under high-contention settings.

Other intra-data center systems focus on in-memory computation [21, 22, 30, 36, 43, 49, 54]. RamCloud [36, 49] is a key-value store that provides exactly once semantics like SEMEL and transactional semantics like MILANA. However, it does not maintain multiple versions of a key. FaRM [21, 22] is optimized for performance over RDMA, it maintains multiple versions of an object and supports strictly serializable distributed transactions. Neither of these systems have MILANA's inconsistent replication.

Calvin [53] buffers transaction requests and creates a transaction schedule from the received requests. All replicas then execute transactions deterministically using the defined schedule. However this approach restricts the type of transactions since it needs the read and write set of a transaction to be pre-declared in the transaction request. MILANA does not have this requirement.

Hyder [10] uses a shared-storage made up of flash chips to store data. Clients record transactions on the flash storage and also broadcast their intent to all the other clients, which allows clients to then determine if a transaction can commit. The broadcast can be a scalability issue and our approach differs since we allow clients or storage servers to scale independently and we try to minimize coordination wherever possible.

Flash based KV stores have been proposed in prior works [4, 17, 18, 39, 41]. Other systems eliminate the FTL indirection [26, 29, 41, 50, 58]. Each of these systems has some aspects included in SEMEL; however SEMEL differs from these works by maintaining multiple versions of a key and providing transactional semantics for updating multiple keys. Previous systems that maintain multiple versions [52, 55, 56] require a snapshot activation to access prior versions. Several studies [13, 14, 29, 41, 46, 47, 51, 59] propose a cooperative hardware and software based approach to exploit the performance of non-volatile memories. SEMEL and MILANA leverage the functionalities accorded by these designs.

7. Conclusion

As cloud services continue to proliferate, the desire for efficient and easy to use persistent storage increases. This paper presents MILANA, a lightweight transactional system layered on top of SEMEL, a persistent multi-version key-value storage service. We exploit precise intra-data center time and software-defined flash to implement a distributed transactional persistent key-value storage service. Precision time enables lightweight multi-version optimistic concurrency and simplified replication protocols. For flash-based storage, MILANA can leverage software-defined flash to unify version and flash management.

Evaluations of our prototype implementations reveal that SEMEL achieves 20%-50% higher IOPs than a traditional separate version and flash management approach. Furthermore, by using PTP, MILANA reduces abort rates by up to 43% over NTP for transactions with high-contention, due to the tighter clock synchronization across servers. We also demonstrate that MILANA's use of local client validation reduces latency by 35% and increases throughput by 55%.

The trend of data centers to exhibit properties of both tightly coupled supercomputers and loosely coupled distributed systems presents unique opportunities to re-examine cloud service implementations. SEMEL and MILANA are examples of these new services that exploit emerging hardware features. Future directions include examining the potential interactions of SDF/Flash storage and PTP with RDMA and instruction set support for hardware transactional memory, and developing other services such as: file systems, distributed lock services, distributed shared memory for in-memory computations, and distributed flash management.

Acknowledgements

We thank Victor Orlikowski for virtual machine support, Matias Bjørling for support with the Open-Channel SSD framework, and the reviewers for their valuable time and feedback. This work supported in part by the National Science Foundation (CNS-1616947) and equipment donations from Intel and Mellanox.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 23–34. ACM, 1995.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70. USENIX Association, 2008.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14. ACM, 2009.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[6] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[7] Ilya Baldin, Jeff Chase, Yufeng Xin, Anirban Mandal, Paul Ruth, Claris Castillo, Victor Orlikowski, Chris Heermann, and Jonathan Mills. *ExoGENI: A Multi-Domain Infrastructure-as-a-Service Testbed*, pages 279–315. Springer International Publishing, Cham, 2016.

[8] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[9] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[10] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - A transactional record manager for shared flash. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 9–20, 2011.

[11] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The linux open-channel ssd subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2017.

[12] Matias Bjrling. *Operating System Support for High-Performance Solid State Drives*. PhD thesis, Denmark, 2016.

[13] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[14] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264. USENIX Association, 2012.

[17] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36. ACM, 2011.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner 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. ACM, 2007.

[20] Bailu Ding, Lucja Kot, Alan Demers, and Johannes Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 262–275. ACM, 2015.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[23] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):5666, 1988.

[24] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems*

Principles, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[25] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 229–240, New York, NY, USA, 2009. ACM.

[26] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 580–591. ACM, 2015.

[27] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'10, pages 11–11. USENIX Association, 2010.

[28] IEEE. Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, 2008.

[29] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, September 2010.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[31] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[32] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[33] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[34] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[35] Costin Leau. Spring data redis retwis-j, 2013. `http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/`.

[36] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 71–86, New York, NY, USA, 2015. ACM.

[37] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *To appear in Proceedings of ACM SIGCOMM*, August 2016.

[38] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, October 1985.

[39] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13. ACM, 2011.

[40] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[41] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[42] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[43] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[44] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[45] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17. ACM, 1988.

[46] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[47] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 301–311, Washington, DC, USA, 2011. IEEE Computer Society.

[48] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.

[49] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[50] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280. ACM, 2012.

[51] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[52] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a flash with iosnap. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 23:1–23:14. ACM, 2014.

[53] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12. ACM, 2012.

[54] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[55] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Vinay Sridhar, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. MjÖlnir: Collecting trash in a demanding new world. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 4:1–4:10. ACM, 2015.

[56] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Anvil: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 111–118. USENIX Association, 2015.

[57] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278. ACM, 2015.

[58] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 1–1. USENIX Association, 2012.

[59] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.