1. BACKGROUND

Milana [3] is an intra-datacenter transactional key-value store in which transactions are executed and coordinated by client machines and validated by storage servers. Milana delivers transactional semantics using a Two-Phase protocol for coordination and Optimistic Concurrency Control (OCC) for isolation. For scalability, an application’s keyspace is divided among shards using Consistent Hashing [2]. Each shard in the key-value store operates using a primary-backup model for fault-tolerance, where reads and writes are handled by the primaries. The primaries also validate transactions and ensure that the replicas’ state is consistent with their own. Applications link with the Milana client library to issue and coordinate transactions on remote data which is stored on the storage servers. In order to facilitate lower latency reads, an extension to Milana introduced the notion of a distributed caching layer that is placed between the client machines and the storage servers. The cache aimed to provide benefits such as:

- Coupling the lower latency of remote DRAMs at the caching layer with the durability of SSDs at the storage layer for better performance.
- Reducing server load for read operations by exploiting the caching layer.
- Offloading validation partially to the caching layer by validating read only and read-write transactions on the distributed cache.

2. MOTIVATION

The client-side cache in Milana uses Consistent Hashing to statically partition the keyspace among the caches to develop a notion of ownership. The cache owning a specific key implies that it is responsible for maintaining state and performing validations on that key. While Consistent Hashing is great for elasticity (i.e., reduces churn when nodes are added or removed), it has been shown to perform poorly when dynamic load-balancing is required [1]. Since the memory and network resources available to a cache are heavily influenced by its workload characteristics, some load-balancing is required in order to ensure fair distribution of keys across the distributed cache. For instance, if client applications demand a certain range of keys, which happen to be owned by the same cache, the cache’s network bandwidth and memory could quite easily get saturated. Under this scenario, the cache starts facing bottlenecks similar to the primaries. Another instance of losing benefits of caching is when the cache needs to keep evicting keys due to capacity misses. Sharer caches help by allowing local DRAM access to keys which have been fetched from remote owner caches. Since the number of keys that can be fetched for local accesses is far greater than the statically assigned(owned) keys, the access to the owned keys may have to be done on the primary, thereby removing the benefit of remote DRAM accesses and distributed validations. In order to resolve these issues, we propose the use of Google’s Slicer [1], which obtains real-time information about the status of all caches in the system and assigns ownership of key ranges to the caches automatically(using auto-sharding techniques) with the goal of minimizing evictions and balancing the request load.

3. DESIGN

The block diagram in Fig 1 describes the system outline. Logically, the system is composed of three components - a cache which is co-located with the client machine for fast DRAM lookups, a key-to-cache assigner inspired by Slicer, and the primary key-value storage system underlying the cache which handles transaction validation and commit requests. The key contribution of this project is exploring the design space tradeoffs in implementing auto-load balancing in the distributed caching layer. We do this by intelligently moving ownership of keys across caches in response to cache metrics obtained from the client machines. Each of these components and its associated design choices are described in the following subsections.

3.1 Cache

Logically, a cache is located between the client and storage layers. The order to provide DRAM-speed gets, the cache is co-located with the client library (and the application) on the same machine. Each cache instance stores the frequently accessed items by the local application’s client library. Since the caching layer is distributed, Milana introduces a notion of cache cache ownership - whereby each cache maintains ownership of a particular set of keys and is responsible for
ensuring cache consistency with the primary for those set of keys. Additionally, the cache has a sharer cache layer, which works with remote owner caches to fetch frequently accessed keys and store them locally. Though this is a logical separation in our implementation, it is possible to implement the sharer and owner cache as two separate caches. Each owner cache maintains a list of sharers that hold a copy of a key it owns, assigning leases to these copies. A key stored in a sharer cache maintains coherence with the owner (and thereby with the primary) using two mechanisms - self-invalidation (carried out by leases assigned by the owners) and explicit invalidations triggered by the owners in case a read-write transaction is about to be performed on the key. Similarly, down the chain, the owner cache also carries leases issued by the primary on each key it caches - this lease information is propagated to remote sharers for self-invalidation.

In addition to caching keys for local reads, the original cache design in Milana also performed validations for read-write transactions. Read-only transactions are validated in the client library. Read-write transactions are validated by both owner caches and the primaries - the owner caches perform validation on the cached keys, and validation for uncached keys is pushed to the primary. However, we observed in our implementation that auto-sharding introduces some interesting complexities in the cache-validation approach.

Our implementation of auto-sharding primarily balances cache load by moving ownership across caches, which introduces tricky race conditions that affect the correctness of transactions and violate serializability.

**Liveness Violation**: Let us say a slice S is being moved (or split, and moved) from cache C_1 to C_2. This change needs to be communicated to all the caches in the system, so that requests can be re-routed to the new owner. However, since C_1 needs to continue servicing and validating in-flight transactions from the clients that have not yet seen the change in ownership, Slicer needs to stagger the change of ownership message by sending it first to C_1, waiting for it to finish validating all in-flight transactions, and then sending the message to the other caches. This leads to a liveness issue in the system - C_1 could potentially keep receiving requests after the change in ownership message, thus blocking Slicer from informing the other caches of the ownership change indefinitely.

**Safety Violation**: Let us consider what happens when Slicer informs the caches of the change simultaneously, instead of informing C_1 first as described above. In implementing this scenario, we found a race between GET and COMMIT for a certain key while ownership changes are taking place. For instance, cache C_1 is in the middle of a transaction T on key K_1, when it receives an ownership change instruction from Slicer for keys in the T’s read-write set. The new owner, C_2, is also informed of this change. Let us assume the transaction finishes committing on C_1, while new GET requests are routed to C_2 (since it is the new owner). We need to ensure that the GET request on C_2 reflects the latest value from T’s commit on K_1, other-
wise we violate serializability. This can be done using two approaches, both of which are unfeasible for different reasons:

1. Keys can be moved along with the latest data from the old owner to the new owner. However, this causes large use in network bandwidth (depending on key and value sizes) and utilizes cache space unnecessarily on the new owner.

2. Allowing C2 to fetch reads from the primary: There is a race between the committed value from C1 making its way to the primary and the GET request from C2 arriving at the primary. If the GET arrives before the committed value is written to the primary, C2 receives stale data, which violates serializability. Also, the primary cannot inform C2 about the ongoing validation, since it takes place on C1.

The second approach mentioned above is infeasible due to the current implementation of Milana - while the owner cache is performing validations and commits, the primary is left exposed to new GETs. The system is designed such that access to a primary is only through a single owner cache. A different implementation where the primary is made privy to committed transactions in the owner before it can service further requests could work with our Slicer-based design. Since such a massive redesign was out of the scope of this project, we stripped the caches of validation. Therefore, the main benefit of our cache is now in offloading the primary’s DRAM - it does not take up any of the transaction validation responsibilities. We show in following sections why auto-assignment in a distributed cache that does not perform validations still makes for an interesting systems project.

### 3.2 Auto-assignment (Slicer)

Auto-assignment of keys to owner caches is done in a manner similar to Google’s Slicer - the three components of our system are described below.

#### 3.2.1 Slicelet

Each cache has a Slicelet which gathers metrics about cache usage and passes them along to the Assigner at an interval. The Assigner uses these metrics to make decisions about slice assignment.

#### 3.2.2 Clerk

A Slicer Clerk is present in each cache as well - the clerk has the simple job of receiving new assignments from the Assigner and storing them, so that local requests are redirected to the appropriate cache based on updated assignments.

#### 3.2.3 Assigner

The assigner performs the initial key assignment, hashing the application key to a 64-bit slice key, and assigning key ranges (slices) to the available caches. It then is responsible for updating slice assignments across caches based on metrics received from the Slicelets. The two most important measures of cache performance are caches misses and evicts (which can sometimes be correlated). The distributed cache only stores a small percentage of total keys in the underlying KV-Store. Since we want to offload memory pressures from the primary to the cache, it is ideal if most GET requests go to the cache, and all caches are being fully occupied, with as few evicts as possible.

With this objective in mind, the Assigner relies on two metrics from the Slicelets - cache load and key load. Cache load is measured in terms of the fraction of the cache which is full, and key load is measured in terms of how hot or cold a key is, in the context of the LRU list of the cache. Each metric is a number between 0 and 1. The cache load is a simple linear fraction, and the key load is a normalized exponential function. So, for example, a cache load of 0.8 implies that the cache is 80% full, and a key load of 0.8 means that the key is closer to being at the top of the LRU list, and less likely to get evicted. With these metrics, the assigner chooses to split or merge slices based on the following criteria:

1. A cache should not be too full (defined by a threshold to the cache load, say <= 0.85) as long as there is at least one empty cache among its peer caches. This ensures that the cache does not overflow and start evicting items.

2. A cache should not be too empty (defined by a threshold to the cache load, say >= 0.5) as long as there is at least one overloaded cache among its peers - this ensures that no cache is underutilized.

3. Together, these criteria most frequently used keys somewhere in the distributed cache.

#### 3.2.4 Weighted Move

Though the high level idea of the weighted-move is inspired by Google’s Slicer, the way in which moves are performed in our system is quite different from the original paper due to the unique nature of the target application (the distributed cache). The weighted move algorithm merges and splits slices based on the cache load and key load. Merging is performed if there exist two caches that are underutilized, and the newly merged slice does not exceed the upper threshold of a single cache’s load. A slice belonging to an overloaded cache is split across two caches if there exists a cache which can handle the newly split slice (i.e., has a low enough load), by removing a slice of the coldest keys in the overloaded cache and merging them with an adjacent slice. If no such cache exists, then the move is not performed. Unlike the original slicer algorithm, the move budget is identified using the cache load. Key churn is avoided by removing slices that primarily cover keys that are cold (i.e., in the bottom of the LRU list in the overloaded cache). This ensures cache load distribution while avoiding moving hot keys, which could cause the system to lose the benefits of DRAM caching, or even remote caching for that matter.

#### 3.2.5 Deciding Slices

Due to the nature of the distributed cache, the mapping between Slice keys and the keys in the cache is not straightforward, and coordinating owner moves between the caches and Slicer requires thought. The complexity arises due to the fact that Slicer and the cache have different views of the same world. A slice is a contiguous range of hashed keys. These keys (exist in unhashed form) in the cache and are preserved in Least Recently Used (LRU) order within the cache, which is different from Slicer’s order. This difference in key representations is important to reconcile when key
assignments are made from Slicer to the caches. Each message sent from the Slicelet in the cache to the Assigner contains two pieces of information:

1. the cache load, and a set of key loads for the bottom 20% of keys (these will have key loads close to 0), and
2. the top 10% of keys (these will have key loads close to 1) in the LRU list.

The key loads, which are calculated using a normalized exponential function of the key ranks in the LRU list, are then used by the Assigner to determine the appropriate Slice which contains a maximum of the keys with low key load and a minimum (ideally zero) number of keys with high key load. Once a slice is determined, it is communicated to the Slice Clerks in the manner described above, such that races are avoided.

In order to remove the right keys, the Cache maintains a second-tier mapping of key information - a map from the hashed key values to an iterator to the actual element in the cache. This ensures that when the Clerk receives a slice that it needs to remove or add, it can find the appropriate slice using the second-tier mapping in constant time.

4. IMPLEMENTATION

Our implementation extends the existing Milana code-base primarily written in C and C++. The project scope lies in our extension of the validation protocol in the cache and primary, and our Assigner, Clerk, and Slicelet implementations. We briefly describe the implementation challenges below.

4.1 Validation Protocol

Since all validations of keys in a transaction were moved from the owner cache to the primaries, it became necessary to keep the latest information about a key on the primaries prior to starting validation. Since clients would access keys from caches, the LatestReadTimestamp (LRT) for the key is only updated on the cache and not on the primary. This value is maintained on a per-key basis and is used to ensure that a transaction’s commit time is greater than the transaction’s read time for each key in the transaction. Therefore, prior to sending the keys for validation to the primaries, the owner cache would invalidate the sharer’s copy of the key on remote sharers and obtain the LRT for each such shared key. The primaries update the LRT which they obtain along with the validation operation for all keys and then begin the validation phase. With this validation protocol in place, the slicer’s change of ownership works as follows:

- Slicer informs the previous owner about a change of ownership, and it invalidates its cache entries along with any sharer’s copies
- The previous owner obtains the LRT or optionally uses lease expiry time as LRT and flushes it to the primary
- The previous owner forwards all future GETs to the primary, and does not service GETs anymore
- The previous owner then sends an acknowledgement to slicer signaling it to inform other caches of the change

If the slicer informs all caches at the same time about a new assignment, it is possible for a transaction to obtain keys from the old owner, but send the validation request through the new owner. This implies that the primary can start validation without having obtained the LRTs for the keys from the old owner, thereby violating serializability.

4.2 Slicer functionality

The Slicer clerk is written as a separate module which is contained within the cache. It maintains a local mapping of slicer assignments to caches, and updates the map as it receives new assignments. The Slicelet sends the cache load and key load metrics to the assigner at a fixed interval (heartbeat). It also helps the Assigner determine if a cache has failed, and immediately move the keys owned by the failed cache over to a cache from which it has recently received a heartbeat (metric) message.

5. EVALUATION PLAN

In order to compare how the auto-assignment performs, we had aimed to compare our system against an identical implementation with a Consistent Hashing assignment instead of Slicer-based assignment. This system does not change assignments on the fly once they have been assigned initially. Our tests were run on 4 client machines with one dedicated machine for the Slicer assigner. The client machines ran owner and sharer caches without validation, with all validation being performed on the primary. All client machines are connected on a single rack using a ToR switch, while the Slicer machine sat in a different machine room, but within the same LAN. Though we have both implementations complete, functionally, we have not been successful in obtaining latency and throughput data for comparison, since we encountered numerous concurrency issues, despite thoughtful design. This is potentially due to the fact that the project scope widened as we explored each aspect of the design, leading to more development and debugging effort and time being spent than we had planned for.

6. CONCLUSION AND FUTURE WORK

Through this project, we have been successful in understanding the design space and implementation hurdles of integrating dynamic key-assignment system like Slicer into a transactional key-value store. We identified load-balancing metrics within the cache, and came up with a key-assignment algorithm to ensure a fair cache distribution that minimizes cache misses and evictions. We hope to soon have a fully working implementation of the system so we can gather data points, and understand performance challenges in comparison to a Consistent Hashing assignment.

7. REFERENCES


[2] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhandina, K. Iwamoto, B. Kim, L. Matkins, and