2018 IEEE 38th International Conference on Distributed Computing Systems

Scalable Transaction Processing Using Functors

Hua Fan [1], Wojciech Golab [2]
Department of Electrical and Computer Engineering, University of Waterloo, Canada
[1] *h27fan@uwaterloo.ca* [2] *wgolab@uwaterloo.ca*

***Abstract*—Distributed transactions, which access data items at multiple sites atomically, face well-known scalability challenges. To avoid the high overhead, in prior work Fan et al. proposed *Epoch-based Concurrency Control* (ECC), which makes transactions visible at epoch boundaries, and presented a system that supports high performance read-only and write-only transactions. However, this idea has a clear difficulty to overcome: the common case of a single transaction that does both reading and writing. This paper proposes ALOHA-DB, a scalable distributed transaction processing system. ALOHA-DB uses a novel paradigm of serializable transaction processing using *functors*, which conceptually resemble futures in modern programming languages. A functor is a placeholder for the value of a key, which can be computed *asynchronously* in the future *in parallel* with other functor computations of the same or other transactions. With multi-versioning in ECC, the functor computations only rely on accessing historical versions, and so the traditional locking mechanism is not needed for concurrency control. Functors elevate ECC to a new level: supporting serializable distributed read-write transactions. This combination of techniques never aborts transactions due to read-write or write-write conflicts, but allows transactions to fail due to logic errors or constraint violations. We used functor-enabled ECC to implement ALOHA-DB and evaluated it using TPC-C and YCSB read-write distributed transactions. Experimental results demonstrate that our system's performance on the TPC-C benchmark is nearly 2 million transactions per second over 20 eight-core virtual machines, which outperforms Calvin, a state-of-the-art transaction processing and replication layer, by one to two orders of magnitude.**

***Keywords*-distributed transactions; transaction processing; functor; epoch-based concurrency control;**

I. Introduction

Transactions in distributed storage systems are inherently costly in two ways: they require concurrency control for isolation in the presence of read-write and write-write conflicts, and they rely on distributed commitment protocols to ensure atomicity in the presence of failures. The cost of such transactions is especially high for update-intensive workloads in a distributed environment, where conventional techniques fare poorly. In particular, two-phase locking leads to aborts due to (suspected or actual) deadlock, optimistic protocols suffer from aborts due to contention, and two-phase commit (2PC) increases the abort rate further by enlarging the "contention footprint" of a transaction [1], [2]. These conventional techniques take a transaction as the basic unit of concurrency control, meaning that a transaction can only commit keys after all conflicts for these keys have been resolved by holding locks or completing backward validation in optimistic protocols. We refer to these techniques as *transaction-level* concurrency control.

Calvin [2]–[4] aims to boost performance for serializable distributed transactions under contention by enforcing *deterministic* transaction execution scheduling on all partitions to prevent aborts, which avoids wasted work due to transaction restarts but introduces its own overhead. By assuming all involved partitions will successfully execute transactions according to the same deterministic scheduling, a transaction is able to commit keys for one partition as long as all conflicts for these keys have been resolved, irrespective of any unresolved conflict on other partitions. We refer to these methods as *partition-level* concurrency control, which gains more parallelism than the transaction-level scheme in high contention distributed transaction processing.

Recently, the *epoch-based concurrency control* (*ECC*) [5] mechanism was introduced for high performance serializable distributed read-only and write-only (RO/WO) transactions. ECC keeps reads and writes completely separated in time, but it has a clear difficulty to overcome: the common case of a single transaction that does both reading and writing.

This paper proposes a new scalable distributed transaction processing system called ALOHA-DB. ALOHA-DB exploits a novel paradigm of serializable transaction processing using *functors*, which conceptually resemble *futures* [6] in modern programming languages. A functor is a placeholder for the value of a key, and this value can be computed *asynchronously* in the future *in parallel* with other functor computations. Functors elevate ECC to a new level: supporting high performance serializable distributed read-write transactions. In functor-enabled ECC, a read-write transaction is executed in two phases: a *write-only phase* that uses a write-only transaction to store a collection of *functors* under the low overhead of ECC; and a *computing phase* that determines the outcomes of the functors asynchronously.

ECC permits transactions to first write operator placeholders (in the form of functors) without any contention in write epochs, and then compute the outcomes of the operators (functors) asynchronously and in parallel after the write epoch when the order of transactions is fixed. This combination of techniques never aborts transactions due to read-write or write-write conflicts, and yet allows transactions to fail due to logic errors or constraint violations, while guaranteeing serializability.

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00101

With multi-versioning in ECC, the functor computations only rely on accessing historical versions, and so the traditional locking mechanism is not needed for concurrency control. ECC uses timestamp ordering and a decentralized timestamp assignment method, which easily resolve transaction ordering across servers. Without lazily materialized functors, conventional concurrency controls need to resolve the read-write conflicts (assuming a transaction reads before it writes) before writing any keys, which may delay both reads and writes in the transaction, thus increasing the contention footprint and hurting performance further.

Functors also enable more fine-grained concurrency control than in partition-level or transaction-level schemes, and distribute the work for a given transaction in a manner that tends to co-locate computation with storage. *First*, functor computing only focuses on how to compute the value of a key, thus only requires *key-level* concurrency control (i.e., resolving conflicts for generating the value of the functor's key) when ECC already guarantees transaction atomicity and resolves transaction orders. *Second*, a functor is computed on the host where the functor's key is stored, thus the transaction processing overhead can be distributed and offloaded to all the participant partitions.

On the TPC-C benchmark for distributed read-write transactions and a YCSB-like microbenchmark, ALOHA-DB outperforms Calvin [2]–[4] – a state-of-the-art high-performance distributed transaction layer that uses deterministic scheduling – by 1-2 orders of magnitude in terms of throughput, while also maintaining lower latency. Furthermore, ALOHA-DB allows transactions to abort due to logic errors, as required by the benchmark.

II. Review of Epoch-based Concurrency Control

ECC [5] is a technique that achieves serializable transaction isolation and high parallelism for RO/WO transactions. ECC was designed for an in-memory database partitioned across multiple servers connected by a high-speed network in a single private cluster. Tight clock synchronization across servers benefits performance but is not required for correctness of ECC. Standard synchronization techniques suffice, such as NTP executed over a low-latency network.

Conceptually, ECC combines two techniques to mitigate conflicts between transactions. *First*, ECC schedules RO/WO transactions into disjoint time slots, called read and write epochs, to eliminate read-write conflicts between the two groups. *Second*, ECC uses multi-versioning to resolve write-write conflicts in write epochs, which allows update transactions to proceed in parallel even when their write sets overlap. In addition, ECC avoids 2PC and enables atomic commitment of distributed transactions in amortized one round trip.

A server can start processing a transaction only if it holds an appropriate *authorization*, which is granted by the *epoch manager (EM)*. An authorization comprises the epoch type

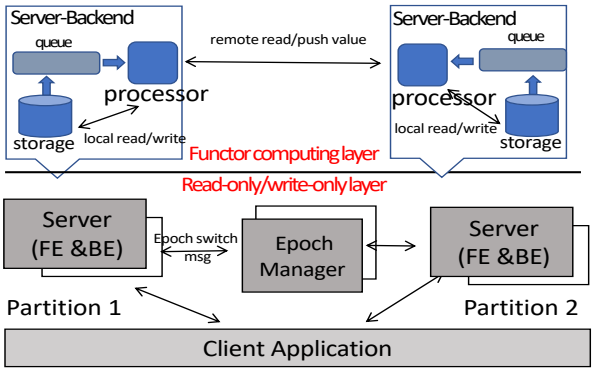

Figure 1. ALOHA-DB architecture.

(read or write), as well as two timestamps indicating a finite *validity period*. Transaction timestamps are obtained using local clocks of servers, and a server can only start a transaction when its local clock is within the validity period. A server may hold at most one authorization at a time, ensuring exclusion between reads and writes. The ECC mechanism guarantees that all write transactions started within an epoch are completed before any of the updates are visible to transactions that start in subsequent epochs. One salient feature of ECC is that it orders transactions using timestamps generated in a decentralized manner. A write transaction is assigned a timestamp within the authorization validity period as the transaction version number when it is started by a server. The server guarantees that the timestamp is globally unique and is within the epoch's validity period to ensure a valid serialization order of transactions [5]. Without using locking and without rejecting obsolete updates, ECC enforces the property that the outcomes of writes can only be observed when the next epoch begins.

III. The ALOHA-DB System

ALOHA-DB is a scalable multi-version in-memory transaction processing system that supports serializable distributed transactions. Using a combination of ECC and *functors* (detailed in the next section), our system minimizes conflicts among transactions. This section describes the system design of ALOHA-DB and focuses on the novel design points as compared to ALOHA-KV [5], which does not support read-write transactions.

A. Architecture

ALOHA-DB is optimized for deployment in a private data center with a high-speed network. Although not necessary for correctness, good network performance and predictability (e.g., low jitter) help our system achieve high throughput and low latency. Specifically, the network latency helps to reduce the epoch switch time, during which no transaction can be started, and benefits clock synchronization among servers when NTP protocol is used. The architecture of ALOHA-DB, illustrated in Figure 1, comprises a collection

of servers and an epoch manager (EM). From the transaction processing perspective, the functor computing layer is built on top of the read-only and write-only transaction processing layer, which is derived from the previous work [5]. Each server performs the functions of both backends (BEs) and frontends (FEs), as in prior work [5]. To facilitate understanding the relationship between ALOHA-DB and [5], we use the terms FE and BE also while describing our system, with the understanding that both terms refer to the same server process.

The EM controls epoch changes by granting and revoking *authorization* at all the FEs, and thus determines when the FEs may start executing transactions. An FE accepts transaction requests from clients, and acts as a transaction coordinator: it starts transaction execution during the correct epoch, generates a timestamp for each transaction, translates transactions to functors (if applicable), communicates with the partitions, and determines the outcome of the BEs. A BE stores the data items in one partition of the database, and serves requests from FEs to read and write these items or functors. BEs also compute functors asynchronously using a component called the *processor*. Whenever there is a functor that is ready to be computed, the BE will push it to a *queue* that will be pulled by the processor. To compute the functors, processors may need to read remotely from other BEs or push data to them. Further details of functor computing are provided in Section IV.

ALOHA-DB is able to leverage the fault tolerance strategies of replication, logging, and checkpointing described in [5] to achieve reliable epoch switching and to avoid data loss in the presence of a single crash failure.

B. Unified Epochs

ALOHA-DB unifies the two epoch types described in Section II to the unified epochs (write epochs), in which write-only transactions and reads accessing old historical versions can be processed at any time. For a read-only transaction requesting the latest version, we adopt an optimization that transforms the transaction to an equivalent read-only transaction for a historical version.

In ALOHA-DB, there is only a series of write epochs in the system. When an FE receives a read-only transaction for the latest version, the FE assigns a timestamp t to the transaction in the write epoch, and delays processing the transaction until the next write epoch begins. Then, the read-only transaction is processed as a read of historical version t. Informally speaking, the read-only transaction is processed as if it happens at t, but it never conflicts with any write transaction within subsequent write epochs.

By eliminating read epochs, ALOHA-DB allows writes to execute faster because there is no longer any read epoch that might block write transactions, while the latest version read latencies may increase because an FE will always delay such read transactions. However, as described in Section IV,

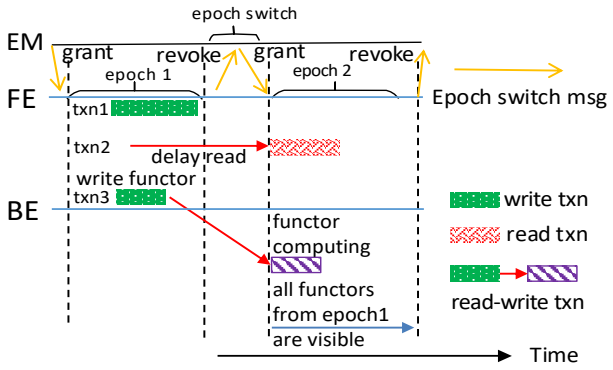

Figure 2. Illustration of unified epochs.

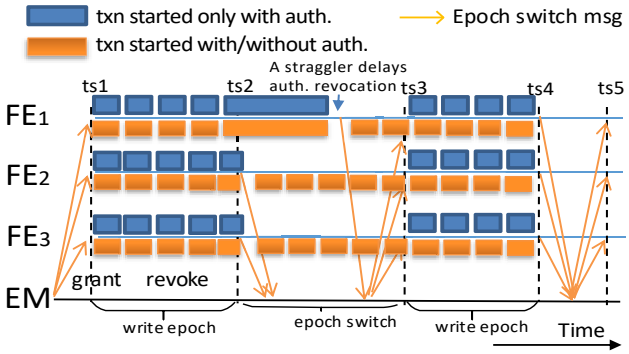

Figure 3. Avoiding straggler side effects by allowing transactions to start without authorization.

a read-write transaction in ALOHA-DB begins reading keys in the read set only after the epoch of the write-only phase completes, and so no additional waiting is required in that case. Moreover, the penalty on read latency for this optimization is bounded by the epoch duration length, which may be tolerable by users when a small epoch duration is used.

Figure 2 illustrates an example of ECC with unified epochs. Transaction 1, a write-only transaction, can be executed during the validity period. Transaction 2, a read-only transaction accessing the latest version, is assigned a timestamp indicating the current version of the data. In the next epoch, the read transaction will be processed as a historical read for the assigned timestamp. Transaction 3, a read-write transaction, has a write-only phase similar to transaction 1 in which it writes the functors to the BEs, and a functor computing phase that may include historical reads similar to transaction 2 using the timestamp assigned in the write-only phase.

C. Avoiding Straggler Side Effects

A straggler transaction is one that prevents an FE from revoking an authorization for a long time. It may delay the start of the next epoch for all FEs, and further degrade the overall throughput. Figure 3 illustrates an example of

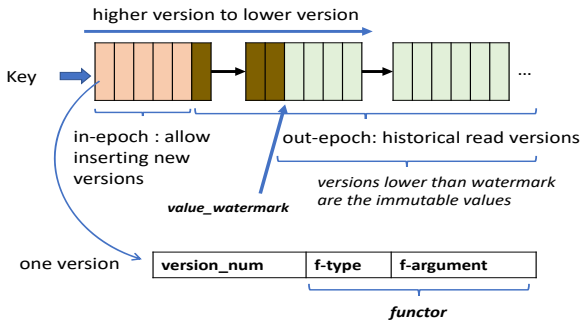

Figure 4. The storage multi-verioning layout for one key.

a delayed epoch due to a straggler at FE_1. In the absence of anomalies, stragglers are unlikely to occur in our system, because the number of in-flight transactions (preventing authorization revocation) decreases to zero after the epoch's finish timestamp is reached, and functor computing occurs asynchronously outside of the epochs.

However, stragglers remain possible in anomalous cases, and for this reason we devise an optimization that avoids the situation where one straggler prevents all FEs from starting the next epoch. It works as follows: FEs can start executing transactions immediately after the authorization is revoked (even without any authorization), as illustrated using the orange bricks between $ts2$ and $ts3$ in Figure 3. These transactions become visible together with transactions started in the following epoch (between $ts3$ and $ts4$). However, this optimization must guarantee that timestamps generated without authorization are smaller than the finish timestamp of the next epoch ($ts4$), as otherwise serializability may be violated. This can be guaranteed by requiring that a transaction without authorization receives a timestamp not exceeding the sum of the previous epoch's finish timestamp ($ts2$) and the duration of the next epoch.

D. Multi-version Storage

ALOHA-DB stores key-functor pairs in a hash-partitioned distributed table. A concrete value of a key is the final form of a functor (see details in Section IV). The functors are versioned to support historical queries, as well as to enable multi-version concurrency control. Figure 4 shows the layout of the versions for one key. For each key, the functors are organized in a logical list ordered by version, implemented as a linked list of arrays. Each *version record* comprises a version number and a functor. The ordered versions favor accessing the latest version not exceeding a given version number and computing functors for a key in ascending order of versions. As explained in Section III-B, the reads in ALOHA-DB are all historical reads, and the writes are assigned a version equal to their timestamp. As a result, the versions are inserted in nearly sorted order, and so ordered versions are maintained easily.

The API functions *Put* and *Get* are used for accessing the storage layer. A *Put* invoked on a new version of a functor requires the version number to be within the epoch validity period. A *Get* returns the latest version of a key's value not exceeding the requested version. All reads in ALOHA-DB only access historical versions which are less than the epoch start timestamp. Thus, the versions for each key are naturally divided into the *in-epoch category* and the *out-epoch category* by the epoch start timestamp. Versions within the in-epoch category are not visible for reading; versions in the out-epoch category are immutable except that functor computing may replace the functor with its final value. *Get* triggers the functor computing if the functor to be read is not a final value, and replaces this functor with its final value.

Each key also maintains a special version number called the *value watermark*, below which all the versions are the final value after the functor computing phase (detailed in Section IV). Accessing a version below the value watermark needs no synchronization because these versions are immutable. In the implementation, we use a lock-free data structure to allow multiple threads to read the storage concurrently.

IV. Functors

Functors resemble *futures* [6] in programming language research, which are objects used to represent the future result of asynchronous computations. Functor computing only reads historical versions, thus no locking mechanism is needed on keys when multi-version storage is used. In contrast to other mechanisms that use transaction-level or partition-level concurrency control, functor-enabled ECC uses functors as placeholders for values in the write set, and the functors of a transaction are computed independently and in parallel. While the basic ECC mechanism provides transaction atomicity and transaction ordering, functor-enabled ECC further adds a key-level concurrency control scheme for read-write transactions that enables high parallelism even under contention.

A. Transaction Lifecycle

From the client's point of view, our transaction model is similar to Calvin [2]. We assume that transactions are submitted "one-shot" (i.e., non-interactively) from clients and processed by invoking stored procedures at servers. A transaction is expressed as a read set and a write set (of keys), as well as a set of arguments supplied by the client. The keys accessed by a transaction must be known ahead of time, which is a restriction also present in Calvin. But ALOHA-DB does not have this restriction for read-only transactions for historical versions, which are common in analytic workloads. Section IV-E discusses the work-around for this restriction.

The lifecycle of a read-write transaction in functor-enabled ECC includes the following phases: (1) The FE transforms the transaction to key-functor pairs that are written to the storage system in a write epoch. Each key in the transaction write set will have a functor representing the value of the key after the transaction is executed. All the functors of a transaction share the same transaction version (timestamp). (2) The functors are computed asynchronously after the write epoch, or on-demand at the time of a read. Once computed, each functor is updated with an immutable final value. Each functor can therefore be computed at most once. (3) Based on the client's request, the FEs can acknowledge the transaction execution result once the write-only phase completes, or when the functor computing phase completes. If the transaction may be aborted in the functor computing phase, the former acknowledgment option still allows clients to learn the transaction outcome (commit or abort) by issuing a separate read request for the result of any of the transaction's functors, because any of the functors will result in abort if the transaction is aborted.

B. Functors for Read-Write Transactions

Interface. A functor is composed of an *f-type* and an *f-argument*. The f-type specifies which handler to call to compute the functor. The f-argument is a blob whose interpretation is based on the f-type. Table I shows some examples of f-types and their f-argument representations.

f-type	f-argument
VALUE	the literal value of the key
ABORTED	none
DELETED	none
ADD/SUBTR	numerical (e.g., increment value by 1)
MAX/MIN	numerical (e.g., update the value if it is smaller)
user-defined ...	read set and arguments

Table I
EXAMPLES OF SOME F-TYPES AND THEIR F-ARGUMENT REPRESENTATIONS.

The f-type **VALUE** denotes that the f-argument itself is the value, hence no computing is needed for this kind of functor. The f-type **ABORTED** means that this version of the value is aborted, while the f-type **DELETED** is a tombstone of the key, denoting that this key is deleted as of this version. The functors of other f-types require computation that may replace the functor by the value of a key.

Programmers can also create user-defined f-types and the corresponding f-arguments. The user-defined f-type indicates which handler to call for computing the functor. The user-defined f-argument has a functor read set and arguments, which indicate the inputs for the handler. In particular, the functor computing phase requires reading all keys in the functor read set for the latest version not exceeding the functor version. The read set of some functors comprises only the key to which the functor was written, in which case the read set is omitted (e.g., **ADD**, **SUBTR**, **MAX**, **MIN**).

Transforming a transaction to functors. In general, to generate a functor for a key in the write set, we can generate the f-argument by taking the transaction read set and any arguments that influence the result of the key. The corresponding functor handler can be generated in a similar way. In the current implementation, transactions are transformed to functors manually and automating this process is future work. We also include an optimization in functor generation: a functor also includes the recipient set, which is the set of keys whose functor's read set includes this key in the transaction. The recipient set is added to a functor whose key should be read in the computing phase of other functors of the transaction. This optimization is used to achieve *proactive remote reads* for other functors, meaning that the computing phase of this functor involves *pushing* the latest value of this key before this functor to other functors. This optimization speeds up functor computation and is not required for correctness.

C. Functor Computing

A transaction is transformed to a collection of functors, and the functors from the same transaction are computed independently and in parallel, as explained shortly.

Computing handler. The functors are computed by handlers in the server-backend based on their f-type. In BE storage, a functor is associated with a version of a key (see Section III-D). Functors may be computed by a scheduled thread pool-based *processor* in the BE, and may also be computed on-demand at the time when the value is requested by a read, whichever occurs first. Further details regarding functor computing in ALOHA-DB are presented in Section IV-D.

Each functor is computed by the *Func* procedure shown in Algorithm 1, which calls the functor computing handler determined by the f-type. Functors that are in their final states with f-type **VALUE**, **ABORTED** or **DELETED** do not need the computing phase. For other types of functors, the computation begins with deciding the required version for each key in the read set, which is the latest version lower than the version of the functor. Reading is achieved by calling the *Get* function with a version number one less than the version number of the functor, which retrieves data from the multi-version storage of the corresponding partition. This is done only after the write epoch in which a functor was written finishes. Thus, the functor computing phase is able to determine the outcome of any transaction with a lower version number whenever a "reads-from" dependency exists, without blocking. Furthermore, functor computing only accesses lower versions (historical versions), which can be read without synchronization if they are final values. If the lower version is a functor that requires computation,

the reading thread will compute that functor first, and then update the functor to its final value (line 21).

After reading the values of keys in the read set of the functor, the handler corresponding to the f-type is called. The values read as well as the f-argument are used as inputs to the handler procedure. The output of the handler is used to update the functor, and the functor is updated with the final value at most once.

Arbitrary abort. In contrast to deterministic transaction processing schemes that ensure transactions never abort, ECC allows a transaction to abort either in the in-epoch phase or in the functor computing phase. In the former case, the FE as the transaction coordinator can send a second round of messages to abort the transaction if any partition fails. In the latter case, the functor computing can decide to abort the transaction as the output, for example due to an error condition (e.g., insufficient funds for debit). In that case, any keys that influence the abort decision must be in the read sets of all the functors, because they influence the result of all functors of the transaction, and all functors must reflect the same abort/commit decision.

Functor computing examples. We present examples of functor computing for three consecutive transactions in Figure 5. We demonstrate the states before functor computing on the left side, and the right side shows the states after functor computing.

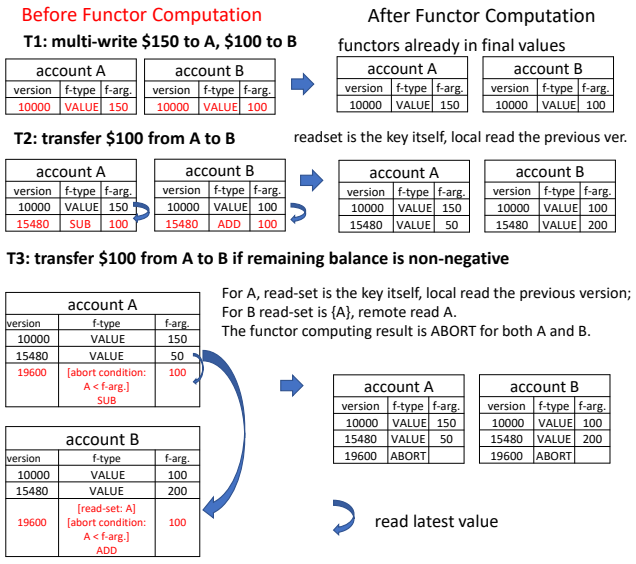

Figure 5. Example of three transactions executed using functors over two data items.

D. Functor Processing In ALOHA-DB

Computing the result of a functor requires reading the previous version of any keys in the f-argument's read set, which means that a higher version functor may depend on one or more lower version functors. In the BE, we use a thread pool-based *processor* to asynchronously compute all uncomputed functors in increasing order of version number. When a new epoch begins, all functors inserted in previous epochs are ready to be processed, and so their meta-data (key and version), which were buffered in the previous epoch, are pushed to a queue for the processor to consume. Each key maintains a watermark, namely a version number below which all versions of functors have already been computed.

Algorithm 1: Functor computing for a specific key k.

```
– records[k]: array of ordered records for key k, each
  record of the form <version v, f-type t, f-arg arg >
– watermarks[k]: value watermark for key k
1  Procedure Compute (k: key, v: version)
2      w ← watermarks[k]
3      // compute functors from version w to version v
         for key k
4      foreach record r ∈ records[k] s.t. r.v ∈ [w, v] do
5          if r.t ∉ {VALUE, ABORT, DELETE} then
6              update r using the result of Func (k, r)
7      while w < v do
8          CmpAndSwap(watermarks[k], w, v)
9          w ← watermarks[k]
10 Procedure Func (k: key, r: record)
11     reads : container (map) for values read
12     for rk ∈ read set of functor in r do
13         reads[rk] ← Get(rk, r.v − 1)
14     f: handler denoted by r.f
15     return f(reads, r)
16 Procedure Get (k: key, v: version)
17     r : latest record for k not exceeding version v
18     if r.t = DELETE then
19         return ⊥ // denotes deleted key
20     if r.t ∉ {VALUE, ABORT} then
21         Compute (k, r.v)
22     if r.t = ABORT then
23         return Get(k, v − 1)
24     return r.arg
```

Algorithm 1 presents the pseudocode for functor computing. For simplicity, in the pseudocode the processor always processes all uncomputed functors of a given key from the watermark to the version obtained from the queue, and then updates the watermark. In the implementation, the version obtained from the queue will be processed first if it does not depend on the previous versions of its key. This is done to boost processing parallelism.

Processors are also responsible for *remote reading* when the functor needs to read a value from another partition, and for *pushing values* whereby the latest value of a key before the functor version is sent to the functors of any keys in the recipient set (see Section IV-B). Pushing a value is a proactive form of reading. As described in Section III, reads may also trigger functor computing on-demand if

the value of a functor is not yet available. At read time, computing a functor may involve a recursively computing dependent functors with lower versions. However, this only happens in the case when a read occurs for the functor before the the asynchronous processing for that functor has been completed.

E. Dependent Transactions

Transactions that must perform reads in order to determine the full read set and write set are called *dependent transactions* [2]. This subsection presents two methods for supporting dependent transactions in ALOHA-DB. Those two methods do not exclude each other, and dependent transactions can choose one or both methods based on the transaction characteristic.

Optimistic approach. ALOHA-DB allows transactions to abort in the functor computing phase. Thus, ALOHA-DB can natively use an *optimistic* approach similar to Hyder [7], which executes transactions by reading from a snapshot and then performs backward validation in the functor computing. In particular, a transaction first reads all required keys for some timestamp (e.g. ts_r), determines the write set, and writes all functors with a timestamp (e.g. ts_w). The functors will check whether any values in the read set have changed between the two timestamps — ts_r and ts_w in the example, abort the transaction if so, and commit the transaction otherwise.

In contrast to Hyder, where the validation procedure visits all data versions in the log order whether or not they are relevant to a given transaction, ALOHA-DB functor computing only requires the latest previous versions of keys in the transaction's read set. This allows multiple transactions to be validated in parallel. A detailed performance comparison between Hyder and ALOHA-DB is left as a future work.

Key dependency. For a dependent transaction, the transformation from a transaction to functors that generates a functor for each key in that write set cannot be achieved until the functor computing phase when the functors can read previous versions of keys. To solve the problem without resorting to optimistic concurrency control (OCC), we defer the write-only phase for the keys that can only be determined as part of the write set during the functor computing phase. These keys are called *dependent keys*, because they are decided in the functor computing phase of functors belonging to other keys in the same transaction. We refer to these functors as *determinate functors*, and the keys that determinate functors belong to are called *determinate keys*.

For example, consider a transaction that will write key B only if the value of key A satisfies some condition. This transaction will choose A as a determinate key, record a determinate functor for A in the write-only phase that will write B (dependent key) in the functor computing phase if the value of A satisfies the condition, and store no functor for key B in write-only phase. If B is written, the version number applied to the dependent key B will be the same as that of the determinate functor, because all the writes belong to the same transaction. Thus, whenever calling Get on the dependent key B for a timestamp ts, the value watermark of key A must be at least ts to guarantee that all "deferred writes" on B have completed. In other words, for any given version, key A's functors must be computed first before reading the same version for key B, otherwise serializability may be violated.

V. Evaluation

To evaluate the performance envelope of functor-enabled ECC, we implemented ALOHA-DB on top of the ALOHA-KV [5] codebase, which is programmed in C++ using the popular open-source RPC framework fbthrift [8]. We run the TPC-C, the scaled TPC-C (detailed in Section V-A), and the YCSB-like microbenchmark on ALOHA-DB and Calvin [2]. ALOHA-DB fully implements the aborting requirements for TPC-C NewOrder transactions, in contrast to the open-source implementation of Calvin. ALOHA-DB outperforms Calvin in terms of throughput under various contention scenarios, indicating that our system has lower overhead for distributed transactions even under high contention. Specifically, our results show that ALOHA-DB achieves around 2 million distributed NewOrder transactions per second across 20 servers, which is 13–112× faster than Calvin.

A. Experimental Setup

1) Workload: **TPC-C** [9] is a standard benchmark for online transaction processing (OLTP). Similarly to prior work [2], [10], [11], our experiments focus on NewOrder and Payment transactions for distributed read-write transactions.

Scaled TPC-C [1], modifies the partition-by-warehouse approach, where each server holds all data related to one or more warehouses, by partitioning data within one warehouse. This workload is more suitable for stress-testing the performance of distributed transactions. The Scaled TPC-C treats the database as a single warehouse, partitions the database by item and district, and simulates the behavior of a large warehouse that spans many hosts. Our experiments implement and evaluate the scaled benchmark, denoted by *Scaled TPC-C*, as well as the conventional partition-by-warehouse benchmark that is used in Calvin papers [2], [3], denoted by *TPC-C*. Payment transactions are only implemented in TPC-C, because the Scaled TPC-C partition strategy in [1] removes the w_ytd field from the warehouse table, which is needed for the Payment transaction. For a fair comparison, (non-scaled) TPC-C transactions are generated in the same way as in Calvin: a distributed transaction always accesses a second warehouse that is not on the same server as the first.

YCSB [12], is a benchmark designed for Internet-scale storage systems. YCSB does not include a standard read-write transaction workload. We choose the YCSB-like mi-

crobenchmark implemented in Calvin, because it has a tuning knob (contention index) that can accurately specify the contention on a partition for a distributed transaction, and it is the main workload used in previous Calvin evaluations [2], [3]. We reproduce the microbenchmark implementation of Calvin [3] for experiments with ALOHA-DB. In the microbenchmark, each server contains a database partition consisting of 1M keys. On each partition, the data items are divided into "hot keys" and "cold keys". Each transaction reads 10 keys then updates the keys by increasing the value by 1, and accesses exactly one hot key at each participant partition. A distributed transaction touches two partitions. When each partition has K hot keys, the contention index (CI) is defined as $1/K$. For example, a CI value of 0.01 denotes that each transaction executed on a partition must access one of the 100 hot keys.

If not specified otherwise, all transactions in our experiments are distributed transactions, which update data items on more than one server.

2) Comparison of systems: If not specified otherwise, ALOHA-DB uses a 25ms unified epoch duration, which we believe provides a performance balance between write throughput and read latency. In comparison, Calvin's sequencer batches requests in epochs of 20ms, because we found that Calvin has no significant throughput improvement with 25ms epochs but only longer latencies, and lower epochs result in lower throughput under the same configuration. Both systems are configured for in-memory storage, and fault tolerance is disabled by default to follow the same convention as in the Calvin papers [2], [3].

ALOHA-DB fully implements the requirement that 1% of NewOrder transactions must abort. Specifically, the aborted transaction includes an item that cannot be found in the corresponding partition, while other partitions process the transaction as usual in the first phase of the transaction commitment protocol. The transaction coordinator (FE) must issue the second round of messages to abort the transaction and roll back the processing on the other partitions. In contrast, Calvin's implementation does not support aborted transactions because of its deterministic design [3]. As a result, Calvin is able to pre-assign the order_id efficiently for a NewOrder transaction, whereas ALOHA-DB must assign the order_id dynamically in the determinate functor processing phase (see Section IV-E). Specifically, the next_order_id is the determinate key of the Order, NewOrder and OrderLine tables.

For an apples-to-apples comparison, ALOHA-DB submits a batch of transaction requests in each RPC call, similarly to Calvin. This ensures that neither system is bottlenecked on the RPC layer.

3) Environment: By default, we use eight Amazon EC2 m4.4xlarge virtual machine instances with hyper-threading disabled (8 cores total). We choose such instances because existing Calvin code is optimized for 8-core machines. However, we believe that ALOHA-DB has potential for higher performance using more powerful machines. A BE/FE pair is co-located in one process at each host and the EM process shares one of these hosts. We present the aggregate throughput and average latency. The transaction latency measurement in both systems is made in the same way: from when the transaction is issued by the client until its functors (ALOHA-DB) or replicated transactions (Calvin) are fully processed. Both systems focus on server-side latency, and the latency results do not include the final response to the client. There are three runs for each combination of parameters, and variation across runs is indicated using vertical error bars representing the min and max measurement. In most cases the error bars are imperceptibly small.

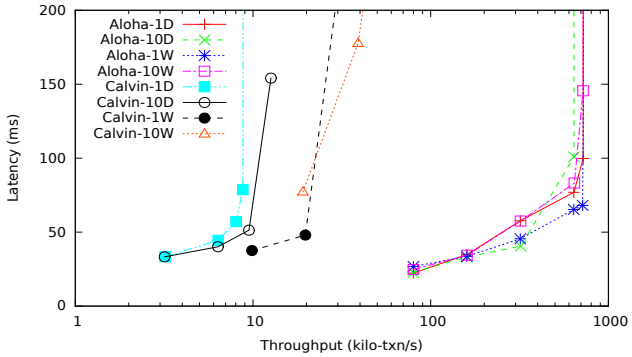

Figure 6. Throughput vs. latency: ALOHA-DB and Calvin experiments for NewOrder transactions. Logarithmic scale used for horizontal axis. **1W** or **10W** denotes 1 or 10 warehouses per host in TPC-C experiments; **1D** or **10D** denotes 1 or 10 districts per host in scaled TPC-C experiments.

B. TPC-C Experiments

1) Throughput vs. Latency: We first present the results for throughput vs. latency of NewOrder transactions under both TPC-C and scaled TPC-C in Figure 6. For TPC-C experiments, two partition settings are used: 1 or 10 warehouses per host, denoted as **1W** or **10W** respectively. Similarly, **1D** and **10D** denote 1 or 10 districts per host in scaled TPC-C experiments.

In terms of peak throughput, ALOHA-DB achieves around 13× (in TPC-C) and 61× (in scaled TPC-C) greater than Calvin. Our Calvin-10W peak throughput, around 60k tps (from a separate investigation which is not shown in Figure 6) is comparable to the results in [3], though higher because we use more powerful virtual machines. However, with fewer warehouses per host or under a scaled TPC-C workload, Calvin suffers a significant throughput drop, while ALOHA-DB's performance under different settings is much more steady.

The latency results show that Calvin has higher latency than ALOHA-DB in light workloads, while sustaining much lower throughput. As explained in [3], preprocessing of requests in the scheduling layer contributes to latency in Calvin. Although, the latency in ALOHA-DB also includes

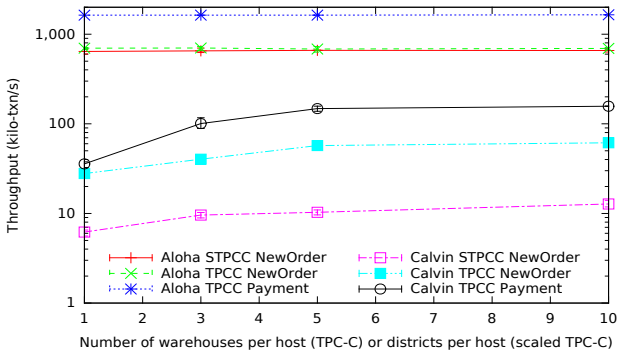

Figure 7. ALOHA-DB and Calvin throughput for NewOrder and Payment transactions under various numbers of warehouses/districts per host. Logarithmic scale used for vertical axis.

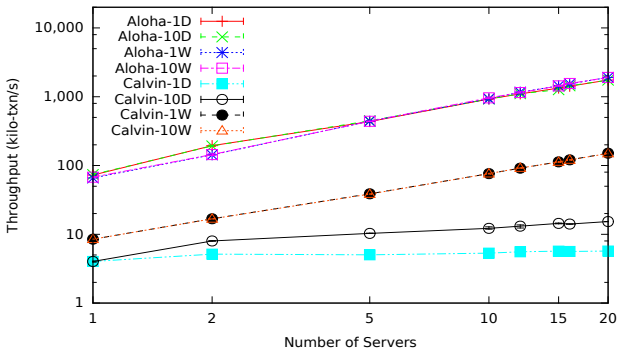

Figure 8. ALOHA-DB and Calvin scale-out performance for NewOrder transactions. Logarithmic scales used for both axes.

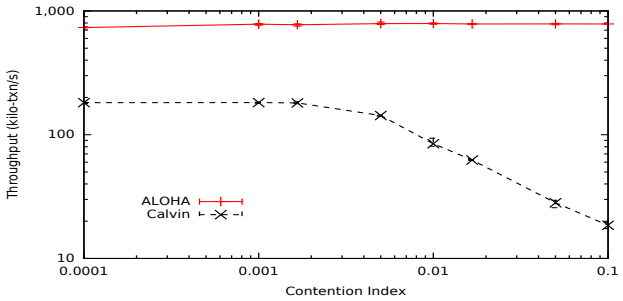

Figure 9. ALOHA-DB and Calvin microbenchmark performance under various values of the contention index. Logarithmic scales used for both axes.

the time functors spend waiting for the epoch to finish before starting computing, thus the average latency must be larger than half of the epoch duration.

2) Database Partitioning: The database partition strategies affect transaction contention and distributed transaction access patterns. Given the same workload in TPC-C experiments, having fewer warehouses at one host creates more contention for each warehouse. In particular, the Payment transaction has higher contention than the NewOrder transaction, as it updates the warehouse table. In terms of distribution, a NewOrder distributed transaction in TPC-C experiments only contacts two partitions, but likely contacts more than two partitions in scaled TPC-C where partitioning is done by item.

Figure 7 shows the results for TPC-C NewOrder and Payment transactions with 1 to 10 warehouses per host, denoted by TPCC, and Scaled TPC-C NewOrder trasnactions with 1 to 10 districts per host, denoted by STPCC. Considering various numbers of warehouses per host in TPC-C, Calvin exhibits a performance drop when the number of warehouses decreases, and the drop is more severe when the number of warehouses is small. For example, the throughput of NewOrder transactions in Calvin drops from 40k to 28k when the number of warehouses changes from 3 to 1. The Payment transactions in Calvin begin to suffer a throughput penalty when the number of warehouses is less than 5, because the contention increases on the warehouse table when each host has fewer warehouses.

In comparison, the performance drop under high contention or high transaction distribution is less than 5% in ALOHA-DB, even in the case of 1 warehouse per host or 1 district per host experiments. ALOHA-DB supports a high performance write-only phase thanks to ECC (see the ALOHA-KV paper [5]), and the functor computing phase uses fine grained (key-level) concurrency control to achieve high parallelism. In the high contention cases, we believe that functor computing also benefits from sequential memory access, when many functors of the same key are processed together (see line 4 in Algorithm 1), while Calvin computes keys belonging to the same transaction together.

3) Scale-Out: Figure 8 presents the NewOrder transaction throughput results using up to 20 servers. We observe nearly linear scalability with the exception of Calvin with Scaled TPC-C. ALOHA-DB achieves up to around 2 million transactions per second in total, which is 13–112× faster than Calvin. For Scaled TPC-C, Calvin does not scale well because a transaction likely needs to contact more partitions as the number of servers increases, while it only contacts two partitions in TPC-C. In contrast, ALOHA-DB's functor processing overheads do not increase significantly when a transaction needs to contact additional partitions, as explained in next subsection.

C. Microbenchmark Experiments

1) Skewed workload: Typically, distributed transaction processing using conventional concurrency control suffers under a skewed workload, because transactions are forced to wait for the contented keys and coordination involving remote servers (e.g., 2PC) is usually slow. Figure 9 demonstrates the throughput of ALOHA-DB and Calvin under various contention index settings. When the CI is less than 0.0017 (600 hot keys per partition), Calvin still performs around the peak throughput. However, the throughput begins to decline with a more skewed workload. ALOHA-DB does

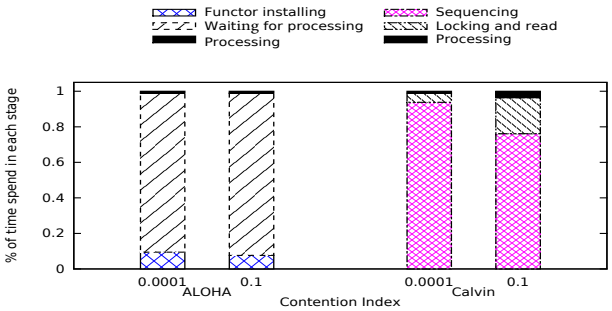

Figure 10. Latency breakdown: latency of different stages of a transaction lifecycle in ALOHA-DB and Calvin under low and high contentions.

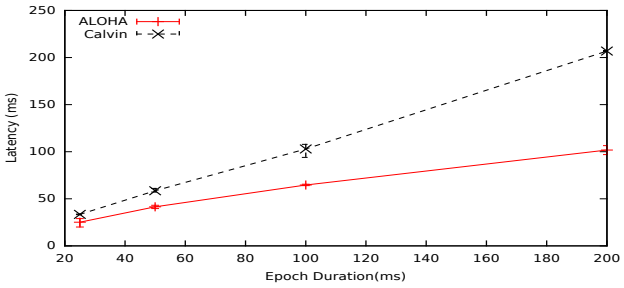

Figure 11. ALOHA-DB and Calvin latency under various epoch duration.

not suffer a significant throughput drop in these settings. In the extremely skewed case (CI is 0.1, 10 hot keys per partition), each partition processes nearly 97k txn/s on average.

We also break down the latency of low and high contention cases (CI 0.0001 and 0.1), under a light workload (5% of peak throughput). Figure 10 shows the percentage of time used in different stages of transaction processing. In ALOHA-DB, the *Functor installing* stage measures the duration from when a transaction is issued to when a functor is installed in the BE (the epoch may be unfinished); *Waiting for processing* describes the duration from when the functor is installed to when the functor is retrieved by processors; *Processing* time denotes the stored procedure running time for the functor computing. For Calvin, *Sequencing* stage denotes the duration from when a transaction is issued to when the partition scheduler begins to process the transaction (comparable to the functor installing and waiting for processing in ALOHA-DB); *Locking and read* duration includes the time of locking all required locks and reading keys in the read-set; *Processing* denotes the stored procedure running time. In both systems, the processing stage takes the minimum time, and the largest part is spent completing the epoch. However, the latency of Calvin is more sensitive to high contention, as in the high contention case it spends more time in the locking phase. We believe Calvin is bottlenecked in the single-threaded lock manager when contention on hot keys is high.

2) Varying epoch duration: Figure 11 presents the latency of ALOHA-DB and Clavin for various epoch durations, under medium contention (CI 0.001) and a light workload. From the results, we can see that the average latency is nearly linear with respect to the epoch duration for both systems, although the slopes are different. For ALOHA-DB, functors need to wait half of the epoch duration on average after they are installed in the BE, and so the linear slope is close to 0.5. The open-source Calvin implementation generates most of the transactions at the beginning of the epoch. Thus, in the figure we observe that the linear slope is close to 1 for Calvin.

D. Discussion

Here we summarize some of the design characteristics of the two systems, and their performance implications, to better understand the experimental results.

(1) Calvin uses partition-level concurrency control. Calvin first replicates a transaction to all involved partitions. Each of these partitions reads all the values in the read-set, *redundantly* executes the same stored procedure on each partition, but only writes the keys belonging to this partition. However, as all partitions read the same read set, some remote reads result in wasted work for a partition where the key read has no influence on the writes performed in this partition. Each functor is computed only once in ALOHA-DB, which avoids the redundant transaction processing that happens in every participating partition in Calvin.

(2) The write-only phase in ALOHA-DB has concurrency control overhead close to eventual consistency. This point was established in [5] by showing that ALOHA-KV has performance close to the baseline of no concurrency control. This is because write-only transactions in write epochs require almost no concurrency control and the epoch switch has minimal overhead.

(3) Multi-versioning and key-level concurrency control in ALOHA-DB provide more parallelism than the single versioned partition-level concurrency control in Calvin, which uses a single-threaded lock manager. Thus, even in the extreme case of 1D or 1W where Calvin may execute transactions one by one in a partition, ALOHA-DB still allows different threads to compute different functors in parallel. In ALOHA-DB, a functor cannot be computed if a version to be read is not a final value. However, in this case the thread will begin to compute that functor on which it depends, rather than blocking until another thread computes this functor (see line 21 in Algorithm 1). This resembles a rescheduling of the functor execution, and ensures that threads are well-utilized.

VI. Related Work

There are many efforts in the research community to support serializable distributed transactions. Many of these use variants of traditional concurrency control mechanisms: for example, Spanner [13] and Sinfonia [14] use two-phase locking, whereas Centiman [15], Rococo [1], Tapir [16],

Hyder [7], and Hekaton [17] implement OCC. However, these concurrency control mechanisms do not scale on many workloads, in particular update-intensive workloads. A recent evaluation [11] shows that for these conventional concurrency control mechanisms (excluding deterministic scheduling in Calvin), the performance of distributed transactions on a cluster only slightly exceeds that of a single machine under a high contention read-write workload. Some other systems trade serializability for scalability and only support restricted transaction types [18], [19], or provide weaker forms of isolation [20]–[22].

Some recent systems address distributed transactions through deterministic scheduling [2]–[4], [23], or by reordering conflicting transaction executions [1], [24]. As shown in our experiments, Calvin [2]–[4] suffers from the latency overheads related to the scheduling phase, and moreover its open-source implementation cannot abort transactions arbitrarily. Faleiro et al. use the placeholder approaches [25], [26] for deterministic databases, but their systems only handle single machine transactions and cannot execute conflicting transactions in parallel using key-level concurrency control. Janus [24] avoids Calvin's redundant execution among partitions by sending transaction logic to partitions as pieces, targeting a more restrictive transaction model (execution of each piece only accesses data on the local partition) and only providing partition-level concurrency control. ROCOCO [1] is a concurrency control mechanism for distributed transactions that uses a two-round protocol to detect conflicts and re-order transaction execution. Our system determines the transaction order by the timestamps generated at the start of transactions, and no conflict detection between transactions is needed. ALOHA-DB has superior performance compared to published results for ROCOCO [1].

The high-level idea of executing read-write transactions on top of a blind write layer (write-only transactions) was previously explored in some scale-out distributed databases [7], [27], [28]. Hyder [7] is a log-structured database that executes transactions optimistically using a snapshot version, and appends writes as "intentions" atomically to the log. Then, a centralized validation phase is needed to decide if the transaction should be committed or aborted. Hyder achieves atomic multi-key writing by atomically writing a log entry in Corfu [29], which uses a centralized *sequencer* to guarantee a total order on the log entries. This sequencer can limit the peak write throughput of Hyder to only sub-million transactions per second. ALOHA-DB instead orders transactions using the timestamps assigned by distributed front-end (FE) servers. The processing of functors differs from the "melding" procedure in Hyder [7] in several ways: functors are placeholders for values whereas "intentions" in Hyder record concrete values; functors are evaluated using the latest version of the data as opposed to a slightly stale snapshot. Thus, Hyder transactions are prone to aborting during the "melding" phase under high contention, whereas ALOHA-DB never aborts transactions due to conflicts. Furthermore, to read the values of some keys, Hyder must "roll forward" all the sequential log entries, even if some of the entries do not affect the keys of interest, whereas ALOHA-DB only needs to compute the functors that are related to the keys of interest.

A recent paper [30] shares some similarity with ALOHA-DB by using the techniques of epochs and a placeholder approach for different purposes, which focuses on replaying recovery logs other than concurrency control. In particular, functor-enabled ECC itself is a concurrency control mechanism, while [30] addresses log replay in primary-backup replication, which requires another concurrency control mechanism (on the primary server) to resolve conflicts and to decide transaction ordering, and create epochs for the backup server. The placeholder in [30] is only an empty value, and it still uses transaction-level parallelism; functors are placeholders and methods to get the final values, and functors make key-level parallelism possible in transaction processing.

The concept of *functors* in this paper resembles *futures* [6] in modern programming languages, which are objects used to represent the future result of asynchronous computations. As far as we are aware, this paper is the first effort to use functors to process distributed transactions using key-level concurrency control.

VII. CONCLUSION

This paper proposes a transaction processing system called ALOHA-DB for supporting high-throughput distributed transactions. We introduce a distributed protocol for serializable read-write transactions by extending the ECC for RO/WO transactions [5]. Our method uses write epochs to record functors, which are objects that represent how to evaluate the corresponding versions of values. A functor is processed either during asynchronous batch processing or at read time, once all versions on which the functor depends are settled. We evaluated ALOHA-DB using the TPC-C benchmark and the YCSB-like microbenchmark. For the TPC-C benchmark, ALOHA-DB achieves close to 2 million transactions per second on 20 servers, which is 1 to 2 orders of magnitude faster than Calvin [2], while also supporting lower latency and allowing transactions to abort due to errors.

ACKNOWLEDGMENT

The authors thank Dr. Brad Morrey (Google) for his feedback on earlier stages of this work, and thank Prof. Angela Demke Brown (University of Toronto), Prof. Bernard Wong, Prof. Lin Tan, Prof. Derek Rayside (University of Waterloo) and the anonymous referees for their helpful comments.

REFERENCES

[1] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 479–494.

[2] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 1–12.

[3] K. Ren, A. Thomson, and D. J. Abadi, "An evaluation of the advantages and disadvantages of deterministic database systems," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 821–832, Jun. 2014.

[4] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 70–80, Sep. 2010.

[5] H. Fan, W. Golab, and C. B. Morrey III, "ALOHA-KV: High performance read-only andwrite-only distributed transactions," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 561–572.

[6] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," *SIGPLAN Not.*, vol. 23, no. 7, pp. 260–267, Jun. 1988.

[7] P. Bernstein, C. Reid, and S. Das, "Hyder - a transactional record manager for shared flash," in *Proceedings of CIDR*, 2011, pp. 9–20.

[8] "fbthrift," https://github.com/facebook/fbthrift.

[9] Transaction Processing Performance Council (TPC), "TPC Benchmark C standard Sepcification Revisiosn 5.11," http://www.tpc.org/tpcc, 2010.

[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 185–196, Nov. 2014.

[11] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 553–564, Jan. 2017.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[14] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," *ACM Trans. Comput. Syst.*, vol. 27, no. 3, pp. 5:1–5:48, Nov. 2009.

[15] B. Ding, L. Kot, A. Demers, and J. Gehrke, "Centiman: Elastic, high performance optimistic concurrency control by watermarking," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 262–275.

[16] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015, pp. 263–278.

[17] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 1243–1254.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 313–328.

[19] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 385–400.

[20] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica, "Scalable atomic visibility with RAMP transactions," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 27–38.

[21] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 113–126.

[22] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, "The end of a myth: Distributed transactions can scale," *Proc. VLDB Endow.*, vol. 10, no. 6, pp. 685–696, Feb. 2017.

[23] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[24] S. Mu, L. Nelson, W. Lloyd, and J. Li, "Consolidating concurrency control and consensus for commits under conflicts," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 517–532.

[25] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 613–624, Jan. 2017.

[26] J. M. Faleiro, A. Thomson, and D. J. Abadi, "Lazy evaluation of transactions in database systems," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 15–26.

[27] P. A. Bernstein, S. Das, B. Ding, and M. Pilman, "Optimizing optimistic concurrency control for tree-structured, log-structured databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1295–1309.

[28] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner, "Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1716–1727, Aug. 2015.

[29] M. Balakrishnan et al., "CORFU: A Distributed Shared Log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, pp. 10:1–10:24, Dec. 2013.

[30] D. Qin, A. D. Brown, and A. Goel, "Scalable replay-based replication for fast databases," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2025–2036, Sep. 2017.