Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle

Adriana Szekeres
University of Washington
aaasz@cs.washington.edu

Michael Whittaker
UC Berkeley
mjwhittaker@berkeley.edu

Jialin Li
University of Washington
lijl@cs.washington.edu

Naveen Kr. Sharma
University of Washington
naveenks@cs.washington.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

Dan R. K. Ports
Microsoft Research
dan@drkp.net

Irene Zhang
Microsoft Research
irene.zhang@microsoft.com

Abstract

Traditionally, the high cost of network communication between servers has hidden the impact of cross-core coordination in replicated systems. However, new technologies, like kernel-bypass networking and faster network links, have exposed hidden bottlenecks in distributed systems.

This paper explores how to build multicore-scalable, replicated storage systems. We introduce a new guideline for their design, called the Zero-Coordination Principle. We use this principle to design a new multicore-scalable, in-memory, replicated, key-value store, called Meerkat.

Unlike existing systems, Meerkat eliminates all cross-core and cross-replica coordination, both of which pose a scalability bottleneck. Our experiments found that Meerkat is able to scale up to 80 hyper-threads and execute 8.3 million transactions per second. Meerkat represents an improvement of 12× *on state-of-the-art, fault-tolerant, in-memory, transactional storage systems built using leader-based replication and a shared transaction log.*

1 Introduction

Replicated, in-memory, transactional storage systems have emerged as an important piece of infrastructure for datacenter applications because they combine fault-tolerance and strong semantics with high throughput and low latency. The performance demands placed on these systems are extreme; e-commerce, social media, and cloud-scale storage workloads, among others, can require tens of millions of transactions per second [23, 43]. Meeting these demands requires a system structured to eliminate coordination bottlenecks both within each server and across the cluster.

For many years, distributed or replicated storage systems have had the luxury of ignoring the cost of *cross-core* coordination within a single node, as it has been masked by the far higher cost of *cross-replica* coordination. Even in a local-area network, the principal bottlenecks have come, by a substantial margin, from network round trips and the cost of processing packets [25, 51]. Indeed, the gap between communication and computation cost has been so large that some have proposed restricting execution to a single core to simplify protocol design [9, 41]. But the era of slow networks has come to an end. The past decade has brought into the mainstream both faster network fabrics and kernel-bypass network technologies that dramatically lower the cost of packet processing. The result is that it is possible to build a system that pushes the limits of cross-core coordination.

In fact, the effects are already visible on today's widely available hardware. Figure 1 compares the performance of a simple key-value storage system implemented using the traditional Linux UDP network stack on a 40 Gbit Ethernet network and with eRPC [15], a recent kernel-bypass network stack. In both cases, increasing the number of cores used allows the system to handle more PUT requests, but the eRPC implementation has 8× higher throughput than the UDP implementation. More importantly, when we introduce an artificial scalability bottleneck in the application – a simple atomic shared counter, incremented on every operation – the effect is quite different. The application bottleneck has no discernable effect (up to 20 cores) with the traditional UDP implementation, as it is masked by bottlenecks in the Linux network stack. With the optimized eRPC stack, however, application-level scalability bottlenecks have a major effect.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387529

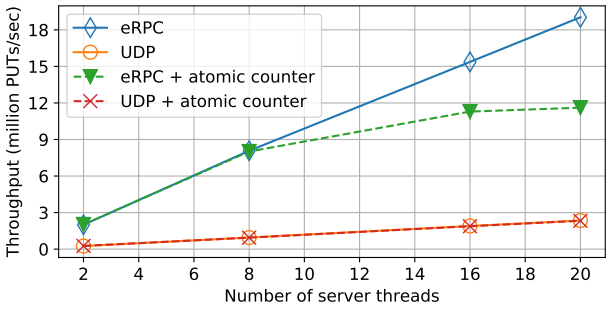


Figure 1: Peak throughput comparison of PUT operations for a simple key-value storage system implemented on both a traditional Linux UDP stack and on a recent kernel-bypass stack, eRPC. The 8× improvement in performance uncovers cross-core scalability bottlenecks in the application – a simple shared atomic counter incremented on every PUT operation bottlenecks the system at 11 million operations/second.

That is, for the first time, the distributed system causes the bottleneck at higher core counts.

Can replicated storage systems scale with increasing core counts? A fruitful line of recent work [17, 18, 26, 46, 48, 50] has made great strides in allowing *single-node* transaction execution to take full advantage of multicore processors. Extending this to the distributed environment, however, runs afoul of some fundamental challenges. Keeping replicated systems consistent requires ensuring that each replica reflects the same order of execution. Multicore execution, however, is inherently non-deterministic, a problem for the state machine approach [40]. Other replicated systems are commonly built with shared-memory data structures (e.g., logs) that are themselves challenging to scale.

This paper takes a principled approach to multicore design that systematically eliminates bottlenecks in distributed systems. We introduce the *Zero-Coordination Principle* (ZCP), which states that a distributed system with no coordination between replicas *or* cores will be multicore-scalable. ZCP extends disjoint access parallelism, a classic architecture property, to the distributed systems realm.

This paper demonstrates that this approach yields multicore-scalable distributed systems using a case study on in-memory replicated storage systems. We present *Meerkat*, the first distributed system in this category that is multicore scalable. Meerkat uses a mix of new and old design techniques to achieve a replicated transaction protocol with no cross-core or cross-replica coordination. Our Meerkat prototype must further ensure that its entire software stack is coordination-free: it uses eRPC for kernel-bypass, x86 atomic instructions, and other optimizations to minimize cross-replica and cross-core latency. Experiments with our prototype demonstrate that Meerkat scales up to 80 hyperthreads and executes 8.3 million YCSB-T [11] transactions/second.

The paper makes the following contributions:

- A new *Zero-Coordination Principle* (ZCP) that guides the design of multicore-scalable, replicated systems.
- The first multicore-scalable, replicated, in-memory, transactional storage system, Meerkat, designed using this principle.
- A performance comparison of Meerkat with existing systems to evaluate the impact of violating ZCP on multicore-scalability and performance.

2 The Zero Coordination Principle

Our goal is to build replicated systems that are both *multicore-scalable* and *replica-scalable*. More precisely, if the system is executing non-conflicting transactions – those whose read and write sets are disjoint – **the number of completed transactions per core should not decrease** even as the total number of cores in the system increases. It should be possible to scale up the system throughput by using nodes with more cores or increase the system fault tolerance by adding more replicas without a performance penalty. That is, the only barriers to scalability should be ones that are fundamental to the workload – those that arise from conflicts between transactions.

Most systems do not achieve both multicore scalability and replica scalability due to shared structures that require coordination for non-conflicting transactions and hence become points of contention. The *Zero-Coordination Principle* states that scalable systems can be built by following two rules:

Multicore scalability: DAP. The first half of ZCP is a familiar one from previous work on designing multicore-scalable applications, namely *disjoint access parallelism* (DAP) [14]. The DAP principle states that a system will be able to achieve ideal scalability on multicore processors if non-conflicting transactions – ones that access disjoint sets of memory locations – access disjoint memory regions. This means that the implementation requires no cross-core coordination. Later work extended the DAP principle to transactional memory [2, 36].

DAP means that there should be no centralized points of contention beyond the data items themselves, such as transaction ID management or database lock managers. Recent work has built single-node transaction processing systems that minimize or eliminate these central contention points [17, 18, 26, 46, 48, 50]. Replicated systems tend to involve further points of contention, such as shared logs or other state tables, that must be eliminated for a replicated system to support ZCP.

Replica scalability: coordination-free execution. Our second performance goal, that performance should not decrease as the number of replicas increases, is addressed by the second half of ZCP. Much as DAP required that non-conflicting transactions not access the same memory regions, ZCP imposes an additional requirement that non-conflicting transactions do not require *cross-replica* coordination, i.e., replicas do not need to send messages to each other in order to commit non-conflicting transactions.

For example, a traditional state machine replication protocol [20, 27, 34, 35] (or shared log), the basis for many transaction processing architectures [3, 5, 8, 39], does not meet this requirement. It requires establishing a total order of operations through a distributed protocol, e.g., by serializing them through a leader replica, a form of cross-replica coordination. This generally causes $O\left(\frac{1}{n}\right)$ throughput scaling in the number of replicas, though some replication and transaction processing protocols have been designed to avoid this [25, 47, 52].

Although sufficient to meet our second performance goal, this ZCP requirement is stricter than necessary and implies a design where the task of replicating transactions is offloaded to the clients. For example, one could potentially employ chain replication [47] to build an equally scalable system but at a higher latency cost (i.e., the message delay is proportional to the number of replicas). The stricter requirement leads to better designs in cases where it is important to reduce the number of message delays.

To summarize, ZCP requires that non-conflicting transactions (1) do not access overlapping memory regions on a single node, and (2) do not require cross-replica coordination. A system that satisfies both requirements will be both multicore-scalable and replica-scalable.

ZCP is inspired by rules about when multicore scalability is achievable on a single node. Beyond DAP itself, the Scalable Commutativity Rule [6] takes the DAP principle a step further by saying that an interface will have a multicore-scalable implementation if its operations commute; it then applies that rule to the design of operating system calls. ZCP extends this approach to the replicated systems context.

3 Meerkat Approach

Enforcing ZCP is a tall order for any distributed system. Meerkat uses the following design techniques to ensure zero coordination.

Replicate transactions, not arbitrary operations. State machine replication, used by popular replication protocols (e.g., Paxos [20], VR [27, 34] and Raft [35]), is a poor match for multicore scalability. Not only does it require determinism, which is impossible to enforce on multicore replicas without coordination, but it also requires a single ordering of operations, typically using a shared log.

Meerkat takes a different approach and directly replicates transactions. This design ensures that only conflicting transactions require coordination, maintaining ZCP. While other protocols use this approach to minimize coordination (e.g., Generalized Paxos [21], EPaxos [31]), they are more general in the way that they detect dependencies between operations and are not designed to minimize cross-core coordination.

Use timestamp-ordered optimistic concurrency control for parallel concurrency control checks. Without a single order of operations among replicas, Meerkat must use a different approach to detect conflicts in transaction ordering. Meerkat uses timestamp ordering and an optimistic concurrency control mechanism to allow conflict detection to happen in parallel. Timestamps directly order transactions, and replicas and cores can independently check for conflicts using only the timestamp without the need for coordination mechanisms. This approach has been used by multicore-scalable single-node transaction protocols [26, 46, 50]; Meerkat extends it to work with a decentralized replication protocol.

Use client-provided timestamps based on loosely synchronized clocks. An obvious challenge for timestamp-ordering is how to efficiently select a timestamp for each transaction. Timestamp selection can be both a multicore bottleneck (contention on the next-assigned timestamp) and a source of coordination between replicas. Meerkat leverages loosely synchronized clocks as a way to avoid this coordination: clients select and propose a timestamp for a transaction, and replicas determine whether they are able to execute the transaction *at that timestamp* without conflicts. Variants of this approach have been used in a variety of recent systems [1, 8, 52]. Importantly, Meerkat does not require clock synchronization for correctness (unlike Spanner [8], for example) but only for performance.

Versioned backing storage. Versioned storage further eliminates the need to coordinate updates to the same key. Timestamp-based concurrency control requires versioned storage, so Meerkat can execute transactions on different versions of the same key out of order: it can execute a read operation at an earlier timestamp without conflicting with a later write, or process certain writes under the Thomas write rule [44].

4 Meerkat Overview

In this section, we describe the design of our new fault-tolerant, multicore transactional storage system dubbed Meerkat. Meerkat provides serializable transactions, replicating operations across multiple commodity servers for fault tolerance. Meerkat follows the Zero-Coordination Principle in its design, combining both a parallelism-friendly storage implementation and a new transaction-oriented replication protocol. Like prior single-node systems, it follows DAP to

provide multicore scalability by avoiding cross-core coordination for non-conflicting transactions. Unlike these systems, it also avoids cross-replica coordination for these transactions, providing replica scalability and achieving improved performance and liveness compared to existing replicated storage systems.

Before we detail the Meerkat transaction processing protocol in the next section, we give an overview of the system model and key data structures.

4.1 System Model and Architecture

We assume a standard asynchronous model of a distributed system. Formally, the system consists of $n = 2f + 1$ multicore replica servers and an unspecified number of client machines, where f is the number of replica failures that the system can tolerate. Replicas and clients communicate through an asynchronous network that may arbitrarily delay, drop, or re-order messages. Replicas can fail only by crashing and can recover later.

Meerkat guarantees one-copy serializability for all transactions: from each client's perspective, the results are equivalent to a serial execution of the transactions on a single system. Like all replicated systems, Meerkat cannot ensure liveness during arbitrary faults; it makes progress as long as fewer than half of the replicas are crashed or recovering, and as long as messages that are repeatedly resent are received in some bounded (but unknown) time.

Each transaction is managed by a distinct *Meerkat transaction coordinator*, which typically runs on the client machines (usually application servers), and each replica runs a local instance of the *Meerkat multicore transactional database*. A Meerkat multicore transactional database instance is a three-layered system consisting of a versioned storage layer, a concurrency control layer, and a replication layer. Every instance can process transactions independently of other instances, perform recovery from various failure scenarios, and synchronize with the other instances to ensure consistency.

4.2 Meerkat Data Structures

	TID	ReadSet	WriteSet	Status	Timestamp
Core 1	455	<a,3>,<b,9>	a,b	COMMITTED	10
	630	<a,10>	a	VALIDATED-OK	11
Core 2	121	<a,3>,<c,5>	c	VALIDATED-ABORT	
	845	<a,10>	a	ABORTED	

Figure 2: An example trecord, partitioned on transaction id among cores. Every Meerkat multicore database manages its own trecord.

Each replica (i.e., Meerkat multicore transactional database instance) runs an algorithm built around two main data structures: the *vstore* and the *trecord*. These structures are organized to preserve ZCP: all state is either partitioned per-key (as in the *vstore*) or purely transaction-local and hence only accessed from a single core (the *trecord*). The result is that cross-core coordination is only necessary between transactions that access the same data.

vstore. The *vstore*, shared among all cores at the replica, implements the versioned storage layer. The vstore can be implemented either as a concurrent hash table (to support efficient gets and puts), or a concurrent tree (to support efficient indexing and range queries). Our implementation uses a hash table; other research has developed suitable multicore-friendly concurrent tree structures [28].

The data stored in the *vstore* is augmented with a per-key version number, or write timestamp, wts, which is the timestamp of the transaction that most recently wrote the value of key, and a read timestamp, rts, which is the timestamp of the transaction that most recently read the value of key. (Since Meerkat serializes transactions in timestamp order, as we show later, wts and rts essentially track the largest timestamps of transactions that wrote and, respectively, read the value of key.) Additionally, each key maintains two lists of all pending transactions that accessed it. $readers[key]$ stores the timestamps of all uncommitted transactions that read key and were successfully validated. Likewise, $writers[key]$ stores the timestamps of all pending transactions that wrote key. These are used in Meerkat's validation protocol.

trecord. As shown in Fig. 2, the *trecord* maintains a set of transaction records for recovery and synchronization. A transaction record contains the following fields: a unique transaction id (*tid*), the transaction's read and write sets constructed during the execution phase (*ReadSet* and *WriteSet*), a (proposed) commit timestamp (*Timestamp*) chosen after the transaction is validated, and the status of the transaction (*Status*). A transaction record also contains two additional fields, *View* and *AcceptView*, that are used to recover from a failure of the transaction's coordinator; these are not illustrated in Fig. 2. We discuss their use in Sec. 5.3.

To avoid unnecessary synchronization and preserve DAP, the trecord is horizontally partitioned among cores by transaction id. That is, each core operates on its own local trecord partition that contains a subset of the transactions. Because synchronous replication inherently requires a multi-round algorithm, a replica is required to process multiple messages for the same transaction, one in each round. We use a mechanism based on the NIC's forwarding capabilities to efficiently route transactions to the correct core. Naively routing transactions to cores can lead to spurious interrupts and unnecessary cross-core communication.

5 Meerkat Transaction Protocol

We begin with an overview of Meerkat's transaction execution, then follow with the protocol in the absence of failures. Finally, we describe failure handling and give a short proof.

5.1 Protocol Overview

Meerkat uses an optimistic protocol for executing and replicating transactions. Like traditional optimistic concurrency control protocols [19], Meerkat's transaction protocol follows a three-phase execute-validate-write protocol. Meerkat integrates its replica coordination protocol with the validation phase to ensure that transactions are executed consistently across replicas. Importantly, this is the only phase that requires coordination, and on its fast path (when there are no conflicts between transactions), it is able to execute without coordination between replicas.

Phase 1: Execute. During the execute phase, or read phase, clients read versioned data from any replica. They buffer any write operations locally; these are not installed until they are ready to commit and have been validated.

Phase 2: Validate. Once the client completes executing the transaction, the transaction enters the validation phase. This is a combined OCC-style validation, ensuring that the transaction commits only if no conflicting transactions have occurred, and a replica coordination protocol, ensuring that each replica has consistent state. In this phase:

1. A transaction coordinator (located either on a client or replica) selects a proposed timestamp for the transaction and sends a VALIDATE request to all replicas.
2. Each replica independently performs OCC validation, checking whether they have committed any other transactions that would conflict with the specified timestamp, and returns either OK or ABORT.
   a. If a supermajority ($> \frac{3}{4}$) of the replicas send matching responses, the transaction commits or aborts under the fast path; the coordinator sends a COMMIT or ABORT message.
   b. Otherwise, an additional round of coordination (the slow path) is needed to ensure consistency.

Phase 3: Write. Once a transaction has been committed, replicas update their versioned storage with the transaction's writes.

5.2 Meerkat Transaction Processing

Meerkat's transaction protocol includes subprotocols for each of its three stages of execution. We first describe these three subprotocols assuming the data is not partitioned across multiple servers. We then discuss Meerkat support for distributed transactions.

5.2.1 Execution Phase

The execution phase is orchestrated by a *Meerkat transaction coordinator*. During the execution phase, the transaction coordinator sends every read to an arbitrary replica. The replica performs the read and responds with the read value and version. These versioned values are buffered by the transaction coordinator into a *ReadSet*. Similarly, the transaction coordinator buffers every write into a *WriteSet*. Writes are not sent to any replica.

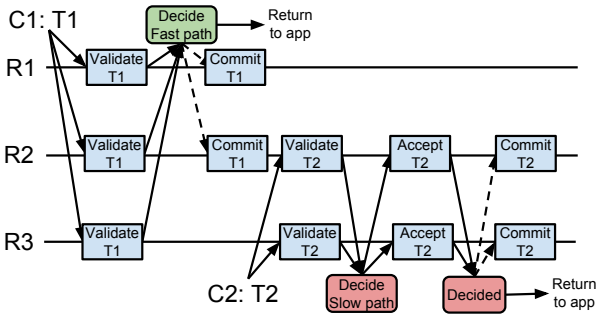


Figure 3: The commit protocol in normal operation mode.

5.2.2 Validation Phase

The execution phase is followed by a commit protocol that performs the validation phase, and, at the same time, replicates the outcome of a transaction. The protocol combines aspects of atomic commitment and consensus protocols. Like atomic commitment protocols, such as two-phase commit, the protocol performs decentralized transaction validation (i.e., each replica is seen as a distinct participant able to validate transactions independently). It uses consensus to allow backup coordinators (in case of coordinator failures) to eventually reach a unique decision to either commit or abort every validated transaction.

A high-level view of the commit protocol and its communication pattern in the normal operation mode is illustrated in Fig. 3. The two illustrated coordinators, *C1* and *C2*, try to commit transactions *T1* and *T2*, respectively, at roughly the same time. After the coordinators receive validation replies from the replicas, they can either decide and commit the transaction on the fast path, if enough replicas already agree on the outcome of the transaction, or must take the slow path (e.g., coordinator *C2* did not receive enough validation replies), on which they must get the replicas to agree on the decided outcome.

Throughout this protocol, we assume that all messages concerning a particular transaction are always processed by the same core on any replica. This core affinity allows the transaction state to be partitioned across different cores in the *trecord*, preventing spurious synchronization across different cores. In practice, the way we achieve this is to have the coordinator select a *coreid* for the replicas to use, and select a UDP port for communication based on the *coreid* to ensure that Receive-Side Scaling NICs deliver messages to the same core.

The commit protocol proceeds as follows:

1. The coordinator first selects a core, *coreid*, to process the transaction, a proposed timestamp *ts* for the transaction, and a unique transaction id *tid*. The proposed timestamp *ts* indicates the proposed serialization point

Algorithm 1 Meerkat validation checks

```
1:  procedure VALIDATE(txn, ts)
2:    ▷ Validate the read set
3:    for r ∈ txn.readSet do
4:      lock(r.key)
5:      e ← vstore[r.key]
6:      if e.wts > r.wts or ts > MIN(e.writers) then
7:        unlock(r.key)
8:        go to abort
9:      end if
10:     e.readers.add(ts)
11:     unlock(r.key)
12:   end for

13:   ▷ Validate the write set
14:   for w ∈ writeSet do
15:     lock(w.key)
16:     e ← vstore[w.key]
17:     if ts < e.rts or ts < MAX(e.readers) then
18:       unlock(r.key)
19:       go to abort
20:     end if
21:     e.writers.add(ts)
22:     unlock(w.key)
23:   end for

24:   return VALIDATED-OK

25: abort:
26:   cleanup_readers_writers(txn)
27:   return VALIDATED-ABORT
28: end procedure
```

of the transaction. To avoid the need for coordination to select the next available timestamp, the coordinator instead proposes a timestamp using its local clock: ts is a tuple of the client's local time and the client's unique id. tid is a tuple of a monotonically increasing sequence number local to the client and the client's unique id. By including the client's unique id, we ensure that both ts and tid are globally unique.

The transaction coordinator then sends $\langle$VALIDATE, $txn, ts\rangle$ to all the replicas,[1] where txn contains the tid as well as the read and write sets of the transaction.

2. Each replica creates a new entry in its core-local *trecord* partition and validates the transaction using the OCC checks illustrated in Alg. 1.

   The OCC checks begin by validating the transaction's reads. The condition $e.wts > r.wts$ checks that the read has read the latest committed version. The condition $ts > MIN(e.writers)$ checks that even if all pending transactions were to commit, the read would still have read the latest committed version as of ts. If either of these conditions does not hold, the transaction is aborted. Otherwise, the transaction's timestamp is added to *readers*.

   The replica core then validates the transaction's writes. The conditions $ts < e.rts$ and $ts < MAX(e.readers)$ check that a write will not interpose itself between a pending read or committed read and the version read by that read. Again, if either condition does not hold, the transaction is aborted. Otherwise, the transaction's timestamp is added to *writers*.

   We designed the parallel OCC checks to have small atomic regions at the cost of precision (i.e., certain valid serializable histories may be rejected). Further optimizations may be possible. For example, in certain abort cases it may be possible to assign a different commit timestamp and still commit the transaction, as in TAPIR [52]. We have not yet explored these.

   Ultimately, the replica core replies to the coordinator with $\langle$VALIDATE-REPLY, $tid, status\rangle$ where *status* is either VALIDATED-OK or VALIDATED-ABORT. If aborted, any changes to the *readers* and *writers* set are backed out.
3. If the coordinator receives a supermajority consisting of $f + \lceil \frac{f}{2} \rceil + 1$ VALIDATE-REPLYS with matching *status* (this is the **fast path** condition), it notifies the client that the transaction is complete. If *status* is VALIDATED-OK, then the transaction is committed, and if *status* is VALIDATED-ABORT, then the transaction is aborted. The coordinator then asynchronously sends $\langle$COMMIT, $tid, status\rangle$ to all replicas. Note that this COMMIT message can be piggy-backed on the client's next message.
4. If the fast path condition is not met, the coordinator takes the **slow path**. It must receive VALIDATE-REPLYS from a majority ($f + 1$) replicas, resending the VALIDATE message if necessary. The coordinator then sends a $\langle$ACCEPT, $tid, status\rangle$ request to replicas, where, if $f+1$ or more VALIDATE-REPLYS have $status = \texttt{VALIDATED} - \texttt{OK}$, the *status* argument is $\texttt{ACCEPT} - \texttt{COMMIT}$ (i.e., the coordinator proposes to commit the transaction), else the *status* argument is $\texttt{ACCEPT} - \texttt{ABORT}$.

   The ACCEPT request has the same role as the phase 2a message in Paxos – to ensure that a single decision is chosen, even when there are multiple proposers. As described so far, there is only one proposer, namely the transaction coordinator; however, the coordinator recovery protocol (Section 5.3.2) introduces *backup coordinators* and the replica recovery protocol (Section 5.3.1) introduces *recovery coordinators* that also act as proposers.
5. On receiving ACCEPT, the replica updates the status of the transaction to *status* and replies to the coordinator with $\langle$ACCEPT-REPLY, $tid\rangle$.

[1] Throughout the protocol, any messages that receive no reply are resent after a timeout. For simplicity, we do not describe this process for each message.

6. Once the coordinator receives a majority (f + 1) of ACCEPT-REPLYS, the transaction is completed. It notifies the client, and asynchronously sends ⟨COMMIT, *tid*, *status*⟩ to all replicas. As before, this COMMIT message can also be piggy-backed on the client's next message.

To keep the description of the commit protocol simple, we defer some details, mostly involving concurrent proposals, which do not occur in the failure-free case, to the failure handling section (Section 5.3).

5.2.3 Write Phase

On receiving a ⟨COMMIT, *tid*, *status*⟩ message, the replica marks the transaction as committed by setting its status in the *trecord* entry to `COMMITTED` if *status* is `VALIDATED-OK`, or `ABORTED` if *status* is `VALIDATED-ABORT`. It then performs OCC's write phase: if *status* is `VALIDATED-OK`, the replica updates the data items to their new values, and sets the version of each item to the commit timestamp. Regardless of whether the transaction commits or aborts, the replica simply cleans up *readers* and *writers* for *tid*.

5.2.4 Distributed Transactions

The protocol we described so far can easily be extended to support distributed transactions (when data is partitioned across servers) since it already includes aspects of atomic commitment protocols (i.e., decentralized validation of transactions). The transaction coordinator would just need to perform, in parallel, the validation phase in all partitions involved in the execution of the transaction.

5.3 Meerkat Failure Handling

Thus far, we have not considered replica or network failures, and have assumed that transaction coordinators do not fail. Of course, Meerkat must remain resilient to such failures.

Meerkat handles replica failure transparently, but for a recovering replica to rejoin the system, Meerkat must run an *epoch change* protocol (Section 5.3.1), where the replicas pause execution of new transactions to agree on a consistent state of pending operations. The epoch change protocol also helps with checkpointing, allowing the replicas to bring themselves up-to-date and safely trim the trecord. Failure of a transaction coordinator is handled by a *coordinator recovery* protocol (Section 5.3.2) that allows a backup coordinator to step in, without affecting any other transaction. The complete specification of these protocols is given in the extended version of the paper [42]; due to space constraints, we provide a brief overview here.

5.3.1 Replica Failure and Recovery

Meerkat uses a leaderless quorum protocol, so it is inherently resilient to failures of a minority of replicas. Provided that there are at least $f + 1$ replicas still available to respond to requests, the system continues to make progress. Depending on the number of replicas in the system, failures may cause the number of available replicas to drop below the $f + \lceil \frac{f}{2} \rceil + 1$ needed for a supermajority, forcing the slow path on every transaction. Note, however, that this still compares favorably to many commonly-used primary-backup protocols, which must stop processing transactions entirely after replica failures until the system is reconfigured.

Replica recovery. We assume that a failed replica rejoins the system without its previous state. To recover correctly, we must ensure not only that the recovering replica learns the outcome of all committed transactions, but that the system state remains consistent for any partially-completed transactions the replica might have participated in a quorum in prior to crashing [29].

Meerkat assures this using a *epoch change* protocol that brings the system to a consistent state. The epoch change protocol is inspired by Viewstamped Replication. It begins by designating a *recovery coordinator*, which serves to poll the replicas to determine the state of ongoing transactions. Each replica maintains an epoch number; the epoch number identifies the epoch's single recovery coordinator (as the (*epoch* mod *n*)th replica). The recovery coordinator takes the role of a *leader* that is in charge of deciding the outcome of all ongoing transactions. After recovery, all replicas will have a consistent *trecord*, which will be used by the restarted replica to recover.

The recovery coordinator first sends a ⟨EPOCH-CHANGE, *e*⟩ request to all replicas. Every replica increments its epoch number to *e*, and does not validate any new transactions until the epoch change completes. It responds to the coordinator with its current *trecord*, aggregated across all cores. Upon receiving at least a majority (f + 1) of replies, the recovery coordinator creates a new trecord (preserving the per-core partitioning), in which it adds transactions using the follow rules:

- It first adds all committed or aborted transactions (i.e., those for which at least one reply returns a *status* of `COMMITTED` or `ABORTED`).
- If any replica has accepted a decision for a transaction from the coordinator (or a backup coordinator, described in Section 5.3.2), it adopts the decision with the latest view number for that transaction, and adds it to the *trecord* with the corresponding status.
- All remaining transactions for which at least a majority (f + 1) of replies reported the same *status* in their respective trecords (either `VALIDATED-OK` or `VALIDATED-ABORT`) are added to the *trecord* (with *status* as `COMMITTED` or `ABORTED`).
- All remaining transactions that might have committed on the fast path, i.e., those where there are at least $\lceil \frac{f}{2} \rceil$ + 1 `VALIDATE-OK` replies, are re-validated using

OCC checks similar to those in Alg. 1 and added with the corresponding status.
- Any other transactions are added with *status* set to ABORTED.

The leader then sends ⟨EPOCH-CHANGE-COMPLETE, e, $trecord$⟩ to all replicas, including the recovering replica. They synchronize their *trecords* with one in the message, and then resume normal operation.

5.3.2 Coordinator Failure and Recovery

The failure of a Meerkat transaction coordinator presents a more subtle problem. In addition to preventing a client from learning the outcome of a transaction, a coordinator failure can leave unfinished transactions on the replicas. These unfinished transactions may degrade performance by preventing other transactions from being successfully validated.

Meerkat addresses this by using a consensus-based coordinator recovery protocol. Meerkat's coordinator strategy follows the approach used in TAPIR [52, 53] and is an instance of Bernstein's cooperative termination protocol [4]. In this protocol, in addition to its normal coordinator, each transaction also has a set of $2f + 1$ *backup coordinators*. The backup coordinators, which are only invoked on coordinator failure, can be shared among all transactions; each replica can run a backup coordinator process, or they can be deployed on a separate cluster. In the event of a coordinator failure, a replica can initiate a coordinator change to activate a backup coordinator and complete (either commit or abort) the transaction.

To ensure that transaction outcomes are consistent even in the presence of backup coordinators, Meerkat uses a consensus protocol. The consensus algorithm guarantees that all (backup) coordinators eventually reach the same decision – to either commit or abort the transaction – even if they concurrently propose different transaction outcomes.

Like most consensus algorithms, the coordinator recovery protocol uses *views* to uniquely identify proposals. Unusually, each view is specific *to a given transaction* (i.e., tid). For each view, one coordinator acts as the proposer. Each transaction starts with $view = 0$, and in this view, the original transaction coordinator is the unique proposer. The unique proposer in $view > 0$ is the ($view$ mod n)$^{\text{th}}$ replica.

To support this protocol, for each transaction, we include two additional fields in the transaction's trecord entry: (1) the current view number, $view$, initially 0; and (2) if this is the case, the view number in which a proposal was last accepted, *acceptView*. Note that this is the same information maintained by Paxos to solve one instance of consensus.

When a replica notices the potential failure of the coordinator of the current view, is starts a view change procedure, similar to Paxos' prepare phase, where a new backup coordinator is established (a majority of replicas agree to ignore proposals from previous coordinators, i.e., containing lower view numbers). After receiving a quorum of replies for the coordinator change request, the new coordinator analyzes the replies to determine a safe outcome for the transaction. This means that it selects any outcome that, in order of priority, has (1) been completed (COMMITTED or ABORTED) at any replica, (2) been proposed by a prior coordinator and accepted by at least one replica, or (3) been VALIDATED-OK or VALIDATED-ABORT by a majority of replicas. It then attempts to complete the transaction with that outcome on the slow path, using a procedure similar to Paxos' accept phase.

5.4 Correctness

Meerkat guarantees serializability of transactions under all circumstances. We give a brief correctness proof sketch here.

Correctness during normal operation. Consider first the non-failure case. Although Meerkat does not guarantee that every replica executes each transaction – no quorum-based protocol can do so – nor even that *any* replica executes every transaction, its correctness stems from the fact that OCC checks can be performed pairwise [52]. Moreover, every successfully committed transaction must have been VALIDATED-OK on a majority ($f + 1$) of replicas. For purposes of contradiction, suppose that there are two conflicting transactions that have both successfully committed. Quorum intersection means that there exists at least one replica that must have VALIDATED-OK both of the two transactions. However, because the two transactions conflict, whichever one arrived later at the replica must have returned VALIDATED-ABORT instead. Thus, no such pair of transactions can exist.

Correctness during replica failure. As noted above, replica failure by itself requires no special handling; it is replica *recovery* that poses challenges. The correctness property for the epoch change protocol is twofold: (1) any client-visible results from previous epochs, either commits or aborts, are reflected as COMMITTED or ABORTED outcomes in the *trecord* adopted by any replica at the start of the new epoch, (2) no further transactions commit in the old epoch. Property (2) is readily satisfied, as the new epoch only begins once $f + 1$ replicas have acknowledged an EPOCH-CHANGE message, and they subsequently process no new transactions in the old epoch; thus, no further transactions can achieve a quorum.

With respect to property (1), consider first a transaction that commits or aborts on the slow path. In this case, $f + 1$ replicas must have processed the ACCEPT message from the coordinator. At least one of these replicas will participate in the epoch change, and so the procedure in Section 5.3.1 will add it to the *trecord* in the appropriate state. Now consider a transaction that committed on the fast path. At least $f + \lceil \frac{f}{2} \rceil + 1$ replica must have returned VALIDATED-OK in the validate phase, and at least $\lceil \frac{f}{2} \rceil + 1$ must have participated in

the epoch change. This transaction too will be added to the *trecord*, per the algorithm, *unless* a conflicting transaction has already been committed; however, since the original transaction previously committed successfully, no such transaction can exist.

6 Evaluation

Our evaluation demonstrates the impact of ZCP on multicore scalability. We break down the cost of cross-core coordination and cross-replica coordination on two workloads with both long and short transactions. We also measure the trade-off between multicore scalability and performance under high contention. Our experiments show the following:

1. Cross-core coordination has a significant impact on multicore scalability regardless of transaction length. Eliminating cross-core coordination improves throughput by 5–7× for up to 80 server threads.
2. Cross-replica coordination depends on transaction length. Eliminating cross-replica coordination can provide a performance improvement ranging from 3% to almost 2× for a three-replica system.
3. ZCP comes at a trade-off in supporting extremely high contention workloads. Using 64 server threads, Meerkat provides better performance on low-to-moderately skewed workloads (up to Zipf coefficients past 0.8), but performance suffers on very highly skewed workloads.

6.1 Prototype Implementation

In addition to our Meerkat prototype, we implemented three other systems to evaluate the relative impact of the two ZCP constraints. First, we implemented a classic, log-based, primary-backup replicated system that requires both cross-core and cross-replica coordination: the primary decides transaction ordering using a shared atomic counter and places each committed transaction into a shared log for replication. Replicas also share the log but read transactions and apply updates in parallel (concurrent log replay). Since the transactions are already ordered, there is no need for determinism at the replicas; however, log replay still requires cross-core coordination for access to the shared log. This mechanism is similar to many primary-backup multi-core databases, like KuaFu [13]. Unlike KuaFu, this variant does not need more synchronization, like barriers, to deal with read inconsistencies (i.e., reads from backups that might be in an inconsistent state after applying updates out of order) since it relies on OCC validation checks at the primary for correctness. Therefore, we refer to this system as KuaFu++.

Next, we implement a leader-less replicated system, designed to emulate TAPIR [52]. The replicas do not coordinate, but each replica uses a shared, cross-core transaction record. Based on the TAPIR protocol, clients issue transaction timestamps, so replicas can perform concurrency control checks and apply transaction updates in parallel.

	Cross-Core Coordination	Cross-Replica Coordination
KuaFu++	Yes	Yes
TAPIR	Yes	No
Meerkat-PB	No	Yes
Meerkat	No	No

Table 1: An overview of our evaluation prototypes. To measure the impact of ZCP, we implemented systems with differing cross-core and cross-replica coordination.

Finally, we implement a primary-backup variant of Meerkat that satisfies DAP, which we label Meerkat-PB. It uses the same data structures and concurrency control mechanism as Meerkat; however, only the primary executes concurrency control checks, i.e., all clients submit their transactions with timestamps to the primary, and the primary decides which conflicting transactions will commit. Since transactions are timestamp-ordered and conflict-free, replicas can commit transactions in any order. To eliminate shared data structures, each backup core is matched to a primary core and processes only its transactions. Meerkat-PB lets us measure the impact of cross-replica coordination without cross-core coordination.

We believe that these four systems are representative for the class of distributed storage systems we target – fast systems that assume low to medium contention. Other approaches are possible, such as deterministic [45] or speculative [16] approaches. However, both of these types of approaches, unlike ours, require that transactions' read and write sets be known a priori. In general, they are most effective for high contention rates, rather than our target.

All of our prototype systems have three layers: (1) a transport layer for message delivery, (2) a storage layer for storing the data and scheduling the transactions, and (3) a replication layer for consistently replicating the data. All systems share the transport layer – ensuring that all systems have access to the same high-speed network library and avoiding differences due to different approaches to serializing and deserializing wire formats. Meerkat and Meerkat-PB also share the storage layer. All replication layers use the same unordered *record* abstraction. Meerkat and Meerkat-PB use one record per core, while KuaFu++ and TAPIR share a single record per replica. To synchronize accesses to the shared record, these solutions use simple mutexes from the C++ standard library.

The shared transport layer is built on eRPC [15], a fast, reliable RPC library that bypasses the kernel. All storage

layers provide the same semi-structured data model and use hash tables to store the key-value pairs. They also all implement interactive transactions, as opposed to stored procedures. Each key-value entry has its own fine-grained lock, and reader and writer lists for concurrency control checks. We use the `unordered_map` container for various data structures and `pthread` read-write locks to protect them for concurrent access.

Transaction Type	# gets	# puts	workload %
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post Tweet	3	5	30%
Load Timeline	rand(1,10)	0	50%

Table 2: Description of the Retwis workload.

6.2 Experimental Setup

We use three replica servers in our experiments. Each server has two 20-core Intel Xeon Gold 6138 processors with 2 hyperthreads per core, supporting up to 80 server threads. In all experiments, we pin each server thread to a distinct logical CPU (i.e., hyperthread). Each core has private L1 and L2 caches (of 64 KB and 1024 KB, respectively) and shares the L3 cache (of 28 MB) with the other cores on its processor. The total DRAM size of 96 GB is evenly split between the two NUMA nodes. To eliminate uneven overheads (e.g., L3 contention, database loaded in one NUMA node), each experimental result was generated by an even number of threads, half of which were pinned on distinct cores of one NUMA node and half of which were pinned on distinct cores of the other NUMA node.

Each server machine has a Mellanox ConnectX-5 NIC, connected using a 40 Gbps link to an Arista 7050QX-32S switch. All machines run Ubuntu 18.04 with Linux kernel version 4.15. To mitigate the effects of frequency scaling, we set the scaling governors (power schemes for CPU) to *performance*, which maintains the CPUs running at 2 GHz, for all 80 logical CPUs (hyperthreads), and we always start all 80 server threads to keep all the cores at full utilization (each server thread polls in a loop on a NIC queue) to prevent the Intel Turbo Boost from unevenly boosting up the frequencies of the cores.

We use the flow steering mechanism of the Mellanox NICs to preserve DAP in the networking stack. Each server thread uses its own send and receive queue to which clients can steer packets based on a port number – the NIC will place packets in the corresponding queue managed by a single core, thus avoiding unnecessary cross-core synchronization.

We run closed loop clients on ten 12-core machines, each a single NUMA node. Each client machine uses a Mellanox ConnectX-4 NIC, connected using a 40 Gbps link to the same switch as the server machines. The client clocks are synchronized with the Precision Time Protocol (PTP). When running the experiments, we used a 5-minute warm-up period to warm-up the caches and the CPUs. Each data point is the average of the results of 3 identical runs.

We use two benchmarks: (1) YCSB-T [11], a transactional version of Yahoo's popular YCSB [7] key-value benchmark and (2) Retwis [52], illustrated in Table 2, a benchmark designed to generate a Twitter-like transactional workload. We use keys and values of size 64 bytes. To illustrate the impact of contention on each system, we perform experiments using a varying Zipf distribution ranging from 0 (uniform, low contention) to 0.6 (medium contention) to >0.9 (highly contended).

We pre-allocate memory for metadata, such as the transaction records and locks, to avoid memory allocation overhead during the measurements. Before each run, we load the entire database into memory, with 1 million data items per core. By increasing the number of keys, we keep the contention level constant as we scale to increasing numbers of cores.

All systems use OCC-based concurrency control, which allows any replica to serve GETs – the OCC concurrency control checks will later establish if they can be ordered at a serializable timestamp. We thus uniformly load balance the clients across replicas and their cores for both GET and COMMIT requests.

Our experiments are focused on measuring throughput – more precisely, goodput, i.e., the number of transactions that successfully commit per second. It should be noted, however, that Meerkat does not sacrifice latency to achieve scalability. Indeed, it achieves low latency because the validation checks are cheap, with small atomic regions – comparable with the ones used by other systems – and the protocol saves one round trip compared to most state-of-the-art systems.

6.3 Impact of ZCP on YCSB-T Benchmark

Using the YCSB-T benchmark, we measure the impact of cross-core and cross-replica coordination using our four prototype systems. We use the transactional variant of the YCSB workload F, where transactions consist of a single read-modify-write operation. These transactions are relatively short with an even mix of read and write operations.

Figure 4 shows the performance of each system. The KuaFu++ system has both cross-core and cross-replica coordination; as a result, it bottlenecks at 6 cores with 600,000 transactions per second. Eliminating cross-replica coordination only improves performance slightly: our TAPIR system scales to 8 cores and 800,000 transactions per second. However, cross-core contention prevents it from scaling further, because it still uses a shared record between server threads. This highlights an important point: despite the significant research that has gone into eliminating cross-replica coordination, cross-core coordination may pose a more significant

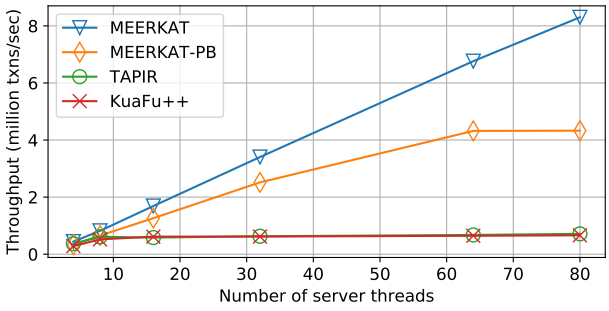


Figure 4: Peak throughputs comparison between Meerkat and the three other systems for uniform key access distribution for YCSB-T transactions containing one read-modify-write operation. While Meerkat is able to scale to 80 cores, the other systems all bottleneck at fewer cores due to violating ZCP.

bottleneck for many deployments, particularly using modern networks.

In contrast, eliminating cross-core coordination in our Meerkat-PB system increases throughput by 7x compared to KuaFu++. Despite the need for cross-replica coordination, the fast network communication between replicas using our eRPC transport lets Meerkat-PB scale to 64 server threads.

Finally, eliminating cross-replica and cross-core coordination in Meerkat improves throughput by 12x compared to our baseline KuaFu++ system. Meerkat is able to continue scaling to 80 threads and 8.3 million transactions per second, with the elimination of cross-core and cross-replica coordination contribution in roughly equal parts to its performance improvements.

6.4 Impact of ZCP on Retwis Benchmark

We perform the same experiment with the Retwis [22, 52] benchmark. Compared to YCSB-T, Retwis offers longer, more complex transactions, a more read-heavy workload and a wider range of transaction types. Figure 5 shows the results for all four systems.

Due to longer transactions, the total transaction throughput is lower for all systems. As a result, violating ZCP has less impact on the systems with more coordination, and even the non-ZCP systems are able to scale to more cores. TAPIR and KuaFu++ are both able to scale to 32 cores because there is more work for the cores to do during the execution period where there is less coordination. However, they still are not able to process more than 600,000-700,000 transactions per second.

Since cross-replica coordination only happens during transaction commit, its impact on performance is reduced with longer transactions. In particular, the majority of the execution phase for this benchmark consists of read operations, which can be executed on any replica. TAPIR does not improve much on KuaFu++ by eliminating cross-replica coordination and Meerkat-PB scales almost as well as Meerkat. As a result, with longer transactions, the effects of cross-core coordination on multicore scalability increase while the effects of cross-replica coordination decrease. Meerkat still scales the best, achieving 2.7 million transactions per second on 80 server threads.

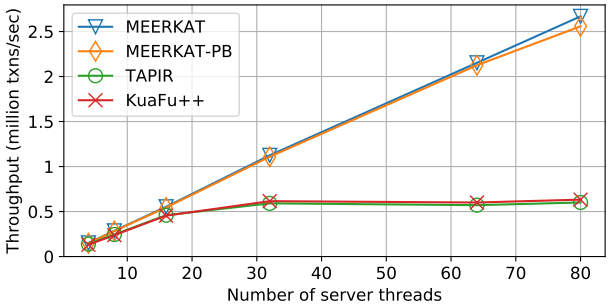


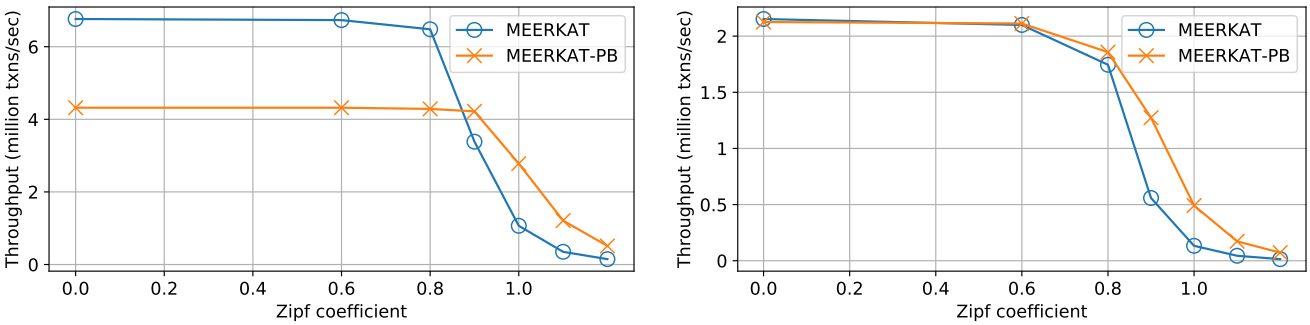
Figure 5: Peak throughputs comparison between Meerkat and the three other systems for uniform key access distribution for Retwis transactions. The comparison systems are able to scale better with coordination due to Retwis's longer transactions and read-heavy workload. However, none of the systems quite scale linearly to 80 cores except Meerkat; the non-ZCP systems continue to be limited by their additional coordination.

6.5 ZCP and Contention

In this section, we evaluate the trade-off between ZCP and performance under high contention workloads. Meerkat uses a highly optimistic concurrency control mechanism to avoid cross-core and cross-replica coordination. Meerkat decentralizes OCC checks at all replicas, while Meerkat-PB centralizes them at the primary. As a result, Meerkat is more likely to abort transactions at higher contention rates because replicas cannot agree. Thus, Meerkat trades-off performance under high contention for better multicore scalability.

To show this trade-off, we vary the Zipf coefficient of both the YCSB-T and Retwis benchmarks. We fix the number of server threads at 64, where Meerkat-PB still scales on both workloads and compare their performance across the range of Zipf coefficients.

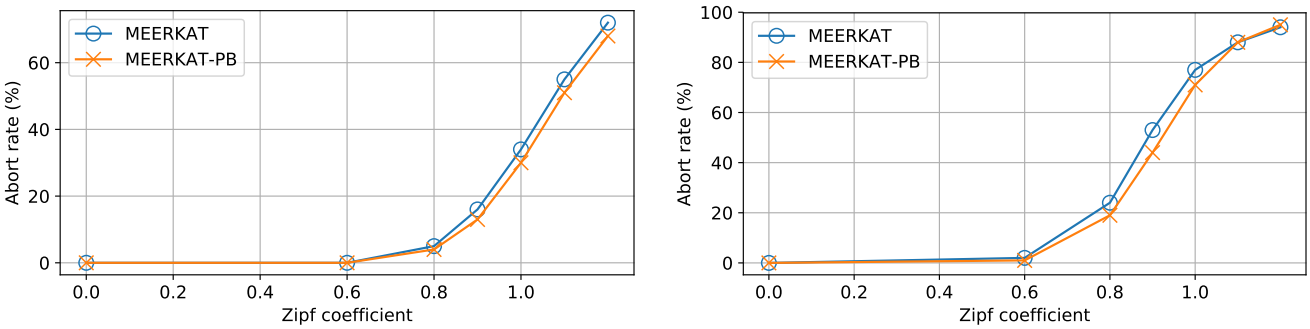
Figure 6 clearly shows the trade-off between the two systems. For YCSB-T, Meerkat provides 50% higher throughput until the zipf coefficient reaches 0.87. After that, Meerkat's OCC mechanism causes the throughput to drop more sharply than Meerkat-PB, making Meerkat's performance worse at higher contention rates. For Retwis's longer transactions, Meerkat-PB is able to match the performance of Meerkat; however, at higher Zipf coefficients, Meerkat-PB outperforms Meerkat.



(a) YCSB-T transactions

(b) RETWIS transactions

Figure 6: Peak throughputs comparison between Meerkat and Meerkat-PB for various zipf coefficients, 64 server threads for (a) YCSB-T transactions containing one read-modify-write operation, and (b) long, read-heavy Retwis transactions. Meerkat outperforms its primary-backup variant for low to medium contended workloads.



(a) YCSB-T transactions

(b) Retwis transactions

Figure 7: Abort rates at peak throughputs comparison between Meerkat and its primary-backup variant for various zipf coefficients, 64 server threads, and 3 replicas for (a) short YCSB transactions containing 1 read-modify-write operation, and (b) the complex, read-heavy Retwis workload. Meerkat has slightly higher abort rate as it needs to collect multiple favorable votes to commit transactions.

Figure 7 shows the reason for the drop in performance. At lower contention rates and with shorter (YCSB-T) transactions, the lack of coordination gives Meerkat higher performance while both systems have low abort rates. However, as the abort rate climbs with more contention and longer transactions (i.e., note that the abort rate climbs faster for Retwis), the lack of coordination hurts Meerkat and causes its throughput to drop faster. High contention workloads require more coordination to support efficiently, which makes them fundamentally at odds with multicore scalability.

7 Related Work

There is significant previous work in transactional storage systems, replication, and multicore scalability. While past research has focused on one or two of these topics, there exists little work on the combination of all three. As we showed in Figure 1, new advances in datacenter networks will make multicore scalability an increasing concern for replicated, transactional storage, forcing researchers to delve into all three topics.

Replicated, transactional storage. Fault-tolerance is crucial for modern transactional storage systems, thus plenty of research has been done to understand how various replication techniques work with various transactional storage systems, but less attention has been paid to the multicore aspect.

Due to its simplicity, the most popular replication technique for transactional storage systems is primary-backup, where only the primary executes and validates transactions and ships updates using a log. Many commercial databases – MySQL [33], PostgreSQL [37], Microsoft SQL Server [30], among others – also support parallel execution at the primary, but the backups consume the log serially, thus becoming a bottleneck, as observed in KuaFu [13]. These primary-backup solutions trivially violate ZCP's cross-replica coordination rule and do not achieve Meerkat's performance/scalability goal. Both KuaFu and Scalable Replay [38] enable

more concurrency at replicas, but both solutions also violate ZCP's first rule – KuaFu still requires log synchronization and Scalable Replay requires a global atomic counter.

State-machine replication, where transactions are first ordered then executed in the same order on all replicas, is another popular option, adopted by many previous systems [24, 45]. In general, these solutions violate both tenets of ZCP, requiring both cross-replica and cross-core on the shared operation log. Calvin [45] improves on state of the art by using deterministic locking on the replicas for better performance but still requires cross-replica coordination.

Chain replication [47] offers an alternative; while it still requires cross-replica coordination, communications between replicas does not increase with increasing numbers of replicas. There has not been a multicore-scalable transactional store deployed on chain replication, but we speculate that it would provide better performance at higher contention than Meerkat at the cost of higher overall latency per transaction.

Several transaction protocols are able to eliminate cross-replica coordination, including the classic Thomas algorithm [44] and recent systems like Janus [32]. However, these systems still use a shared log, as they were generally not designed to be multicore scalable, and so cannot avoid cross-core coordination. It may be possible to redesign these systems to meet ZCP, which represents an interesting future direction.

Multicore Scalability for Unreplicated, In-memory Storage. Meerkat follows a long line of research on unreplicated, multicore, in-memory storage systems. Silo [46], TicToc [50], Cicada [26], ERMIA [17], MOCC [48] all strive to achieve better multicore scalability using a variety of designs. These include both single-version optimistic concurrency control [46, 50] and multi-version concurrency control designs [17, 26]. Some use more sophisticated concurrency control mechanisms to ensure serializability; for example, ERMIA uses a Serial Safety Net to reuse a snapshot isolation mechanism while guaranteeing serializability [17]. Others use careful design to provide scalability properties beyond DAP; for example, Silo not only avoids cross-core coordination for committing disjoint read-write transactions, but it also eliminates cross-core synchronization for all read-only transactions. Wu et al [49] provide a good overview of the design choices involved in these in-memory concurrency control protocols and their relative benefits.

While these systems perform extremely well, they cannot be easily extended to a replicated environment. In particular, the concurrency control protocol in each system cannot be trivially extended to support coordination between replicas to ensure consistency. Replication further requires data structures that may not be multicore scalable and are not easily made to scale.

General Multicore Replication. Replication for general multicore applications remains a difficult problem. Without insight into application operations, replicas must use some form of determinism to ensure consistency. For example, Paxos Made Transparent [10] captures all system calls and uses deterministic multi-threading. Eve [16] takes a more speculative approach, letting replicas execute in parallel and rolling back operations in case of inconsistencies. Rex [12] executes a transaction first at a primary, and logs all non-deterministic decisions the primary made during the execution and replays them at the replicas. These heavyweight mechanisms require significant cross-core coordination, making it impossible to scale these systems as the number of cores grows.

8 Conclusion

With the proliferation of kernel-bypass technologies and faster datacenter networks, existing distributed system designs will be able to scale to the large number of cores being deployed in the datacenter. This paper presented a new guideline for the design of multicore-scalable distributed systems – ZCP – and a new multicore-scalable, replicated, in-memory, transactional storage system.

As we showed in our experiments, cross-core and cross-replica coordination will increasingly dominate and limit the performance of existing systems. A state-of-the-art replicated, in-memory storage system can only scale to 6-8 server threads, while Meerkat is able to scale to 10× the number of cores with a corresponding increase in throughput. We hope that these results encourage researchers to focus more on the design of multicore-scalable distributed systems in the future.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Marta Patiño, for their comments and feedback. This work is supported in part by NSF (CNS-1714508 grant) and Futurewei.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *Proc. of SIGMOD*, 1995.

[2] H. Attiya, E. Hillel, and A. Milani. Inherent Limitations on Disjoint-access Parallel Implementations of Transactional Memory. In *Proc of SPAA*, 2009.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of CIDR*, 2011.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[5] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *Proc. of NSDI*, 2011.

[6] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proc. of SOSP*, 2013.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of SOCC*, 2010.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, 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. In *Proc. of OSDI*, 2012.

[9] J. Cowling and B. Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proc. of USENIX ATC*, 2012.

[10] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang. Paxos Made Transparent. In *Proc. of SOSP*, 2015.

[11] A. Dey, A. Fekete, R. Nambiar, and U. Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *Proc. of ICDEW*, 2014.

[12] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the Speed of Multi-core. In *Proc. of EuroSys*, 2014.

[13] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou. KuaFu: Closing the Parallelism Gap in Database Replication. In *Proc. of ICDE*, 2013.

[14] A. Israeli and L. Rappoport. Disjoint-access-parallel Implementations of Strong Shared Memory Primitives. In *Proc. of PODC*, 1994.

[15] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *Proc. of NSDI*, 2019.

[16] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All About Eve: Execute-verify Replication for Multi-core Servers. In *Proc. of OSDI*, 2012.

[17] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proc. of SIGMOD*, 2016.

[18] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proc. of SIGMOD*, 2015.

[19] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 1981.

[20] L. Lamport. Paxos made simple. *ACM SIGACT News*, 2001.

[21] L. Lamport. Generalized consensus and Paxos. Technical Report 2005-33, Microsoft Research, 2005.

[22] C. Leau. Spring Data Redis – Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[23] F. Li. Cloud-native database systems at Alibaba: Opportunities and challenges. In *Proc. of VLDB*, 2019.

[24] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. of SOSP*, 2017.

[25] J. Li, E. Michael, A. Szekeres, N. K. Sharma, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proc. of OSDI*, 2016.

[26] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proc. of SIGMOD*, 2017.

[27] B. Liskov and J. Cowling. Viewstamped replication revisited, 2012.

[28] Y. Mao, E. Kohler, and R. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proc. of EuroSys*, 2012.

[29] E. Michael, D. R. K. Ports, N. K. Sharma, and A. Szekeres. Recovering Shared Objects Without Stable Storage. In *Proc. of DISC*, 2017.

[30] Microsoft SQLServer. https://www.microsoft.com/en-us/sql-server/default.aspx.

[31] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in Egalitarian parliaments. In *Proc. of SOSP*, 2013.

[32] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proc. of OSDI*, 2016.

[33] MySQL. https://www.mysql.com/.

[34] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[35] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. of USENIX ATC*, 2014.

[36] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations. In *Proc. of PODC*, 2015.

[37] PostgreSQL. http://www.postgresql.org/.

[38] D. Qin, A. Goel, and A. D. Brown. Scalable Replay-Based Replication For Fast Databases. In *Proc. of VLDB*, 2017.

[39] J. Rao, E. J. Shekita, and S. Tata. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. In *Proc. of VLDB*, 2011.

[40] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 1990.

[41] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an Architectural Era: (It's Time for a Complete Rewrite). In *Proc. of VLDB*, 2007.

[42] A. Szekeres, M. Whittaker, J. Li, N. K. Sharma, A. Krishnamurthy, D. R. K. Ports, and I. Zhang. Meerkat: Multicore-scalable replicated transactions following the zero-coordination principle (extended version). Technical Report UW-CSE-19-11-02 v2, University of Washington CSE, Nov. 2019. Available at http://syslab.cs.washington.edu/papers/meerkat-tr-v2.pdf.

[43] D. Tahara. Cross shard transactions at 10 million requests per second. https://blogs.dropbox.com/tech/2018/11/cross-shard-transactions-at-10-million-requests-per-second/, Nov. 2018.

[44] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[45] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. of SIGMOD*, 2012.

[46] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proc. of SOSP*, 2013.

[47] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of OSDI*, 2004.

[48] T. Wang and H. Kimura. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. In *Proc. of VLDB*, 2016.

[49] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. In *Proc. of VLDB*, 2017.

[50] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proc. of SIGMOD*, 2016.

[51] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The End of a Myth: Distributed Transactions Can Scale. In *Proc. of VLDB*, 2017.

[52] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurhty, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. of SOSP*, 10 2015.

[53] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication (extended version). Technical Report 2014-12-01 v2, University of Washington CSE, Sept. 2015. Available at http://syslab.cs.washington.edu/papers/tapir-tr-v2.pdf.