Concurrent Prefix Recovery: Performing CPR on a Database

Guna Prasaad[§]
University of Washington
guna@cs.washington.edu

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Donald Kossmann
Microsoft Research
donaldk@microsoft.com

ABSTRACT

This paper proposes a new recovery model based on group commit, called *concurrent prefix recovery* (*CPR*). CPR differs from traditional group commit implementations in two ways: (1) it provides a semantic description of committed operations, of the form "all operations until time t_i from session i"; and (2) it uses asynchronous incremental checkpointing instead of a WAL to implement group commit in a scalable bottleneck-free manner. CPR provides the same consistency as a point-in-time commit, but allows a scalable concurrent implementation. We used CPR to make two systems durable: (1) a custom in-memory transactional database; and (2) FASTER, our state-of-the-art, scalable, larger-than-memory key-value store. Our detailed evaluation of these modified systems shows that CPR is highly scalable and supports concurrent performance reaching hundreds of millions of operations per second on a multi-core machine.

1. INTRODUCTION

The last decade has seen huge interest in building extremely scalable, high-performance multi-threaded data systems – both databases and key-value stores. Main memory databases exploit multicores (up to 1000s of cores [14]) as well as NUMA, SIMD, HTM, and other hardware advances yielding orders-of-magnitude higher performance than traditional databases. In the open-source FASTER research project [1], we have been developing key-value store technologies that push performance even further. FASTER achieves more than 150M ops/sec on one machine for point updates and lookups, while supporting larger-than-memory data and caching the hot working set in memory [5].

Applications using such systems generally require some form of durability for the changes made to application state. Modern systems can handle extremely high update rates in memory but struggle to retain their high performance when durability is desired. Two broad approaches address this requirement for durability today.

WAL with Group Commit. The traditional approach to achieve durability in databases is to use a *write-ahead log* (*WAL*) that records every change to the database. Group commit amortizes the cost of writing the log to disk as large chunks, but update-intensive applications still stress disk write bandwidth. Even without the I/O bottleneck, a WAL introduces overhead – one study [7] found that 30% of CPU cycles are spent in generating log records due to lock contention, excessive context switching, and buffer contention during logging. Recent research has improved the traditional WAL algorithm along dimensions such as buffer allocation [9], by using thread-local REDO logs [15], and optimizing for small I/Os on flash storage [6]. Johnson et al. [8] propose a distributed group commit using Lamport clocks which reduces the concurrency bottleneck but still incurs log writes. Overall, the overheads of WAL continue to affect scalability today.

Checkpoint-Replay. An alternate to WAL, popular in streaming databases, is to take periodic, consistent, point-in-time checkpoints, and use them with input replay for recovery. Cao et. al. [3] propose asynchronous checkpointing algorithms for applications that are frequently physically consistent i.e. the state of the application is transactionally consistent at a physical point in time. Such a consistent state cannot be attained without quiescing the database in most common scenarios. Traditionally, databases obtain a *fuzzy* checkpoint of its state asynchronously and use the WAL to recover a consistent snapshot during recovery. However, as noted earlier, this approach limits throughput due to the WAL bottleneck. VoltDB [10] uses an asynchronous checkpointing technique which takes checkpoints by making every database record "copy-on-write", and this approach is shown to be expensive in update-intensive workloads [5]. CALC [13] obtains asynchronous consistent checkpoints using an atomic commit log (instead of WAL), in which case the atomic log becomes the new bottleneck. To summarize, existing checkpoint-replay based durability solutions are unable to support the ever growing need for scalability.

These alternatives are depicted in Figs. 1(a) and (b). Both WAL and point-in-time checkpoints have scalability issues. To validate this point, we augmented FASTER with a WAL. An in-memory workload that previously achieved more than 150M ops/sec dropped to around 15M ops/sec after the WAL was enabled, even when writing the log to memory. Creating a copy of data on the log for every update is expensive and stresses contention on the log's tail. Further, we built an in-memory transactional database with WAL and point-in-time checkpoints and found both techniques to bottleneck at around 20M single-key txns/sec (see Sec. 6 for details). This huge performance gap has caused many real deployments to forego durability altogether, e.g., by disabling WAL in RocksDB, or by using workarounds such as approximate recovery and quiesce-and-checkpoint [4]. These approaches introduce complexity, latency, quality, and/or performance penalties.

Our Solution

In this paper, we advocate a different approach. We adopt the semantics of group commit, which commits operations as a batch, as our user

[§]Work started during internship at Microsoft Research.

DOI: https://doi.org/10.1145/3299869.3300090

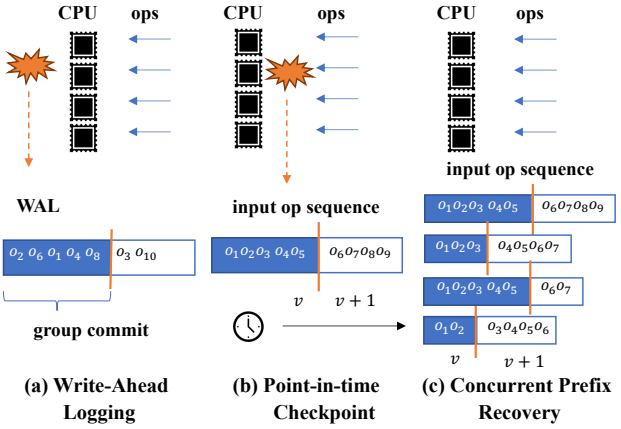


Figure 1: Approaches to Durability

model for durability. However, instead of acknowledging individual commits, we notify commit as "all operations issued up to time t": we call this model *prefix recovery*. Clients can use this acknowledgment to prune[1] their in-flight operation log until t and expose commit to users. Based on this model, we make the following contributions:

- We argue that it is not possible to guarantee a system-wide prefix recovery without quiescing or introducing a central bottleneck. To address this problem, we propose *concurrent prefix recovery* (*CPR*). In CPR (see Fig. 1(c)), the system periodically notifies each client (or session) S_i of a commit point t_i in its *local* operation timeline, such that all operations before t_i are committed, but none after. We show that CPR has the same consistency as prefix recovery, but allows a scalable asynchronous implementation.
- Traditional group commit is implemented using a WAL. Instead, we implement CPR commits using *asynchronous consistent checkpoints* that capture all changes between commits without introducing any scalability bottleneck. However, this solution requires the ability to take incremental checkpoints very quickly. Fortunately, systems such as FASTER store data in an in-place-updatable log-structured format, making incremental checkpoints very quick to capture and commit. Our approach unifies the worlds of (1) asynchronous incremental checkpoints; and (2) a WAL with group commit, augmented with in-place updates on the WAL between commits.
- While CPR makes it theoretically possible to perform group commit in a scalable asynchronous fashion, it is non-trivial to design systems that achieve these properties without introducing expensive runtime synchronization. To complete the proposal, therefore, we use CPR to build new scalable, non-blocking durability solutions for (1) a custom in-memory transactional database; and (2) FASTER, our state-of-the-art larger-than-memory key-value store. We use an extended version of epoch framework as our building block for loose synchronization, and introduce new state-machine based protocols to perform a CPR commit. As a result, our simple main-memory database implementation scales linearly up to 90M txns/sec – an order-of-magnitude higher than current solutions – while providing periodic CPR commits. Further, our implementation of FASTER with CPR reaches up to 180M ops/sec (the higher throughput compared to [5] is due to a better machine used in this paper) while supporting periodic CPR commits.

To recap, we identify the scalability bottleneck introduced by durability on update-intensive workloads, and propose CPR to alleviate this bottleneck. We then develop solutions to realize CPR in two broad classes of systems: an in-memory database and a larger-than-memory key-value store. Our detailed evaluation shows that it is possible to achieve very high performance in both these CPR-enabled systems, incurring no overhead during normal runtime, and low overhead during commit (in terms of throughput and latency).

[1] Prefix recovery and CPR also work with reliable messaging systems e.g. Kafka, which prunes input messages until some point in time.

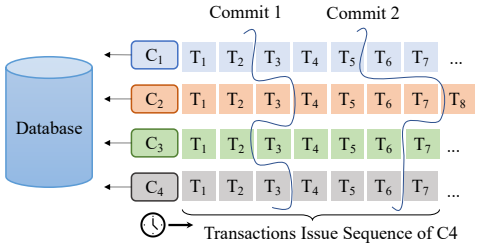


Figure 2: Concurrent Prefix Recovery Model

2. CONCURRENT PREFIX RECOVERY

A database snapshot is "transactionally consistent" if it reflects all changes made by committed transactions, and none made by uncommitted or in-flight transactions. When the database fails, it can recover to a consistent state using the snapshot, but some in-flight transactions may be lost.

A stricter recovery guarantee is "prefix recovery," where the database – upon failure – can recover to a systemwide prefix of transactions accepted for processing by the database. A naïve method to obtain a prefix recovery snapshot is to stop accepting new transactions until we obtain a consistent snapshot. This technique, called commit-consistent checkpointing [2], forcefully creates a physical point in time at which the database state is consistent, but reduces availability. An alternate method [13] achieves this asynchronously using multiversioning and an atomic commit log. The commit log records every transaction commit and is key to demarcating a prefix that determines which transactions are part of the snapshot. However, the log introduces a scalability bottleneck.

Current state-of-the-art techniques to obtain a prefix recovery snapshot quiesces the database or impedes scalability, neither of which is desirable. We indeed argue that one cannot obtain such a snapshot without these limitations. The key insight is that to obtain the snapshot, we must create a virtual time-point t corresponding to a prefix. As incoming transactions are processed simultaneously, depending on whether they are issued before or after t, they must be executed differently. For example, consider two transactions: T that is accepted before t and T' that is accepted after. Threads must execute T and T' differently as the effect of T must reflect in the snapshot, whereas that of T' should not. So, all threads must agree on a common protocol to determine this unique t, when chosen. To guarantee prefix recovery, threads must coordinate before executing every transaction, which is not possible without introducing a serial communication bottleneck.

To circumvent this fundamental limitation, we introduce CPR. In a prefix recovery snapshot, the database commits all transactions issued before a time-point t. CPR relaxes this requirement by eliminating the need for a "system-wide" time across all clients. Instead, it provides a client-local time, t_C, to each client C, such that all transactions issued by C before t_C are committed and none after t_C are.

Definition 1 (CPR Consistency). *A database state is CPR consistent if and only if, for every client C, the state contains all its transactions committed before a unique client-local time-point t_C, and none after.*

Consider the example shown in Fig. 2. The database has 4 clients issues transactions, each assigned a client-local sequence number. A CPR commit, commit 1 (marked as curve) for instance, commits the transactions $C_1 : \{T_1, T_2\}$, $C_2 : \{T_1, T_2, T_3\}$, $C_3 : \{T_1, T_2\}$, and

$C_4 : \{T_1, T_2, T_3\}$. Upon failure, the database recovers the appropriate prefix for each client: for instance, the effects of $\{T_1, T_2, T_3\}$ for client C_2. $C_2 : T_4$ cannot be recovered using commit 1. A later commit, commit 2, persists the effects of transactions until $C_2 : T_7$ including $C_2 : T_4$, and hence $C_2 : T_4$ can then be recovered.

It is desirable to be able to commit the database state at client determined time t_C. For example, concurrent clients issuing update requests as batches of transactions might want to commit at batch boundaries. We claim that client-determined CPR commit cannot be performed without quiescing the database. Let the client-determined set of CPR points for a commit with k clients be $s_1, s_2, ..., s_k$. A transaction request s' by client C_i just after s_i can be executed only when all transactions issued before each of $s_1, s_2, ..., s_k$ have been executed. Hence, s' is blocked till then. Extending this to all clients, the entire database is blocked until all transactions before $s_1, ..., s_k$ have been processed. As a result, client-determined CPR commits are unattainable without blocking. The fundamental limitation here is that s' is blocked because it must read the effects of transactions before CPR points of every client, and these are predetermined (e.g. at a batch boundary). However, in case of CPR, we could circumvent this problem by flipping the roles: clients request for a commit, and the database determines the CPR points for each client *collaboratively* while obtaining the snapshot.

3. EPOCH FRAMEWORK

The epoch framework helps *avoid synchronization between threads whenever possible*. An epoch managed thread executes user operations (e.g. transactions) independently most of the time. It uses thread-local data structures to maintain system state, letting threads lazily synchronize over critical systemwide events. The epoch framework is a key building block in CPR commit protocols. We describe its abstract function here (Refer [5, 12] for details).

We extended the standard epoch framework with custom trigger actions. Threads can register to lazily execute arbitrary global actions, called *trigger actions*, after a global event has occurred. For instance, a thread can register to execute a global action A (e.g. close a file) after a certain thread-local event E happens in every thread (e.g. a thread-local `done` flag set after reading a partition of the file). The key guarantee provided by the framework is that A is executed once and only after all thread-local events have occurred. This functionality is exposed using the following interface:

- `Acquire`: Add the current thread to the epoch managed threads.
- `Refresh`: All epoch managed threads must invoke `Refresh` periodically, but never during an user operation (e.g. only in-between and never in the middle of a transaction).
- `BumpEpoch(cond, action)`: Register ⟨`cond, action`⟩ with the framework; `action` is executed only after `cond` is satisfied.
- `Release`: Remove the current thread from epoch managed threads.

4. CPR COMMIT PROTOCOL

We now present an asynchronous protocol for performing CPR commit in a simple in-memory transactional database. The database has a shared-everything architecture where any thread can access any record. It uses strict 2-Phase Locking with No-Wait deadlock prevention policy for concurrency control. We chose this setup for ease of exposition, and we believe that our algorithm can be easily extended for other protocols as well. We also assume memory twice the size of the database to simplify explanation of the key benefit of CPR.

4.1 Commit Algorithm

Each record in the database has two values, *stable* and *live*, and an integer that stores its current *version*. In steady state, the database is at some version v. A CPR commit corresponds to shifting the database version from v to $(v + 1)$ and capturing its state as of version v. To simplify explanation, we assume a one-to-one mapping between threads and clients: each client C has a dedicated thread T_C to handle all its transactions serially in the order it was issued as shown in Alg. 1. A CPR commit is coordinated using the epoch framework (Sec. 3) as shown in Alg. 2 and its global state machine is shown in Fig. 3.

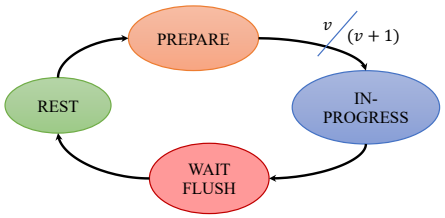


Figure 3: State Machine for CPR Commit in DB

```
Function Run()
    phase, version = Global.phase, Global.version;
    while true do
        repeat
            if inputQueue.TryDequeue(txn) then
                if not Execute(txn, phase, version) then
                    if txn aborted due to CPR then
                        break;
        until k times;
        Refresh();
        newPhase, newVersion = Global.phase, Global.version;
        if phase is PREPARE and newPhase is IN_PROGRESS then
            Record time t_T for thread T;
        phase, version = newPhase, newVersion;

Procedure Execute(txn, phase, version)
    foreach (record, accessType) in txn.ReadWriteSet() do
        if record.TryAcquireLock(accessType) then
            lockedRecords.Add(record);
            if phase is PREPARE then
                if record.version > version then
                    Unlock all lockedRecords;
                    Abort txn due to CPR;
            else if phase is IN_PROGRESS or WAIT_FLUSH then
                if record.version < version + 1 then
                    Copy record.live to record.stable;
                    record.version = version + 1;
        else
            Unlock all lockedRecords;
            Abort txn;
    Execute txn using live values;
    Add txn to thread-local staged transactions;
    Unlock all lockedRecords;
```

Algorithm 1: Pseudo-code for Execution Threads

A CPR Commit is lazily coordinated using the epoch framework over three phases: Prepare, In-Progress and Wait-Flush. The protocol state is maintained using two shared global variables, `Global.phase` and `Global.version`. They denote the current phase and version of the database respectively. Threads have a thread-local view of these variables that are updated only during `Refresh`. Avoiding frequent atomic synchronization over these variables is key to the scalability of CPR-based systems and is only possible due to the epoch framework.

Rest Phase. A commit request is issued when the database is in v, Rest. When in Rest, transactions execute normally using strict 2PL with No-Wait policy, the default high-performance phase. The algorithm is triggered by invoking the `Commit` function (Alg. 2). This updates the global state to Prepare and adds an epoch trigger action

```
Function Commit()
    Atomically set Global.phase = PREPARE;
    BumpEpoch(all threads in PREPARE, PrepareToInProg);
Procedure PrepareToInProg()
    Atomically set Global.phase = IN_PROGRESS;
    BumpEpoch(all threads in IN_PROGRESS,
      InProgToWaitFlush);
Procedure InProgToWaitFlush()
    Atomically set Global.phase = WAIT_FLUSH;
    foreach record in database do
        if record.version == Global.version + 1 then
            Capture record.stable;
        else
            Capture record.live;
    Atomically set Global.phase, Global.version = REST,
      Global.version + 1;
    Commit all staged transactions;
```

Algorithm 2: Epoch-based State Machine

PrepareToInProg, which is triggered automatically after all threads have entered Prepare. Execution threads update their local view of the phase during subsequent epoch synchronization and enter Prepare.

Prepare Phase. The Prepare phase 'prepares' threads for a CPR Commit. A transaction is executed in Prepare only if all its instructions can be executed on version v of the database. Such transactions are part of the commit and can be recovered on failure. To ensure CPR consistency, they must not read the effects of transactions that are not part of the commit. Upon encountering any record with version greater than v, the transaction immediately aborts, and the thread refreshes its thread-local view of system phase and version. At most one transaction per thread is aborted this way for every commit, since the thread advances to the next phase immediately.

In-Progress Phase. PrepareToInProg action is executed automatically after all threads enter Prepare. It updates the system phase to In-Progress and adds another trigger action, InProgToWaitFlush. When a thread refreshes its thread-local state now, it enters In-Progress. An In-Progress thread executes transactions in database version $(v+1)$; it updates the version of records it reads/writes to $(v+1)$ when it is $\leq v$. This prevents any transaction belonging to the commit from reading the effects of those that are not. To process $(v+1)$ transactions without blocking, and at the same time capture the record's final value at version v, we copy the live value to the stable value.

Wait-Flush Phase. Once all threads enter In-Progress, the epoch framework executes trigger action InProgToWaitFlush. First, it sets the global phase to Wait-Flush, then it captures version v: if a record's version is $(v+1)$, then its stable value is captured, else its live value is captured as part of the commit. Meanwhile, incoming transactions in Wait-Flush are processed similar to those in In-Progress. After all records are captured and persisted, the global phase and version are updated to Rest and $(v+1)$ respectively.

This concludes the CPR commit of version v of the database, resulting in the following theorem (proof sketch in [12]).

Theorem 1 (Correctness). *Algorithms 1 and 2 together produce a transactionally consistent snapshot:*

- *For every thread T, the snapshot reflects all transactions committed before a time t_T, and none after.*
- *The snapshot is conflict-equivalent to a point-in-time snapshot.*

Recovery. Recovery in a CPR-based database is straightforward: we simply load the database back into memory from the latest commit. Unlike traditional WAL-based recovery, there is no need for UNDO processing since the value of each record captured in Alg. 2 is transactionally consistent, and it is the final value after all v transactions have been executed. So, this corresponds to a database state when all transactions issued before time t_T for every thread T have been committed. Transactions issued after t_T by thread T are lost, as per the definition of CPR-consistency.

Time	Database State (Before)	Thread 1	Thread 2
1	$A:\langle 1,3,-\rangle, B:\langle 1,2,-\rangle$	$A=5$	$B=3$
2	1,Rest → 1,Prepare		
3	$A:\langle 1,5,-\rangle, B:\langle 1,3,-\rangle$	$B=2$	$\otimes$
4	$A:\langle 1,5,-\rangle, B:\langle 1,2,-\rangle$	$\otimes$	$B=1$
5	1,Prepare → 1,In-Progress		
6	$A:\langle 1,3,-\rangle, B:\langle 1,1,-\rangle$	$A=5$	$\otimes$
7	$A:\langle 1,5,-\rangle, B:\langle 1,1,-\rangle$	$B=7$	$A=9$
8	$A:\langle 2,9,5\rangle, B:\langle 1,7,-\rangle$	~~$A=3$~~ $\Longrightarrow \otimes$	$B=5$
9	1,In-Progress → 1,Wait-Flush		
10	$A:\langle 2,9,5\rangle, B:\langle 2,5,7\rangle$	$\otimes$	$A=3$
11	$A:\langle 2,3,5\rangle, B:\langle 2,5,7\rangle$	$A=9$	$\otimes$
12	1,Wait-Flush → 2,Rest		
13	$A:\langle 2,9,5\rangle, B:\langle 2,5,7\rangle$	$\otimes$	$A=1$
14	$A:\langle 2,1,5\rangle, B:\langle 2,5,7\rangle$	$B=4$	$\otimes$
15	$A:\langle 2,1,5\rangle, B:\langle 2,4,7\rangle$		

Rest Prepare In-Progress
Wait-Flush $\otimes$ Epoch-Refresh key: $\langle$version, live, stable$\rangle$

Figure 4: Sample Execution of CPR Algorithm

4.2 CPR By Example

As an example, we illustrate CPR on two threads for a database that has two records, A and B, see Fig. 4. Each row denotes a time step in which threads execute a 1-key write transaction: for instance $A=5$ is a transaction that updates A's value to 5. A thread updates its thread-local state during epoch refresh (denoted using $\otimes$). Initially, both threads are in Rest, processing transactions by updating the live values. We receive a commit request at $t=2$, which updates the global phase to Prepare. Threads 1 and 2 enter Prepare at $t=4$ and $t=3$ respectively. Prepare threads also check if record version $>$ current database version (i.e. 1), before executing the transactions.

Since all threads have entered Prepare, the system advances to the In-Progress phase at $t=5$. Thread 2 enters In-Progress by refreshing its epoch at $t=6$. This transition from Prepare to In-Progress demarcates its CPR-point. When a record version is 1, In-Progress threads copy its live value to stable value and update the version before processing the transaction. At $t=7$, thread 2 copies 5, the live value of A, to stable value, updates version to 2 and writes 9 to live value. Thread 1, which is still in Prepare, tries to update A at $t=8$ but aborts since its version is greater than 1 and immediately refreshes its epoch. Thread 1 enters In-Progress now, marking its CPR-point. As all threads are in In-Progress, the system enters the Wait-Flush phase. We capture the stable values, $A=5$ and $B=7$, in the background while threads execute transactions belonging to version 2 on the live values. For other records with version ≤ 1, the live value is captured as part of the commit. Once the captured values are safely persisted on disk, the system transits to Rest with version 2. This ends the CPR commit of version 1 of the database with CPR-points $t=8$ and $t=6$.

5. CPR IN FASTER

We next show how CPR-based durability is added to FASTER [5], our recent open-source concurrent latch-free hash key-value store. It supports reads, blind upserts, and read-modify-write (RMW) operations over larger-than-memory data. In the FASTER paper, we report a scalable in-memory throughput of more than 150M ops/sec for the working set in memory, making it a good candidate to apply CPR.

FASTER has two main components, a *hash index* and a *log-structured record store* called HybridLog. HybridLog defines a *logical address space* that spans secondary storage and main memory. Each record contains some metadata, a key, and a value. Records corresponding to keys that share the same slot in the hash index are organized as a reverse linked list: each record's metadata contains the logical address of the previ-

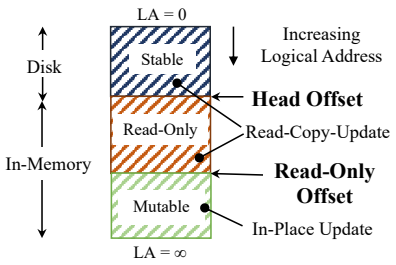


Figure 5: HybridLog Organization in FASTER

ous record mapped to that slot. The hash index points to the logical address of the latest (tail) record in this linked list.

The HybridLog address space (Fig. 5) is divided into an immutable stable region (on disk), an immutable read-only region (in memory), and a mutable region (also in memory). The *head offset* tracks the smallest logical address available in memory. The *read-only offset* divides the in-memory portion of the log into *immutable* and *mutable* regions. The *tail offset* points to the next free address at the tail of the log. FASTER threads perform in-place updates in the hot mutable region for high in-memory performance. Updates to the immutable region use read-copy-update, where a new mutable copy of the record is created at the end of tail to update it. FASTER uses epoch protection to control access to shared memory in a latch-free manner.

5.1 Towards Adding Durability

By default, the index and in-memory portion of HybridLog is lost on failure. We added the ability to periodically commit in-flight operations in the mutable region using CPR, by adding a session-based persistence API to FASTER. Clients can start and end a *session*, identified by a unique Guid, using StartSession and StopSession. Every operation such as Upsert on FASTER occurs within a session, and carries a monotonic session-local serial number. On failure, a client can re-establish a session by invoking ContinueSession with its session Guid as parameter. This call returns the last serial number (CPR point) that FASTER has recovered on that session. As described earlier, CPR commits are session-local, and FASTER recovers to a specific CPR point for every session. The client can also register a callback to be notified of new CPR points whenever FASTER commits.

FASTER provides threads unrestricted access to records in the mutable region of HybridLog, letting user code control concurrency. As CPR enforces a strict *only and all* policy, it is challenging to obtain a CPR-consistent checkpoint without compromising on fast concurrent memory access.

5.2 Asynchronous I/O and CPR

FASTER supports disk-resident data using an asynchronous model: an I/O request is issued in the background, while the requesting thread processes future requests. The user-request is executed later once the record is retrieved from disk. FASTER supports two CPR modes. In the *strict* mode, pending operations logically occur at the point they were originally issued. We also support a *relaxed* mode, where pending operations are re-ordered to logically occur at the time of continuation after I/O completion.

Asynchronous I/O complicates strict CPR in a fundamental way since some requests before a CPR point may be pending. Recall that in CPR, a request r_1 not belonging to the commit must not be executed before a request r_2, potentially from a different session, belonging to the commit. This requirement can lead to quiescing when handled naively; we assume strict CPR and address the issue in our solution.

5.3 HybridLog Checkpoint

We augmented the per-record header in HybridLog to include a version number v for a record. During normal processing, FASTER is in the Rest phase and at a particular version v. HybridLog checkpointing involves (1) shifting the version from v to $(v+1)$; and (2) capturing modifications made during version v. We leverage our epoch framework (Sec. 3) to loosely coordinate a global state machine (see Fig. 6a) for CPR checkpointing without affecting user-space performance. It consists of 5 states: Rest, Prepare, In-Progress, Wait-Pending, and Wait-Flush; state transitions are realized by FASTER threads lazily, when they refresh their epochs. A sample execution with 4 threads is shown in Fig. 6b. Following is a brief overview of each phase:

- Rest: Normal processing on FASTER version v, with identical performance to unmodified FASTER.
- Prepare: Requests accepted before and during the Prepare phase for every thread are part of v commit.
- In-Progress: Transition from Prepare to In-Progress demarcates a CPR point for a thread: requests accepted in In-Progress (or later) phases do not belong to v commit.
- Wait-Pending: Complete pending v requests (in strict CPR only).
- Wait-Flush: Unflushed v records are written to disk asynchronously.
- Rest: Normal processing on FASTER version $(v + 1)$.

A CPR commit request (from user or triggered periodically) first records the current tail offset of HybridLog, say L_s^h, and updates the global state from Rest to Prepare. Threads enter Prepare during their subsequent epoch refresh.

Prepare. A Prepare thread T processes an incoming user-request under a shared latch on the key's bucket. When the shared-latch acquisition fails or when the record version is $> v$, T detects that the CPR shift has begun and refreshes its epoch immediately, entering the In-Progress phase. If it never encounters such a scenario, the CPR shift happens during a subsequent epoch refresh. Additionally, in strict CPR, all pending requests are associated with a held shared latch.

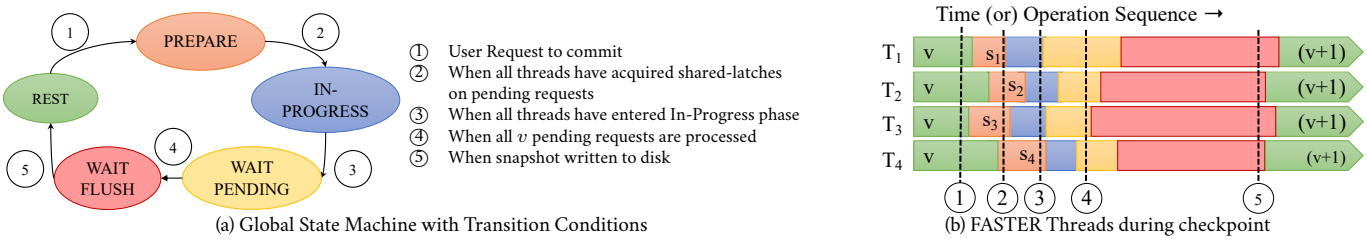
In-Progress. After all threads enter the Prepare phase, the state machine advances to In-Progress. A thread demarcates its CPR point at its transition from Prepare to In-Progress. It now processes requests as belonging to version $(v + 1)$. Accessed records in the mutable region are handled carefully. If the record version is $(v + 1)$, the thread modifies it in-place as usual. If the record has version $\leq v$, it acquires an exclusive latch on the key's bucket, performs a read-copy-update, creating an updated $(v + 1)$ record at the tail, and releases the latch. If exclusive-latch acquisition fails, the request is added to a thread-local pending list corresponding to version $(v + 1)$.

Wait-Pending. When all threads enter In-Progress, FASTER enters Wait-Pending in strict CPR, where pending I/Os in version v get completed by all threads, releasing shared latches.

Wait-Flush. Once all v requests have been completed, we record the tail offset of HybridLog, say L_e^h, and shift the read-only offset to L_e^h, which asynchronously flushes HybridLog until L_e^h to disk. Once the asynchronous write to disk is complete, system moves back to Rest with version $(v + 1)$. This concludes the HybridLog checkpoint.

5.4 Index Checkpoint

In addition to the HybridLog checkpoint, we obtain a fuzzy checkpoint of the hash index that maps key-hash to logical addresses on HybridLog. The main reason for checkpointing the index is to reduce recovery time by replaying a smaller suffix of the HybridLog during recovery (similar to database checkpoints for WAL truncation). Hence, it can be done much less frequently, particularly with slower log growth due to in-place updates in HybridLog. Since hash bucket entries are updated only using atomic compare-and-swap instructions, the index is always physically consistent. To obtain a fuzzy checkpoint, we write the hash index pages to storage using asynchronous I/O. We also record the tail offset of HybridLog before starting (L_s^i)



(a) Global State Machine with Transition Conditions

(b) FASTER Threads during checkpoint

Figure 6: Overview of CPR for FASTER

and after completion (L_e^i) of the fuzzy checkpoint. We use these offsets during recovery, which is described next.

5.5 Recovery

FASTER recovers to a CPR-consistent state using a combination of a fuzzy hash index and HybridLog checkpoint (say of version v). During recovery, we scan through records in a section of HybridLog, from logical address $S = \min(L_s^i, L_s^h)$ to $E = \max(L_e^i, L_e^h)$, updating the hash index appropriately. The recovered index must point to the latest record with version $\leq v$ for each slot. Due to the fuzzy nature of our index checkpoint, it could point to $(v+1)$ records or records that are not the latest.

For records in the section of HybridLog between S and E: If the version is $\leq v$, we update the index slot to point to the record's logical address, L_R. When the version is $> v$, we mark the record invalid as it does not belong to v commit of FASTER. Additionally, when the address in the slot is $\geq L_R$, we update the index to point to the previous address stored in the record header. This fix-up may be considered the UNDO phase of our recovery in FASTER. As noted earlier, each slot in the hash index points to a reverse linked-list of records stored in the HybridLog. The copy-on-update scheme in FASTER ensures that records in this list have decreasing logical addresses, while the HybridLog checkpoint design ensures that $(v+1)$ records occur only before all v records in the list. Together, these two invariants result in a consistent FASTER hash index after recovery.

6. EVALUATION

We evaluate CPR in two ways. First, we compare CPR with two state-of-the-art asynchronous durability solutions for a main-memory database: CALC [13] and WAL [11]. Next, we evaluate CPR on our key-value store, FASTER. We present only the key results here and refer the reader to our full paper [12] for a detailed evaluation.

Implementation. For the first part, we implemented a stand-alone main-memory database, that supports three recovery techniques (CPR, CALC, and traditional WAL). Both CALC and CPR implementations have two values, *stable* and *live*, for each record, while WAL only has a single value. An optimal implementation of CPR does not require two values for each record; we do this for a head-to-head comparison with CALC [13]. The entire database is written to disk asynchronously during a CPR/CALC checkpoint. We do not obtain fuzzy checkpoints for WAL but periodically flush the log to disk. All three versions use the main-memory version of FASTER [5] as the data store and implement two-phase locking with NO-WAIT deadlock avoidance policy.

We added CPR to FASTER and that constitutes the second part of our evaluation. Threads first load the key-value store with data, and then issue a sequence of operations. Commit requests are issued periodically. We report system throughput and latency every two seconds. We point FASTER to our SSD, and employ the default expiration based garbage collection scheme (not triggered in these experiments). The total in-memory region of HybridLog is set at 32GB, large enough that reads never hit storage for our workloads, with the mutable region set to 90% of memory at the start. By default, FASTER hash index has #keys/2 hash-bucket entries. We do not directly compare with existing solutions since prior work [5] has shown that other persistent key-value stores such as RocksDB achieve an order of magnitude lower performance (< 1M ops/sec) even when WAL is disabled.

Setup. The first set of experiments are conducted on a Standard D64s v3 machine on Microsoft Azure. The machine has 2 sockets and 16 cores (32 hyperthreads) per socket, 256GB memory and runs Windows Server 2018. Experiments on CPR with FASTER are carried out on a local Dell PowerEdge R730 machine with 2.3GHz Intel Xeon Gold 6140 CPUs, running Windows Server 2016. The machine has 2 sockets and 18 cores (36 hyperthreads) per socket, 512GB memory and a 3.2TB FusionIO NVMe SSD drive. The two-socket experiments shard threads across sockets. We preload input datasets into memory.

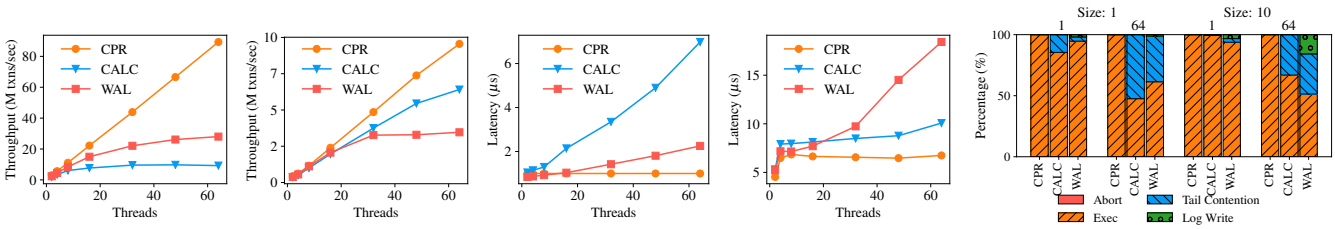
Workloads. For our stand-alone database, we use a mix of transactions based on the Yahoo! Cloud Serving Benchmark (YCSB). Transactions are executed against a single table with 250 million 8 byte keys and 8 byte values. Each transaction is a sequence of read/write requests on these keys, which are drawn from a Zipfian distribution. A request is classified as read or write randomly based on a read-write ratio written as W:R; a read copies the existing value, and a write replaces the value in the database with a provided value. We mainly focus on a low contention ($\theta = 0.1$) workload here since it incurs the most performance penalty due to logging or tail contention.

For FASTER with CPR, we use an extended version of the YCSB-A workload, with 250 million distinct 8 byte keys, and value sizes of 8 and 100 bytes. After pre-loading, records occupy 6GB of HybridLog space and the index is 8GB. Workloads are described as R:BU for the fraction of reads and blind updates. We add *read-modify-write* (RMW) updates in addition to the blind updates supported by YCSB. Such updates are denoted as 0:100 RMW in experiments (we only experiment with 100% RMW updates for brevity). RMW updates increment a value by a number from a user-provided input array with 8 entries, to model a per-key "sum" operation. We use the standard Uniform and Zipfian ($\theta = 0.99$) distributions in our workloads.

6.1 Evaluation on Transactional Database

We first plot average throughput (Figs. 7a, 7b) and latency (Figs. 7c, 7d) of the three systems against a varying number of threads for a mixed read-write (50 : 50) workload – for 1- and 10-key transactions. We also profiled the experiment; the breakdown for 1 and 64 threads are shown in Fig. 7e. "Exec" refers to the cost of in-memory transaction processing including acquiring and releasing locks, "Tail-Contention" is the overhead of LSN allocation (in WAL) and appending to the commit log (in CALC), while "Log Write" denotes the cost of writing WAL records on the log.

Scalability. CPR scales linearly up to 90M txns/sec on 64 threads for 1-key transactions, whereas CALC and WAL reach a maximum of



(a) Scalability; Size:1 (b) Scalability; Size:10 (c) Latency; Size:1 (d) Latency; Size:10 (e) Analysis

Figure 7: Scalability and Latency on Low Contention (θ= 0.1) YCSB workload

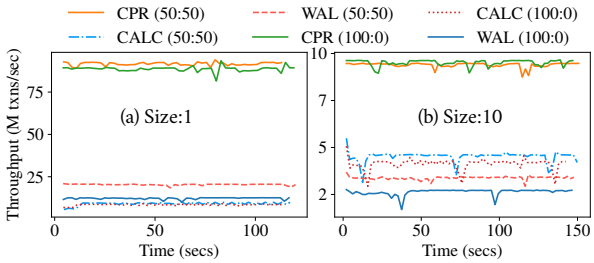


Figure 8: Throughput during Checkpoint

10M txns/sec and 25M txns/sec respectively. The breakdown analysis reveals that tail contention in WAL and in CALC's atomic commit log are a scalability bottleneck. WAL performs better than CALC here since every transaction is appended to the commit log, while 50% read-only 1-key transactions do not generate any WAL records. In case of 10-key transactions, CPR again scales linearly up to 10M txns/secs, while WAL and CALC scale only up to 3.5 and 6.2M txns/sec. Tail contention is still a bottleneck (about $30-40\%$) for both CALC and WAL, while WAL incurs an additional 20% overhead for writing log records. Unlike the 1-key case, CALC outperforms WAL since most transactions contain at least one write resulting in a WAL record.

Latency. 1-key transactions (Fig 7c) in CPR are executed in approximately 700 nanoseconds and the latency almost remains constant as we increase the number of threads. This is due to the highly efficient design of the underlying FASTER hash index [5]. Due to tail contention, latency in CALC and WAL increases as we scale. CALC results in a latency of 6μs on 64 threads, while WAL incurs an average latency of only 2μs due to 50% read-only transactions. In CPR, 10-key transactions (Fig 7c) incur a cost of $7\mu s$, which is 10x that of a 1-key transaction. CALC latency, even though higher than CPR due to tail contention in the atomic commit log, remains almost constant because the cost of execution is higher in 10-key transactions. Since most 10-key transactions result in a WAL record, the effect of tail contention and writing log records is evident from the increasing trend.

Throughput vs. Time. We now plot average throughput during the lifetime of a run for CPR, CALC and WAL on 64 threads, with checkpoints at 30, 60 and 90 secs both for mixed $(50:50)$ and write-only $(100:0)$ workloads; Fig. 8a and Fig. 8b correspond to 1- and 10-key transactions respectively. In all three systems, there is no observable drop in throughput during checkpointing. This is due to the asynchronous nature of the solutions. Even for 10-key transactions, the effect of copying over records from live to stable values is minimal as they are already available in upper levels of the cache. CPR design scales better overall and does not involve any serial bottlenecks, yielding a checkpoint throughput of 90M txns/sec. As noted earlier, WAL is better than CALC in $50:50$ 1-key transactions due to 50% read-only transactions. The minor difference between write-only and mixed workloads is because writes are more expensive than reads.

6.2 Evaluation of FASTER with CPR

Throughput and Log Size. We plot throughput vs. wall-clock time during the lifetime of a FASTER run. We perform two "full" (index and log) commits during the run, at the 10 sec and 40 sec mark respectively, and plot results for two key distributions (Uniform and Zipf). We evaluate both our commit techniques – *fold-over* and *snapshot* to separate file – in these experiments.

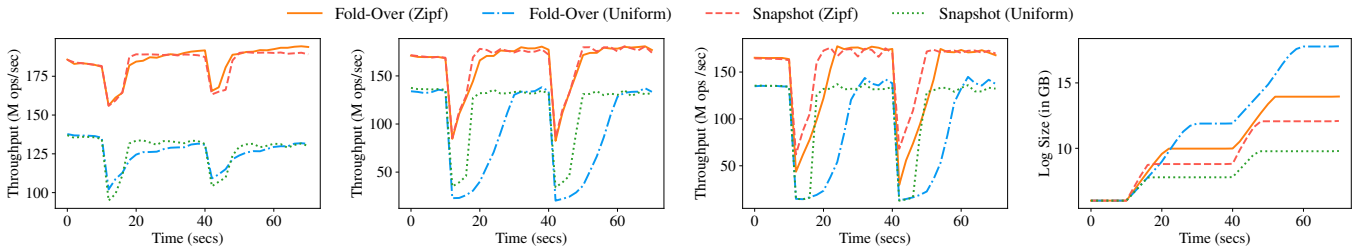
Fig. 9a shows the result for a 90:10 workload (i.e., with 90% reads). Overall, Zipf outperforms Uniform due to better locality of keys in Zipf, reaching up to 180M ops/sec. After commit, both snapshot and fold-over slightly degrade in throughput because of read-copy-updates. It takes 6 secs to write 14GB of index and log, close to the sequential bandwidth of our SSD. After the second commit, the Zipf throughput of fold-over returns to normal faster than snapshot because of its incremental nature. With a 50:50 workload, in Fig. 9b, fold-over drops in throughput after commit, because of the overhead of read-copy-update of records to the tail of `HybridLog`. Performance increases as the working set migrates to the mutable region, with Zipf increasing faster than Uniform as expected. For this workload, snapshot does better than fold-over as it is able to dump the unflushed log to a snapshot file and quickly re-open `HybridLog` for in-place updates. A 0:100 workload with only blind updates demonstrates similar effects, as shown in Fig. 9c. We also profiled execution for the time taken in each CPR phase: each phase lasted for around 5ms, except for Wait-Flush, which took around 6 secs as described above.

Fig. 9d depicts the size of `HybridLog` vs. time, for a 0:100 workload. We note that (1) `HybridLog` size grows slowly with snapshot, as the snapshots are written to a separate file; and (2) `HybridLog` for Uniform grows faster than for Zipf, because more records need to be copied to the tail after a commit for Uniform.

We also experimented with checkpointing only the log, with more frequent commits, since the index is usually checkpointed infrequently. The results are in [12]; briefly, we found CPR commits to have much lower overhead as expected, with a similar trend overall.

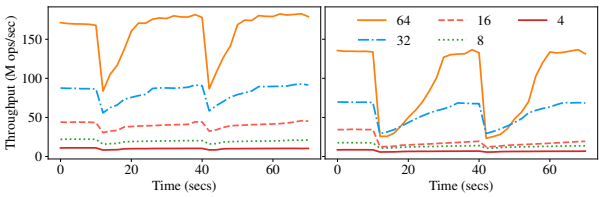
Varying number of threads. We plot throughput vs. time for varying number of threads from 4 to 64, for a 50:50 workload. We depicts the results for Zipf and Uniform distributions in Figs. 10a and 10b respectively, with full fold-over commits taken at the 10 sec and 40 sec mark. Both figures show linear throughput improvement with increasing number of threads, indicating that CPR does not affect scalability. In fact, normal (Rest phase) performance is unaffected by the introduction of CPR. At lower levels of scale, the effect of CPR commits is minimal due to lower Rest phase performance. Further, performance recovery after a commit is faster with more threads, since hot data migrates to mutable region faster.

End-to-End Experiment We evaluate an end-to-end scenario with 36 client threads feeding a 50:50 YCSB workload to FASTER. Each client has a *buffer* of in-flight (uncommitted) requests. When a buffer



(a) Throughput; YCSB 90:10 (b) Throughput; YCSB 50:50 (c) Throughput; YCSB 0:100 (d) Log growth; YCSB 0:100

Figure 9: FASTER Throughput and Log Growth vs. Time; Full Fold-over and Snapshot Commits at **10** and **40** secs



(a) 50:50 Zipf distribution (b) 50:50 Uniform distribution

Figure 10: Throughput vs. Time; Varying #Threads

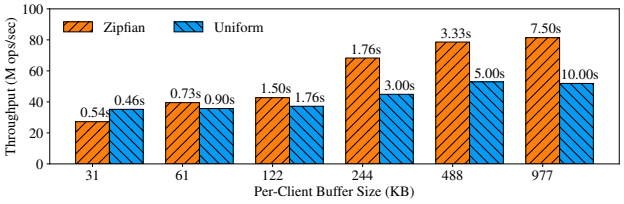


Figure 11: End-to-end Experiment; YCSB **50**:**50**

reaches 80% capacity, we issue a log-only fold-over commit request, which allows clients to trim their buffers based on CPR points. Clients block if their buffers are full. Each entry in the buffer takes up 16 bytes (for the 8 byte key and value). Fig. 11 shows the results for Zipf and Uniform workloads, as we vary the per-client buffer size. Above each bar is the corresponding average checkpoint interval, or the latency of CPR commit, observed for the given buffer size. We take one full checkpoint, and report average throughput over the next 30 secs.

Increasing the buffer size allows more in-flight operations, which improves throughput for both workloads. Even a small buffer is seen to provide high throughput. For small buffer sizes, commits are issued more frequently (e.g., every 0.5 secs for a 30KB buffer) as expected. The Zipf workload reaches a higher maximum throughput with a larger buffer because the smaller working set reaches the mutable region faster between commits. With the smallest buffer, Uniform outperforms Zipf due to the higher contention faced in Zipf when moving items to the mutable region after every (frequent) commit.

7. CONCLUSION

Modern databases and key-value stores have pushed the limits of multi-core performance to hundreds of millions of operations per second, leading to durability becoming the central bottleneck. Traditional durability solutions have scalability issues that prevent systems from reaching very high performance. We propose a new recovery model based on group commit, called *concurrent prefix recovery* (*CPR*), which is semantically equivalent to a point-in-time commit, but allows a scalable implementation. We present CPR commit protocols for a custom in-memory transactional database and FASTER, our key-value store that supports larger-than-memory data. A detailed evaluation of both systems shows that CPR supports highly concurrent and scalable performance, while providing durability. FASTER with CPR is available as open-source software [1].

8. REFERENCES

[1] FASTER Project. `https://github.com/microsoft/FASTER`.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[3] T. Cao, M. A. V. Salles, B. Sowell, Y. Yue, A. J. Demers, J. Gehrke, and W. M. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD 2011*.

[4] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, Dec. 2014.

[5] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD 2018*.

[6] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[7] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD 2008*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[8] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB J.*, 21(2):239–263, 2012.

[9] H. Jung, H. Han, and S. Kang. Scalable database logging for multicores. *PVLDB*, 11(2):135–148, 2017.

[10] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE 2014*.

[11] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[12] G. Prasaad, B. Chandramouli, and D. Kossmann. Concurrent Prefix Recovery: Performing CPR on a Database. In *SIGMOD 2019*.

[13] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *SIGMOD 2016*.

[14] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[15] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI 2014*.