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

With increasing multi-core parallelism, modern databases and key-value stores are designed for scalability and presently yield very high throughput for the in-memory working set. These systems typically depend on group commit using a *write-ahead log* (*WAL*) to provide durability and crash recovery. However, a WAL is expensive, particularly for update-intensive workloads, where it also introduces a concurrency bottleneck (the log) besides log creation and I/O overheads. In this paper, we propose 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.

ACM Reference Format:
Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. 2019. Concurrent Prefix Recovery: Performing CPR on a Database. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300090

*Work performed during internship at Microsoft Research.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300090

1 INTRODUCTION

The last decade has seen huge interest in building extremely scalable, high-performance multi-threaded data processing systems – both databases and key-value stores. For instance, main memory databases [9, 10, 18] exploit multiple cores (up to thousands of cores in some cases [30]) as well as NUMA, SIMD, HTM, and other hardware advances to drive performance to orders-of-magnitude higher levels than those achieved by traditional databases. Key-value stores are pushing performance even further: for instance, Masstree [24] achieves high in-memory throughput – up to 30M ops/sec on one machine – compared to traditional range indices. Our recent open-source key-value store, FASTER [6], 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.

1.1 New Bottleneck and Today's Solutions

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*), in which every change to the database is recorded. Techniques such as *group commit* [8, 13] enable writing the log to disk in larger chunks, but update-intensive applications stress disk write bandwidth. Even without the I/O bottleneck, a WAL introduces overhead – one study found that 30% of CPU cycles are spent in generating log records [15] due to lock contention, excessive context switching, and buffer contention during logging.
- *Checkpoint-Replay*: An alternate to using a WAL, popular in streaming databases, is to take periodic, consistent, point-in-time checkpoints, and use them with input replay for recovery. Taking an asynchronous checkpoint is trivial with a WAL: we could take a *fuzzy* checkpoint and use the log to recover a consistent snapshot. However, as noted earlier, this approach limits throughput due to the WAL bottleneck. We could avoid this if we quiesce the database

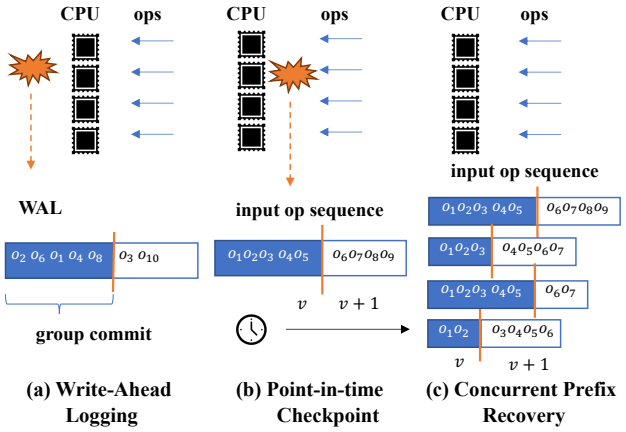


Figure 1: Approaches to Durability

but would lose asynchronicity and create downtime. A recent proposal, CALC [26], uses asynchronous consistent checkpoints but depends on an atomic commit log (instead of the WAL) to define the consistency point. However, the atomic commit log then becomes the new bottleneck, precluding scalable multi-threaded performance.

These alternatives are depicted in Fig. 1(a) and (b). Both write-ahead logging and point-in-time checkpoints have scalability issues due to the WAL and commit log bottlenecks respectively. To validate this point, we augmented our key-value store, FASTER [6], with a WAL. Results showed that 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. We also built an in-memory transactional database and found its throughput with both a WAL and point-in-time checkpointing to bottleneck at around 20M single-key txns/sec (see Fig. 2 and Sec. 7 for details).

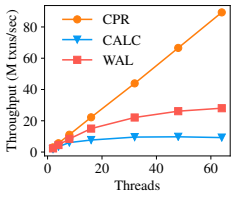


Figure 2: Scalability

This huge performance gap has caused many real deployments to forego durability altogether, e.g., by disabling the WAL in RocksDB [27], or by using workarounds such as approximate recovery and quiesce-and-checkpoint [5]. These approaches introduce complexity, latency, quality, and/or performance penalties (see Sec. 8 for related work).

1.2 Our Solution

In this paper, we advocate a different approach. We adopt the semantics of group commit, which commits a group of operations at once, as our user model for durability. However, instead of acknowledging individual commits, we convey commit as "all operations issued up to time t": we call this model *prefix recovery*. Clients can use this information to prune[1] their in-flight operation log until t and expose commit to users. Based on this durability model, we make the following contributions:

- We argue that in a multi-threaded system, it is not possible to provide a prefix recovery guarantee over a global operation timeline without introducing system blocking or a central bottleneck. To address this problem, we propose an augmented model called *concurrent prefix recovery* (*CPR*). In CPR (see Fig. 1(c)), the system periodically notifies each user *thread* (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.
- CPR commits all operations before a per-thread point in time and none after. While it appears desirable for applications to choose specific CPR points, e.g., at input batch boundaries, we show that if the application were to ask for a *specific* set of commit points, we would be unable to satisfy the request without causing some threads to block. Instead, we flip the request: the application requests the system to commit, and the system coordinates the global construction of *some* commit point for each thread without losing asynchronicity or creating a central bottleneck.
- 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 protection [19] as our building block for loose synchronization, and introduce new state-machine based protocols to perform a CPR

[1] Prefix recovery and CPR also work with reliable messaging systems such as Kafka [12], which can prune input messages until some point in time.

commit. As a result, our simple main-memory database implementation scales linearly (see Fig. 2) 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[2] reaches up to 180M ops/sec, while supporting larger-than-memory data and 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).

The rest of the paper is organized as follows. Sec. 2 defines CPR. We review epochs (a key building block for our designs) in Sec. 3. Sec. 4 designs CPR for an in-memory database. Next, we design CPR for FASTER in Secs. 5 and 6. Finally, we evaluate our solutions (Sec. 7), survey related work (Sec. 8), and provide concluding remarks (Sec. 9).

2 CONCURRENT PREFIX RECOVERY

A database snapshot is said to be *transactionally consistent* if it reflects all changes made by committed transactions, and none made by uncommitted or in-flight transactions. In the event of a failure, the database can recover to a consistent state using such a 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 system-wide prefix of all issued transactions, i.e., those that have been accepted for processing by the database. The database state at any given moment may not be transactionally consistent since transactions are always being executed. A naïve method to obtain a snapshot for prefix recovery is to stop accepting new transactions until we obtain a consistent snapshot. This technique, called *commit-consistent checkpointing* [3], forcefully creates a physical point in time at which the database state is consistent, but reduces availability.

An alternate method achieves this asynchronously by maintaining two versions of the database, called *stable* and *live*. Logically, transactions that belong to the prefix update both the live and the stable version, whereas transactions that do not belong to the prefix update only the live version. The stable version is then captured asynchronously as the snapshot for prefix recovery. The challenge, however, is in determining an appropriate prefix since transactions are being executed concurrently. Previous approaches [26] have used an *atomic commit log* that records every transaction commit to dynamically choose the prefix. However, as Secs. 1 and 7 reveal, such a log introduces a serial bottleneck that greatly impedes system scalability.

[2]Get FASTER with CPR at `https://github.com/Microsoft/FASTER`.

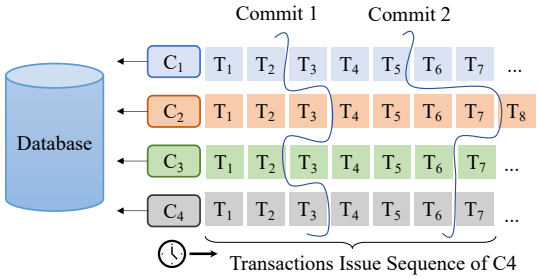


Figure 3: Concurrent Prefix Recovery Model

We now argue that one cannot obtain a snapshot for prefix recovery without introducing a serial bottleneck. The key insight is that to obtain such a snapshot, we must create a virtual time-point t corresponding to a prefix. As incoming transactions are processed simultaneously, depending on whether they occur before or after t, they must be executed differently. For example, consider two transactions: T that executes before t and T' that executes 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 mechanism 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 bottleneck.

To overcome this limitation, we introduce *Concurrent Prefix Recovery* (*CPR*). Recall that on obtaining a prefix snapshot, the database commits all transactions issued before a time-point t by each of its clients. CPR consistency relaxes this requirement by eliminating the need for a *system-wide* time t. 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. We formally define CPR below:

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. 3. The database has 4 clients issuing 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. Transaction T_4 of C_2 cannot be recovered. A later commit, commit 2, persists the effects of transactions until T_7 for C_2, including T_4, and hence T_4 can then be recovered.

It is desirable to be able to commit at client-determined CPR points. 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 a 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.

A key insight from the preceding argument 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). To circumvent this problem, we flip the roles: clients request for a commit, and the database collaboratively determines the CPR point for each client while trying to perform a CPR commit. We review the basics of epoch protection framework next, which forms the basis of our CPR commit algorithms.

3 EPOCH FRAMEWORK BACKGROUND

Epoch protection helps *avoid coordination whenever possible*. A thread performs user operations independently without any synchronization most of the time. It uses thread-local data structures extensively and maintains system state by lazily synchronizing over the state. Maintenance operations (e.g., deciding to flush a page to disk) can be performed collaboratively by leveraging an extended epoch protection framework. We use epochs as a key building block to design CPR commit protocols in this paper.

Epoch Basics. We maintain a shared atomic counter **E**, called the *current epoch*, that can be incremented by any thread. Every thread T has a thread-local version of **E**, denoted by E_T . Threads refresh their local epoch values periodically. All thread-local contexts, including the epoch values E_T, are stored in a shared epoch table, with one cache-line per thread. An epoch c is said to be *safe*, if all threads have a strictly higher thread-local epoch value than c, i.e., $\forall T : E_T > c$. Note that if epoch c is safe, all epochs less than c are safe as well. We additionally maintain a global counter $\mathbf{E_s}$, which tracks the current maximal safe epoch. $\mathbf{E_s}$ is computed by scanning all entries in the epoch table and is updated whenever a thread refreshes its epoch. The system maintains the following invariant: $\forall T : \mathbf{E_s} < E_T \leq E$.

Trigger Actions. Threads can register to execute arbitrary global actions called *trigger actions* when an epoch becomes safe and a certain condition is satisfied by all thread-local contexts. When incrementing the current epoch, say from e to $e + 1$, threads can optionally associate with it a condition C on the thread-local context and an action A. The epoch framework automatically triggers A when e becomes safe and when all thread-local contexts in the system satisfy condition C. This is enabled using a *drain-list*, a list of $\langle$epoch, cond, `action`$\rangle$ tuples, where `action` is the callback code fragment that must be invoked after epoch is safe and cond is the condition that must be satisfied by all thread-local contexts. We expose epoch protection using the following operations that can be invoked by any thread T :

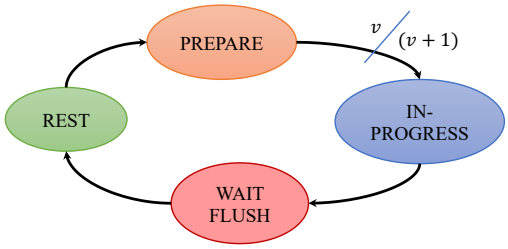


Figure 4: State Machine for CPR Commit in DB

- `Acquire`: Reserve an entry for T and set E_T to **E**
- `Refresh`: Update E_T to **E**, E_s to current maximal safe epoch and trigger any ready actions in the drain-list
- `BumpEpoch(cond, action)`: Increment counter **E** from e to $(e + 1)$ and add $\langle e, cond, action\rangle$ to drain-list
- `Release`: Remove entry for T from epoch table

4 CPR COMMIT IN AN IN-MEMORY TRANSACTIONAL DATABASE

We now present an asynchronous algorithm for performing CPR commit in a simple in-memory transactional database that uses strict 2-Phase Locking with No-Wait deadlock-prevention policy for concurrency control. We chose this specific protocol for ease of exposition, and we believe that our algorithm can be extended for other protocols as well. We also assume memory twice the size of the database to simplify explanation of the key benefit of adding CPR. Sec. 6 covers CPR for a real system without this assumption.

4.1 Commit Algorithm

The database has a shared-everything architecture where any thread can access any record. All transactions issued by a client C are processed by the same thread T_C, and different threads handle transactions from different clients using pseudo-code in Alg. 1. A CPR commit is coordinated using the epoch framework (Sec. 3) as shown in Alg. 2.

The algorithm eliminates the need to coordinate among execution threads and instead, relies on fine-grained information maintained at the records to perform a CPR commit. Each record in the database has two values: *stable* and *live*,

```
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

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. This is lazily coordinated using the epoch framework. The algorithm executes over three phases: PREPARE, IN-PROGRESS and WAIT-FLUSH. We maintain two shared global variables, `Global.phase` and `Global.version`, to denote the database's current phase and version. Threads have a thread-local view of these variables and update them only during epoch synchronization. Avoiding frequent synchronization of these global state variables is key to the scalability of CPR-based systems. The global state machine corresponding to Alg. 2 is shown in Fig. 4.

Rest Phase. A commit request is issued when the database is in REST phase and at some version v. 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 `PrepareToInProg`, which is triggered automatically

```
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

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' execution threads for a shift in database version. A transaction is executed in PREPARE only if its entire set of read-write instructions can be executed on version v of the database. Such transactions are part of the commit and hence 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. Interestingly, at most one transaction per thread is aborted this way for every commit, since the thread advances to the next phase, IN-PROGRESS, when it refreshes.

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 location as shown in Alg. 1.

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

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	$A:\langle 1,5,-\rangle, B:\langle 1,3,-\rangle$	$B=2$	⊗
4	$A:\langle 1,5,-\rangle, B:\langle 1,2,-\rangle$	⊗	$B=1$
5	1,PREPARE → 1,IN-PROGRESS		
6	$A:\langle 1,3,-\rangle, B:\langle 1,1,-\rangle$	$A=5$	⊗
7	$A:\langle 1,5,-\rangle, B:\langle 1,1,-\rangle$	$B=7$	$A=9$
8	$A:\langle 2,9,5\rangle, B:\langle 1,7,-\rangle$	~~$A=3$~~ ⟹ ⊗	$B=5$
9	1,IN-PROGRESS → 1,WAIT-FLUSH		
10	$A:\langle 2,9,5\rangle, B:\langle 2,5,7\rangle$	⊗	$A,3$
11	$A:\langle 2,3,5\rangle, B:\langle 2,5,7\rangle$	$A=9$	⊗
12	1,WAIT-FLUSH → 2,REST		
13	$A:\langle 2,9,5\rangle, B:\langle 2,5,7\rangle$	⊗	$A=1$
14	$A:\langle 2,1,5\rangle, B:\langle 2,5,7\rangle$	$B=4$	⊗
15	$A:\langle 2,1,5\rangle, B:\langle 2,4,7\rangle$		

REST PREPARE IN-PROGRESS
WAIT-FLUSH ⊗ Epoch-Refresh key: ⟨version, live, stable⟩

Figure 5: Sample Execution of CPR Algorithm

version v of the database: 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. Note that we may reduce commit size by capturing only records that changed since last commit; this is an orthogonal optimization covered in prior work [26].

4.2 CPR By Example

We now illustrate CPR on two threads for a simple database that has two records, A and B, using Fig. 5. 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. Threads update their thread-local state during epoch refresh (denoted using ⊗). 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. Threads in PREPARE additionally check if record version is >1, the current version of database, 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

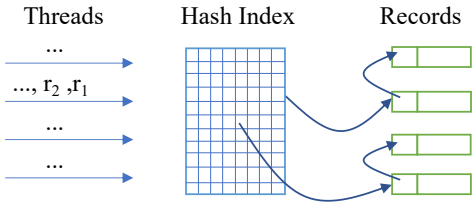


Figure 6: FASTER Overall Architecture

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 of records, $A=5$ 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 $t=6$.

4.3 Correctness

THEOREM 1. *The snapshot of the database obtained using algorithms 1 and 2 has the following properties:*

(a) It is transactionally consistent.
(b) For every thread T, it reflects all transactions committed before t_T, and none after.
(c) It is conflict-equivalent to a point-in-time snapshot.

We prove the above theorem in Appendix A.

4.4 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, according to the definition of CPR-consistency.

5 BACKGROUND ON FASTER

We present an overview of FASTER [6], our recent open-source concurrent latch-free hash key-value store that combines in-place updates with larger-than-memory data handling capabilities, by efficiently caching the hot working set in memory. It supports reads, blind upserts, and read-modify-write (RMW) operations. In the FASTER paper, we report a scalable in-memory throughput of more than 150M ops/sec, making it a good candidate to apply CPR-based durability.

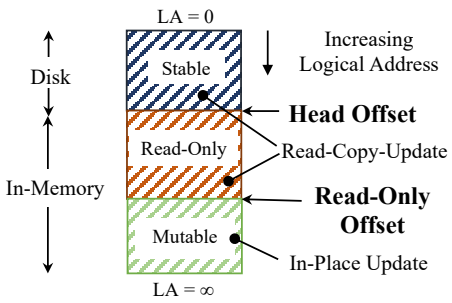


Figure 7: `HybridLog` Organization in FASTER

FASTER has two main components (Fig. 6): a *hash index* and a *log-structured record store* called `HybridLog`. The index is a map from the key hash to an address in a logical address space. Keys with the same hash share a single 64-bit slot in the hash index. All reads and updates to the slots are atomic and latch-free. The `HybridLog` record store defines a *logical address space* that spans main memory and secondary storage. The tail portion of the logical address space is present in memory. Each record in `HybridLog` 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 (see Fig. 6): each record's metadata contains the logical address of the previous record mapped to that slot. The hash index points to the tail record of this linked list.

5.1 Hybrid Log

FASTER stores records in `HybridLog` (Fig. 7), a log-structured record store that spans main memory and disk. The logical address space 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 on records in the hot mutable region of the log. If a record is in the immutable region, a new mutable copy of the record is created at the end of tail to update it. This organization captures the hot working set for in-place updates in the mutable region. This key design choice makes FASTER capable of reaching high throughput by avoiding an atomic increment of the tail offset, a read-copy-update of the record, and an update of the index entry for records in the hot mutable region – creating a WAL entry in this critical path would bring down performance.

The head and read-only offsets are maintained at a constant lag from the tail offset. As the log grows, the head and read-only offsets shift. FASTER threads use epochs to lazily synchronize offset values, so one thread may read a stale value of the read-only offset and update a record in-place,

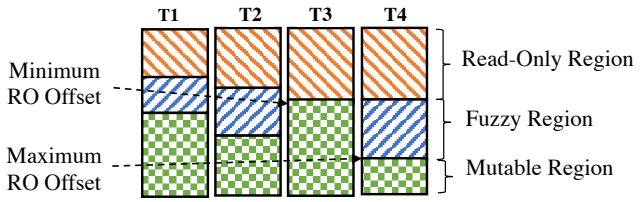


Figure 8: Thread-Local View of `HybridLog` Regions

while a second thread sees the record to be immutable and copies it over to the tail: this leads to a *lost update anomaly*, where the update by the first thread is lost. To avoid this anomaly, FASTER maintains an additional marker called the *safe read-only offset*, which tracks the largest read-only offset seen by all threads (using the epoch framework). A FASTER thread copies a record to the tail only if its logical address is less than the safe read-only offset. If the record falls in between the read-only and the safe read-only offset (called the *fuzzy region*; see Fig. 8), the request is added to a pending list to be processed later.

The update scheme is summarized next (we focus on RMW for brevity). To process an RMW request for key k, FASTER first obtains the logical address l from the hash index. If l is $\perp$ (invalid), it creates a new record by allocating required space at the tail of the hybrid log (say l') and update the slot corresponding to k atomically using a *compare-and-swap* operation to point to l'. If l is less than the head offset, it issues an asynchronous I/O request. If l is more than the head offset, the record is present in memory. FASTER updates the record in-place when it is mutable, i.e., when l is greater than the read-only offset. If l is less than the safe read-only offset, it creates a new updated copy at the tail and updates the hash index atomically. If l is in the fuzzy region (between the read-only and safe read-only offsets), the operation goes pending as described earlier.

5.2 Towards Adding Durability

By default, the in-memory portion of `HybridLog` is lost on failure. We added the ability to 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 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 with FASTER, to be notified of new CPR points whenever FASTER commits.

6 ADDING DURABILITY TO FASTER

The unflushed in-memory portion of HybridLog and the hash index are lost on failure. For durability, we augment FASTER with CPR-based group commit, by periodically persisting a CPR-consistent checkpoint on disk. On failure, FASTER recovers from the most recent commit. Because of the log-structured nature of HybridLog, checkpoints can be incremental. This enables CPR to be a frequent group-commit mechanism, with fast in-place updates to records in the log between commits.

For ease of exposition, assume a one-to-one mapping between a FASTER thread and a user-session. Each user request (e.g. RMW, Read, Upsert) is associated with a strictly increasing session-local serial number. For FASTER with N threads, we wish to create a CPR-consistent checkpoint with commit points $s_0, s_1, \ldots, s_N$ such that for every thread T_i, the commit makes only and all requests before serial number s_i durable.

6.1 Challenges

There are two main challenges in adding CPR to FASTER:

- FASTER provides threads unrestricted access to records in the mutable region of HybridLog, letting user code control concurrency. Handled naïvely, this could lead to a lost-update anomaly (see Sec. 5). Since CPR enforces a strict *only and all* policy, it is challenging to obtain a CPR-consistent checkpoint without compromising on fast concurrent memory access.
- 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. This model improves performance, but complicates CPR in a fundamental way since some requests before a CPR point may be pending. CPR consistency enforces that, semantically, 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 leads to quiescing when handled naively.

6.2 HybridLog Checkpoint

We augmented the 64-bit per-record header in HybridLog (used previously to store the 48-bit previous address and status bits such as invalid and tombstone; see [6] for details) to include a 13-bit 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. 9a) 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. 9b. 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 requests of version v.
- WAIT-FLUSH: All unflushed v records are written to disk asynchronously.
- REST: Normal processing on FASTER version $(v + 1)$.

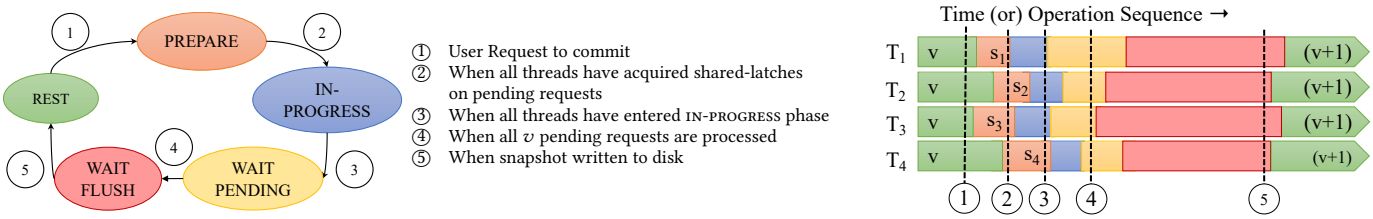
A CPR commit request (from user or triggered periodically) first records the current tail offset of HybridLog, say L_s^h, and updates the global state from REST to PREPARE. Threads enter PREPARE during their subsequent epoch refresh.

6.2.1 PREPARE. A thread in PREPARE 'prepares' to handle a shift in version. Recall that every FASTER thread has a thread-local list of pending requests since some of them are processed asynchronously. When a thread enters PREPARE, it acquires a *shared-latch* on the key's bucket (we may instead use a per-record latch as well) for each pending request. These latches are released only after the request is completed.

A PREPARE thread T processes an incoming user-request as follows (see Alg. 4 of Appendix B): It first acquires a shared-latch on the key's bucket. When the record is in memory with version $\leq v$, T processes the request based on which HybridLog region the record belongs to — modifies it in-place when in mutable region, performs a copy-on-write when in safe read-only region, and adds it to a thread-local pending list to retry later when in fuzzy region. When not in memory, it adds the request to its pending list and issues an asynchronous I/O request to retrieve it from disk. A pending request holds on to the shared-latch until it is processed later, while it is released immediately in all other cases.

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.

6.2.2 IN-PROGRESS. After all threads enter the PREPARE phase and acquire shared latches for its pending requests, the epoch framework (using the BumpEpoch mechanism from Sec. 3) advances the state machine to IN-PROGRESS. A thread transits from PREPARE to IN-PROGRESS during a subsequent epoch



(a) Global State Machine with Transition Conditions

(b) FASTER Threads during checkpoint

Figure 9: Overview of CPR for FASTER

refresh, thus demarcating its CPR point. It now processes incoming requests as belonging to version $(v+1)$. All user requests received before this point by the thread are part of the commit, and none after. The pseudo-code for IN-PROGRESS (and later) phases are shown in Alg. 5 of Appendix B. The key idea here is not to let the thread modify any record of version $\leq v$ in-place and instead, force a read-copy-update without violating CPR-consistency.

When the record is in memory with version $\leq v$, an IN-PROGRESS thread acquires an exclusive latch on the key's bucket, performs a copy-on-update creating an updated $(v+1)$ record at the tail and releases the latch. Exclusive-latch acquisition succeeds only when no other request holds a shared-latch. When it fails, the request is added to a thread-local pending list corresponding to version $(v+1)$.

If record version is $(v+1)$, the thread modifies it in-place when in mutable region, performs a copy-on-write when in safe read-only region, and adds to thread-local pending list for version $(v+1)$ when in fuzzy region. When the record is not in memory, it issues an asynchronous I/O request and adds the user-request to $(v+1)$ pending list.

6.2.3 WAIT-PENDING. When all threads enter IN-PROGRESS, FASTER enters WAIT-PENDING. CPR-consistency requires that *all* requests accepted before the CPR points be recoverable on failure. FASTER stays in the WAIT-PENDING phase until all pending requests - if any - that are part of the commit are completed. These requests hold a shared-latch on the key's bucket, which is released as they are completed.

When the system is in the WAIT-PENDING phase, all FASTER threads are either in IN-PROGRESS or WAIT-PENDING. A WAIT-PENDING thread processes incoming user-requests similar to an IN-PROGRESS thread. Since there are no PREPARE threads in the system, it performs a copy-on-update for record version $\leq v$ when the shared latch count is zero, without acquiring any exclusive-latch. When the count is non-zero, there is a pending v request belonging to the bucket. Since processing the $(v+1)$ request now may violate CPR consistency, it is added to the thread-local pending list to be processed later.

6.2.4 WAIT-FLUSH. Once all v requests have been completed, we record the tail offset of HybridLog, say L^h_e, and shift the read-only offset to L^h_e, which asynchronously flushes HybridLog until L^h_e to disk. All v records on the HybridLog occur before L^h_e. There may also be some $(v+1)$ records in this section, but they are invalidated during recovery (see Sec. 6.4). We add a trigger action to the epoch framework that advances the global state machine to WAIT-FLUSH. Incoming requests in WAIT-FLUSH are processed identical to the REST phase since all v records are in the safe read-only region and hence immutable. Once the asynchronous write to disk is complete, system moves back to REST with version $(v+1)$. This concludes the HybridLog checkpoint for CPR commit.

6.3 FASTER Hash 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 index can be checkpointed independently from 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^i_s) and after completion (L^i_e) of the fuzzy checkpoint. We use these offsets during recovery, which is described next.

6.4 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 $S = \min(L^i_s, L^h_s)$ to $E = \max(L^i_e, L^h_e)$, 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.

```
S = min(L_s^i, L_s^h); E = min(L_e^i, L_e^h);
foreach record R at L_R between S and E do
    if R.version ≤ v then
        update slot address to L_R;
    else
        R.invalid = true;
        addr = address in slot of R.key;
        if addr ≥ L_R then
            update slot address to R.previousAddress;
```

Algorithm 3: Pseudo-code for Recovery

The pseudo-code for recovery is shown in Alg. 3. 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 (see Fig.6) 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 – each slot points to the latest record with version $\leq v$ in its linked-list.

6.5 Solution Variants

Implementing CPR on FASTER presents a range of design choices. We discuss two major ones here. A critical transition in CPR is from the PREPARE to IN-PROGRESS phase: when a thread demarcates its CPR point and starts processing incoming requests as version $(v + 1)$. The algorithm in Sec. 6.2 uses fine-grained bucket-level latches to enable this. An alternate coarse-grained approach is to use HybridLog offsets for this purpose, which is explained in Appendix C.

Another design consideration is how we capture the volatile v-records on storage. The solution in Sec. 6.2, called a *fold-over* commit, advances the read-only offset of HybridLog to its tail, to offload these records to disk. While this provides incremental checkpoints, it forces a copy-on-update for every record after a commit, resulting in a slower recovery of performance. We could instead obtain a snapshot of entire volatile HybridLog in a separate file, called *snapshot* commit, without advancing the read-only offset: the section of HybridLog remains immutable only during the commit, and can be updated in-place afterwards. This design is described in Appendix D.

These two choices impact the design of FASTER in significant ways. We compare them empirically in Sec. 7.3.

7 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 [26] and WAL [25]. Next, we evaluate CPR on FASTER, our high-performance, hash-based key-value store for larger-than-memory data.

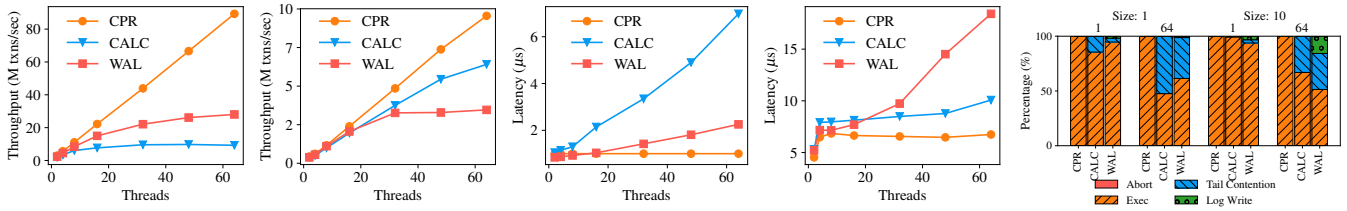
7.1 Implementation, Setup, Workloads

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 [26]. 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 [6] as the data store and implement two-phase locking with NO-WAIT deadlock avoidance policy.

We added CPR to the C# and C++ versions of FASTER, and use the C# version for our evaluation. Threads first load the key-value store with data, and then issue a sequence of operations. Commit requests are issued periodically. We measure and report system throughput and latency every 2 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, the FASTER hash index has #keys/2 hash-bucket entries. Prior work [6] has shown that other persistent key-value stores such as RocksDB achieve an order of magnitude lower performance (less than 1M ops/sec) even when WAL is disabled. Therefore, we omit comparisons to these systems in this paper.

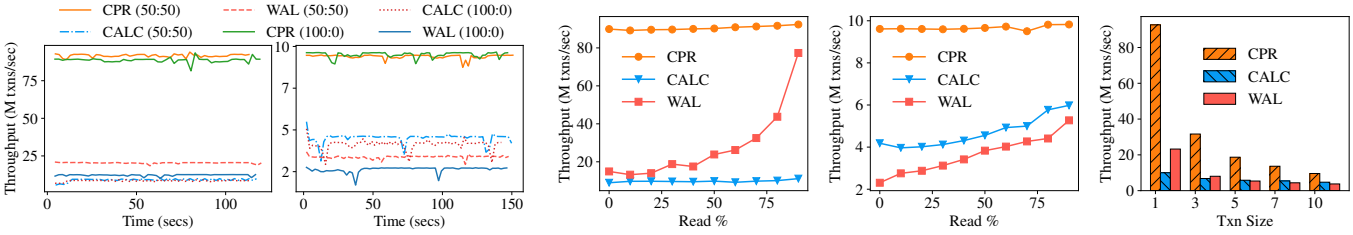
Setup. The transactional database experiments are conducted on a Standard D64s v3 machine [2] on Microsoft Azure. The machine has 2 sockets and 16 cores (32 hyperthreads) per socket, 256GB memory and runs Windows Server 2018. Experiments on CPR with FASTER are carried out on a local Dell PowerEdge R730 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 pre-load input datasets into memory for all experiments.

Workloads. For our stand-alone database, we use a mix of transactions based on the Yahoo! Cloud Serving Benchmark



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

Figure 10: Scalability and Latency on Low Contention (θ= 0.1) YCSB workload



(a) Throughput; Size:1 (b) Throughput; Size:10 (c) Throughput; Size:1 (d) Throughput; Size:10 (e) Throughput; Mix: 50:50

Figure 11: Throughput during Checkpoint and Performance on Different Transaction Mixes

(YCSB) [7]. 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 (θ = 0.1) workload here since it incurs the most performance penalty due to logging or tail contention. Additional experiments on high contention YCSB and TPC-C [28] benchmarks are in Appendix E.

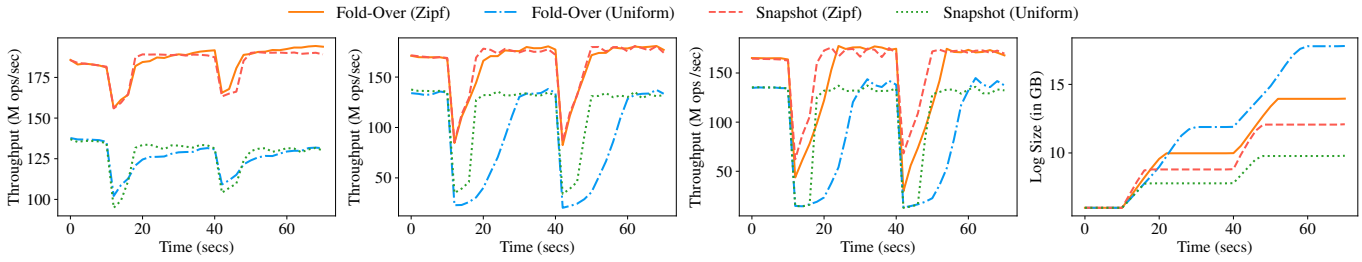
For FASTER with CPR, we use an extended version of the YCSB-A workload, with 250 million distinct 8 byte keys, and value sizes of 8 bytes and 100 bytes. After pre-loading, records take up 6GB of `HybridLog` space and the index takes up 8GB of space. Workloads are described as R:BU for the fraction of 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 running per-key "sum" operation. We use the standard Uniform and Zipfian (θ = 0.99) distributions in our workloads.

7.2 Transactional Database Comparisons

We first plot average throughput (Figs. 10a, 10b) and latency (Figs. 10c, 10d) 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. 10e. "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.

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

7.2.2 Latency. 1-key transactions (Fig 10c) 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 [6]. 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 10c) incur a cost of 7μs, which is 10x that of a 1-key transaction. CALC latency, even though higher than CPR due to tail contention in the atomic commit



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

Figure 12: FASTER Throughput and Log Growth vs. Time; Full Fold-over and Snapshot Commits at 10 and 40 secs

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.

7.2.3 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. 11a and Fig. 11b 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. There is a minor difference between write-only and mixed workloads for all three systems, since writes are more expensive than reads.

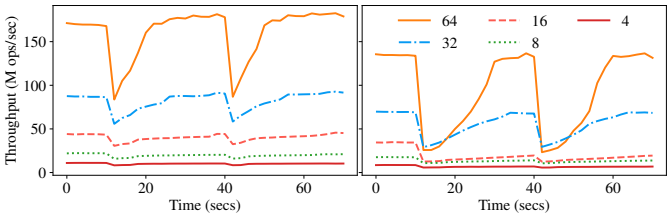
7.2.4 Varying transaction mix. We now investigate the impact of read-write ratio on performance by varying the read percent from 0% to 90% on 64 threads. We plot results for both 1- and 10-key transactions in Fig. 11c and Fig. 11d respectively. In both cases, the read-write ratio does not impact the performance of CPR greatly; it increases marginally as reads are cheaper than writes. In a 1-key transaction, the read-write ratio directly affects the contention on tail of WAL: performance of WAL improves with increasing reads due to fewer log records generated. CALC appends every transaction to an atomic commit log resulting in a serial bottleneck and that outweighs any gains due to more reads. In case of 10-key transactions, contention at log reduces as throughput of entire system is much smaller. Tail contention, even though still a bottleneck, is not the most significant one, and so cheaper reads result in performance improvement for both WAL and CALC. Performance gain in WAL is higher with increasing read percent since it eliminates two writes: an update to the record value and during WAL record generation.

7.2.5 Varying transaction size. The cost of executing a transaction is directly proportional to the amount of computation and number of read/write accesses. We present average throughput for transactions of size 1, 3, 5, 7 and 10 for mixed (50 : 50) workload on 64 threads in Fig. 11e. CPR throughput, which is an order of magnitude higher than the other two, drops linearly as there is no other bottleneck in the system. Since CALC throughput is already restricted at 10M txns/sec due to a significant overhead (tail contention), there is not much reduction in its performance with increasing size. WAL, on the other hand, writes a lot more data in the WAL record as we increase transaction size. WAL outperforms CALC for smaller transactions, and as size increases the trend reverses due to log write overhead.

7.3 Evaluation of FASTER with CPR

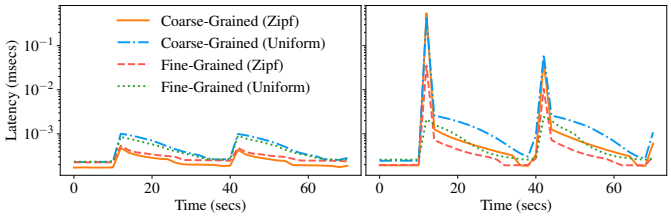
7.3.1 Throughput and Log Size. We plot throughput vs. wall-clock time during the lifetime of a FASTER run. We perform two "full" (index and log) commits during the run, at the 10 sec and 40 sec mark respectively, and plot results for two key distributions (Uniform and Zipf). We evaluate both our commit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

Fig. 12a 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. 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. 12b, 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



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

Figure 13: Throughput vs. Time; Varying #Threads



(a) 0:100 blind updates (b) 0:100 RMW updates

Figure 14: Latency vs. Time; Log-only Fold-over

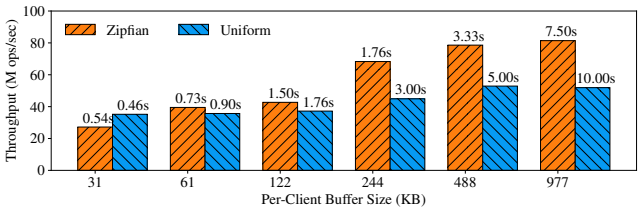


Figure 15: End-to-end Experiment; YCSB 50:50

to dump the unflushed log to a snapshot file and quickly reopen `HybridLog` for in-place updates. A 0:100 workload with only blind updates demonstrates similar effects, as shown in Fig. 12c. 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. 12d depicts the size of `HybridLog` vs. time, for a 0:100 workload. We note that (1) `HybridLog` size grows much more 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 Appendix E; briefly, we found CPR commits to have much lower overhead as expected, with a similar trend overall.

7.3.2 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. 13a and 13b 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.

7.3.3 Operation Latency; Fine- vs. Coarse-Grained. Fig. 14a shows average latency (in msecs) vs. time, for a 0:100 blind update workload using the fold-over strategy, log-only checkpoints, for Uniform and Zipf distributions. We depict latency for the two version transfer schemes introduced in this paper: coarse-grained and fine-grained. Latency during the REST phase is around 100-300ns as the working set is entirely in memory. During CPR commit, latency increases to 1μs for Zipf and around 0.5μs for Uniform because of slightly more contention on index entries for hot keys in Zipf. Since there is no data-dependency between v and $(v+1)$ version for blind updates, contended operations do not go pending in this experiment.

We finally repeat this experiment with a 0:100 RMW workload. With RMW, due to the data dependency between v and $(v+1)$ versions of a record, we expect the hand-off to have a greater effect as $(v+1)$ operations may have to wait for records being updated by v operations (using a lock in fine-grained or going pending in coarse-grained). This is validated in Fig. 14b, where the latency spikes during CPR are more pronounced than before. It is clear that coarse-grained introduces significantly more latency than fine-grained, because requests go pending. Further, the latency spike is higher for Zipf than Uniform because of greater contention when handing off records for the hot keys. Overall, latency degrades slightly as expected, but with fine-grained, it stays below 2.2μs even during CPR.

7.3.4 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 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. 15 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, and hence 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.

8 RELATED WORK

This paper falls in the realm of "Durability and Recovery", which are well-studied problems in the database community. Our work is inspired by prior work in this area.

Write-Ahead Logging. The write-ahead logging (WAL) protocol originally defined in [1] has been the standard for providing durability in databases. Nearly all DBMSs use WAL based on ARIES [25], which integrates concurrency control with transaction rollback and recovery. Database archive checkpointing [3] techniques often use epochs, similar to CPR, but depend on a WAL to bring the database to the committed state. Shadow paging [22] maintains a dual mapping between pages and their "shadow" on disk, which speeds up checkpointing and provides page integrity, but requires a WAL to restore transactional consistency. Many [14, 20] have proposed adopting fuzzy checkpointing to main memory databases, which requires a WAL. WAL has been reported to be a significant overhead [15] (roughly 12% of total time in a typical OLTP workload) even in a single-threaded database engine. Johnson et. al. [16] identify I/O related delays, log-induced lock contention, log buffer contention, and excessive context switches to be key factors resulting in overhead. To reduce log contention at the tail, Aether [16] consolidates log allocation. Jung et. al. [17] address the same problem by creating a scalable data structure for log buffer allocation. I/O has become less of a bottleneck due to modern hardware. Some recent [11, 29] studies demonstrate speedups due to better response times and better handling of small I/Os. However, even the fastest flash drives do not eliminate all overhead. Zhen et al. present SiloR [31] that avoids centralized logging bottleneck by letting each worker thread copy transaction-local redo logs to per-thread log buffers after validation. In spite of such attempts, recording every update remains expensive at scale.

Distributed Logging. While traditional log implementations enforce a particular serial order of log entries using LSNs, distributed logging exploits the fact that there is more than one correct order for a given serializable order. Lomet et. al. [21] propose a redo logging method where individual nodes can maintain a private log and upon a crash failure recover from its own private log; in case of a media failure, they merge other existing logs in any order that results in the same serializable order. This method is found to be infeasible in a multi-socket scenario as the protocol requires dirty page writes during migrations. Johnson et. al. [16] propose a distributed logging protocol based on Lamport Clocks instead of traditional LSNs, which is then used to derive an appropriate serial order during recovery. While this alleviates log contention due to LSNs, it does not address the problem of expensive log writes, which is also a significant overhead in update-intensive scenarios. CPR checkpoints are transactionally consistent and do not need any WAL, thus allowing fast in-place updates in memory.

Transactionally-Consistent Checkpoints. Another approach to provide durability is to obtain point-in-time snapshots that are transactionally-consistent. Applications such as multiplayer games naturally reach points during normal execution when no new transactions are being executed. Cao et. al. [4] propose two different checkpoint algorithms, zig-zag and interleaved ping-pong, that capture entire snapshots of a database without key locking using multiple versions. Similarly, VoltDB [23] uses an asynchronous checkpointing technique which takes checkpoints by making every database record "copy-on-write". This approach is shown to be expensive [6] in update-intensive scenarios. CALC [26] uses an atomic commit log and limited multi-versioning to create a virtual point of consistency, which is then captured asynchronously while transactions are being processed simultaneously. As discussed earlier, these techniques suffer from a serial bottleneck that limits scalability.

9 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 design solutions to make two systems durable using CPR, a custom in-memory database and FASTER, our larger-than-memory hash-based key-value store. A detailed evaluation of both systems shows that we can support highly concurrent and scalable performance, while providing CPR-based durability.

Acknowledgments. We would like to thank Phil Bernstein, James Hunter, and the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1 (June 1976), 97–137. Issue 2.

[2] Microsoft Azure. 2018. Azure Windows VM Sizes. https://aka.ms/AA1t6ge. [Online; accessed 09-July-2018].

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

[4] Tuan Cao, Marcos Antonio Vaz Salles, Benjamin Sowell, Yao Yue, Alan J. Demers, Johannes Gehrke, and Walker M. White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011.* 265–276.

[5] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401–412.

[6] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18).* ACM, New York, NY, USA, 275–290.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10).* ACM, New York, NY, USA, 143–154.

[8] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84).* ACM, New York, NY, USA, 1–8.

[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13).* ACM, New York, NY, USA, 1243–1254.

[10] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Foundations and Trends in Databases* 8, 1-2 (2017), 1–130.

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

[12] Apache Software Foundation. 2017. Apache Kafka. https://kafka.apache.org/. [Online; accessed 30-Oct-2017].

[13] Dieter Gawlick and David Kinkade. 1985. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull.* 8 (1985), 3–10.

[14] R. B. Hagmann. 1986. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Trans. Comput.* C-35, 9 (Sept 1986), 839–843.

[15] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08).* ACM, New York, NY, USA, 981–992.

[16] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB J.* 21, 2 (2012), 239–263.

[17] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable Database Logging for Multicores. *PVLDB* 11, 2 (2017), 135–148.

[18] Alfons Kemper, Thomas Neumann, Jan Finis, Florian Funke, Viktor Leis, Henrik Mühe, Tobias Mühlbauer, and Wolf Rödiger. 2013. Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *IEEE Data Eng. Bull.* 36, 2 (2013), 41–47.

[19] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382.

[20] Jun-Lin Lin and Margaret H. Dunham. 1996. Segmented Fuzzy Checkpointing for Main Memory Databases. In *Proceedings of the 1996 ACM Symposium on Applied Computing (SAC '96).* ACM, New York, NY, USA, 158–165.

[21] David B. Lomet and Mark R. Tuttle. 1995. Redo Recovery After System Crashes. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 457–468.

[22] Raymond A. Lorie. 1977. Physical Integrity in a Large Segmented Database. *ACM Trans. Database Syst.* 2, 1 (March 1977), 91–104.

[23] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014.* 604–615.

[24] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12).* ACM, New York, NY, USA, 183–196.

[25] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162.

[26] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. 2016. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16).* ACM, New York, NY, USA, 1539–1551.

[27] Facebook Open Source. 2017. RocksDB. http://rocksdb.org/. [Online; accessed 30-Oct-2017].

[28] Transaction Processing Performance Council. 2010. http://www.tpc.org/tpcc/. [Online; accessed 31-Oct-2018].

[29] Jason Tsong-Li Wang (Ed.). 2008. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008.* ACM.

[30] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220.

[31] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14).* USENIX Association, Berkeley, CA, USA, 465–477.

```
if bucket.TrySharedLatch() then
    if L ≥ HeadOffset then
        if record.version ≤ v then
            if L ≥ ReadOnlyOffset then
                update record in-place concurrently;
                bucket.ReleaseLatch();
                return OK;
            else if L ≥ SafeReadOnlyOffset then
                add request to v pending list;
                return PENDING;
            else
                create updated v record at tail;
                bucket.ReleaseLatch();
                return OK;
    else
        issue async IO request;
        add request to v pending list;
        return PENDING;
    bucket.ReleaseLatch();
return CPR_SHIFT_DETECTED;
```

Algorithm 4: Pseudo-code for PREPARE in FASTER

```
if L ≥ HeadOffset then
    if record.version ≤ v then
        switch thread.Phase do
            case IN-PROGRESS do
                if bucket.TryExclusiveLatch() then
                    create updated (v + 1) record at tail;
                    ReleaseLatch();
                    return OK;
            case WAIT-PENDING do
                if bucket.SharedLatchCount == 0 then
                    create updated (v + 1) record at tail;
                    return OK;
            case WAIT-FLUSH do
                create updated (v + 1) record at tail;
                return OK;
        add to (v + 1) pending list;
        return PENDING;
    else
        if L ≥ ReadOnlyOffset then
            update record in-place concurrently;
            return OK;
        else if L ≥ SafeReadOnlyOffset then
            add to (v + 1) pending list;
            return PENDING;
        else
            create updated (v + 1) record at tail;
            return OK;
else
    issue async IO request;
    add to (v + 1) pending list;
    return PENDING;
```

Algorithm 5: Pseudo-code for Other FASTER Phases

A PROOF OF THEOREM 1

THEOREM. *The snapshot of the database obtained using algorithms 1 and 2 has the following properties:*

(a) It is transactionally consistent.
(b) For every thread T, it reflects all transactions committed before t_T, and none after.
(c) It is conflict-equivalent to a point-in-time snapshot.

First, the snapshot obtained is transactionally consistent because it ensures that the current version of *all* records in the read-write set of a transaction are either v (in PREPARE) or $(v + 1)$ (in IN-PROGRESS or later) when executed. So, a transaction can either belong to the commit or not.

All transactions executed in PREPARE belong to version v, while those executed in IN-PROGRESS or later belong to $(v + 1)$. Since, for every thread T, there is a unique point in time t_T when it shifts from PREPARE to IN-PROGRESS, which also marks the set of transactions for T that are included, the snapshot is CPR-consistent.

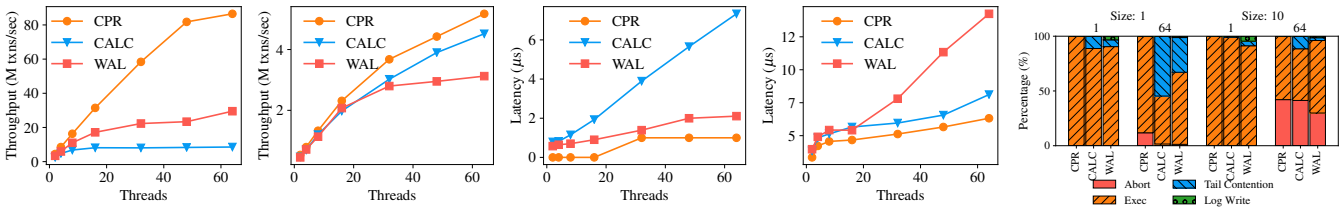
Consider two transactions, T_1 and T_2, such that T_1 belongs to the commit and T_2 does not. If both T_1 and T_2 belong to the same thread, $T_1 \prec T_2$ in the serial order. If T_1 and T_2 are executed on different threads, then there are two possibilities depending on their read-write sets: when they do not intersect, $T_1 \prec T_2$ is a valid serial order as is $T_2 \prec T_1$, since they are conflict-equivalent; if they intersect on a set of records R, T_1 must have executed when version of each record r in R is v; similarly T_2 must have executed when the record versions are $(v + 1)$. We know that r's version shifts from v to $(v + 1)$, and hence $T_1 \prec T_2$. Thus, the snapshot obtained is conflict-equivalent to a point-in-time snapshot where all v and none of $(v + 1)$ transactions have executed.

B PSEUDO-CODE FOR CPR ON FASTER

Alg. 4 depicts the pseudo-code for executing incoming requests during PREPARE phase on FASTER, while all other phases are handled in Alg. 5.

C COARSE VS. FINE-GRAINED SHIFT

Threads demarcate their CPR points when they shift PREPARE to IN-PROGRESS and start processing incoming requests as version $(v + 1)$. An IN-PROGRESS thread performs a copy-on-write of a v-record only when it is in the immutable region, or by acquiring an exclusive-latch on the bucket if in the mutable region. Alg. 4 uses a fine-grained bucket-level latch to ensure CPR-consistency of updates. PREPARE threads acquire a shared-latch even for in-place updates to records in the mutable region, which may be expensive.



(a) Scalability; Size:1 (b) Scalability; Size:10 (c) Latency; Size:1 (d) Latency; Size:10 (e) Analysis; Mix: 50:50

Figure 16: Scalability and Latency on High Contention ($\theta = 0.99$) YCSB Workload

An alternate approach is to acquire a shared-latch only when a request is added to pending list. However, this may cause an inconsistency since IN-PROGRESS threads might copy over a v record when a PREPARE thread is updating it in place, similar to the lost-update anomaly in Sec. 5. To prevent this, an IN-PROGRESS thread performs a copy-on-write only when the v record is in safe read-only region. The safe read-only offset, in this context, is used as a coarse-grained marker for records that are eligible for a version shift. For v records that are in the mutable region, the request is added to the thread-local pending list. This approach trades-off making a request pending in IN-PROGRESS (which incurs higher operation latency) against the overhead of bucket-level latching for in-place updates in PREPARE.

D FOLD-OVER VS. SNAPSHOT

Another design choice deals with how we capture the v records. Since the organization of data in FASTER is log-structured, we can fold-over the log by shifting the read-only offset to tail during CPR commit and by design all records in that portion of `HybridLog` are written to disk. We call this the *fold-over commit* and is described earlier in Sec. 6. However, with this approach, even after the commit, a request upon encountering a v record does a copy-on-write, since it is immutable. Moreover, folding-over `HybridLog` unnecessarily forces some records to the disk further delaying performance recovery. The key advantage of fold-over is the amount of data written to disk during the commit — it is fully incremental, i.e., only data that changed since the previous commit is written out.

An alternate design is to capture all volatile v records into a separate snapshot file on disk *without* modifying the `HybridLog` offsets. Once captured, we can then move to the REST phase, where a v record in mutable region can be updated in-place, avoiding copy-on-write for such records. In effect, the period during which the system performs copy-on-write heavily is limited to the time it takes to write the volatile portion of `HybridLog` to disk. The downside of this technique is that every commit writes all records in the volatile region to secondary storage, which may include records that were not updated since the previous commit. This can be wasteful if CPR commits are frequent.

E ADDITIONAL EXPERIMENTS

E.1 High Contention YCSB Benchmark

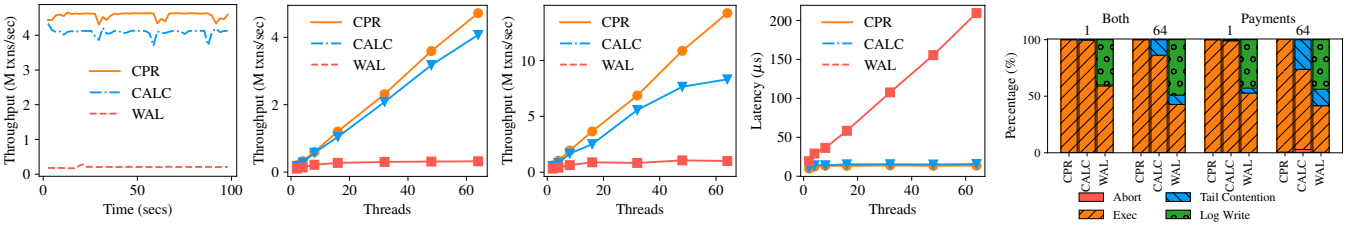
We evaluate CPR, CALC and WAL on a YCSB high-contention ($\theta = 0.99$) read-write (50 : 50) workload; we report average throughput (Figs. 16a, 16b) and latency (Figs. 16c, 16d) as we vary number of threads for 1- and 10- key transactions along with a breakdown analysis (Fig. 16e).

The high contention 1-key case is similar to its low contention counterpart: CPR yields a throughput of 95M txns/sec, an order of magnitude better than CALC and WAL, with a maximum of 10M txns/sec and 30M txns/sec respectively. The marginal improvement in throughput relative to the low contention scenario (refer Sec. 7.2) is due to better caching of hot records since unique, frequently-accessed records are fewer. Tail contention serves as a significant bottleneck in CALC and WAL, while CPR scales linearly; this is evident from the linearly increasing trend for average latency in CALC and 'Tail Contention' overhead in the breakdown. WAL performs better since 50% read-only transactions do not generate a WAL record.

For 10-key high contention workload, on the other hand, aborts due to conflicting accesses are a significant overhead. CPR, while still better (by 15%), performs almost similar to CALC as the overheads are elsewhere. WAL is twice as expensive as CPR or CALC due to the additional log write overhead; WAL's increasing latency trend reveals that this is significant. There is not much difference between low and high-contention for 10-key workloads in WAL as the logging overhead outweighs that of aborts.

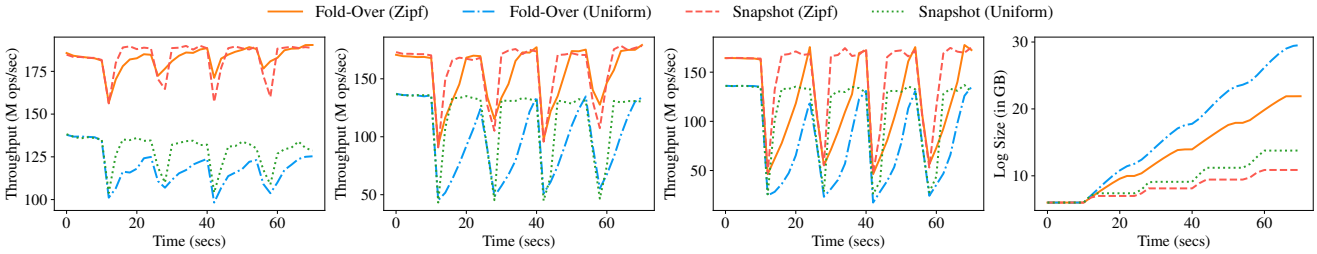
E.2 TPC-C Benchmark

We also compared the three systems on two workloads derived from the TPC-C [28] benchmark: a mixed (50 : 50) and payments-only (100 : 0) mixture of Payment and New-Order transactions. Transaction inputs were generated as per the standard specifications. Payment is a short transaction writing to 3 records, while New-Order is longer accessing 23 records on average. We use 256 warehouses in our experiments to reduce the impact of contention (covered in Sec. E.1). We plot throughput on 64 threads during the lifetime of a



(a) Throughput; 50:50 (b) Scalability; 50:50 (c) Scalability; 100:0 (d) Latency; 50:50 (e) Analysis

Figure 17: Throughput during Checkpoint, Scalability and Latency on TPC-C Benchmark



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

Figure 18: FASTER Throughput and Log Growth vs. Time; Log-only Fold-over and Snapshot Commits

run for the mixed workload with checkpoints at 30, 60 and 90 secs in Fig. 17a. We observe a marginal drop in performance for CPR and CALC due to the extra overhead of copying over the live to stable value during checkpoint; the value sizes are considerably larger compared to 8 bytes in YCSB. We now present the average throughput (Figs. 17b and 17c) and latency (Fig. 17d: note the log-scale) as we vary number of threads along with a breakdown analysis (Fig. 17e). CPR is better than CALC while the difference is smaller since cost of transaction execution compared to tail contention is higher. Latency of CPR and CALC remains constant further validating this. Writing WAL records is an important overhead in WAL since value sizes are larger, leading to much poorer performance. For the payment-only workload, CPR outperforms CALC by almost 50% since payment transactions are smaller making tail contention a significant bottleneck again.

E.3 Frequent CPR Commits

The FASTER hash index can be recovered independently from `HybridLog` and hence frequent commits of FASTER does not require the index checkpoint. We plot throughput vs. wall-clock time as in Sec. 7.3, but take a checkpoint every 15 secs, starting at the 10 sec mark. Further, since the index is typically checkpointed less frequently, we checkpoint only `HybridLog` in these experiments. We plot the results for Uniform and Zipf distributions, for the fold-over and snapshot techniques, as we vary the workload mix.

Fig. 18a shows the result for a read-dominated 90:10 workload. We see that throughput is affected very slightly due to commits, with Zipf recovering the working set quickly. While all checkpoints take the same time with snapshot (having to dump the in-memory snapshot to disk every time), subsequent fold-over commits (being incremental) have little effect on throughput. The performance with Uniform is similar. Fold-over takes longer to recover performance compared to snapshot, since the working set takes longer to migrate to the mutable region. With the 50:50 and 0:100 workloads (Figs. 18b and 18c), the Zipf distribution with fold-over is able to ramp up post-commit performance quicker than Uniform, as it captures the working set quickly. As before, system recovers full performance quickly after the snapshot is written out. We depict the size of `HybridLog` vs. time for frequent log-only checkpoints over a 0:100 workload in Fig. 18d. The log growth is more rapid with fold-over checkpoints and with Uniform distributions, as expected.