Concurrent Prefix Recovery: Performing CPR on a Database

Anonymous Author(s)

ABSTRACT

With increasing multi-core parallelism, modern databases and key-value stores have been designed for scalability, and yield very high throughput for the in-memory working set. These systems typically depend on group commit using write-ahead-logging to provide durability and crash recovery. However, write-ahead logging incurs significant overhead, particularly for update-intensive workloads, where it introduces a concurrency bottleneck (the log), and incurs log creation and flush I/O overhead. 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) we provide users a semantic description of committed operations, of the form "all operations until time T_i on thread i"; and (2) we use asynchronous incremental checkpointing instead of a WAL to implement the group commit in a scalable bottleneck-free manner. CPR provides the same consistency as a point-in-time commit, but allows a concurrent and scalable implementation. We design and implement protocols and solutions to make two systems durable using CPR: (1) a custom in-memory transactional database; and (2) a state-of-the-art scalable larger-than-memory hash-based key-value store. A detailed evaluation of these modified systems shows that CPR is capable of supporting highly concurrent and scalable performance, reaching hundreds of millions of operations per second on a multi-core machine.

1 INTRODUCTION

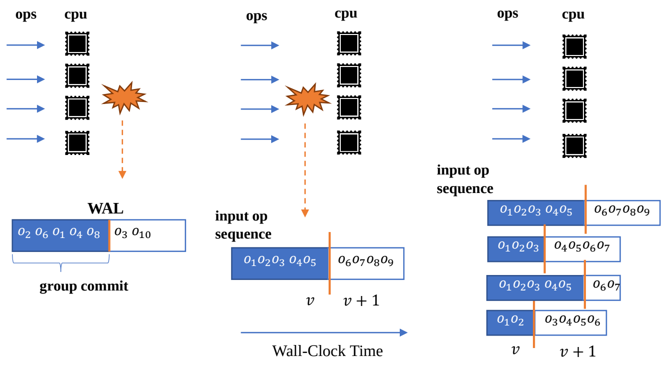
Over the last decade, we have seen huge interest in building extremely scalable high performance multi-threaded data processing systems – both databases and key-value stores. For instance, main memory databases [9, 11, 19] exploit multiple cores (up to thousands of cores in some cases [31]), as well as NUMA, SIMD, HTM, and other hardware advances to drive performance to levels that are orders-of-magnitude higher than those achieved by traditional databases. Key-value stores have pushed performance even further: for instance, Masstree [24] achieves very high in-memory throughput – up to 30M ops/sec on one machine – compared to traditional range indices. A recently proposed system, FASTER [6], reports up to 160M ops/sec on one machine for point updates and lookups, while supporting larger-than-memory data and caching the hot working set in main memory.

Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 New Bottleneck and Today's Solutions

Applications using such systems usually require some form of durability for the changes made to application state. Modern systems are able to handle extremely high update rates in memory, but struggle to retain their high performance when durability is desired. There are two broad approaches that today's systems follow for durability:

- *WAL with Group Commit*: The traditional approach to achieving durability in databases is to use a *write-ahead log* (*WAL*), where every change to the database is firmed into a log update. Techniques such as group commit [8, 13] make it easier to write logs to disk in coarse-grained chunks, but update-intensive applications stress disk write bandwidth. Even without the I/O bottleneck, WAL introduces overheads – one study indicated that 30% of CPU cycles are devoted to generating log records [15] due to issues such as log-induced lock contention, excessive context switching, and log buffer contention.
- *Checkpoint-Replay*: An alternate to WAL, popular in streaming databases, is to take periodic consistent point-in-time checkpoints of the database, and use it with input replay for recovery. The checkpoint captures all changes between commit points. If we had a WAL, taking an asynchronous checkpoint would be trivial: we could take a *fuzzy checkpoint* and use the WAL to recreate a consistent snapshot. As described earlier, this approach kills performance due to the bottleneck of the log. If we could quiesce the database, we could avoid the log bottleneck, but would lose asynchronicity and create downtime. A recent proposal, CALC [26], takes asynchronous consistent checkpoints, but depends on an atomic commit log (instead of the WAL) in order to define the consistency point. However, the atomic commit log becomes the new bottleneck, precluding scalable multi-threaded performance.



(a) Write-Ahead Logging (b) Point-in-time Checkpoint (c) Concurrent Prefix Recovery

Figure 1: Approaches to Durability

These alternatives are depicted in Fig. 1(a) and (b) – both WAL and point-in-time checkpoints suffer from scalability issues because of the bottleneck of WAL and commit log respectively. To validate this point, we implemented the Faster key-value store based on details provided in the paper [6], and augmented it 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 main memory. The reason is that creating a copy of data on every update is expensive due to read-copy-update, and stresses contention on the tail of the log. We also built an in-memory transactional database, and found its throughput (see Fig. 2) with both a WAL and point-in-time checkpointing to bottleneck at around 20M single-key txns/sec (Sec. 7 has the details).

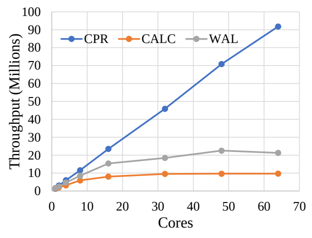


Figure 2: Scalability

This huge performance gap has resulted in many real deployments foregoing durability altogether, e.g., by disabling the WAL in RocksDB [28], or using workarounds such as approximate recovery and quiesce-and-checkpoint [5], which 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 committing individual operations, we convey commit as "all input operations issued up to time T": we call this model *prefix recovery*. Clients may use the prefix recovery information to prune[1] their in-flight operation log until T, and expose commit to users. Based on this durability model, this paper makes the following contributions:

- We show 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, and none after. We show that CPR has the same consistency as prefix recovery, but allows for a scalable asynchronous implementation.
- Traditional group commit is implemented using a WAL. Instead, CPR commits are implemented using *asynchronous checkpoints*. Such checkpoints can capture all changes between two commit points without creating a scalability bottleneck, because they do not require atomic access to a central log that captures every change. However, this solution requires the ability to take incremental checkpoints very quickly. Fortunately, systems such as Faster store operational data in an in-place-updatable record-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 commit points.
- CPR commits all operations before a per-thread point in time, and none after. While it would appear desirable for applications to choose specific CPR points, e.g., at input batch boundaries, we show that if the application were to ask the system 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 provide *some* CPR commit point, and the system co-ordinates the global construction of the CPR commit point for all threads without losing asynchronicity or creating a central bottleneck.
- While CPR makes it theoretically possible to perform group commit without blocking or creating a central bottleneck, it is non-trivial to design systems that achieve these properties without introducing expensive runtime synchronization. To complete the proposal, therefore, we use CPR as the basis for building new scalable non-blocking durability solutions for (1) a custom in-memory transactional database; and (2) a state-of-the-art scalable larger-than-memory hash-based key-value store (Faster). We use an extended version of epoch protection [20] as our building block for loose synchronization, and introduce

[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.

new state-machine based protocols to perform CPR commit. As a result, our simple main-memory database implementation scales linearly (see Fig. 2) up to 90 million operations per second – 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, 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 overheads during normal runtime, and low overheads during commit (in terms of throughput and latency).

The rest of the paper is organized as follows. Sec. 2 defines CPR. We overview 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 conclude the paper (Sec. 9).

2 CONCURRENT PREFIX RECOVERY

A database is said to be at a *point of consistency* if its state reflects all changes made by committed transactions, and none made by uncommitted or in-flight transactions. A checkpoint obtained at a particular point of consistency is said to be *transactionally-consistent*. Upon failure, the database can be recovered from the checkpoint to a consistent state, but some in-flight transactions could be lost.

A slightly stricter notion of consistency is *prefix recovery*, where the database commits, and can recover to, a system-wide prefix of all issued transactions. The database state at any given physical time-point might not be transactionally-consistent as transactions are always being executed. A naive method to obtain a checkpoint for prefix recovery is to stop accepting new transactions until we obtain a snapshot. This technique forcefully creates a physical point-in-time where the database state is consistent, but affects availability.

Ren et. al. [26] propose a multi-versioning based checkpointing scheme that creates a snapshot of the database at a virtual point-in-time, while simultaneously processing incoming transaction requests. They create this virtual point-of-consistency by using an atomic commit log. The identifier of every transaction is appended to an atomic log before commit. They choose a virtual point-in-time t and coordinate transaction execution across the various threads using the commit log to capture the snapshot values as of t. Note that as this scheme is based on a commit log (rather than an

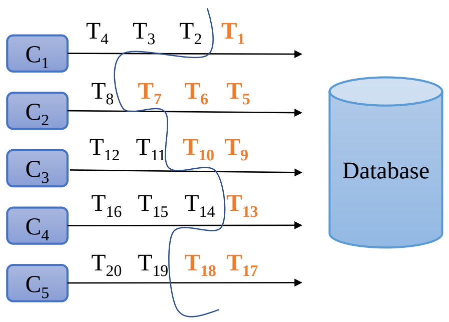


Figure 3: Concurrent Prefix Recovery Model

input log), it does not provide the prefix recovery guarantee outlined above. However, as Sec. 1 and Sec. 7 reveal, a single commit log (or input log) introduces a serial bottleneck in the system and greatly impedes scalability.

We now argue that one cannot obtain a prefix recovery consistent checkpoint of a database without introducing a serial bottleneck. The key insight is that to obtain such a checkpoint, we must create a virtual point of consistency t. As incoming transactions are processed simultaneously, depending on whether they belong to the checkpoint or not, they must be executed differently. For example, consider two transactions T that is committed before t and T' that commits 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 point-in-time t, when chosen. To guarantee prefix recovery, this must be reflected in all threads atomically, 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 recovery-based point-in-time snapshot, the database commits all transactions issued before a single time-point t to each of its clients/sessions. CPR consistency relaxes this requirement by eliminating the need for a *system-wide* time t. Instead, CPR provides a client-local time t_C to each client, such that all transactions issued by the client 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/session 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 5 clients that have issued a sequence of transaction requests $T_1 - T_{20}$. A database state that reflects the effects of transactions T_1 issued by client C_1, T_5, T_6, T_7 issued by C_2, T_9, T_{10} of C_3, T_{13} of C_4 and T_{17}, T_{18} of C_5 is CPR-consistent with respect to the CPR points (marked as curve) shown in figure. Given such a checkpoint, the database can recover the effects of T_5, T_6, T_7 with respect to C_2 and so C_2 can assume they are durable. However, even though T_2 may be completed, it is not committed and cannot be recovered from a CPR checkpoint.

It may appear desirable to be able to obtain checkpoints at client-determined CPR-points. For example, concurrent clients issuing update requests as batches of transactions might want to obtain a snapshot of the database at batch boundaries. We claim that client-determined CPR checkpoints cannot be obtained without quiescing the database. Let a client-determined set of CPR points for a database with k clients be $s_1, s_2, ..., s_k$. A thread T_i handling client C_i can execute the transaction request just after s_i, say s', if and only if all transactions that occur before each of $s_1, s_2, ..., s_k$ have been committed. Hence, s' cannot be executed until that is completed and extending this to all threads, the entire database is blocked until all transactions specified by $s_1, ..., s_k$ are committed. As a result, client-determined CPR checkpoints are unattainable without blocking.

A key insight from the above argument is that s' cannot be executed only because it must read the effects of transactions that are part of the checkpoint and that is pre-determined. To circumvent the issue of blocking, instead, we flip the roles: clients request for a checkpoint and the database collaboratively figures out the CPR-points for threads while trying to obtain a CPR-consistent checkpoint. In rest of the paper, we leverage this idea along with lazy synchronization using epochs to design checkpointing algorithms for a custom in-memory transactional database and a highly concurrent larger-than-memory key-value store. We first review the basics of epoch protection framework that forms the basis of both our CPR algorithms in the next section.

3 EPOCH FRAMEWORK BACKGROUND

Epoch protection helps *avoid coordination whenever possible.* Threads perform user operations independently with no synchronization most of the time. It uses thread-local data structures extensively and maintains system state by lazily synchronizing over them. 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. The system maintains a shared atomic counter $\mathbf{E}$, called the current epoch, that can be incremented by any thread. Every thread T has a thread-local version of $\mathbf{E}$, 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 additionally associate 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 the drain-list, a list of ⟨`epoch`, `condition`, `action`⟩ 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 :

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

4 CHECKPOINTING AN IN-MEMORY TRANSACTIONAL DATABASE

In this section, we present an asynchronous checkpointing algorithm for a simple in-memory transactional database to highlight the key benefit of CPR. We assume strict two-phase locking concurrency control with `NO-WAIT` deadlock-prevention policy. We chose this specific concurrency control for ease of exposition, and believe that the algorithm can be extended appropriately for other protocols. Each record in the database has two versions: *stable* and *live*, and an integer that denotes the current *version* of the record. Note that this formulation assumes twice the memory for the database, but helps simplify the explanation of our key contribution of adding CPR (Sec. 6 covers CPR for a real system that does not make this assumption).

Transactions issued by a client C are processed by a unique execution thread T_C, with pseudo-code shown in Algorithm 1. A separate checkpointing thread coordinates the asynchronous checkpointing of the database. The key idea behind CPR-based checkpoint is that it eliminates the need to coordinate among the various execution threads to identify the virtual point of consistency, instead the correctness relies on fine-grained version information maintained at the records.

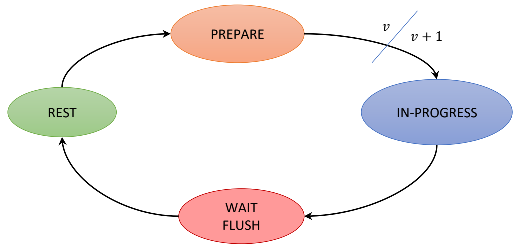


Figure 4: State Machine for CPR DB Checkpointing

In steady state, the database is at some version v. Checkpointing the database corresponds to shifting the database from version v to $(v + 1)$ and capturing the database state as of version v. The algorithm achieves this by using three additional phases: PREPARE, IN-PROGRESS and WAIT-FLUSH. The checkpoint is coordinated across multiple threads lazily using the epoch synchronization of Sec. 3. We maintain two shared global variables `Global.phase` and `Global.version` to denote the current phase and version of the database. Threads have a thread-local view of the these variables and update them only during epoch synchronization. Not having to synchronize with these global state variables atomically is key to scalability of the CPR-based checkpointing. The pseudo-code for the global state machine (Fig. 4) is shown in Algorithm 2.

REST Phase. A checkpointing request is issued when the database is in the REST phase and at some version v. During this phase, transactions are executed normally using strict 2PL with `NO-WAIT` policy. This is the default high-performance phase. The checkpointing thread triggers the algorithm by invoking the `TakeCheckpoint` function, which updates the global state to PREPARE and adds a trigger action to shift to IN-PROGRESS phase when all threads have entered PREPARE phase. Execution threads update their local view of the phase eventually, entering the PREPARE phase.

PREPARE Phase. In the PREPARE phase, execution threads are *prepared* for a shift in the database version. To ensure transactional-consistency of the checkpoint, a transaction is executed in the PREPARE phase only if the entire set of read-write instructions can be executed on version v of the database. Since it belongs to version v such a transaction is part of the checkpoint and can be recovered on failure. The algorithm achieves this by exploiting the fine-grained version information present in the records: if an execution thread in PREPARE phase encounters a record with version greater than v, it immediately aborts the current transaction and refreshes its thread-local view of phase and version.

IN-PROGRESS Phase. The checkpointing thread waits until all threads enter the PREPARE phase the trigger action PrepareToInProgress is executed. Now the global state is updated to IN-PROGRESS and a trigger action to shift to WAIT-FLUSH is added. Since all execution threads are now prepared to handle a shift in the database version, the checkpointing thread atomically updates the global phase to IN-PROGRESS. A thread in IN-PROGRESS phase is responsible for organically shifting the versions of records that it encounters thereby ensuring that a transaction processed in IN-PROGRESS phase executes entirely in version $(v + 1)$ of the database. An IN-PROGRESS thread updates the version of all the records it reads/writes to $(v + 1)$. However, to capture the final value of the record at version v called the *stable value*, the thread copies over the live value of the record when it updates the version of the record.

Once all threads are in IN-PROGRESS phase, trigger action InProgressToWaitFlush is executed automatically as part of epoch synchronization. It starts capturing the version v of the database. In cases where the record has already shifted to version $(v+1)$, the stable v value has been recorded and hence can be captured; while in other cases, the 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;
        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

```
Function TakeCheckpoint()
    Atomically set Global.phase = PREPARE;
    BumpEpoch(all threads in PREPARE, PrepareToInprogress);
Procedure PrepareToInprogress()
    Atomically set Global.phase = IN_PROGRESS;
    BumpEpoch(all threads in IN_PROGRESS,
      InProgressToWaitFlush);
Procedure InProgressToWaitFlush()
    foreach updated 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 transcations;
```
Algorithm 2: Epoch-based State Machine

value can be captured as the stable v value. Once stable v values of all records have been captured and persisted on disk, the checkpointing thread updates the phase and version to REST and $(v + 1)$ respectively. All staged transactions can now be committed as they can be safely recovered from the checkpoint in the event of a failure.

4.1 Correctness

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

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

PROOF SKETCH. Firstly, the snapshot obtained from the algorithm is transactionally-consistent as 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) when it is executed. So, a transaction can either belong to the checkpoint or not.

All transactions executed in PREPARE phase belong to version v and all those executed in IN-PROGRESS or WAIT-FLUSH phase belong to $(v + 1)$. Since for every thread there is a unique point in time t_C when it shifts from PREPARE to IN-PROGRESS for version v, which also marks the set of transactions for the client C that are included and none after, the snapshot is CPR-consistent.

Consider two transactions T and T' such that T belongs to the checkpoint and T' does not. If both T and T' belong to the same client and executed by a single thread, $T \prec T'$ in the serial order. If T and T' belong to different clients, then there are two possibilities depending on their read-write sets. When their read-write sets do not intersect, $T \prec T'$ is a valid serial order as it is conflict equivalent to $T' \prec T$. If the read-write sets intersect on a set of records R, we know that T could have executed only when version of each record r in R is v and similarly T' could have executed only when they are $(v + 1)$. We know that version of r shifts from v to $(v + 1)$ and hence $T \prec T'$. Thus, schedule of execution resulting from the algorithm is conflict-equivalent to a point-in-time snapshot where all v transactions are executed before $v + 1$ transactions. □

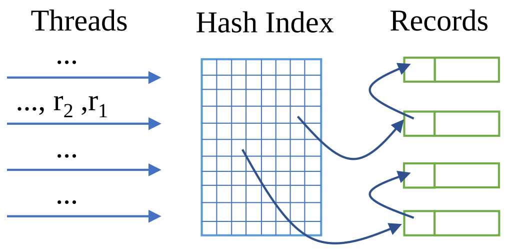


Figure 5: FASTER Overall Architecture

5 BACKGROUND ON FASTER

This section overviews FASTER [6], a 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 shared memory. It supports reads, blind upserts, and read-modify-write (RMW) operations. FASTER reports a scalable in-memory throughput of up to 160 million ops/sec, making it a good candidate for extending with CPR-based durability.

FASTER has two main components (Fig. 5): a *hash index* and a log-structured record store called `HybridLog`. The index serves as a map from the hash of a key to an address in a logical address space. Keys that have the same hash value 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: each record contains a previous address in its metadata that is 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. 6), a log-structured record store that spans disk and main memory. 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

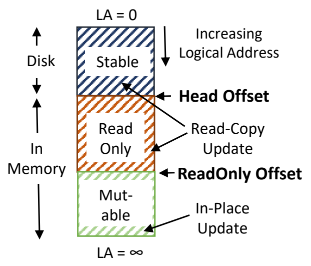


Figure 6: `HybridLog` Organization in FASTER

the lowest 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 a 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 could read a stale value of the read-only offset and update a record in-place, 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-the only and the safe read-only offset (called the *fuzzy region*; see Fig. 7), the request is added to a pending list to be processed later.

The update scheme is summarized next (we focus on RMW for simplicity). 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 address, the record is present in memory. We update the record in-place when it is mutable i.e. l is greater than the read-only offset. If l is less than safe read-only offset, we create a new updated copy at the tail and update the

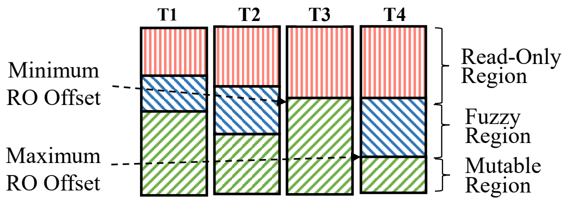


Figure 7: Thread-Local View of `HybridLog` Regions

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 using `StartSession` and `StopSession`. FASTER uniquely identifies each session using a `Guid`. All user requests (`RMW`, `Read`, `Upsert`) to FASTER occur within a session, and carry a unique (session-local) serial number.

```
// Session API
Guid StartSession();
StopSession(Guid guid);
long ContinueSession(Guid);
void RegisterCallback(Action<long>);
```

In the event of a failure, the client can re-establish the lost session using its `Guid` by invoking `ContinueSession`. On successful connection, FASTER returns a serial number s which specifies that FASTER has recovered *all* and *only* those requests from the session until s. As described earlier, CPR commits are session-local, and FASTER recovers to a specified persistence point for every session. Optionally, the client can register a callback with FASTER, that notifies the client of the new session-local persistence point whenever FASTER group-commits its state using CPR.

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 allowing periodic creation of a CPR-consistent checkpoint that is persisted on disk. In the event of a failure, FASTER recovers from the most recent commit point (all uncommitted in-flight requests after this CPR "line in the sand" are lost).

In FASTER, every session maps to a single thread. For ease of exposition, assume that every FASTER thread handles one session, and there are N threads. Each user request (e.g. RMW, Read, Upsert) on thread T_i is associated with a strictly increasing session-local serial number s_i. We wish to create a CPR-consistent commit point $s_0, s_1, \ldots, s_N$ for FASTER, such that for every thread T_i, the commit makes only and all requests before 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 shared memory, letting user code control concurrency. Handled naively, this design can lead to the lost-update anomaly, as shown in Sec. 5. Since CPR enforces a strict *only and all* policy, it is challenging to achieve a CPR-consistent checkpoint without compromising on fast concurrent memory access.
- FASTER is designed to support disk-resident data using an asynchronous model: the request is issued and gets completed by an I/O thread in the background, while the requesting thread processes future requests. On I/O completion, the *continuation* is handed back to the same thread for further processing. This model helps reduce the overhead of accessing data from storage, but further complicates CPR in a fundamental sense: at any point, there can be some requests before the identified CPR point, that have not yet been completed. A request r_1 not belonging to the CPR checkpoint cannot be executed before a request r_2, potentially from a different session, that belongs to the checkpoint. When handled naively, this can lead to a quiescing of the key-value store.

6.2 Hybrid Log Checkpoint

We augmented the 64-bit per-record header in `HybridLog` (used 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 every record. During normal processing, FASTER is in the REST phase and at a particular version v. Obtaining a checkpoint of FASTER involves (1) shifting its version from v to $(v + 1)$ and (2) capturing all modifications made during version v. We leverage our epoch framework (Sec. 3) to loosely coordinate this transition using a global state machine. The global state machine for CPR in FASTER, shown in Figure 8a, consists of 5 states: REST, PREPARE, IN-PROGRESS, WAIT-PENDING and WAIT-FLUSH. These phases help orchestrate the shift without affecting the performance of user-space requests while a CPR-consistent commit is made durable. State transitions in the global state machine are realized by FASTER threads lazily, when they refresh their epochs periodically. An example execution with 4 threads is shown in Figure 8b. Following is a brief overview of each phase:

- REST: Normal processing on FASTER version v. This phase provides identical performance to unmodified FASTER, making CPR a purely pay-as-you-go operation.
- PREPARE: Requests accepted before and during the PREPARE phase for every thread are part of v checkpoint.
- IN-PROGRESS: Transition from PREPARE to IN-PROGRESS demarcates CPR point for a thread: requests accepted in IN-PROGRESS (or later) phases do not belong to v checkpoint.
- 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 $(v + 1)$.

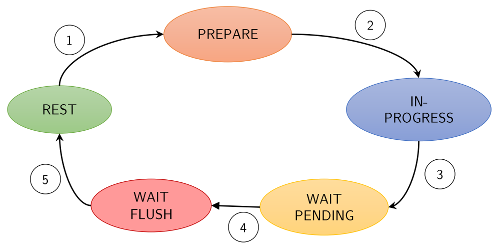
The request to perform a CPR commit (from the user or triggered periodically) first records the current tail address of `HybridLog`, say $\mathbf{L_s}$, and updates the global state from REST to PREPARE. A thread subsequently updates its thread-local state during an epoch refresh, and enters the PREPARE phase.

6.2.1 PREPARE. A thread T in PREPARE phase 'prepares' to handle a shift in version. In FASTER, some requests are processed asynchronously and hence every thread has a thread-local list of pending requests. As soon as T enters PREPARE, it acquires a *shared-latch* on the bucket (note that we may instead use a per-record latch as well) corresponding to each such pending request. These latches are released when the request is completed. T processes an incoming user-request on key k with record at logical address L on `HybridLog` using the scheme below (Pseudo-code in Algorithm 3 of Appendix A).

T first tries to acquire a shared-latch on the bucket corresponding to k. If the record is available in memory with a version $\leq v$, the request is processed depending on which region of `HybridLog` it belongs to: (1) a record in mutable region is updated in-place; (3) a new updated record with version v is created at end of tail when the record is in safe-read-only region; and (3) when record is in the fuzzy region, the request is added to the thread-local pending list; (4) T issues an asynchronous I/O request to retrieve the record from secondary storage when it is not available in memory. In cases (1) and (2), the shared-latch is released immediately, whereas in (3) and (4) it is released only when the requests are completed later.

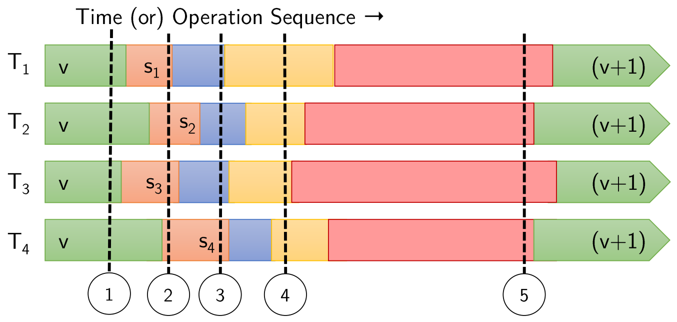
When T is unable to acquire a shared-latch on the bucket or when the version of the record in memory is $> v$, T detects that the CPR shift has begun. The key optimization that prevents blocking here is that the thread immediately refreshes its thread-local state and enters the IN-PROGRESS phase. It is possible that a PREPARE phase thread never encounters such a scenario, in which case the CPR shift for T happens during its periodic epoch synchronization.

6.2.2 IN-PROGRESS. The state machine advances to IN-PROGRESS phase when all threads have entered PREPARE phase and acquired shared latches corresponding to their thread-local pending requests. This shift is performed using



① User Request to Take Checkpoint
② When all threads have acquired shared-latches on pending requests
③ When all threads have entered IN-PROGRESS phase
④ When all v pending requests have been processed
⑤ When snapshot written to disk

(a) Global State Machine with Transition Conditions



(b) FASTER Threads during checkpoint

Figure 8: Overview of CPR for FASTER

BumpEpoch with condition and action, described in Sec. 3. A thread T shifts to IN-PROGRESS from PREPARE phase during the periodic epoch synchronization or when it encounters any record with version $> v$. This shift from PREPARE to IN-PROGRESS demarcates CPR shift for thread T. All user requests received by T before this point are part of the commit, and none after. T processes incoming requests as belonging to checkpoint version $(v + 1)$. The key idea here is to not let T modify any record of version v (or lesser) in-place and force it to perform a read-copy-update without violating CPR-consistency. The pseudo-code for IN-PROGRESS and later phases are shown in Algorithm 4 of Appendix A.

When the record is in memory with a version $\leq v$, T tries to acquire an exclusive-latch on the bucket for the key. The acquisition succeeds only when no other active request in the PREPARE phase holds a shared-latch on the bucket. The request is then processed using the copy-on-write scheme by creating an updated $(v+1)$-record at the end of tail. The latch is released when update is completed. If latch acquisition fails, the request is added to a thread-local pending list that corresponds to version $(v + 1)$.

If record is of version $(v + 1)$, T 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 it is in fuzzy region. When it is not available in memory, T issues an asynchronous I/O request and adds the request to $(v + 1)$ pending list.

6.2.3 WAIT-PENDING. The global state transits to WAIT-PENDING when all threads have entered IN-PROGRESS (using the BumpEpoch mechanism from Sec. 3). CPR-consistency requires that *all* requests accepted before a certain session-local point in time be recoverable on failure. However, some requests before the CPR-point could still be pending. FASTER stays in the WAIT-PENDING phase until such pending requests that are part of the checkpoint have completed. Whenever a pending v-request is completed, its corresponding bucket-level shared-latch is released.

A thread T in WAIT-PENDING, processes incoming user requests in a manner similar to the IN-PROGRESS phase (4). When the system is in WAIT-PENDING, all other FASTER threads are either in the IN-PROGRESS or WAIT-PENDING phase. Since no thread is in PREPARE, T creates a $(v + 1)$ record when the count of shared-latches is zero. When the shared latch count is greater than zero, there is an active pending request corresponding to that bucket and hence creating a $(v + 1)$ record could violate the CPR consistency.

6.2.4 WAIT-FLUSH. Once pending requests that belong to the checkpoint have been completed, we record the current tail address of HybridLog, say L_e, and shift the read-only offset of HybridLog to L_e, which causes an asynchronous flush of HybridLog until L_e to storage. Note that there are no v records after L_e, whereas there may be some $(v + 1)$ records before L_e that need to be ignored during recovery (see Sec. 6.4). We then bump the current epoch with an action to shift the global state machine to WAIT-FLUSH. During WAIT-FLUSH, user requests are processed identically to the REST phase because all v records are guaranteed to be in the safe read-only region.

Once the asynchronous write to secondary storage is complete, FASTER moves back to REST, now with version $(v + 1)$, thus completing the multi-phase CPR checkpoint of FASTER.

6.3 FASTER Hash Index Checkpoint

In addition to the checkpoint of HybridLog, we obtain a fuzzy checkpoint of the hash index that stores the map from key-hash to logical address of records on HybridLog. The index can be checkpointed independently from, and usually less frequently than, the HybridLog. Since hash bucket entries in the FASTER index are always updated using compare-and-swap, the index is always physically consistent. So, to obtain a fuzzy checkpoint, we flush the entire hash index (and overflow buckets) to storage using asynchronous I/O. We record the tail address of `HybridLog` before starting (L_h) and after completing (L_i) the fuzzy checkpoint. These offset are using during recovery, described next.

6.4 Recovery

Recovering from a CPR-consistent checkpoint is straightforward. We first recover the FASTER hash index from its fuzzy checkpoint. All updates to the hash index during its fuzzy checkpoint correspond to `HybridLog` records between L_h and L_i, because in-place updates do not modify the index. However, some of these updates may be part of the fuzzy checkpoint and some may not. During recovery, therefore, we scan through the records between L_h and L_i on the `HybridLog` in order, and update the recovered fuzzy index wherever necessary. The resulting index is a logically consistent hash index that corresponds to `HybridLog` until L_i, because all updates to hash index entries after completing the fuzzy checkpoint (and recording the tail-offset L_i) correspond only to records after L_i in `HybridLog`.

Every CPR commit (checkpoint) of `HybridLog` corresponds to a version v and the meta-data contains the tail address at the start L_s and end L_e of the CPR checkpoint. We first resurrect the `HybridLog` until L_e. We then scan the records from L_i (the end of the last index checkpoint) to L_s, updating hash entries as in index recovery. Next, we scan `HybridLog` from L_s to L_e. If we encounter a record with version $(v + 1)$, we simply mark the record as invalid and do not update the index. If we encounter a record with version less than $(v + 1)$, we update the index as before. Finally, recover the contexts of all the active sessions from the recorded meta-data. The clients can now continue their session by invoking `ContinueSession(Guid)`.

6.5 Solution Variants

Implementation of the CPR checkpointing algorithm with support for asynchronous requests presents a range of design choices, out of which we discuss two major ones in Appendix B and C. Appendix B deals with the granularity at which the shift from version v to $(v + 1)$ happens in FASTER and appendix C deals with how the stable v records are captured on secondary storage. These two choices impact the design of FASTER in significant ways and hence we compare them empirically in Sec. 7.3.

7 EVALUATION

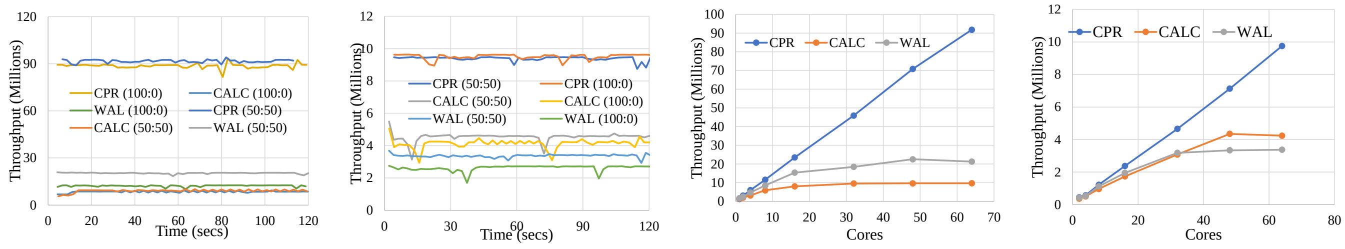
We evaluate CPR in two ways. First, we use a main-memory database implementation to compare CPR to two state-of-the-art asynchronous checkpointing techniques (CALC and WAL). Next, we evaluate the addition of CPR to our implementation of FASTER, a high-performance hash key-value store for larger-than-memory data.

7.1 Implementation, Setup, Workloads

Implementation. For the database evaluation, we implemented a stand-alone main memory database, that supports three recovery techniques (CPR, CALC, and traditional database WAL). Both CALC and CPR uses has a *stable* and *live* value for each record, while WAL has only a single value for each record. An optimal implementation of CPR does not require replicating every record. However, we chose this design to perform a head-on-head comparison with the CALC design presented by Ren et. al. [26]. Checkpointing of CPR and CALC happens asynchronously and every checkpoint writes the entire database to disk. We did not obtain incremental checkpoints as it is an orthogonal optimization that can be applied both to CPR and CALC. We do not obtain fuzzy checkpoints for WAL but periodically flush the generated log records to disk asynchronously, thus removing I/O related delays. All three versions use the main-memory version of FASTER [6] as its data store, and implements two-phase locking with NO-WAIT deadlock-avoidance strategy on top. In all our experiments, we are mainly concerned about throughput of the system during normal and checkpoint time.

We also added CPR to our C# implementation of FASTER (based on the description in [6]). Threads first load the key-value store with data, and then issue a sequence of operations. Checkpoints are issued periodically. We measure and report system throughput and latency every 2 seconds. We point FASTER to our SSD, and employ an expiration-based garbage collection scheme (not triggered in these experiments). We size the total in-memory region of `HybridLog` 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, we size the FASTER hash index with #keys/2 hash bucket entries.

Setup. Experiments comparing recovery techniques using our custom stand-alone main-memory database implementation are conducted on a Standard D64s v3 machine [2] on Microsoft Azure. The machine has 64 vCPUs and 256 GB memory, and runs Windows Server 2018. Experiments on CPR with FASTER are carried out 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, and the two-socket experiments shard threads across sockets. The machine has 512GB of RAM and a 3.2TB FusionIO NVMe SSD drive for the log and checkpoints. We preload input datasets into memory for all experiments.



(a) Throughput; Txn-Size:1 **(b) Throughput; Txn-Size:10** **(c) Scalability; Txn-Size:1** **(d) Scalability; Txn-Size:10**

Figure 9: Comparing performance of CPR, CALC and DB on low-contention YCSB workload

Workloads. For our stand-alone database, we use a mix of transaction operations based on YCSB. Transactions are executed against a table with 250 million 8 byte keys and values of size 8 bytes. Each transaction is a sequence of read-write requests on these keys. We vary transaction size by varying number of requests. Keys are generated from a low-contention zipfian distribution ($\theta = 0.1$), so as to minimize concurrency control overheads. Each request is classified as read or write randomly based on a read-write ratio. A write operation replaces the value in the database with a provided value. We mainly use update-only (100:0) and equal ($50:50$) mixture.

For FASTER wth CPR, we use an extended version of the YCSB-A workload from the Yahoo Cloud Serving Benchmark [7], with 250 million distinct 8 byte keys, and value sizes of 8 bytes and 100 bytes. After pre-loading the keys, the records take up 6GB of `HybridLog` space, whereas the index takes up 8GB of space. Operation workloads are described as R:BU for the fraction of reads and blind updates in the workload. 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 ($\theta = 0.99$) distributions for operations.

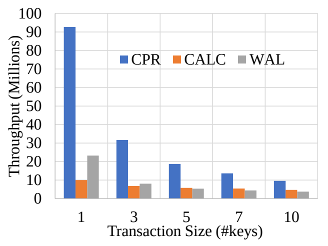
7.2 Database Checkpointing Comparisons

7.2.1 Throughput vs. Time. We plot throughput of in-memory transaction processing vs. time during the lifetime of a run for CPR, CALC and WAL on 64 cores. We take three checkpoints – at 30, 60 and 90secs. Fig. 9a and Fig. 9b shows the result for a 1-key and 10-key transactions respectively for update-only ($100:0$) and mixed read-write ($50:50$) YCSB workload with $\theta = 0.1$.

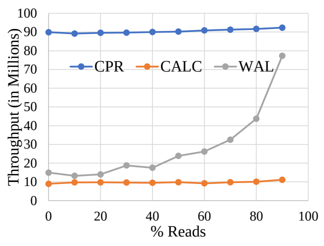
Throughput of CPR is an order of magnitude ($\approx$ 90 Mops/sec) higher than throughput of CALC and WAL on 1-key update-only workload, due to contention at log tail for WAL and atomic commit log for CALC. There is almost no impact on performance during checkpoints at 30, 60 and 90 secs as all three designs obtain checkpoints asynchronously. Throughput of CPR and CALC does not vary significantly due to transaction mix, whereas WAL performs 2x better on mixed than update-only workload as no WAL records are created for read-only transactions. CPR outperforms both WAL and CALC by 3x and 2.5x on 10-key transactions respectively. CPR throughput drops from 90M for 1-key transaction to 10M for 10-key transactions as the latter is 10x heavier. Unlike 1-key case, CALC outperforms WAL even on mixed workload as the number of read-only transactions are smaller, even though 50% of individual accesses in each are reads. On top of the contention at log tail, WAL incurs cost of writing large update information and updating the PageLSN for each record page.

7.2.2 Varying number of threads. We plot average throughput for varying number of threads to compare scalability of the three database designs. We present results for mixed workload ($50:50$) on 1- and 10-key transactions in Fig. 9c and Fig. 9d respectively. We did not obtain any checkpoint during the run as the difference in performance is insignificant. CPR design scales linearly upto 90Mops/sec whereas, CALC reaches a maximum of 10Mops/sec due to the serial bottleneck introduced by atomic commit log. WAL reaches 20Mops/sec as the contention on tail is half as compared to CALC. Similarly for 10-key transactions, CPR scales linearly upto 10 Mops/secs, while CALC and WAL scale only upto 3.5 and 4.2Mops/sec. CALC and WAL scale until 48 cores but flatten beyond due to contention at log tail, especially across NUMA nodes. WAL performs poorer than CALC due to the additional overhead of writing old and new values to be used for recovery.

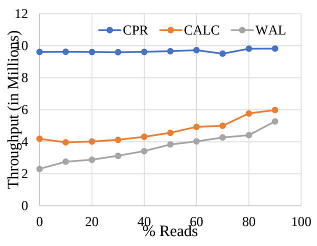
7.2.3 Varying transaction size. We present average throughput for transactions of size 1, 3, 5, 7 and 10 for mixed ($50:50$) workload on 64 cores. Cost of executing a transaction is directly proportional to the amount of computation and number of accesses. Fig. 10a depicts throughput of CPR, CALC and WAL for various transaction sizes. CPR throughput, which is an order of magnitude higher than the other two,



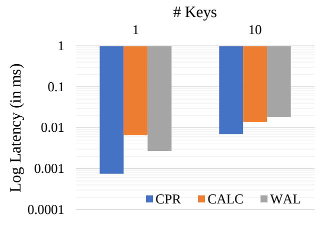
(a) Throughput; Mix: 50:50



(b) Throughput; Txn-Size:1



(c) Throughput; Txn-Size:10



(d) Latency; Mix: (50:50)

Figure 10: Comparing throughput, latency of CPR, CALC and DB for different transaction mixes

drops linearly as there is no other bottleneck in the system. CALC throughput is already bottlenecked at 10Mops/sec by the contention at log tail and hence does not result a drastic reduction in performance. WAL, on the other hand, has to write a lot more data to record the modifications done to the database as we increase transaction size. WAL outperforms CALC for smaller transactions and as the size increases the trend reverses due to this overhead.

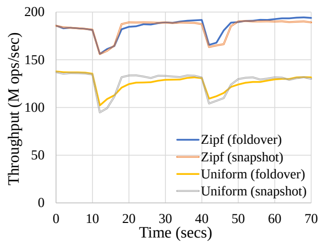
7.2.4 Varying transaction mix. We investigate the impact of read-write ratio on performance by varying the transaction mix from 0% to 90% reads on 64 cores. We plot results for both 1-key and 10-key transactions generated using YCSB zipfian distribution with $\theta = 0.1$ in Fig. 10b and Fig. 10c respectively. In both cases, read-write ratio does not impact performance of CPR. In a 1-key transaction, read-write ratio directly affects the contention on tail of WAL. As we increase the read percentage, performance of WAL improves due to fewer log records. Performance of CPR increases marginally as reads are cheaper than writes. 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 serial bottleneck, is not the most significant one, and so effect of cheaper reads result in performance improvement for both WAL and CALC. Gains in WAL is higher for higher read percentage as it eliminates two writes: update to record value and during log record generation.

7.2.5 Latency. We now present average latency of transaction execution for CPR, CALC and WAL for a mixed (50 : 50) workload both on 1 and 10-key transactions on 64 cores. Fig. 10d depicts the log of latency in milli-seconds as a y-axis reversed bar graph. 1-key transactions in CPR are executed in approximately 700 nanoseconds, which is as expensive as a few cache misses in modern CPUs. Such efficient in-memory performance is made possible by the underlying FASTER hash index [6]. Since CPR is almost linearly scalable, this yields an in-memory performance of 90 Mops/sec on 64 cores. Due to contention at tail, CALC and WAL incur a latency of around $6\mu s$ and $2\mu s$ respectively. Note however, increasing number of cores results in increasing latency as well due to increased contention. 10-key transactions incur a cost of $7\mu s$, which is 10x that of 1-key transaction for CPR. Since, most transactions result in creation of a WAL log record, the effect of log tail contention is evident for WAL with a latency of $21\mu secs$. The latency of CALC is slightly lesser as it appends only the transaction identifier rather than all modifications to the database like in WAL.

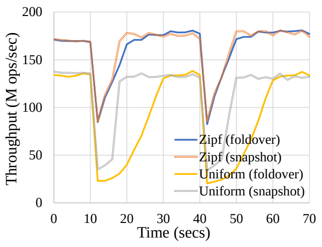
7.3 Evaluation of FASTER with CPR

7.3.1 Throughput and Log Size. We plot throughput vs. wall-clock time during the lifetime of a FASTER run. We take two "full" (index and log) checkpoints 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 checkpointing techniques – *fold-over* and *snapshot* to separate file – in these experiments.

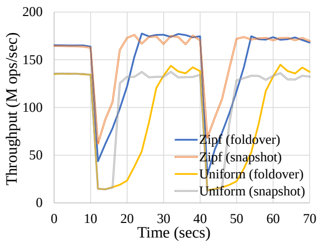
Fig. 11a shows the result for a 90:10 workload (i.e., with 90% reads). Overall, Zipf outperforms Uniform due to the better locality of keys in Zipf. After taking a checkpoint, both snapshot and fold-over slightly degrade in throughput because of read-copy-update during the checkpoint operation. It takes 6 secs to write 14GB of index and log, close to the sequential bandwidth of our SSD. After the second checkpoint, the Zipf throughput of fold-over returns to normal faster than snapshot because of its incremental nature (only the few records updated after the first checkpoint need to be flushed). With a 50:50 workload, in Fig. 11b, fold-over drops in throughput after taking a checkpoint, 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, avoiding having to move the entire working set to a new mutable region. A 0:100 workload with only blind updates demonstrates similar effects, as shown in Fig. 11c. We also measured the time taken in each of the CPR phases:



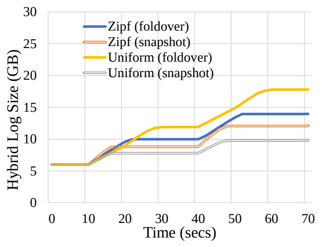
(a) Throughput; YCSB 90:10.



(b) Throughput; YCSB 50:50.

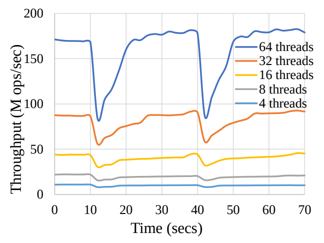


(c) Throughput; YCSB 0:100.

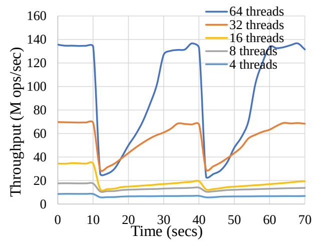


(d) Log growth; YCSB 0:100.

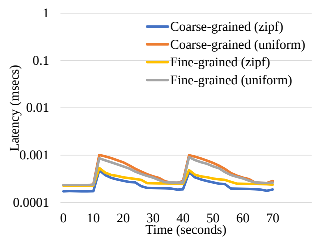
Figure 11: FASTER throughput and log growth vs. time; full fold-over and snapshot checkpoints at 10 **and** 40 **secs.**



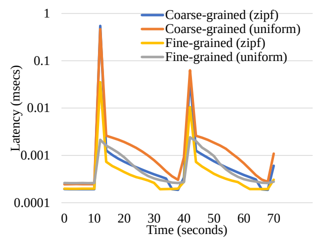
(a) 50:50 **Zipf distr.**



(b) 50:50 **Uniform distr.**



(a) 0:100 **blind updates.**



(b) 0:100 **RMW updates.**

Figure 12: Throughput vs. time; varying #threads.

Figure 13: Latency vs. time; log-only fold-over.

each phases lasted for around 5ms, except for WAIT-FLUSH, which took around 6 secs as described above.

Fig. 14c 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 checkpoint for Uniform.

We also experimented with checkpointing only the log, with more frequent checkpoints, since the index is usually checkpointed infrequently. The results are in Appendix D; briefly, we found checkpoints to have much lower overhead as expected, with overall trends remaining similar.

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 results the Zipf and Uniform distributions in Figs. 12a and 12b respectively, with full fold-over checkpoints taken at the 10 sec and 40 sec mark. Both figures show linear throughput improvement with increasing number of cores, indicating that CPR support 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 checkpointing is minimal due to lower rest phase performance. Further, the recovery to maximum throughput after a checkpoint is faster with more threads.

7.3.3 Operation Latency; Fine- vs. Coarse-Grained. Fig. 13a 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 two the 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, latency increases to 1 microsecs for Zipf and around 0.5 microsecs for Uniform because of slightly more contention on index entries for hot keys in Zipf. Since there is no data-dependency between the v and $v + 1$ version with 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 the v and $v+1$ versions of a record, we expect the hand-off to have a greater effect as operations in the $v + 1$ phase may have to waiting for records being updated in the X phase (using a lock in fine-grained or going pending in coarse-grained). This is validated in Fig. 13b, 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 of items going pending. Further, the effect of Zipf on latency is higher than the effect of Uniform because of greater contention while handing off records for the hot keys. Overall, latency degrades slightly during CPR as expected, but with fine-grained, latency stay below 2.2 microsecs even during CPR.

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 the prior work in this area:

Write-Ahead Logging. The write-ahead logging (WAL) protocol originally defined in [1] has been the defacto standard for providing durability in traditional databases. Nearly all database systems use WAL based on ARIES [25], which integrates concurrency control with transaction rollback and recovery. WAL has been reported to be a significant overhead [16] (roughly 12% of total time in a typical OLTP workload) even in the case of single-threaded database engine. Johnson et. al. [17] identify I/O related delays, log-induced lock contention log buffer contention and excessive context switches to be key factors resulting in overheads. To reduce log contention at the tail, Aether [17] consolidates log allocation. Jung et. al. [18] 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 [4, 30] 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 overheads. Zhen et al. present SiloR [32] that avoids centralized logging bottleneck by letting each worker thread copy transaction-local redo logs to per-thread log buffer after validation and is based on optimistic concurrency control (OCC). Even though several attempts have been made to make WAL faster, recording every update is expensive, particularly when operating at hundreds of millions of operations per second. Stonebraker et. al. [29] recommend completely eliminating WAL, but instead suggest replication across distributed main memory, which is complex and uses more resources.

Distributed Logging. While traditional log implementations enforce a particular serial order of log entries using LSNs, distributed logging exploits the fact that there are more than one correct orders for a given serializable order. Lomet et. al. [22] propose a redo logging method where individual nodes can maintain a private log and upon a crash failure recover from its own private log and merge other existing logs in any order that results in same serializable order in case of a media failure. This method is found to be infeasible in a multi-socket scenario as the protocol requires dirty page writes during migrations. Johnson et. al. [17] 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, they do not address the problem of every update being logged which is a significant overhead in highly update-intensive scenarios. CPR checkpoints are transactionally consistent and do not need any WAL, thus allowing fast in-place updates in memory.

Fuzzy and WAL-based Checkpointing. Many [14, 21] have proposed adopting fuzzy checkpointing to main memory databases and some have argued [27] that it is the most efficient as it does not interfere with main stream performance. A fuzzy checkpoint requires the WAL to recover a consistent state of the database. Microsoft's main memory database system, Hekaton [10], asynchronously obtain periodic partial checkpoints of the database by scanning tail of the log. Both these approaches depend on the WAL, which we seek to avoid in our work due to its overheads.

Transactionally-Consistent Checkpoints. Another popular approach to provide durability is to obtain point-in-time snapshots that are transactionally-consistent. Certain applications such as multiplayer online games naturally reach points in the process of normal execution when no new transactions are being executed. Cao et. al. [3] propose two different checkpoint algorithms zig-zag and interleaved ping-pong that captures entire snapshots of a database without key locking using multiple version. Similarly, VoltDB [23] uses an asynchronous checkpointing technique which takes checkpoints by making every database record "copy-on-write". We have seen that this approach is expensive in update-intensive scenarios. CALC [26], on the other hand, 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.

9 CONCLUSION

Modern databases and key-value stores have pushed the limits of multi-core scalability to hundreds of millions of operations per second, leading to durability becoming the new 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 a larger-than-memory hash key-value store. A detailed evaluation of both these systems shows that we can support highly concurrent and scalable performance, reaching hundreds of millions of operations per second on a multi-core machine, while providing CPR-based durability.

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. https://doi.org/10.1145/320455.320457

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

[3] 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. https://doi.org/10.1145/1989323.1989352

[4] Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). 2009. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009.* ACM.

[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. https://doi.org/10.14778/2735496.2735503

[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. https://doi.org/10.1145/3183713.3196898

[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. https://doi.org/10.1145/1807128.1807152

[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. https://doi.org/10.1145/602259.602261

[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. https://doi.org/10.1145/2463676.2463710

[10] 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. https://doi.org/10.1145/2463676.2463710

[11] 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. https://doi.org/10.1561/1900000058

[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. https://doi.org/10.1109/TC.1986.1676845

[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. https://doi.org/10.1145/1376616.1376713

[16] 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. https://doi.org/10.1145/1376616.1376713

[17] 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. https://doi.org/10.1007/s00778-011-0260-8

[18] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable Database Logging for Multicores. *PVLDB* 11, 2 (2017), 135–148. https://doi.org/10.14778/3149193.3149195

[19] 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. http://sites.computer.org/debull/A13june/hyper1.pdf

[20] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. https://doi.org/10.1145/320613.320619

[21] 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. https://doi.org/10.1145/331119.331168

[22] 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. http://dl.acm.org/citation.cfm?id=645921.673170

[23] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering.* 604–615. https://doi.org/10.1109/ICDE.2014.6816685

[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. https://doi.org/10.1145/2168836.2168855

[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. https://doi.org/10.1145/128765.128770

[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. https://doi.org/10.1145/2882903.2915966

[27] Kenneth Salem and Hector Garcia-Molina. 1989. Checkpointing Memory-Resident Databases. In *Proceedings of the Fifth International Conference on Data Engineering.* IEEE Computer Society, Washington, DC, USA, 452–462. http://dl.acm.org/citation.cfm?id=645474.653875

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

[29] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007.* 1150–1160. http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf

[30] 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.

[31] 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. https://doi.org/10.14778/2735508.2735511

[32] 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. http://dl.acm.org/citation.cfm?id=2685048.2685085

A PSEUDO-CODE FOR CPR ON FASTER

Algorithm 3 depicts the pseudo-code for executing incoming user requests during PREPARE phase on FASTER. Algorithm 4 depicts the pseudo-code for all other phases during CPR checkpoint.

```
if bucket.TrySharedLatch() then
    if $L \geq HeadAddress$ then
        if $record.version \leq v$ then
            if $L \geq ReadOnlyAddress$ then
                update record in-place concurrently;
                bucket.ReleaseLatch();
                return OK;
            else if $L \geq SafeReadOnlyAddress$ 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;
ReleaseLatch();
return CPR_SHIFT_DETECTED;
```

Algorithm 3: Pseudo-code for PREPARE Phase

B COARSE- VS. FINE-GRAINED SHIFT

An important transition in CPR is from the PREPARE to IN-PROGRESS phase, when threads demarcate the CPR points and shift from processing incoming requests as version v to $(v + 1)$. A thread in IN-PROGRESS phase performs a copy-on-write update to a v-record only when it is either in the immutable region or by acquiring an exclusive-latch on the bucket if it in the mutable region. The algorithm described in Sec. 6.2 uses a fine-grained bucket- or record-level latch to ensure CPR-consistency of updates. PREPARE phase threads acquire a shared-latch even for in-place updates to records in the mutable region, which may be expensive.

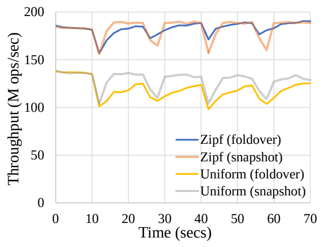
An alternate approach is to acquire a shared-latch only when a request is added to the pending list. However, this could cause an inconsistency as IN-PROGRESS threads might copy over a v-record when a PREPARE thread is updating it in-place, similar to the lost-update anomaly in Sec. 5. In order to prevent this, an IN-PROGRESS thread performs a copy-on-write update only when the v-record is in the safe read-only region. The safe read-only address, in this context, is used as a coarse-grained marker for records that are eligible for a version shift as they cannot be updated concurrently. For records that are in the mutable region, however, the request is instead added to the thread-local pending list. This approach trades-off making a request pending in the IN-PROGRESS phase (which can incur extra operation latency) against the overhead of bucket- or record-level pinning for in-place updates in the PREPARE phase.

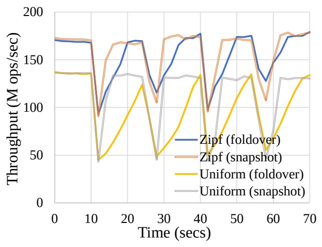
```
if $L \geq HeadAddress$ then
    if $record.version \leq v$ then
        switch thread.Phase do
            case IN-PROGRESS do
                if bucket.TryExclusiveLatch() then
                    create updated $(v + 1)$ record at tail;
                    ReleaseLatch();
                    return 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 $L \geq ReadOnlyAddress$ then
            update record in-place concurrently;
            return OK;
        else if $L \geq SafeReadOnlyAddress$ 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;
```

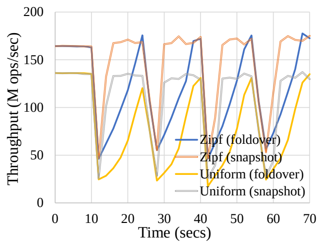
Algorithm 4: IN-PROGRESS, WAIT-PENDING and WAIT-FLUSH Phases

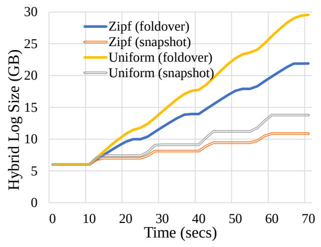
C HYBRID LOG FOLD-OVER VS. SNAPSHOT

Another design dimension that presents two alternatives 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 address to end tail address of the CPR checkpoint and by design of HybridLog all records









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

Figure 14: FASTER throughput and log growth vs. time; log-only fold-over and snapshot checkpoints.

in that portion of the log are written as part of `HybridLog` on secondary storage. We call this the *fold-over checkpoint*, and is the version described earlier. However, with this approach, even after the checkpoint, a request on encountering a v record does a copy-on-write update by creating a new $(v + 1)$ record at end of tail. Moreover, folding-over `HybridLog` unnecessarily forces some warm records to the disk and thus increasing the number of asynchronous I/O read requests further delaying performance recovery. The key advantage of fold-over checkpoints is the amount of data written to disk during the checkpoint, because it is fully incremental, i.e., only data that changed after the previous checkpoint needs to be written out.

Another variant is to obtain a snapshot of all the volatile records in memory into a separate snapshot file on disk *without* modifying the state of the `HybridLog`. After the snapshot of memory is taken, we can move to the REST phase, where a v-record in the mutable region can now be updated in-place, allowing us to avoid read-copy-update for such records. In effect, the period of time during which the system is in read-copy-update mode is limited to the time it takes to write the mutable part of `HybridLog` to a separate snapshot file. This time is dependent on available storage write bandwidth, rather than being dependent on how long it takes to read-copy-update the working set to the mutable region. The downside of this technique is that every checkpoint must write all records in the volatile region to secondary storage, which may include records that were not updated after the previous checkpoint, which can be expensive if CPR commits are performed frequently.

D FREQUENT LOG CHECKPOINTS

The FASTER hash index can be recovered independently from `HybridLog` and hence frequent checkpoints 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 before, as we vary the workload mix.

Fig. 14a shows the result for a read-dominated 90:10 workload. We see that throughput is affected very slightly due to checkpoints, 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 checkpoints (being incremental) have little effect on throughput. The performance with Uniform is similar. Fold-over takes longer to get back to full throughput as compared to snapshot, as the working set needs to migrate to the new mutable region. With the 50:50 and 0:100 workloads (Figs. 14b and 14c), the Zipf distribution with fold-over is able to ramp up post-checkpoint performance quicker than Uniform, as it captures the working set quickly. As before, snapshot can quickly get back full performance 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. 14d. The log growth is more rapid with fold-over checkpoints and with Uniform distributions, as expected.