Concurrent Prefix Recovery

Guna Prasaad, Badrish Chandramouli, Donald Kossman

Motivation

Extremely scalable data processing systems powered by:

- Main memory databases
- Thousands of cores, NUMA, SIMD, HTM

State of the art key-value stores:

- Masstree: 30M ops/sec, in-memory
- FASTER: 150M ops/sec, larger-than-memory, cached hot working set

How to get scalable durability?

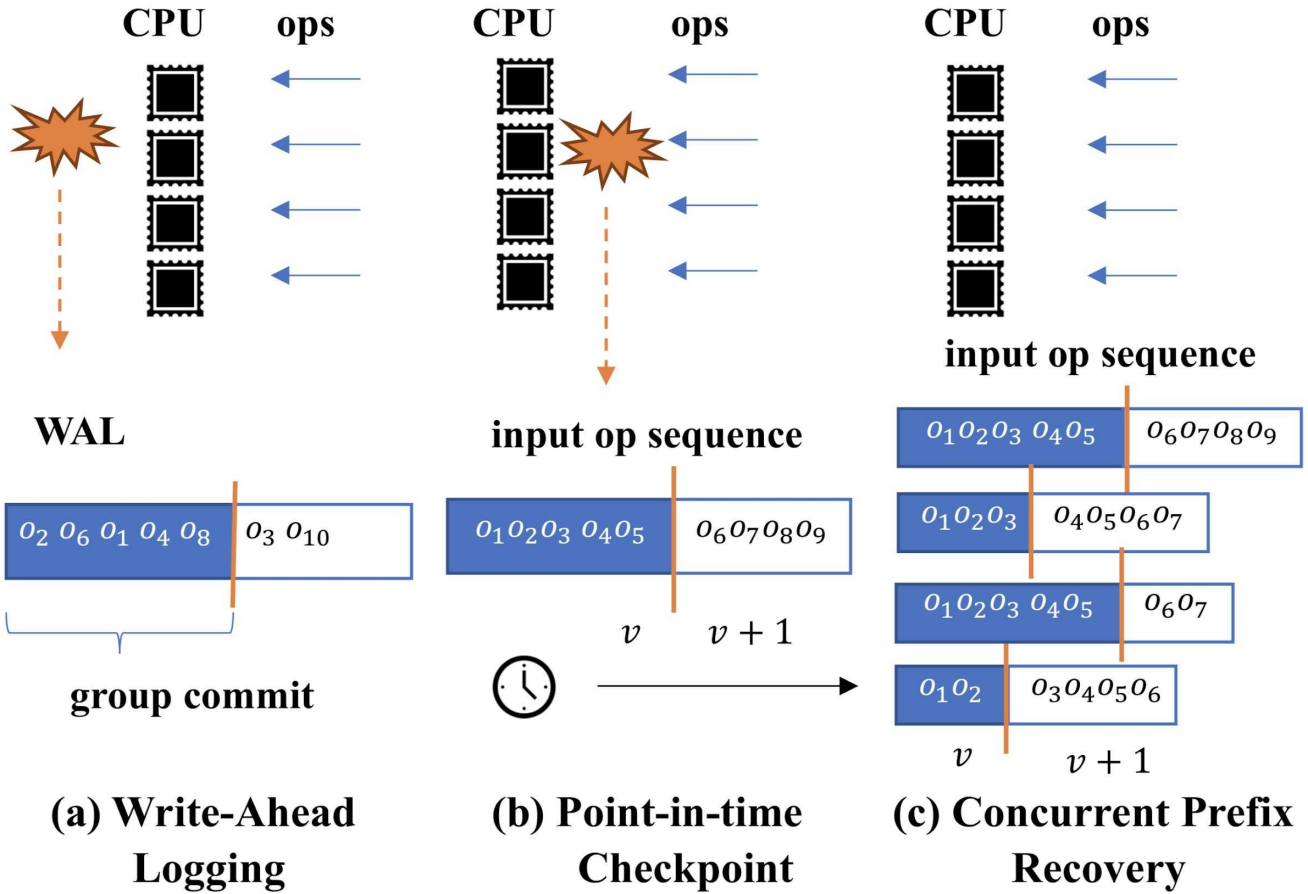


Figure 1: Approaches to Durability

Concurrent Prefix Recovery

- *Prefix recovery*: commit as “all operations issued up to time *t*”
  - Problem: cannot provide a prefix recovery guarantee over a global operation timeline without introducing system blocking or a central bottleneck
- *Concurrent prefix recovery*: 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
- *Asynchronous consistent checkpoints*: incremental checkpoints very quick to capture and commit due to in-place-updatable log-structured format
- Application requests commit, and the system coordinates the global construction of some commit point for each thread without losing asynchronicity or creating a central bottleneck

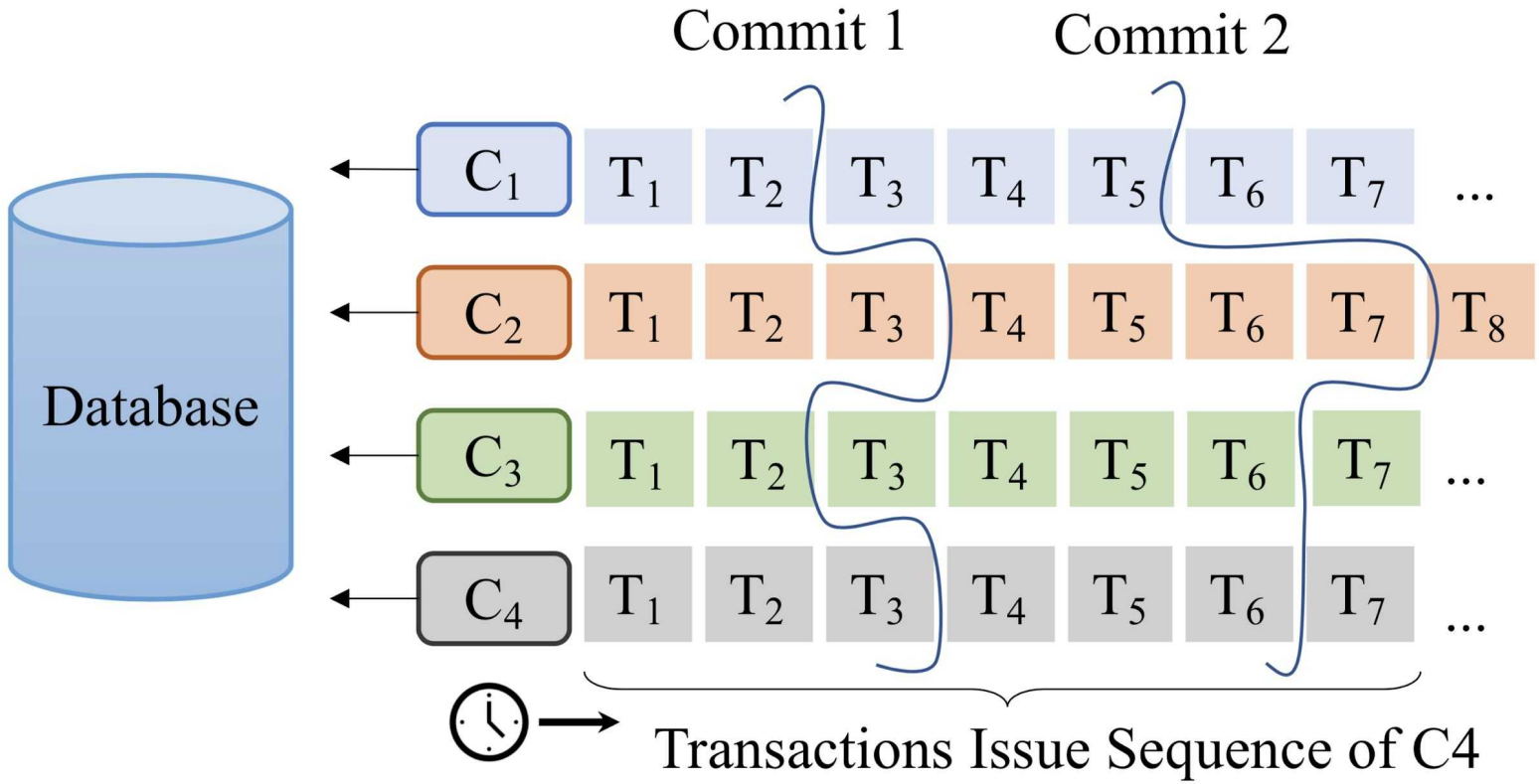


Figure 3: Concurrent Prefix Recovery Model

Epoch Framework

Epochs

- **E:** shared global epoch counter
- E_T: thread-local version of **E**
- All thread-local contexts stored in shared epoch table, one cache line/thread
- $\mathbf{E_s}$: maximal safe epoch. $\forall T : \mathbf{E_s} < E_T \leq \mathbf{E}$

Triggers

- *Drain-list* of a list of ⟨epoch, cond, action⟩ tuples

In-Memory Transactional Database

- Shared-everything, strict 2PL, no-wait
- Every record has *stable* and *live* values, with integer for current *version*
- Shared variables Global.phase and Global.version, thread local versions updated at epoch synchronization
- Different threads handle txns from different clients

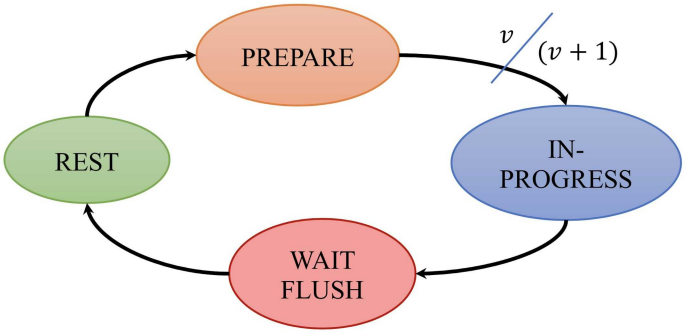


Figure 4: State Machine for CPR Commit in DB

Commit Phases

- Rest: commit requested at some version *v*, default high performance mode
- Prepare: txns in *v* run, encountering *v+1* forces abort and thread refresh
- In-Progress: txns in *v+1* run, updating record versions, copy *live* to *stable*
- Wait-Flush: capture version *v*, records at *v+1* capture *stable*, otherwise *live*

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

REST PREPARE IN-PROGRESS
WAIT-FLUSH $\otimes$ Epoch-Refresh key: $\langle$version, live, stable$\rangle$

Figure 5: Sample Execution of CPR Algorithm

Correctness

Theorem 1. The snapshot of the database 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

Recovery

- Simply load database back into memory from the latest commit
- No UNDO processing due to transactionally-consistent record captures after all *v* txns
- Database state when all txns before time t_T for every thread *T* have been committed, txns issued after t_T lost according to definition of CPR consistency

CPR in FASTER

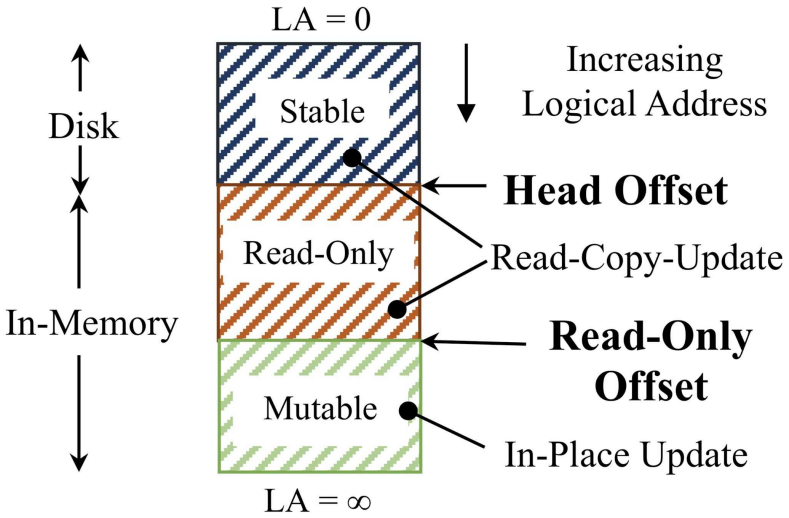
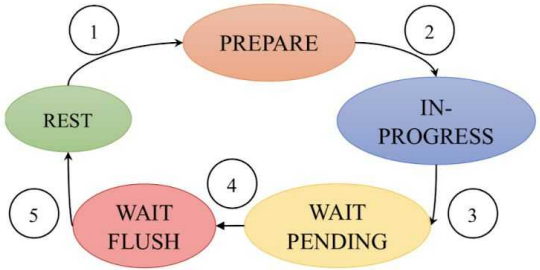


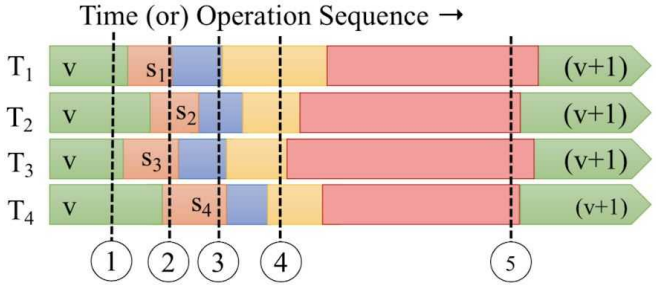
Figure 7: `HybridLog` Organization in FASTER

CPR in FASTER



① User Request to commit
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 are processed
⑤ When snapshot written to disk

(a) Global State Machine with Transition Conditions



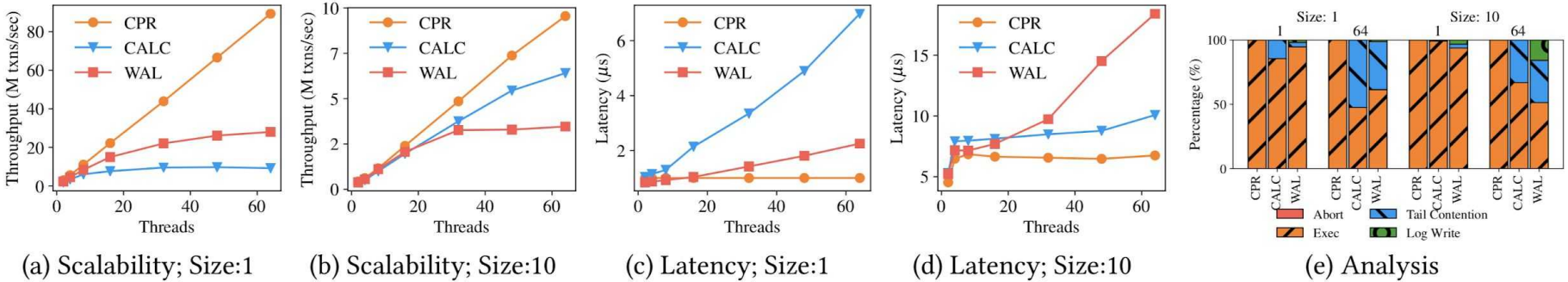
(b) FASTER Threads during checkpoint

Figure 9: Overview of CPR for FASTER

Evaluation

- In-memory transactional database
  - CPR, CALC, and WAL
  - Data store provided by FASTER
- FASTER with CPR
- YCSB

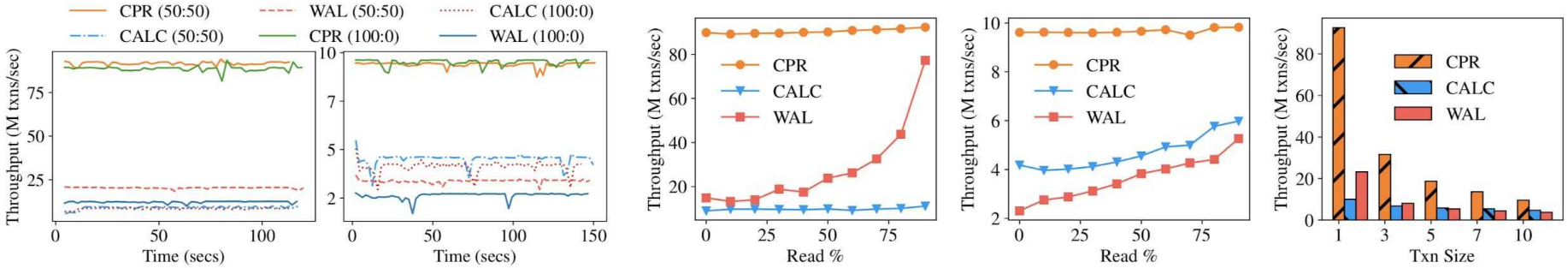
In-memory transactional database



(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

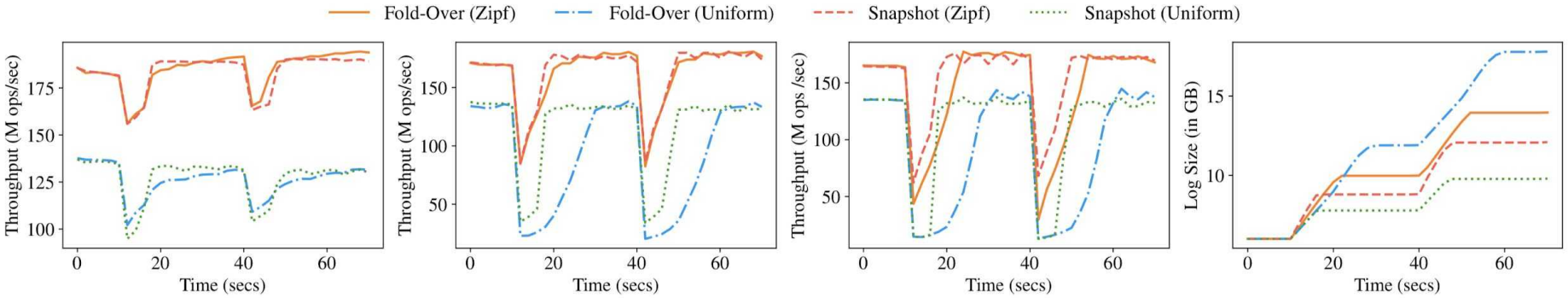
In-memory transactional database (continued)



(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

FASTER



(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

Related Work

- Write-Ahead Logging
  - 12% of total time in OLTP workload
  - SiloR: decentralized redo logs, but expensive at scale
- Distributed logging:
  - Lomet: private redo logs, infeasible in multi-socket scenarios due to dirt pages writes during migrations
  - Johnson: distributed logging approach does not solve expensive log writes
- Transactionally-Consistent Checkpoints
  - VoltDB: asynchronous checkpointing expensive on update-intensive workloads
  - CALC: atomic commit log creates serial bottleneck that limits scalability