


Fast Databases with Fast Durability and Recovery Through Multicore Parallelism

Wenting Zheng and Stephen Tu, Massachusetts Institute of Technology;
Eddie Kohler, Harvard University;
Barbara Liskov, Massachusetts Institute of Technology

OSDI’14

12/07/2022

Overview

- Background
- Silo Overview
- SiloR: Logging
- SiloR: Checkpoints
- SiloR: Recovery
- Evaluation

Background

- Crash Recovery
  - Ensure database consistency, transaction atomicity, and durability despite failures
  - Three parts
    - Log: actions during normal transaction processing to ensure that the DBMS can recover from a failure
    - Checkpoint: periodic actions to truncate logs
    - Recovery: actions after a failure to recover the database

Silo Overview

- A variant of OCC
  - Lock write-set
  - Validate read-set
  - Merge modifications into db
- Epoch
  - Global epoch number E
  - Workers user E to compute new TID
  - Serialization point

SiloR: Logging

- Basic logging
  - "Workers" run txns
  - "Loggers" handle logging and checkpointing (separate logging threads)

  - A log record: <TID, table, key, value>

  - Log flow
    - Each worker constructs log records in disk format when committing txns (stored in a memory buffer taken from a per-worker buffer pool)
    - When a buffer fills or at an epoch boundary, worker passes the buffer to logger (over a shared-memory queue)
    - Logger commits logs to disk via fsync (allow worker to send txn results to clients)

SiloR: Logging

- Value logging vs. operation logging
  - Value logging
    - Disadvantage: more log data, might slow txn execution
    - Advantage: easy to replay in parallel —— the largest TID per value wins
  - Operation logging
    - Disadvantage: txns need to be replayed in original serial order and hard to parallelize
    - Advantage: Big txns may generate fewer log data
  - Disadvantage solving
    - Add hardware and use wisely

SiloR: Logging

- Workers and loggers
  - CPU work: much vs. less —— many worker threads vs. few logger threads
  - One logger thread per disk
  - Threads mapping
    - Each logger a partition of db
      - Workers do more work to analyze txns and split updates appropriately
      - Every worker may have to communicate with every logger (remote writes or reads)
    - Each worker subset to one logger
      - Core pinning is used to ensure that a logger and its workers run on the same socket
      - Log buffers allocated on a socket are only accessed by that socket

SiloR: Logging

- Buffer management
  - Loggers allocate a maximum number of buffers per worker core
  - Worker flushes a buffer to its logger when
    - the buffer is full, or
    - a new epoch begins
  - Each log buffer is 512KB
    - Big enough to obtain some benefit from batching
    - Small enough to avoid wasting much space when a partial buffer is flushed

SiloR: Logging

- File management
  - *data.log* : current log file
  - Every 100 epochs, logger
    - Renames data.log to *old_data.e* (*e* is the largest epoch the file contains)
    - Starts a new *data.log*
  - A distinguished logger thread
    - Maintains another file *pepoch* (containing the current persistent epoch)
    - Ensures that all txns in epochs <= *pepoch* are durably stored in some log

SiloR: Logging

- Calculating $pepoch$
  - 1. Each worker w
    - Advertises its current epoch e_w and guarantees that all future txns it sends to its logger will have epoch $\geq e_w$
    - Updates e_w by setting $e_w \leftarrow E$ after flushing its current log buffer to its logger
  - 2. Each logger l
    - Reads log buffers from workers and writes them to log files
  - 3. Each logger
    - Regularly decides to make its writes durable
    - At that point, it calculates the minimum of the e_w for each of its workers and the epoch number of any log buffer it owns that remains to be written
    - This is the logger’ s current epoch e_l
    - Synchronizes all its writes to disk

SiloR: Logging

- Calculating *pepoch*
  - 4. The logger
    - Publishes e_l after completing synchronization
    - This guarantees that all associated transactions with epoch < e_l have been durably stored for this logger’ s workers
  - 5. The distinguished logger thread
    - Periodically computes a persistence epoch $e_p = \min\{ e_l \} - 1$ over all loggers
    - Writes e_p to the *pepoch* file and then synchronizes that write to disk
  - 6. The distinguished logger thread
    - Publishes e_p to a global variable once *pepoch* is durably stored
    - At that point all txns with epochs ≤ e_p have become durable and workers can release their results to clients

SiloR: Checkpoints

- Overview
  - Checkpointer threads (one per checkpoint disk)
  - Logs and checkpoints are stored on the same disks
  - Loggers and checkpointers execute on the same cores

  - A distinguished checkpoint manager
    - Assign slices to checkpointers
    - Each checkpointer: 1/n of db (n disks)

  - $[e_l, e_h]$: Each checkpointer walks over its assigned database slices in key order, writing records as it goes.

SiloR: Checkpoints

- Inconsistent checkpoint
  - Concurrent txns continue to execute during the checkpoint period
  - Both restore a checkpoint and replay a portion of log

- An important optimization
  - Checkpointer threads skip any records with current epoch >= e_l
  - Given any checkpoint started in e_l , it is necessary to replay the log [e_l , e_x], e_x >= e_h

SiloR: Checkpoints

- Writing the checkpoint
  - Checkpointers walk over index trees
  - All checkpointers walk over all tables
  - Partition the keys of each table into n subranges, one per checkpointer
  - For each table, each checkpointer divides its assigned key range into m files (stored on the same disk)
  - Blocks are assigned to files in round-robin order

SiloR: Checkpoints

- Writing the checkpoint
  - Checkpointer manager thread
    - Start a new checkpoint every C seconds
    - Record the partition for each table into a shared array
    - Record e_l
    - Start n checkpointer threads, per one disk
  - Checkpointer thread
    - Construct a block of record using a range scan on index tree
    - Record format: <TID, key, value>
    - Write block to checkpoint file (next block to next file in round-robin order)
    - Every 32MB writes, one fsync
    - …
    - Final fsync
    - Report current epoch E to manager

SiloR: Checkpoints

- Writing the checkpoint
  - Checkpointer manager thread
    - After all checkpointers have reported, computes e_h (maximum epoch reported)
    - Wait until $e_h < e_p$
    - Install checkpoint on disk by writing a final record to a special checkpoint file (record e_l , e_h and checkpoint metadata such as table/checkpoint file names)

SiloR: Checkpoints

- Cleanup
  - Remove previous checkpoints and log files that contain only txns with epochs < e_l
  - Next checkpoint is begun roughly 10 seconds
    - Minimize log replay by taking frequent checkpoints

SiloR: Recovery

- Checkpoint recovery
  - A recovery manager thread
    - Read the latest checkpoint metadata file
  - Recovery is carried out by n * m threads
    - Each thread read and process T files from disk (one file per index tree)
    - For each <TID, key, value>, the key is inserted in the index tree identified by the file name, with the given value and TID
    - Threads reconstruct the tree in parallel with little interference and benefit from locality

SiloR: Recovery

- Log recovery
  - Manager thread
    - Read the $pepoch$ file to obtain e_p (the most recent persistent epoch)
    - Read the directory for each disk and create a varible L_d per disk (to track which log files from that disk have been processed)
    - Start up g log processor threads for each disk
    - g = ⌈ N / n ⌉ (N cores and n disks)
  - Log processor thread
    - Update L_d for its disk
    - If L_d <= 0, the thread has no more work to do, informs manager and stops
    - Otherwise, read next file

SiloR: Recovery

- Log recovery
  - Log processor thread
    - Process the files in the opposite order than they were written (avoid overwriting)
    - Read entries in the file sequentially
      - Each entry contain a TID t and a set of table/key/value tuples
      - If t contains an epoch number < e_l or > e_p , skip the entry
      - Otherwise, insert a record into the table if its key isn’ t there yet
      - When a version of the record is already in the table, overwrite if it has a larger TID

Evaluation

- Experimental setup
  - Four 8-core Intel Xeon E7-4830 processors clocked at 2.1 GHz
  - Cache size
    - Each core has a private 32 KB L1 cache and a private 256 KB L2 cache
    - The eight cores on a single processor share a 24 MB L3 cache
  - 256 GB of DRAM with 64 GB of DRAM attached to each socket
  - Without networked clients
  - 3 separate Fusion ioDrive2 flash drives and 1 RAID-5 disk array
  - Measurement
    - SiloR: 28 worker threads
    - LogSilo: 28 worker threads
    - MemSilo: 32 worker threads

Evaluation

- Key-value workload

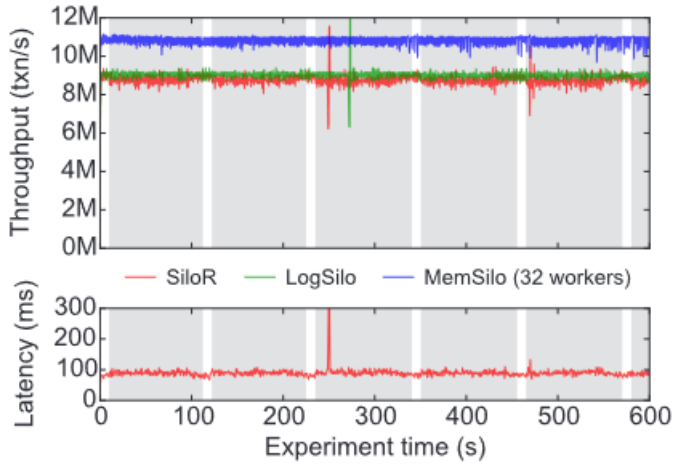

Figure 1: Throughput and latency of SiloR, and throughput of LogSilo and MemSilo, on our modified YCSB benchmark. Average throughput was 8.76 Mtxn/s, 9.01 Mtxn/s, and 10.83 Mtxn/s, respectively. Average SiloR latency was 90 ms/txn. Database size was 43.2 GB. Grey regions show those times when the SiloR experiment was writing a checkpoint.

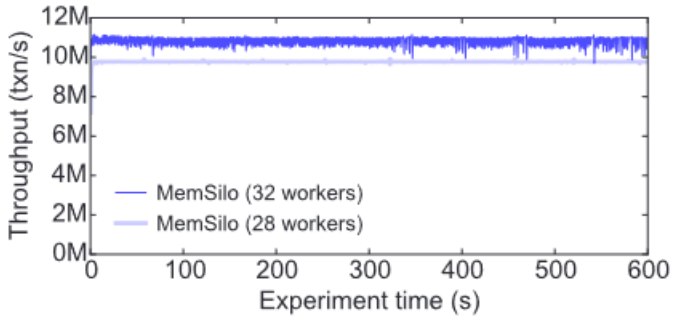

Figure 2: Throughput of MemSilo on YCSB with 32 and 28 workers. Average throughput was 10.83 Mtxn/s and 9.77 Mtxn/s, respectively.

	Read/Write	Record Size
YCSB-A	50/50	1000B
Modified	70/30	100B

Evaluation

- Key-value workload

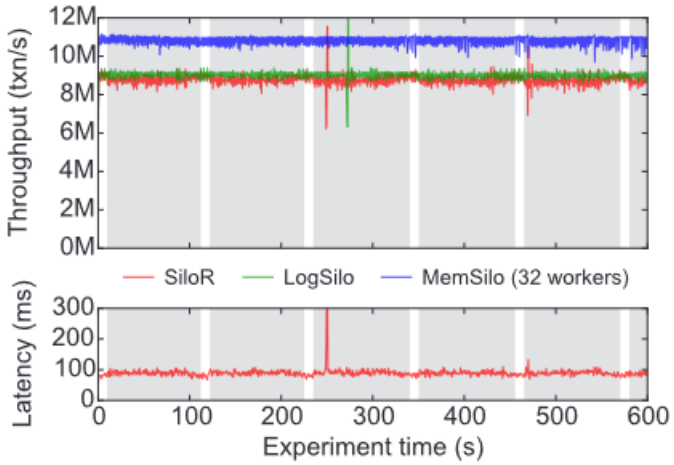

Figure 1: Throughput and latency of SiloR, and throughput of LogSilo and MemSilo, on our modified YCSB benchmark. Average throughput was 8.76 Mtxn/s, 9.01 Mtxn/s, and 10.83 Mtxn/s, respectively. Average SiloR latency was 90 ms/txn. Database size was 43.2 GB. Grey regions show those times when the SiloR experiment was writing a checkpoint.

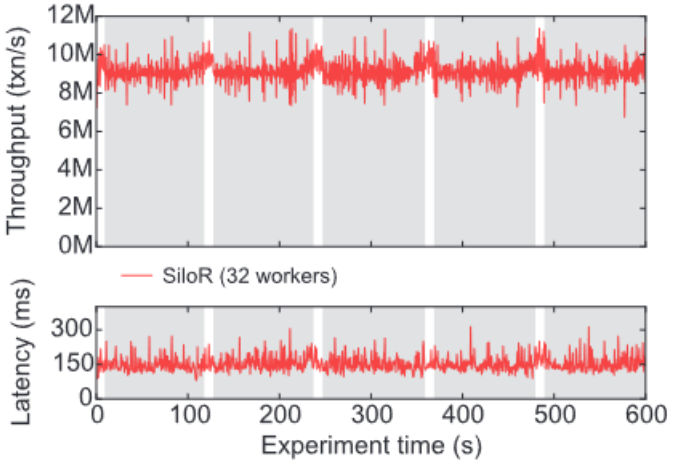

Figure 3: Throughput and latency of SiloR on YCSB with 32 workers. Average throughput was 9.14 Mtxn/s and average latency 153 ms.

Evaluation

- Importance of regular synchronization

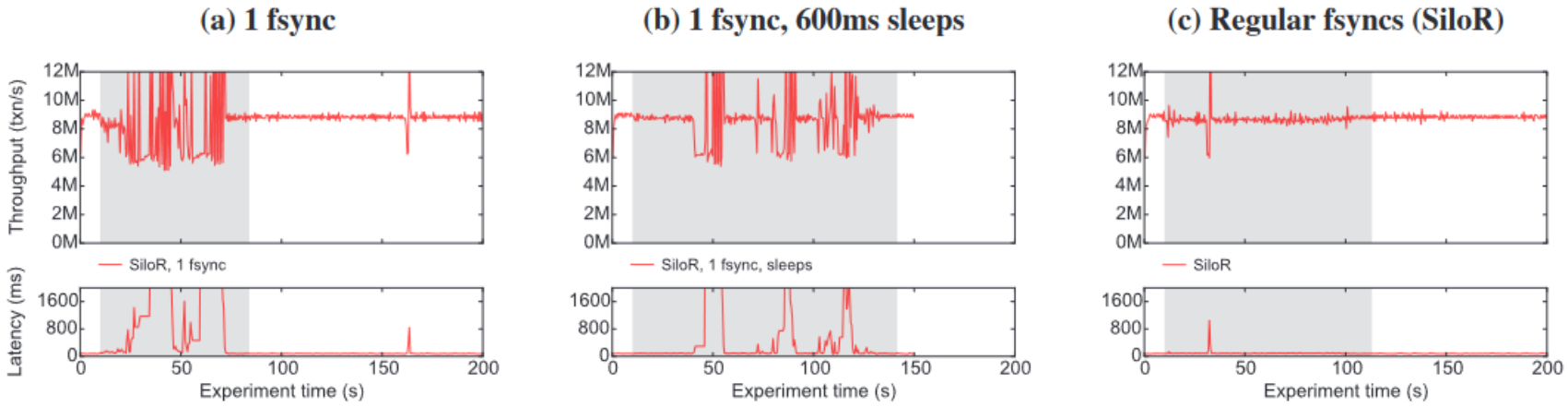

Figure 4: Importance of regular disk synchronization. In (a), one fsync call synchronizes the checkpoint; throughput and latency are extremely bursty (note the latency axis tops out at 2 sec). In (b), regular sleep calls in the checkpoint threads reduce burstiness, but do not eliminate it. In (c), SiloR, regular calls to fsync almost entirely eliminate burstiness. Here we run the modified YCSB benchmark.

Evaluation

- Compression
  - We also experimented with compressing the database checkpoints via lz4 before writing to disk
  - This didn’ t help either latency or throughput, and it actually slowed down the time it took to checkpoint
  - Our storage is fast enough that the cost of checkpoint compression outweighed the benefits of writing less data

Evaluation

- Online transaction processing workload

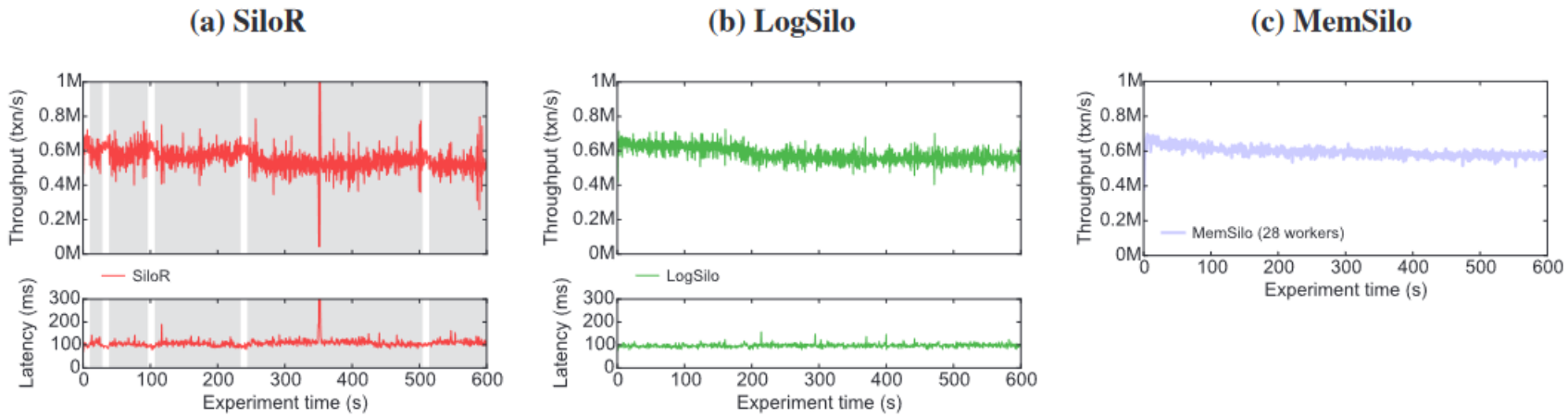

Figure 5: Throughput and latency of SiloR and LogSilo, and throughput of MemSilo, on a modified TPC-C benchmark. Average throughput is 548 Ktxn/s, 575 Ktxn/s, and 592 Ktxn/s, respectively. Average SiloR latency is 110 ms/txn; average LogSilo latency is 97 ms/txn. The database initially contains 2 GB of record data, and grows to 94 GB by the end of the experiment. All experiments run 28 workers.

Evaluation

- Recovery
  - Environment
    - Crash the database immediately before a checkpoint completes
    - 6 threads per disk (24 threads total) to restore the checkpoint
    - 8 threads per disk (32 threads total) to recover the log

Evaluation

- Recovery
  - YCSB-A
    - Recover 36 GB of checkpoint and 64 GB of log to recreate a 43.2 GB database
    - Recovery takes 106 s, or about 1.06 s/GB of recovery data
    - 31% of this time (33 s) is spent on the checkpoint and the rest (73 s) on the log
  - TPC-C
    - Stop a SiloR run of TPC-C immediately before its fourth checkpoint completes
    - 72.2 GB of record data (not including keys)
    - Recover 15.7 GB of checkpoint and 180 GB of log to recreate this database
    - Recovery takes 211 s, or about 1.08 s/GB of recovery data
    - 8% of this time (17 s) is spent on the checkpoint and the rest (194 s) on the log
  - Recovery time is proportional to the amount of data that must be read to recover
  - Log replay is the limiting factor in recovery