Fast Databases with Fast Durability and Recovery Through Multicore Parallelism

Wenting Zheng, *MIT** Stephen Tu, *MIT**
Eddie Kohler, *Harvard University* Barbara Liskov, *MIT*

checkpoint can hurt performance

Abstract

Multicore in-memory databases for modern machines can support extraordinarily high transaction rates for online transaction processing workloads. A potential weakness, however, is recovery from crash failures. Can classical techniques, such as checkpoints, be made both efficient enough to keep up with current systems' memory sizes and transaction rates, and smart enough to avoid additional contention? Starting from an efficient multicore database system, we show that naive logging and checkpoints make normal-case execution slower, but that frequent disk synchronization allows us to keep up with many workloads with only a modest reduction in throughput. We design throughout for parallelism: during logging, during checkpointing, and during recovery. The result is fast. Given appropriate hardware (three SSDs and a RAID), a 32-core system can recover a 43.2 GB key-value database in 106 seconds, and a $>$ 70 GB TPC-C database in 211 seconds.

1 Introduction

In-memory databases on modern multicore machines [10] can handle complex, large transactions at millions to tens of millions of transactions per second, depending on transaction size. A potential weakness of such databases is robustness to crashes and power failures. Replication can allow one site to step in for another, but even replicated databases must write data to persistent storage to survive correlated failures, and performance matters for both persistence and recovery.

Crash resistance mechanisms, such as logging and checkpointing, can enormously slow transaction execution if implemented naively. Modern fast in-memory databases running tens of millions of small transactions per second can generate more than 50 GB of log data per minute when logging either values or operations. In terms of both transaction rates and log sizes, this is up to several orders of magnitude more than the values reported in previous studies of in-memory-database durability [2, 14, 24]. Logging to disk or flash is at least theoretically fast, since log writes are sequential, but sequential log replay is not fast on a modern multicore machine. Checkpoints are also required, since without them, logs would grow without bound, but checkpoints require a walk over the entire database, which can cause data movement and cache pollution that reduce concurrent transaction performance. Recovery of a multi-gigabyte database using a single core could take more than 90 minutes on today's machines, which is a long time even in a replicated system.

Our goal in this work was to develop an in-memory database with full persistence at relatively low cost to transaction throughput, and with fast recovery, meaning we hoped to be able to recover a large database to a transactionally-consistent state in just a few minutes without replication. Starting from Silo [27], a very fast in-memory database system, we built *SiloR*, which adds logging, checkpointing, and recovery. Using a combination of logging and checkpointing, we are able to recover a 43.2 GB YCSB key-value-style database to a transactionally-consistent snapshot in 106 seconds, and a more complex $>$ 70 GB TPC-C database with many tables and secondary indexes in 211 seconds.

Perhaps more interesting than our raw performance is the way that performance was achieved. We used concurrency in all parts of the system. The log is written concurrently to several disks, and a checkpoint is taken by several concurrent threads that also write to multiple disks. Concurrency was crucial for recovery, and we found that the needs of recovery drove many of our design decisions. The key to fast recovery is using all of the machine's resources, which, on a modern machine, means using all cores. But some designs tempting on the logging side, such as operation logging (that is, logging transaction types and arguments rather than logging values), are difficult to recover in parallel. This drive for fast parallel recovery affected many aspects of our logging and checkpointing designs.

Operation logs are difficult to recover in parallel? Why?

Starting with an extremely fast in-memory database, we show:

- All the important durability mechanisms can and should be made parallel.
- Checkpointing can be fast without hurting normal transaction execution. The fastest checkpoints introduce undesired spikes and crashes into concurrent throughput, but through good engineering and by pacing checkpoint production, this variability can be reduced enormously.

*Currently at University of California, Berkeley.

- Even when checkpoints are taken frequently, a high-throughput database will have to recover from a very large log. In our experiments, log recovery is the bottleneck; for example, to recover a 35 GB TPC-C database, we recover 16 GB from a checkpoint and 180 GB from the log, and log recovery accounts for 90% of recovery time. Our design allows us to accomplish log replay at roughly the maximum speed of I/O.

- The system built on these ideas can recover a relatively large database quite quickly.

2 Silo overview

We build on Silo, a fast in-memory relational database that provides tables of typed records. Clients issue one-shot requests: all parameters are available when a request begins, and the request does not interact with its caller until it completes. A request is dispatched to a single database *worker* thread, which carries it out to completion (commit or abort) without blocking. Each worker thread is pinned to a physical core of the server machine. Most cores run workers, but SiloR reserves several cores for logging and checkpointing tasks.

Silo tables are stored in efficient, cache-friendly concurrent B-trees [15]. Each table uses one primary tree and zero or more secondary trees for secondary indexes. Key data is embedded in tree structures, and values are stored in separately-allocated *records*. All structures are stored in shared memory, so any worker can access the entire database.

Silo uses a variant of optimistic concurrency control (OCC) [11] to serialize transactions. Concurrency control centers on *transaction IDs* (TIDs). Each record contains the TID of the transaction that most recently modified it. As a worker runs a transaction, it maintains a *read-set* containing the old TID of each read or written record, and a *write-set* containing the new state of each written record. On transaction completion, a worker determines whether the transaction can commit. First it locks the records in the write-set (in a global order to avoid deadlock). Then it computes the transaction's TID; this is the serialization point. Next it compares the TIDs of records in the read-set with those records' current TIDs, and aborts if any TIDs have changed or any record is locked by a different transaction. Otherwise it commits and overwrites the write-set records with their new values and the new TID.

2.1 Epochs

Silo transaction IDs differ in an important way from those in other systems, and this difference impacts the way SiloR does logging and recovery. Classical OCC obtains the TID for a committing transaction by effectively incrementing a global counter. On modern multicore hardware, though, any global counter can become a source of performance-limiting contention. Silo eliminates this contention using time periods called *epochs* that are embedded in TIDs. A global epoch number E is visible to all threads. A designated thread advances it periodically (every 40 ms). Worker threads use E during the commit procedure to compute the new TID. Specifically, the new TID is (a) greater than any TID in the read-set, (b) greater than the last TID committed by this worker, and (c) in epoch E.

This avoids false contention on a global TID, but fundamentally changes the relationship between TIDs and the serial order. Consider concurrent transactions T1 and T2 where T1 reads a key that T2 then overwrites. The relationship between T1 and T2 is called an *anti-dependency*: T1 must be ordered before T2 because T1 depends on *the absence* of T2. In conventional OCC, whose TIDs capture anti-dependencies, our example would always have TID(T1) $<$ TID(T2). But in Silo, there is no communication whatsoever from T1 to T2, and we could find TID(T1) $>$ TID(T2)! This means that replaying a Silo database's committed transactions in TID order might recover the wrong database.

Epochs provide the key to correct replay. On total-store-order (TSO) architectures like x86-64, the designated thread's update of E becomes visible at all workers simultaneously. Because workers read the current epoch at the serialization point, the ordering of TIDs *with different epochs* is always compatible with the serial order, even in the case of anti-dependencies. Epochs allow for a form of group commit: SiloR persists and recovers in units of epochs. We describe below how this impacts logging, checkpointing, and recovery.

3 Logging

This section explains how SiloR logs transaction modifications for persistence. Our design builds on Silo, which included logging but did not consider recovery, log truncation, or checkpoints. The SiloR logging subsystem adds log truncation, makes changes related to liveness, and allows more parallelism on replay.

3.1 Basic logging

The responsibility for logging in SiloR is split between workers, which run transactions, and separate logging threads ("loggers"), which handle only logging, checkpointing, and other housekeeping tasks. Workers generate log records as they commit transactions; they pass these records to loggers, which commit the logs to disk. When a set of logs is committed to disk via fsync, the loggers inform the workers. This allows workers to send transaction results to clients.

A log record comprises a committed transaction's TID plus the table, key, and value information for all records modified by that transaction. Each worker constructs log records in disk format and stores them in a memory buffer taken from a per-worker buffer pool. When a buffer fills, or at an epoch boundary, the worker passes the buffer to the logger over a shared-memory queue.

3.2 Value logging vs. operation logging

SiloR uses value logging, not operation or transaction logging. This means that SiloR logs contain each transaction's output keys and values, rather than the identity of the executed operation and its parameters.

The choice of value logging is an example of recovery parallelism driving the normal-case logging design. Value logging has an apparent disadvantage relative to operation logging: for many workloads (such as TPC-C) it logs more data, and therefore might unnecessarily slow transaction execution. However, from the point of view of recovery parallelism, the advantages of value logging outweigh its disadvantages. Value logging is easy to replay in parallel—the largest TID per value wins. This works in SiloR because TIDs reflect dependencies, i.e., the order of writes, and because we recover in units of epochs, ensuring that anti-dependencies are not a problem. Operation logging, in contrast, requires that transactions be replayed in their original serial order. This is always hard to parallelize, but in Silo, it would additionally require logging read-sets (keys and TIDs) to ensure anti-dependencies were obeyed. Operation logging also requires that the initial pre-replay database state be a transactionally consistent snapshot, which value logging does not; and for small transactions value and operation logs are about the same size. These considerations led us to prefer value logging in SiloR. We solve the problem of value logging I/O by adding hardware until logging is not a bottleneck, and then using that hardware wisely.

3.3 Workers and loggers

Loggers have little CPU work to do. They collect logs from workers, write them to disk, and await durability notification from the kernel via the fsync/fdatasync system call. Workers, of course, have a lot of CPU work to do. A SiloR deployment therefore contains many worker threads and few logger threads. We allocate enough logger threads per disk to keep that disk busy, one per disk in our evaluation system.

But how should worker threads map to logger threads? One possibility is to assign each logger a partition of the database. This might reduce the data written by loggers (for example, it could improve the efficacy of compression), and it might speed up replay. We rejected this design because of its effect on normal-case transaction execution. Workers would have to do more work to analyze transactions and split their updates appropriately. More fundamentally, every worker might have to communicate with every logger. Though log records are written in batches (so the communication would not likely introduce contention), this design would inevitably introduce *remote writes or reads*: physical memory located on one socket would be accessed, either for writes or reads, by a thread running on a different socket. Remote accesses are expensive and should be avoided when possible.

Our final design divides workers into disjoint subsets, and assigns each subset to exactly one logger. Core pinning is used to ensure that a logger and its workers run on the same socket, making it likely that log buffers allocated on a socket are only accessed by that socket.

3.4 Buffer management

Although loggers should not normally limit transaction execution, loggers must be able to apply backpressure to workers, so that workers don't generate indefinite amounts of log data. This backpressure is implemented by buffer management. Loggers allocate a maximum number of log buffers per worker core. Buffers circulate between loggers and workers as transactions execute, and a worker blocks when it needs a new log buffer and one is not available. A worker flushes a buffer to its logger when either the buffer is full or a new epoch begins, whichever comes first. It is important to flush buffers on epoch changes, whether or not those buffers are full, because SiloR cannot mark an epoch as persistent until it has durably logged *all* transactions that happened in that epoch. Each log buffer is 512 KB. This is big enough to obtain some benefit from batching, but small enough to avoid wasting much space when a partial buffer is flushed.

We found that log-buffer backpressure in Silo triggered unnecessarily often because it was linked with fsync times. Loggers amplified file system hiccups, such as those caused by concurrent checkpoints, into major dips in transaction rates. SiloR's loggers instead recirculate log buffers back to workers as soon as possible—after a write, rather than after the following epoch change and fsync. We also increased the number of log buffers available to workers, setting this to about 10% of the machine's memory. The result was much less noise in transaction execution rates.

3.5 File management

Each SiloR logger stores its log in a collection of files in a single directory. New entries are written to a file called **data.log**, the current log file. Periodically (currently every 100 epochs) the logger renames this file to **old_data.***e*, where *e* is the largest epoch the file contains, then starts a new **data.log**. Using multiple files simplifies the process of log truncation and, in our measure-

ments, didn't slow logging relative to Silo's more primitive single-file design.

Log files do not contain transactions in serial order. A log file contains concatenated log buffers from several workers. These buffers are copied into the log without rearrangement; in fact, to reduce data movement, SiloR logger threads don't examine log data at all. A log file can even contain *epochs* out of order: a worker that delays its release of the previous epoch's buffer will not prevent other workers from producing buffers in the new epoch. All we know is that a file **old_data.***e* contains no records with epochs $> e$. And, of course, a full log comprises multiple log directories stored independently by multiple loggers writing to distinct disks. Thus, no single log contains enough information for recovery to produce a correct database state. It would be possible to extract this information from *all* logs, but instead SiloR uses a distinguished logger thread to maintain another file, **pepoch**, that contains the current *persistent epoch*. The logger system guarantees that all transactions in epochs $\leq$ **pepoch** are durably stored in some log. This epoch is calculated as follows:

1. Each worker w advertises its *current* epoch, e_w, and guarantees that all future transactions it sends to its logger will have epoch $\geq e_w$. It 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. The logger then synchronizes all its writes to disk.

4. After this synchronization completes, the logger publishes e_l. 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 as $\min\{e_l\} - 1$ over all loggers. It writes e_p to the **pepoch** file and then synchronizes that write to disk.

6. Once **pepoch** is durably stored, the distinguished logger thread publishes e_p to a global variable. At that point all transactions with epochs $\leq e_p$ have become durable and workers can release their results to clients.

This protocol provides a form of group commit. It ensures that the logs contain all information about transactions in epochs $\leq e_p$, and that no results from transactions with epoch $> e_p$ were released to clients. Therefore it is safe for recovery to recover all transactions with epochs $\leq e_p$, and also necessary since those results may have been released to clients. It has one important disadvantage, namely that the critical path for transaction commit contains two fsyncs (one for the log file and one for **pepoch**) rather than one. This somewhat increases latency.

4 Checkpoints

Although logs suffice to recover a database, they do not suffice to recover a database in bounded time. In-memory databases must take periodic *checkpoints* of their state to allow recovery to complete quickly, and to support log truncation. This section describes how SiloR takes checkpoints.

4.1 Overview

Our main goal in checkpoint production is to produce checkpoints as quickly as possible without disrupting worker throughput. Checkpoint speed matters because it limits the amount of log data that will need to be replayed at recovery. The smaller the distance between checkpoints, the less log data needs to be replayed, and we found the size of the log to be the major recovery expense. Thus, as with log production, checkpointing uses multiple threads and multiple disks.

Checkpoints are written by *checkpointer* threads, one per checkpoint disk. In our current implementation checkpoints are stored on the same disks as logs, and loggers and checkpointers execute on the same cores (which are separate from the worker cores that execute transactions). Different checkpointers are responsible for different slices of the database; a distinguished *checkpoint manager* assigns slices to checkpointers. Each checkpointer's slices amount to roughly $1/n$th of the database, where n is the number of disks. A checkpoint is associated with a range of epochs $[e_l, e_h]$, where each checkpointer started its work during or after e_l and finished its work during or before e_h.

Each checkpointer walks over its assigned database slices in key order, writing records as it goes. Since OCC installs modifications at commit time, all records seen by checkpointers are committed. This means that full ARIES-style undo and redo logging is unnecessary; the log can continue to contain only "redo" records for committed transactions. However, concurrent transactions continue to execute during the checkpoint period, and they do not coordinate with checkpointers except via per-record locks. If a concurrent transaction commits multiple modifications, there is no guarantee the checkpointers will see them all. SiloR checkpoints are thus inconsistent or "fuzzy": the checkpoint is not necessarily

a consistent snapshot of the database as of a particular point in the serial order. To recover a consistent snapshot, it is always necessary both to restore a checkpoint and to replay at least a portion of the log.

We chose to produce an inconsistent checkpoint because it's less costly in terms of memory usage than a consistent checkpoint. Silo could produce consistent checkpoints using its support for snapshot transactions [27]. However, checkpoints of large databases take a long time to write (multiple tens of seconds), which is enough time for all database records to be overwritten. The memory expense associated with preserving the snapshot for this period, and especially the allocation expense associated with storing new updates in newly-allocated records (rather than overwriting old records), reduces normal-case transaction throughput by 10% or so. We prefer better normal-case throughput. Our choice of inconsistent checkpoints further necessitates our choice of value logging; it is impossible to recover from an inconsistent checkpoint without either value logging or some sort of ARIES-style undo logging.

Another possible design for checkpoints is to avoid writing information about keys whose records haven't changed since the previous checkpoint, for example, designing a disk format that would allow a new checkpoint to elide unmodified key ranges. We rejected this approach because ours is simpler, and also because challenging workloads, such as uniform updates, can cause any design to effectively write a complete checkpoint every time a checkpoint is required. We wanted to understand the performance limits caused by these workloads.

In an important optimization, checkpointer threads skip any records with current epoch $\geq e_l$. Thus, the checkpoint contains those keys written in epochs $< e_l$ that were not overwritten in epochs $\geq e_l$. It is not necessary to write such records because, given any inconsistent checkpoint started in e_l, it is always necessary to replay the log starting at epoch e_l. Specifically, the log must be complete over a range of epochs $[e_l, e_x]$, where $e_x \geq e_h$, for recovery of a consistent snapshot to be possible. There's no need to store a record in the checkpoint that will be replayed by the log. This optimization reduces our checkpoint sizes by 20% or more.

4.2 Writing the checkpoint

Checkpointers walk over index trees to produce the checkpoint. Since we want each checkpointer to be responsible for approximately the same amount of work, yet tables differ in size, we have all checkpointers walk over all tables. To make the walk efficient, we partition the keys of each table into n subranges, one per checkpointer. This way each checkpointer can take advantage of the locality for keys in the tree.

The checkpoint is organized to enable efficient recovery. During recovery, *all* cores are available, so we designed the checkpoint to facilitate using those cores.

For each table, each checkpointer divides its assigned key range into m files, where m is the number of cores that would be used during recovery for that key range. Each of a checkpointer's m files are stored on the same disk. As the checkpointer walks over its range of the table, it writes blocks of keys to these m files. Each block contains a contiguous range of records, but blocks are assigned to files in round-robin order. There is a tension here between two aspects of fast recovery. On the one hand, recovery is more efficient when a recovery worker is given a continuous range of records, but on the other hand, recovery resources are more effectively used when the recovery workload is evenly distributed (each of the m files contain about the same amount of work). Calculating a perfect partition of an index range into equal-size subranges is somewhat expensive, since to do this requires tree walks. We chose a point on this tradeoff where indexes are coarsely divided among checkpointers into roughly-equal subranges, but round-robin assignment of blocks to files evens the workload at the file level.

The checkpoint manager thread starts a new checkpoint every C seconds. It picks the partition for each table and writes this information into a shared array. It then records e_l, the checkpoint's starting epoch, and starts up n checkpointer threads, one per disk. For each table, each thread creates the corresponding checkpoint files and walks over its assigned partition using a range scan on the index tree. As it walks, it constructs a block of record data, where each record is stored as a key/TID/value tuple. When its block fills up, the checkpointer writes that block to one of the checkpoint files and continues. The next full block is written to the next file in round-robin order.

Each time a checkpointer's outstanding writes exceed 32 MB, it syncs them to disk. These intermediate syncs turned out to be important for performance, as we discuss in §6.2.

When a checkpointer has processed all tables, it does a final sync to disk. It then reads the current epoch E and reports this information to the manager. When all checkpointers have reported, the manager computes e_h; this is the maximum epoch reported by the checkpointers, and thus is the largest epoch that might have updates reflected in the checkpoint. Although, thanks to our reduced checkpoint strategy, new tuples created during e_h are not stored in the checkpoint, tuples removed or overwritten during e_h are also not stored in the checkpoint, so the checkpoint can't be recovered correctly without complete logs up to and including e_h. Thus, the manager waits until $e_h \leq e_p$, where e_p is the persistence

epoch computed by the loggers (§3.5). Once this point is reached, the manager *installs* the checkpoint on disk by writing a final record to a special checkpoint file. This file records e_l and e_h, as well as checkpoint metadata, such as the names of the database tables and the names of the checkpoint files.

4.3 Cleanup

After the checkpoint is complete, SiloR removes old files that are no longer needed. This includes any previous checkpoints and any log files that contain only transactions with epochs $< e_l$. Recall that each log comprises a current file and a number of earlier files with names like **old_data**.e. Any file with $e < e_l$ can be deleted.

The next checkpoint is begun roughly 10 seconds after the previous checkpoint completed. Log replay is far more expensive than checkpoint recovery, so we aim to minimize log replay by taking frequent checkpoints. In future work, we would like to investigate a more flexible scheme that, for example, could delay a checkpoint if the log isn't growing too fast.

5 Recovery

SiloR performs recovery by loading the most recent checkpoint, then correcting it using information in the log. In both cases we use many concurrent threads to process the data and we overlap processing and I/O.

5.1 Checkpoint recovery

To start recovery, a recovery manager thread reads the latest checkpoint metadata file. This file contains information about what tables are in the system and e_l, the epoch in which the checkpoint started. The manager creates an in-memory representation for each of the T index trees mentioned in the checkpoint metadata. In addition it deletes any checkpoint files from earlier or later checkpoints and removes all log files from epochs before e_l.

The checkpoint is recovered concurrently by many threads. Recall that the checkpoint consists of many files per database table. Each table is recorded on all n disks, partitioned so that on each disk there are m files for each table. Recovery is carried out by $n \times m$ threads. Each thread reads from one disk, and is responsible for reading and processing T files from that disk (one file per index tree). Processing is straightforward: for each key/value/TID in the file, the key is inserted in the index tree identified by the file name, with the given value and TID. Since the files contain different key ranges, checkpoint recovery threads are able to reconstruct the tree in parallel with little interference; additionally they benefit from locality when processing a subrange of keys in a particular table.

5.2 Log recovery

After all threads have finished their assigned checkpoint recovery tasks, the system moves on to log recovery. As mentioned in §3, there was no attempt at organizing the log records at runtime (e.g. partitioning the log records based on what tables were being modified). Instead it is likely that each log file is a jumble of modifications to various index trees. This situation is quite different than it was for the checkpoint, which was organized so that concurrent threads could work on disjoint partitions of the database. However, SiloR uses value logging, which has the property that the logs can be processed in any order. All we require is that at the end of processing, every key has an associated value corresponding to the last modification made up through the most recent persistent epoch prior to the failure. If there are several modifications to a particular key k, these will have associated TIDs T1, T2, and so on. Only the entry with the largest of these TIDs matters; whether we happen to find this entry early or late in the log recovery step does not.

We take advantage of this property to process the log in parallel, and to avoid unnecessary allocations, copies, and work. First the manager thread reads the **pepoch** file to obtain e_p, the number of the most recent persistent epoch. All log records for transactions with TIDs for later epochs are ignored during recovery. This is important for correctness since group commit has not finished for those later epochs; if we processed records for epochs after e_p we could not guarantee that the resulting database corresponded to a prefix of the serial order.

The manager reads the directory for each disk, and creates a variable per disk, L_d, that is used to track which log files from that disk have been processed. Initially this variable is set to the number of relevant log files for that disk, which, in our experiments, is in the hundreds. Then the manager starts up g log processor threads for each disk. We use all threads during log recovery. For instance, on a machine with N cores and n disks, we have $g = \lceil N/n \rceil$. This can produce more recovery threads than there are cores. We experimented with the alternative $m = \lfloor N/n \rfloor$, but this leaves some cores idle during recovery, and we observed worse recovery times than with oversubscription.

A log processor thread proceeds as follows. First it reads, decrements, and updates L_d for its disk. This update is done atomically: this way it learns what file it should process, and updates the variable so that the next log processor for its disk will process a different file. If the value it reads from L_d is ≤ 0, the log processor thread has no more work to do. It communicates this to the manager and stops. Otherwise the processor thread reads the next file, which is the *newest* file that has not yet been processed. In other words, we process the files in the opposite order than they were written. The proces-

sor thread on disk d that first reads L_d processes the current log file **data.log**; after this files are read in reverse order by the epoch numbers contained in their names. The files are large enough that, when reading them, we get good throughput from the disk; there's little harm in reading the files out of order (i.e., in an order different from the order they were written).

The processor thread reads the entries in the file sequentially. Recall that each entry contains a TID t and a set of table/key/value tuples. If t contains an epoch number that is $< e_l$ or $> e_p$, the thread skips the entry. Otherwise, the thread inserts a record into the table if its key isn't there yet; when a version of the record is already in the table, the thread overwrites only if the log record has a larger TID.

Value logging replay has the same result no matter what order files are processed. We use reverse order for reading log files because it uses the CPU more efficiently than forward order when keys are written multiple times. When files are processed in strictly forward order, every log record will likely require overwriting some value in the tree. When files are processed in roughly reverse order, and keys are modified multiple times, then many log records don't require overwriting: the tree's current value for the key, which came from a later log file, is often newer than the log record.

5.3 Correctness

Our recovery strategy is correct because it restores the database to the state it had at the end of the last persistent epoch e_p. The state of the database after processing the checkpoint is definitely not correct: it is inconsistent, and it is also missing modifications of persistent transactions that ran after it finished. All these problems are corrected by processing the log. The log contains all modifications made by transactions that ran in epochs in e_l up through e_p. Therefore it contains what is needed to rectify the checkpoint. Furthermore, the logic used to do the rectification leads to each record holding the modification of the last transaction to modify it through epoch e_p, because we make this decision based on TIDs. And, importantly, we ignore log entries for transactions from epochs after e_p.

It's interesting to note that value logging works without having to know the exact serial order. All that is required is enough information so that we can figure out the most recent modification. That is, log record "version numbers" must capture dependencies, but need not capture anti-dependencies. Silo TIDs meet this requirement. And because TID comparison is a simple commutative test, log processing can take place in any order. In addition, of course, we require the group commit mechanism provided by epochs to ensure that anti-dependencies are also preserved.

6 Evaluation

In this section, we evaluate the effectiveness of the techniques in SiloR, confirming the following performance hypotheses:

- SiloR's checkpointer has only a modest effect on both the latency and throughput of transactions on a challenging write-heavy key-value workload and a typical online transaction processing workload.
- SiloR recovers 40–70 GB databases within minutes, even when crashes are timed to maximize log replay.

6.1 Experimental setup

All of our experiments were run on a single machine with four 8-core Intel Xeon E7-4830 processors clocked at 2.1 GHz, yielding a total of 32 physical cores. 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. The machine has 256 GB of DRAM with 64 GB of DRAM attached to each socket, and runs 64-bit Linux 3.2.0. We run our experiments without networked clients; each database worker thread runs with an integrated workload generator. We do not take advantage of our machine's NUMA-aware memory allocator, a decision discussed in §6.5.

We use three separate Fusion ioDrive2 flash drives and one RAID-5 disk array. Each disk is used for both logging and checkpointing. Each drive has a dedicated logger thread and checkpointer thread, both of which run on the same core. Within a drive, the log and checkpoint information reside in separate files. Each logger or checkpointer writes to a series of files on a single disk.

We measure three related databases, SiloR, LogSilo, and MemSilo. These systems have identical in-memory database structures. SiloR is the full system described here, including logging and checkpointing. LogSilo is a version of SiloR that only logs data: there are no checkpointer threads or checkpoints. MemSilo is Silo run without persistence, and is a later version of the system of Tu et al. [27] Unless otherwise noted, we run SiloR and LogSilo with 28 worker threads and MemSilo with 32 worker threads.

6.2 Key-value workload

To demonstrate that SiloR can log and checkpoint with low overhead, we run SiloR on a variant of YCSB workload mix A. YCSB is a popular key-value benchmark from Yahoo [4]. We modified YCSB-A to have a read/write (get/put) ratio of 70/30 (not 50/50), and a record size of 100 bytes (not 1000). This workload mix was originally designed for MemSilo to stress database internals rather than memory allocation; though the read/write ratio is somewhat less than standard YCSB-A, it is still quite high compared to most workloads. Our read and write transactions sample keys uniformly.

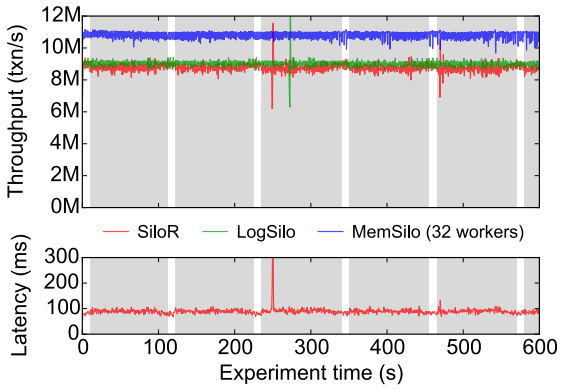

Figure 1: Throughput and latency of SiloR, and throughput of LogSilo and MemSilo, on our modified YCSB benchmark. Average throughput was 8.76 Mtxn/s, 9.01 Mtxn/s, and 10.83 Mtxn/s, respectively. Average SiloR latency was 90 ms/txn. Database size was 43.2 GB. Grey regions show those times when the SiloR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

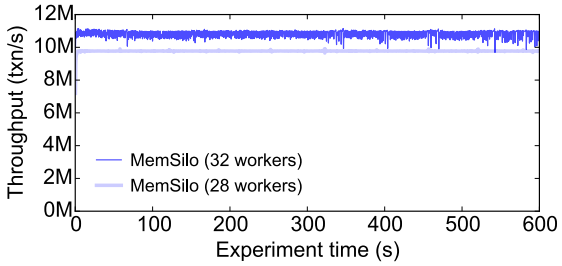

Figure 2: Throughput of MemSilo on YCSB with 32 and 28 workers. Average throughput was 10.83 Mtxn/s and 9.77 Mtxn/s, respectively.

There are 400M keys for a total database size of roughly 43.2 GB (3.2 GB of key data, 40 GB of value data).

Figure 1 shows the results over a 10-minute experiment. Checkpointing can be done concurrently with logging without greatly affecting transaction throughput. The graph shows, over the length of the experiment, rolling averages of throughput and latency with a 0.5-second averaging window. For SiloR and LogSilo, throughput and latency are measured to transaction persistence (i.e., latency is from the time a transaction is submitted to the time SiloR learns the transaction's effects are persistent). Intervals during which the checkpointer is running are shown in gray. Figure 1's results are typical of our experimental runs; Figure 6 in the appendix shows two more runs.

SiloR is able to run multiple checkpoints and almost match LogSilo's throughput. Its throughput is also close to that of MemSilo, although MemSilo does no logging or checkpointing whatsoever: SiloR achieves

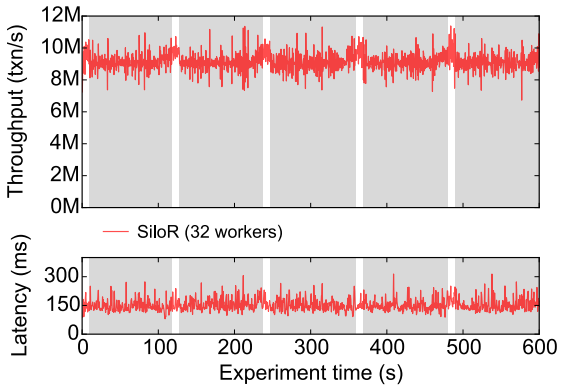

Figure 3: Throughput and latency of SiloR on YCSB with 32 workers. Average throughput was 9.14 Mtxn/s and average latency 153 ms.

8.76 Mtxn/s, 80% the average throughput of MemSilo (10.83 Mtxn/s). Average latency is affected by logging and checkpointing somewhat more significantly; it is 90 ms/transaction.[1] Some of this latency is inherent in Silo's epoch design. Since the epoch advances every 40 ms, average latency cannot be less than 20 ms. The rest is due to a combination of accumulated batching delays (workers batch transactions in log buffers, loggers batch updates to synchronizations) and delays in the persistent storage itself (i.e., the two fsyncs in the critical path each take 10–20 ms, and sometimes more). Nevertheless, we believe this latency is not high for a system involving persistent storage.

During the experiment, SiloR generates approximately 298 MB/s of IO per disk. The raw bandwidth of our Fusion IO drives is reported as 590 MB/s/disk; we are achieving roughly half of this.

SiloR and LogSilo's throughput is less than MemSilo's for several reasons, but as Figure 2 shows, an important factor is simply that MemSilo has more workers available to run transactions. SiloR and LogSilo require extra threads to act as loggers and checkpointers; we run four fewer workers to leave cores available for those threads. If we run MemSilo with 28 workers, its throughput is reduced by roughly 10% to 9.77 Mtxn/s, making up more than half the gap with SiloR. We also ran SiloR with 32 workers. This bettered the average throughput to 9.13 Mtxn/s, but CPU oversubscription caused wide variability in throughput and latency (Figure 3).

As we expect, the extensive use of group commit in LogSilo and SiloR make throughput, and particularly latency, more variable than in MemSilo. Relative to Mem-

[1] Due to a technical limitation in SiloR's logger implementation, the latency shown in the figure is the (running) average latency for *write* transactions only; we believe these numbers to be a converative upper bound on the actual latency of the system.

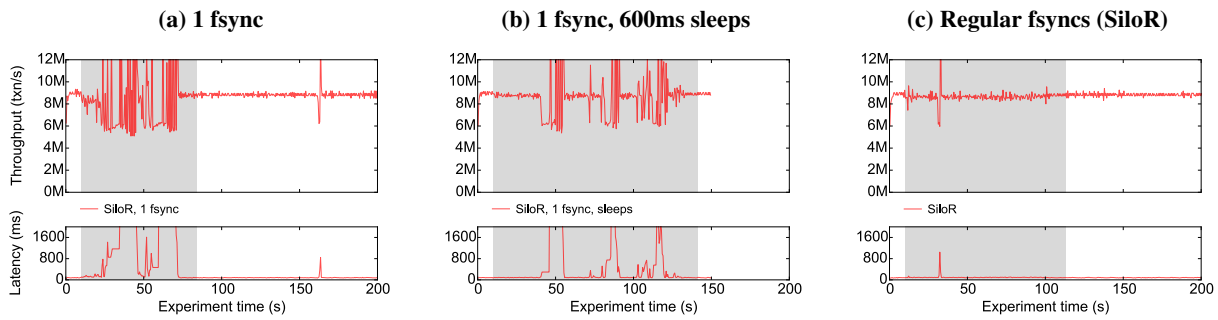

Figure 4: Importance of regular disk synchronization. In (a), one fsync call synchronizes the checkpoint; throughput and latency are extremely bursty (note the latency axis tops out at 2 sec). In (b), regular sleep calls in the checkpoint threads reduce burstiness, but do not eliminate it. In (c), SiloR,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

Silo with 28 cores, LogSilo's performance is more variable, and SiloR's more variable still. The spike in latency visible in Figure 1, which happened at one time or another in most of our runs, is discussed below in §6.5.

Importance of regular synchronization. A checkpoint is useless until it is complete, so the obvious durability strategy for a checkpointer thread is to call fsync once, after writing all checkpoint data and before reporting completion to the manager. But SiloR checkpointers call fsync far more frequently—once per 32 MB of data written. Figure 4 shows why this matters: the naive strategy, (a), is very unstable on our Linux system, inducing wild throughput swings and extremely high latency. Slowing down checkpointer threads through the occasional introduction of sleep() calls, (b), reduces the problem, but does not eliminate it. We believe that, with the single fsync, the kernel flushed old checkpoint pages only when it had to—when the buffer cache became full—placing undue stress on the rest of the system. Frequent synchronization, (c), produces far more stable results; it also can produce a checkpoint more quickly than can the version with occasional sleeps.

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.

6.3 On-line transaction processing workload

YCSB-A, though challenging, is a well-behaved workload: all records are in one table, there are no secondary indexes, accesses are uniform, all writes are overwrites (no inserts or deletes), all transactions are small. In this section, we evaluate SiloR on a more complex workload, the popular TPC-C benchmark for online transaction processing [26]. TPC-C transactions involve customers assigned to a set of districts within a local warehouse, placing orders in those districts. There are ten primary tables plus two secondary indexes (SiloR treats primary tables and secondary indexes identically). We do not model client "think" time, and we run the standard workload mix. This contains 45% "new-order" transactions, which contain 8–18 inserts and 5–15 overwrites each. Also write-heavy are "delivery" transactions (4% of the mix), which contain up to 150 overwrites and 10 removes each.[2] Unmodified TPC-C is not a great fit for an in-memory database: very few records are removed, so the database grows without bound. During our 10-minute experiments, database record size (not including keys) grows from 2 GB to 94 GB. Nevertheless, the workload is well understood and challenging for our system.

Figure 5 shows the results. TPC-C transactions are challenging enough for Silo's in-memory structures that the addition of persistence has little effect on throughput: SiloR's throughput is about 93% that of MemSilo. The MemSilo graph also shows that this workload is more inherently variable than YCSB-A. We use 28 workers for MemSilo, rather than 32, because 32-worker runs actually have lower average throughput, as well as far more variability (see Figure 7 in the appendix: our 28-worker runs achieved 587–596 Mtxn/s, our 32-worker runs 565–583 Mtxn/s). As with YCSB-A, the addition of persistence increases this variability, both by batching transactions and by further stressing the machine. (Figure 7 in the appendix shows that, for example, checkpoints can happen at quite different times.) Throughput degrades over time in the same way for all configurations. This is because the database grows over time, and Silo tables are stored in trees with height proportional to the log of the table size. The time to take a check-

[2] It is common in the literature to report TPC-C results for the standard mix as "new order transactions per minute." Following Silo, we report transactions per second for *all* transactions.

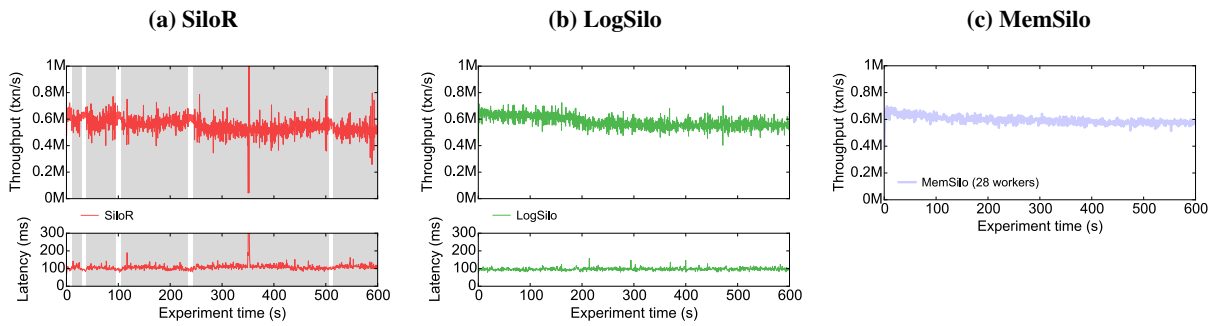

Figure 5: Throughput and latency of SiloR and LogSilo, and throughput of MemSilo, on a modified TPC-C benchmark. Average throughput is 548 Ktxn/s, 575 Ktxn/s, and 592 Ktxn/s, respectively. Average SiloR latency is 110 ms/txn; average LogSilo latency is 97 ms/txn. The database initially contains 2 GB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

point also grows with database size (3.5 s or so per GB of record data). Latency, which is 110 ms/txn average for SiloR, is higher than in YCSB-A, but not by much, even though TPC-C transactions are far more complex. In summary, SiloR can handle more complex workloads with larger transactions as well as it can handle simple workloads with small transactions.

6.4 Recovery

We now show that SiloR checkpoints allow for fast recovery. We run YCSB-A and TPC-C benchmarks, and in each case, crash the database immediately before a checkpoint completes. This maximizes the length of the log that must be recovered to restore a transactionally-correct state. We use 6 threads per disk (24 threads total) to restore the checkpoint, and 8 threads per disk (32 threads total) to recover the log.

For YCSB-A, SiloR must 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. The TPC-C database grows over time, so checkpoints have different sizes. We stop a SiloR run of TPC-C immediately before its fourth checkpoint completes, at about 465 s into the experiment, when the database contains about 72.2 GB of record data (not including keys). SiloR must 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. Thus, recovery time is proportional to the amount of data that must be read to recover, and log replay is the limiting factor in recovery, justifying our decision to checkpoint frequently.

6.5 Discussion

This work's motivation was to explore the performance limits afforded by modern hardware. However, there are other limits that SiloR would encounter in a real deployment. At the rates we are writing, our expensive flash drives would reach their maximum endurance in a bit more than a year!

In contrast with the evaluation of Silo, we disable the NUMA-aware allocator in our tests. When enabled, this allocator improves average throughput by around 25% on YCSB (to 10.91 Mtxn/s for SiloR) and 20% on TPC-C (to 644 Ktxn/s for SiloR). The cost—which we decided was not worth paying, at least for TPC-C—was performance instability and dramatically worse latency. Our TPC-C runs saw sustained latencies of over a second in their initial 40 s so, and frequent latency spikes later on, caused by fsync calls and writes that took more than 1 s to complete. These slow file system operations appear unrelated to our storage hardware: they occur only when two or more disks are being written simultaneously; they occur at medium write rates as well as high rates; they occur whether or not our log and checkpoint files are preallocated; and they occur occasionally on each of our disks (both Fusion and RAID). Turning off NUMA-aware allocation greatly reduces the problem, but traces of it remain: the occasional latency spikes visible in our figures have the same cause. NUMA-aware allocation is fragile, particularly in older versions of Linux like ours;[3] it is possible that a newer kernel would mitigate this problem.

7 Related work

SiloR is based on Silo, a very fast in-memory database for multicore machines [27]. We began with the publicly available Silo distribution, but significantly adapted the logging implementation and added checkpointing and recovery. Silo draws from a range of work in databases

[3]For instance, to get good results with the NUMA allocator, we had to pre-fault our memory pools to skirt kernel scalability issues; this step could take up to 30 minutes per run!

and in multicore and transactional memory systems more generally [1, 3, 6–9, 11, 12, 15, 18, 19, 21].

Checkpointing and recovery for in-memory databases has long been an active area of research [5, 20, 22–24]. Salem et al. [24] survey many checkpointing and recovery techniques, covering the range from fuzzy checkpoints (that is, inconsistent partial checkpoints) with value logging to variants of consistent checkpoints with operation logging. In those terms, SiloR combines an action-consistent checkpoint (the transaction might contain some, but not all, of an overlapping transaction's effects) with value logging. Salem et al. report this as a relatively slow combination. However, the details of our logging and checkpointing differ from any of the systems they describe, and in our measurements we found that those details matter. In Salem et al. action-consistent checkpoints either write to all records (to paint them), or copy concurrently modified records; our checkpointers avoid all writes to global data. More fundamentally, we are dealing with database sizes and speeds many orders of magnitude higher, and technology tradeoffs may have changed.

H-Store and its successor, VoltDB, are good representatives of modern fast in-memory databases [10, 13, 25]. Like SiloR, VoltDB achieves durability by a combination of checkpointing and logging [14], but it makes different design choices. First, VoltDB uses command logging (a variant of operation logging), in contrast to SiloR's value logging. Since VoltDB, unlike Silo, partitions data among cores, it can recover command logs somewhat in parallel (different partitions' logs can proceed in parallel). Command logging in turn requires that VoltDB's checkpoints be transactionally consistent; it takes a checkpoint by marking every database record as copy-on-write, an expense we deem unacceptable. Malviya et al. also evaluate a variant of VoltDB that does "physiological logging" (value logging). Although their command logging recovers transactions not much faster than it can execute them—whereas physiological logging can recover transactions 5x faster—during normal execution command logging performs much better than value logging, achieving 1.5x higher throughput on TPC-C. This differs from the results we observed, where value logging was just 10% slower than a system with persistence entirely turned off. Our raw performance results also differ from those of Malviya et al. For command logging on 8 cores, they report roughly 1.3 Ktxn/s/core for new-order transactions, using a variant of TPC-C that entirely lacks cross-warehouse transactions. (Cross-warehouse transactions are particularly expensive in the partitioned VoltDB architecture.) Our TPC-C throughput with value logging, on a mix including cross-warehouse transactions and similar hardware, is roughly 8.8 Ktxn/s/core for new-order transactions. Of course, VoltDB is more full-featured than SiloR.

Cao et al. [2] describe a design for frequent consistent checkpoints in an in-memory database. Their requirements align with ours—fast recovery without slowing normal transaction execution or introducing latency spikes—but for much smaller databases. Like Malviya et al., they use "logical logging" (command/operation logging) to avoid the expense of value logging. The focus of Cao et al.'s work is two clever algorithms for preserving the in-memory state required for a consistent checkpoint. These algorithms, Wait-Free ZigZag and Wait-Free Ping-Pong, effectively preserve 2 copies of the database in memory, a current version and a snapshot version; but they use a bitvector to mark on a per-record basis which version is current. During a checkpoint, updates are directed to the noncurrent version, leaving the snapshot version untouched. This requires enough memory for at least 2, and possibly 3, copies of the database, which for the system's target databases is realistic (they measure a maximum of 1.6 GB). As we also observe, the slowest part of recovery is log replay, so Cao et al. aim to shorten recovery by checkpointing every couple seconds. This is only possible for relatively small databases. Writing as fast as spec sheets promise, it would take at least 10 seconds for us to write a 43 GB checkpoint in parallel to 3 fast disks, and that is assuming there is no concurrent log activity, and thus that normal transaction processing has halted.

The gold standard for database logging and checkpointing is agreed to be ARIES [16], which combines undo and redo logging to recover inconsistent checkpoints. Undo logging is necessary because ARIES might flush uncommitted data to the equivalent of a checkpoint; since SiloR uses OCC, uncommitted data never occurs in a checkpoint, and redo logging suffices.

The fastest recovery times possible can be obtained through hot backups and replication [14, 17]. RAMCloud, in particular, replicates a key-value store node's memory across nearby disks, and can recover more than 64 GB of data to service in just 1 or 2 seconds. However, RAMCloud is not a database: it does not support transactions that involve multiple keys. Furthermore, RAMCloud achieves its fast recovery by fragmenting failed partitions across many machines. This fragmentation is undesirable in a database context because increased partitioning requires more cross-machine coordination to run transactions (e.g., some form of two-phase commit). Nevertheless, 1 or 2 seconds is far faster than SiloR can provide. Replication is orthogonal to our system and an interesting design point we hope to explore in future work.

8 Conclusions

We have presented SiloR, a logging, checkpointing, and recovery subsystem for a very fast in-memory database. What distinguishes SiloR is its focus on performance for extremely challenging workloads. SiloR writes logs and checkpoints at gigabytes-per-second rates without greatly affecting normal transaction throughput, and can recover $>$ 70 GB databases in minutes.

For future work, we would like to investigate checkpointers that cycle through logical partitions of the database. We believe this approach will allow us to substantially reduce the amount of log data that needs to be replayed after a crash. Another possibility is to investigate a RAMCloud-like recovery approach in which data is fragmented during recovery, allowing quick resumption of service at degraded rates, but then reassembled at a single server to recover good performance.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Allen Clement, for helpful comments and patience. This work was supported by the NSF under grants 1302359, 1065219, and 0704424, and by Google and a Microsoft Research New Faculty Fellowship.

References

[1] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. PPoPP '10*, Jan. 2010.

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

[3] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *Proc. VLDB '01*, Sept. 2001.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. ACM Symp. on Cloud Computing*, June 2010.

[5] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. SIGMOD '84*, June 1984.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proc. SIGMOD 2013*, June 2013.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. DISC '06*, Sept. 2006.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11), 1976.

[9] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. 12th Conf. on Extending Database Tech.*, Mar. 2009.

[10] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1:1496–1499, 2008.

[11] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2), 1981.

[12] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4), 2011.

[13] A.-P. Liedes and A. Wolski. SIREN: a memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In *Proc. ICDE '06*, Apr. 2006.

[14] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *Proc. ICDE '14*, Mar. 2014.

[15] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys '12*, Apr. 2012.

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

[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. SOSP 2011*, Oct. 2011.

[18] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2), 2010.

[19] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page latch-free shared-everything OLTP. *Proc. VLDB Endow.*, 4(10), 2011.

[20] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1:271–287, 1986.

[21] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *Proc. VLDB Endow.*, 6(2), 2012.

[22] D. Rosenkrantz. Dynamic database dumping. In *Proc. SIGMOD '78*, May 1978.

Appendix

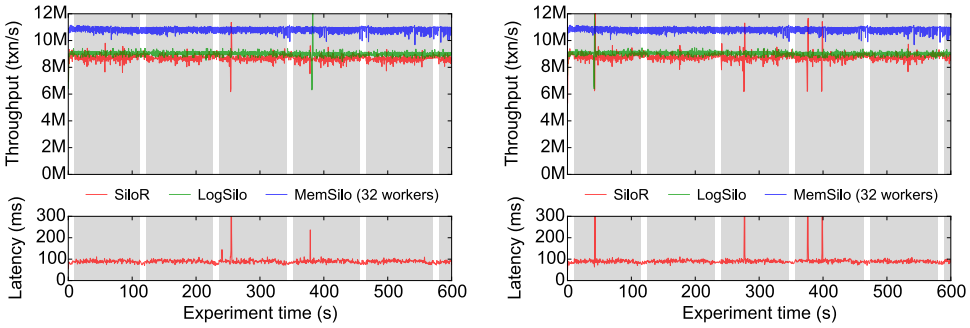

Figure 6: Performance of SiloR, LogSilo, and MemSilo on our modified YCSB benchmark: additional runs.

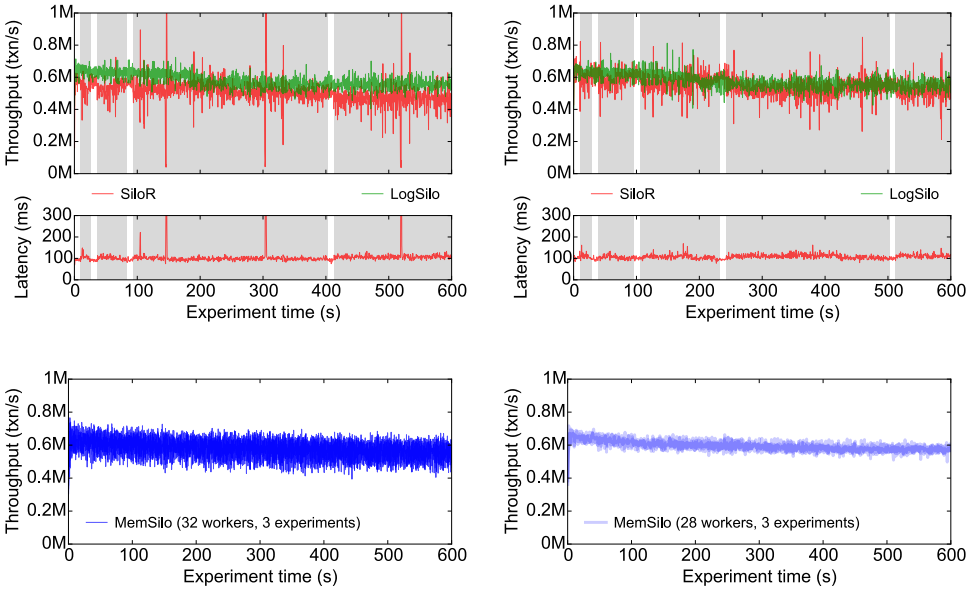

Figure 7: Performance of SiloR, LogSilo, and MemSilo (with 32 and 28 workers) on our modified TPC-C benchmark: additional runs.

[23] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *Proc. ICDE '89*, Feb. 1989.

[24] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Trans. Knowledge and Data Eng.*, 2(1), Mar. 1990.

[25] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. VLDB '07*, Sept. 2007.

[26] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

[27] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. SOSP '13*, Nov. 2013.