Fast Checkpoint and Recovery Techniques for an In-Memory Database

by

Wenting Zheng

S.B., Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2014

Certified by ....
Barbara Liskov
Professor
Thesis Supervisor

Accepted by ....
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Fast Checkpoint and Recovery Techniques for an In-Memory Database

by

Wenting Zheng

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Multicore in-memory databases for modern machines can support extraordinarily high transaction rates for online transaction processing workloads. A potential weakness of such databases, however, is recovery from crash failures. We show that techniques for disk-based persistence can be efficient enough to keep up with current systems' huge memory sizes and fast transaction rates, be smart enough to avoid additional contention, and provide fast recovery.

This thesis presents SiloR, a persistence system built for a very fast multicore database system called Silo. We show that naive logging and checkpoints make normal-case execution slower, but that careful design of the persistence system allows us to keep up with many workloads without negative impact on runtime performance. We design the checkpoint and logging system to utilize multicore's resources to its fullest extent, both during runtime and during recovery. Parallelism allows the system to recover fast. Experiments show that a large database ($\approx$ 50 GB) can be recovered in under five minutes.

Thesis Supervisor: Barbara Liskov
Title: Professor

Acknowledgments

I would like to thank my advisor, Barbara Liskov, for always giving me wonderful ideas and wise advice when I need it the most. I am very fortunate to have worked with her and she has helped me become a much better researcher.

This thesis is not possible without help from Eddie Kohler, Stephen Tu, and Sam Madden. I would like to thank Eddie for his great advice on implementation of the system and helpful guidance that made debugging much easier. Stephen's project made my thesis possbile, and he always points me in the right direction whenever I encounter problems.

I would like to thank my parents for giving me unconditional love and continued support and encouragement throughout the last five years at MIT.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents SiloR, a checkpoint and recovery system for a very fast multi-core in-memory database called Silo. The goal of this project is to provide persistence for Silo while maintaining its runtime performance and provide fast and parallelizable recovery.

1.1 Problem statement

In-memory databases on modern multicore machines can handle complex, large transactions amazingly well. These databases can execute millions to tens of millions of transactions per second, depending on transaction size. However, such databases need to be robust to failures. Database persistence can be guaranteed in the form of logging to disk. Database systems also use checkpoints in addition to logging because periodic checkpoints can reduce the log size, speeding up recovery and preventing the system from running out of disk space.

Implementing persistence for a fast and large in-memory database has several challenges. Logging is fast because it uses sequential writes, thus making full use of the disk throughput. However, checkpoints require a walk over the entire database. The data movement and cache pollution caused by that walk can reduce database performance, as can interference with logging caused by writing the database to disk.

Recovery for such a system is also challenging. Modern in-memory databases that can handle tens of millions of small transactions per second can generate more than 50 GB of log data per minute. 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 [1, 6, 13]. Such a huge log size makes recovery challenging because of the sheer amount of data to be read and recovered.

1.2 Proposed solution

This thesis presents SiloR, a persistence system built on top of Silo, a very fast multi-core in-memory database [15]. SiloR adds logging, checkpointing, and recovery to Silo. The goal of SiloR is to provide full persistence for Silo with low impact on runtime performance and also provide fast recovery. This means that the runtime throughput and latency should not be adversely affected, and that a crashed "big" database ($\approx$50 GB in size) should recover to a transactionally-consistent snapshot in reasonable time. We are able to achieve these goals in SiloR.

To achieve such performance, SiloR uses concurrency in all parts of the system. We design the runtime system to utilize parallelism to minimize checkpoint time and maintain reasonable log IO. The log is written concurrently to several disks, and a checkpoint is taken by several concurrent threads that also write to multiple disks.

Concurrency is also crucial for recovery. The key to fast recovery is to use all of the machine's resources, in particular all of the cores on a multi-core machine. Some logging designs, such as *operation logging* (logs contain the operations performed by the transactions instead of the modified records), are tempting because they could reduce the amount of logs written to disk during runtime. However, operation logging is difficult to recover is parallel. The desire for fast parallel recovery affected many aspects of our logging and checkpointing designs.

We propose a design and an implementation for SiloR to deal with the challenges summarized above. In particular, we show:

- All the important durability mechanisms can and should be made parallel.
- Checkpointing and logging combined can be fast without hurting normal transaction execution.
- Log replay becomes a bottleneck during recovery due to the high throughput of this database. SiloR uses concurrency to recover a large database with a large log in a reasonable amount of time.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents related work. Chapter 4 discusses the design of logging in SiloR. Chapter 5 discusses the design of checkpoints. Chapter 6 talks about how recovery can be executed from logging and checkpointing. Chapter 7 presents the performance of checkpoint and recovery in SiloR. Chapter 8 concludes.

Chapter 2

Related work

The gold standard for database persistence is ARIES [8]. ARIES handles disk-based database persistence by implementing a range of techniques, such as write-ahead logging, fuzzy checkpointing, and REDO/UNDO logging. However, ARIES was designed for disk-based databases, which keep all of the data on disk. To improve runtime performance, these databases load and keep a small portion of the active data in memory. For every piece of data the database system loads into memory, there is a corresponding disk page for that data.

In-memory databases are different from disk-based databases. For in-memory database, the size of the memory is big enough that the entire database can fit in memory. This means that operations are performed in memory – disk is only used for persistence and recovery in case of a crash. Checkpointing therefore has to flush out a large amount of data to disk. For the rest of the related work section, we will focus on literature that deals with recovery for in-memory database systems.

2.1 Persistence for in-memory databases

Checkpointing and recovery for in-memory databases has long been an active area of research [2, 10–13]. One of the early survey papers on in-memory database persistence techniques was written by Salem and Garcia-Molina [13]. This paper listed several different checkpoint consistency guarantees – $FUZZY$, where there are no guarantees about the state of the values being checkpointed; AC (action consistent), where each value checkpointed is guaranteed to be atomic and to contain a correct value; TC (transaction consistent), the most consistent out of all three guarantees, where the state of the checkpoint is transactionally consistent. Their paper describes System M, in which they experimented with three checkpoint methods – fuzzy, black/white checkpoints, and copy-on-update. SiloR's checkpoint method is closest to fuzzy checkpointing, but our consistency guarantees are action consistent. This is because SiloR uses optimistic concurrency control [4] to commit records, and only results of transactions that will commit are written into the checkpoint.

The details of our logging and checkpointing differ from any of the systems Salem and Garcia-Molina describe, and in our measurements we found that those details matter. In System M's implementation, action-consistent checkpoints either write to all records to mark 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 have changed. It is also interesting to note that System M's performance results for the different methods of checkpointing. While the underlying hardware is different from what SiloR uses, the results support our decision to avoid a transactionally consistent checkpoint because of the high overhead.

There has been a lot of recent work on in-memory databases [3, 9, 14, 15]. H-Store and its commercial counterpart, VoltDB, are good representatives of such systems [3, 14]. Like SiloR, VoltDB uses a combination of logging and checkpointing to achieve persistence [6]. However, VoltDB makes different design choices; it utilizes transactionally consistent checkpointing and *command logging* (also known as operation logging). Malviya et al. [6] discusses and argues for the benefits of command logging by comparing the performance of ARIES-type value logging (also referred to as *physiological logging*) with command logging on VoltDB. The authors found that command logging out-performed value logging for Voter and TPC-C benchmarks. This makes sense because command logging writes out less data compared to value logging for these workloads. However, command logging requires a transactionally consistent checkpoint design. VoltDB uses copy-on-update to keep such a checkpoint. Because copy-on-update could potentially cause a 2x overhead for the system, we deem such memory overhead unacceptable for a large in-memory database.

Malviya et al. observed a 33% loss in runtime performance for value logging implementation compared to the no persistence implementation. This result is different from what we observed in SiloR, where there was only a 16% reduction in throughput due to the use of value logging. Both results used the TPC-C benchmark, where command logging would have likely improved throughput.

Malviya et al. utilizes parallelism for recovery for both command logging and value logging. For the value logging implementation, the REDO phase is parallelized; for the command logging implementation, each partition replays the operations in parallel. However, Malviya et al.'s performance tests used a variant of TPC-C that lacks cross-warehouse transactions because these transactions are expensive in a partitioned VoltDB. With cross-warehouse transactions excluded, Malviya et al. found that value logging recovered 1.5x faster than command logging. It is likely that normal TPC-C with cross-warehouse transactions will cause the command logging implementation to recover even slower.

Cao et al. [1] described two algorithms for taking consistent checkpoints for in-memory databases [1]. These algorithms, Wait-Free Zigzag and Wait-Free Ping-Pong, take ideas from the copy-on-update method and introduce ways to use more memory in order to avoid locking and bulk bit-array reset overhead. Wait-Free Zigzag has a 2x memory overhead, while Wait-Free Ping-Pong has a 3x memory overhead. While Wait-Free Zigzag and Wait-Free Ping-Pong will work for Silo, these algorithms will not be effective. Cao et al. ran experiments for state sizes of *at most* 1.6 GB, but SiloR is designed to handle databases with size around 50 GB. A 2x or 3x memory overhead can be tolerated when the database size is a few gigabytes, but not when it is 50 GB.

SIREN [5] is an in-memory database that reduces the size of checkpoints by using incremental checkpoints. It uses the concept of pages, taken from disk-based databases, to achieve this. SIREN organizes tuples in a table into logical pages, which can be split and joined like on-disk B-tree pages. At the start of a checkpoint, SIREN freezes all dirty pages in order to take a consistent checkpoint, but uses pending operations (similar to copy-on-update) so that normal transactions are not blocked. Because records are organized into pages, SIREN can effectively determine which pages are dirty since the last checkpoint was performed, making incremental checkpointing easy to implement.

SIREN's system uses copy-on-update to keep extra copies of records in memory. SIREN's incremental checkpointing could be very effective for skewed workloads that update a subset of the pages, but will not work well when the workload consists of mostly random modifications. Their approach should work well for skewed workloads that update only a small subset of the pages. SiloR uses a "brute-force" checkpointing method, in which all records are copied even if they have not been modified; we would like to explore incremental checkpointing in the future (chapter 8).

Chapter 3

Silo Overview

In this section, we present an overview of Silo in order to provide a context for the rest of the thesis. Silo [15] is a very fast multi-core in-memory database, able to support millions to tens of millions of transactions per second.

Clients of Silo 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 in SiloR, we use several cores for housekeeping tasks related to logging and checkpointing.

Each Silo table is implemented as one primary index tree and zero or more secondary index trees, where each tree is an efficient, cache-friendly concurrent B-tree variant [7] that points to separately-allocated records. Because trees are stored in (shared) main memory, any worker can access the entire database.

Silo uses optimistic concurrency control (OCC [4]) to perform transactions. An OCC transaction runs "optimistically," which means it runs without acquiring locks until the validation phase. 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* that contains each record that was read, along with its TID at the time it was accessed. Modified records occur in the *read-set*, but the worker additionally maintains a *write-set* that stores the new state of the 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 assigns the transaction's TID; this is the serialization point. Next it compares the TIDs of the records in the *read-set* with these records' current TIDs. If any TID has changed or the record is locked, the transaction aborts. Otherwise it commits: the worker overwrites the *write-set* records with their new values and the transaction's TID.

3.1 Epochs

Silo transaction IDs differ in an important way from those in other systems, and this difference impacts the way we do logging and recovery in SiloR.

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 by using time periods called *epochs*. A global epoch number E is visible to all threads. A designated thread periodically advances E, currently set to once every 40 ms.

Each Silo transaction is assigned a transaction ID. Worker threads access E while committing transactions, and a TID consists of an epoch number and a counter. The TID for a transaction uses the value of E visible at the serialization point, plus a counter. The counter is selected so that the TID of thet transaction is:

- larger than the TID of any record read or written by the transaction
- larger than the worker's most recently chosen TID

Silo TIDs capture the transaction serial order most of the time, but not always. Consider two transactions, T1 and T2, where T1 happens before T2. T1 reads record A, adds 1 to the value of A, and stores the result in record B. T2 only modifies A.

Silo does not guarantee that T2 will necessarily have a greater transaction ID than T1.

Epochs provide the key to correct replay, provided the designated transaction's update of E becomes visible at all workers simultaneously, which is what happens on a total-store-order (TSO) machine like an x86-64. Because workers read the current epoch at the serialization point, the ordering of TIDs with different epochs is compatible with the serial order. In the next few chapters, we will describe how we use epochs to achieve correct persistence.

Chapter 4

Logging

Logging is the most basic form of maintaining persistence in a database. In SiloR we use value logging instead of command logging. This section describes the logging design and justifies why we believe that value logging is more suitable for Silo than command logging.

4.1 Basic logging

The responsibility for logging in SiloR is split between worker threads and logging threads. Workers generate log records as they commit transactions. After a certain amount of log records has been generated, workers pass these records to logging threads, which write the records to disk. When a set of logs is made persistent on disk via `fsync`, the loggers inform the workers. This allows workers to send transaction results to clients.

Each transaction's log record starts with a transaction ID, followed by the number of records logged in this transaction and a list of `(table ID, key, value)` records; each record modified by the transaction is contained in this list. Each worker constructs log records in disk format and stores them in a memory buffer taken from a per-worker buffer pool. When there is no more space in the buffer or the worker is at an epoch boundary, the worker passes the buffer to a logger over a shared-memory queue. Each logger processes buffers from several workers and commits the logs to disk using `fsync`.

Durability is a transitive property: a transaction is durable if all of its modifications are recorded on durable storage *and* all transactions serialized before it are also durable. SiloR uses epochs to provide this guarantee. For example, a transaction T1 with epoch e_1 will be persistent once all of the transactions from epochs up to and including e_1 are also persistent on disk. Thus, the loggers periodically compute a global system *persistence epoch*: the computation guarantees that results of *all* transactions running up through that epoch have been synchronized to disk. Workers return transaction results to clients when the epoch in which that transaction ran becomes persistent.

4.2 Value logging vs. operation logging

SiloR uses value logging instead of operation logging. This means that SiloR logs contain the modifications made by a transaction instead of the operations performed by the transaction. Consider a transaction that performs `increment_range`(10, 15), which adds 1 to all keys between 10 and 15. Value logging would store the keys and new values for all updated keys, while operation logging would store the transaction type (`increment_range`) and its arguments.

The choice of value logging is an example of recovery parallelism driving the normal-case logging design. Value logging has an apparent disadvantage: in many workloads, such as TPC-C, value logging logs more data than operation logging, and therefore may slow down transaction execution more than operation logging. However, from the point of view of recovery, the advantages of value logging outweigh its disadvantages. Value logging is easy to replay in parallel — the largest TID per key wins. Operation logging, in contrast, requires that transactions be replayed *in their original serial order*, which is much more difficult to parallelize. An additional difficulty is that, as mentioned in chapter 3, Silo's transaction IDs do not reflect serial order. Thus, correct replay would require logging transactions' *read-sets* (keys and TIDs) as well as operation IDs and arguments, so that the operations could be replayed. This will increase the log size. Yet another point is that operation logging also requires that the initial pre-replay database state be a transactionally consistent snapshot, whereas value logging does not. Furthermore, our experiments (see chapter 7) show that performance with value logging enabled is only slightly degraded over performance in the absense of any support for persistence. These considerations led us to prefer value logging in SiloR.

4.3 Workers and loggers

Loggers have little CPU-intensive work to do. They collect logs from workers, write them to disk, and await durability notification from the kernel. Workers have a lot of CPU-intensive work to do. A SiloR deployment therefore contains many worker threads and few logger threads.

How should worker threads be mapped 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 might speed up replay. We rejected this design because of its adverse 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. In our experiments, we found that one logger thread per disk is sufficient to handle the buffers from workers. Multiple logger threads would simply interfere with each other, competing for disk IO.

4.4 Buffer management

Our measurements of Silo's primitive logging system revealed bottlenecks in the logging subsystem, especially under high loads from small transactions. Although loggers should not normally limit transaction execution, loggers must be able to apply backpressure to workers so that workers don't continuously generate log data, causing the system run out of memory. This backpressure is implemented by buffer management. Loggers allocate a maximum number of log buffers per worker. Buffers circulate between loggers and workers as transactions execute; a worker will block when it needs a new log buffer and all of its buffers are currently used by the logger. During our initial experiments, this backpressure blocked workers unnecessarily often. File system hiccups, such as those caused by concurrent checkpoints, were amplified by loggers into major dips in transaction rates.

In SiloR, we redesigned the log buffer subsystem to recirculate log buffers back to workers as soon as possible. In Silo, buffers are returned to the workers after `fsync` is done. This is unnecessary. As soon as a `write` has been performed on a buffer, that buffer can be immediately returned to the workers. We also increased the number of log buffers available to workers, setting this to about 10% of the machine's memory. The result was fewer and smaller dips in transaction throughput.

A worker flushes a buffer to its logger when either the buffer is full or a new epoch begins. 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 1 MB. This is big enough to obtain some benefit from batching, but small enough to avoid wasting much space when a partial buffer is flushed.

4.5 Details of logging

This section provides a detailed description of how logging works.

1. Workers send buffers to loggers when the buffers become full *or* at epoch boundaries. This is accomplished by publishing a pair $< e_w, B >$, where e_w is the current epoch being run at the worker. The worker guarantees that it has completed all epochs less than e_w, and that all of its future log buffers will contain transactions with epochs $\geq e_w$.

2. Loggers periodically process the information pushed by the workers. Each logger l writes the associated buffer to disk, and uses the epoch information to calculate e_{max_l} and e_{min_l}, where $e_{max_l} = max\{e_w\}$, and $e_{min_l} = min\{e_w\} - 1$.

3. Logger l synchronizes the writes to disk using `fsync`.

4. After synchronization completes, the logger publishes e_{min_l}. These published values are used to compute the persistence epochs (described in the next section §4.6).

5. If necessary, logger l performs log rotation (see §4.7).

4.6 Persistent epoch

SiloR uses a distinguished logger thread to periodically compute a *persistence epoch* e_p as $\min\{e_{min_l}\}$ over all of the loggers. It writes e_p to a file named `persist_epoch` and synchronizes that write to disk using `fsync`. After `fsync` is complete, the logger thread atomically renames `persist_epoch` to `pepoch`. This atomic rename guarantees that the `pepoch` file will never contain partially updated epoch information, even if there is a system crash during the update.

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 committed and their results are released to clients.

This protocol ensures that it will be safe during recovery to recover all transactions that ran in epochs up to and including e_p. The log contains all effects of transactions up to and including e_p, and furthermore we can be sure that clients have not received results of any transactions that ran after this epoch.

4.7 Log file management

Each SiloR logger manages its log as a collection of files. The logger writes to the current log file, which is always named `data.log` and holds the most recent entries.

Periodically (currently every 100 epochs) the logger renames the current log file to `old_data.`e, where e is e_{max_l} (see step 2 in §4.5). e_{max_l} stores the maximum epoch seen across *all* workers for a logger l. Right before renaming, `data.log` contains records for transactions up to and including e_{max_l}, but none from any epoch after e_{max_l}. After the renaming, the logger starts writing to a new `data.log` file. Using multiple files simplifies the process of log truncation during checkpointing (explained in section 5.3).

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 greater than e.

Chapter 5

Checkpointing

In a runtime database system, logging takes care of the main persistence guarantees. However, logging alone is not sufficient because disks have limited space. If log truncation is not performed periodically, the disks will run out of space, crashing the database. Thus, checkpointing is required to take a "snapshot" of the runtime system and use that information to truncate log records.

Checkpointing in a database is also a necessary tool for speeding up recovery. More frequent checkpointing means fewer log records to replay during recovery. This is especially important for a large in-memory database that has a very high transaction throughput because even a slightly longer checkpoint period may mean a lot more log records have accumulated.

5.1 Basic design

SiloR takes a checkpoint by walking over all of the tables in the database. Because Silo uses optimistic concurrency control, all of the records in a checkpoint are *certain to be committed*. We do not require the records to be *persistent* at the time of the tree walk, although they must be persistent by the time the checkpoint completes. This will be explained in more detail in section 5.2.

It is important to note that SiloR's checkpoint is *inconsistent*. This means that a checkpoint does not hold a consistent snapshot of the database as of a particular point in the serial order. For example, a transaction that updated two keys might have only one of its updates represented in the checkpoint. Or, a checkpoint might contain the modifications of transaction T but not of some transaction that preceded T in the serial order. However, each record that is checkpointed will never be read in a partially updated state.[1] This means we do not need to perform UNDO during recovery.

As mentioned in section 4, SiloR uses value logging instead of operation logging. This checkpointing design works well with the logging design we have chosen. In fact, an inconsistent checkpointing design only works with value logging: we *cannot* perform operation logging and still be able to restore the database into a consistent state upon recovery.

We chose to produce an inconsistent checkpoint because it is less costly in terms of memory usage than a consistent checkpoint. Checkpoints of large databases take a long time to write (up to a few minutes), which is enough time for all database records to be overwritten. The allocation expense associated with storing new updates in newly-allocated records (rather than overwriting old records), can reduce normal-case transaction throughput by approximately 10%. The extra memory needed for preserving a transactionally consistent checkpoint can be at large as the database size. Such an approach would unnecessarily constrain the size of the database SiloR can support.

A checkpoint executes across a range of epochs. We define the checkpoint start epoch to be e_l, and the checkpoint end epoch to be e_h. A checkpoint that was written during $[e_l, e_h]$

- *will* contain the effects of any transactions executed in epochs $< e_l$ that are not

[1]In Silo it is possible that a record is modified while being read because readers do not set locks. Silo is able to recognize when this happens and redo the read. We use the same technique to read a valid record before copying it into the buffer.

superceded by later transactions' updates

- *may* contain the effects of any transactions executed in epochs with $e_l \leq e \leq e_h$
- *will not* contain the effects of any transactions executed in epochs $> e_h$.

To recover a consistent database state, it is necessary to load the checkpoint and then replay the log. The log must be complete over a range of epochs $[e_l, E]$, where $E \geq e_h$, or recovery is not possible.

5.2 Writing the checkpoint

5.2.1 Parallelism in checkpointing

SiloR is intended to support large databases, so it is desirable to produce a checkpoint concurrently and to store it on multiple disks. We have experimented with writing only with a single checkpointer thread, but the checkpoint period was extremely long. This called for parallel checkpointing.

We have a total of $n + 1$ threads running during active checkpointing, where n is the number of disks available. We have one *checkpoint manager* that is in charge of periodically starting a new checkpoint. Whenever the checkpoint starts, the checkpoint manager starts n *checkpoint worker threads* (also referred to as checkpoint threads), one per checkpoint disk. In our current implementation checkpoints are stored on the same disks as the log, and loggers and checkpointers execute on the same cores (which are separate from the worker cores that execute transactions).

SiloR takes a checkpoint by walking over the trees that store the database tables. The checkpointer uses a low-level scan operation provided by Masstree [7] to retrieve all of the keys and records currently in the tree. To perform the scan operation, we need to provide a key range. Since the underlying btree is ordered lexicographically, it is possible to provide a minimum key, but more difficult to provide a maximum key. To provide a maximum key, `maxkey`, we do an initial tree walk down the right side of the tree.

A possible way to split up the work of the checkpointers is to split up the tables among the n threads. However, since tables often differ in size, it is better to split up every table's key range into n subranges, designating one share from each table to one checkpoint thread. Therefore, each checkpoint worker thread stores roughly $1/n$ th of the database, where n is the number of disks. The checkpoint manager first walks through all of the active tables to find the maximum key. Each Silo key is stored in a variable-length key structure, consisting of a series of 8-byte key slices. The checkpointer uses the first key slice `kslice` to split the key range. More specifically, the range is split up in the following way:

```
$step \leftarrow kslice/n$
$range_1 \leftarrow [0, step]$
$range_2 \leftarrow [step, 2 * step]$
...
$range_n \leftarrow [(n-1) * step, maxkey]$
```

This scheme is an approximation of the keyspace since it is based on the assumption that the checkpoint range is uniformly distributed. The results from our YCSB and TPC-C experiments show that the sizes of the split checkpoint images deviate by at most 10%.

5.2.2 Recovery-aware checkpoint

Our system also recovers the checkpoint in parallel. During recovery, we use more threads than are used to take the checkpoint; experiments showed that limiting the number of threads to what was used for taking the checkpoint caused an unnecessarily long checkpoint recovery. Unlike the runtime system, we do not need to worry about impact on transaction processing throughput during recovery – instead all machine resources can be used to speed up recovery time. Therefore, the checkpoint design needs to consider recovery performance. We design the checkpoint mechanism to enable parallel recovery while minimizing impact on runtime performance.

During runtime, each checkpointer divides each of its table subranges into m files, where m is the number of cores that would be used during recovery for that subrange. Each file is named `checkpoint`ID where ID is the table ID. The m files are located on the same disk under different directories. For example, the m files may be located at `checkpoint_disk1/dir1/checkpoint100` and `checkpoint_disk1/dir2/checkpoint100`, where $m = 2$ and `100` is the table ID.

As the checkpointer walks over its range of the table, it copies the encountered records into a buffer of B in size. In our system, B is set to 2 MB. Each buffer contains a contiguous range of records in the format `(key, TID, value)`. When the checkpoint thread encounters a record that cannot fit into the current buffer, it writes the accumulated bytes to disk. After the buffer has been written, the checkpoint thread continues tree walking, again copying records into its buffer. When the buffer fills up again, the checkpoint thread writes to the next file. Each checkpointer writes to the m files in round-robin order.

When a checkpoint thread needs to write out a buffer, it does not call `write` directly. We use an optimization here to minimize the checkpoint thread's interference with the logger thread. The first optimization is a set of global flags around the `fsync` call in the logger. We loosely define the `fsync` region in the logger threads to be "critical" sections. These critical sections are useful because any interference with loggers' `fsync` calls may slow down `fsync` and increase latency. There is one global flag per disk, shared between that disk's logger thread and checkpointer thread. When the logger enters a critical section, it sets the global flag to 1. The flag is reset back to 0 when the `fsync` is done. The checkpointer reads this flag and only tries to execute `write` to disk if the flag is not set. Since the checkpointer fills up a buffer of size 2MB, we execute `write` in small buffers of size 4KB. In addition, each time a file descriptor's outstanding writes exceed 32 MB, the checkpoint thread synchronizes them to disk. These intermediate `fsync` calls turned out to be important for performance, as we discuss in section 7.

When a checkpoint thread has processed all tables, it performs a final `fsync` for each open file.

5.3 Cleanup

For a checkpoint file to be valid, it should only contain records that are persistent on disk. When the checkpoint thread performs its tree walk, it copies the latest version of each record, but these versions are not necessarily persistent at that time of the copy. Thus, each checkpoint thread must do the following:

```
$e_{wait} \leftarrow e_{global}$
**while** true **do**
    $e_p \leftarrow$ global persistent epoch
    **if** $e_{wait} \leq e_p$ **then** break
    **end if**
**end while**
```

Without this wait, the checkpoint may not be recovered correctly with the log records. Note that e_{wait} is an upper bound. We can compute e_{wait} more accurately by keeping track of the maximum of each record that the checkpoint thread encounters. This might potentially reduce the wait time if the persistent epoch is lagging behind the regular epoch a lot, but that is unlikely to happen during normal processing.

After the wait, the checkpoint thread installs the checkpoint files by performing atomic rename. For each of the its m files, it renames `checkpointID` to `ckp_ID`. Continuing the example from the previous section, `checkpoint_disk1/dir1/checkpoint100` would be renamed to `checkpoint_disk1/dir1/ckp_100`. After renaming is done, the checkpoint thread exits.

Once all checkpoint threads have exited, the checkpoint manager installs a file containing the checkpoint start epoch, e_l. It uses atomic rename again by first writing to a file named `checkpoint_epoch`, performing `fsync`, and renaming the file to `cepoch`. The recovery thread uses `cepoch` file's epoch number as the checkpoint epoch number.

Finally, the checkpoint manager thread performs log truncation. Recall that each log is comprised of a current file and a number of earlier files with names like `old_data.`e. Since each such log file guarantees that it contains only log records with epochs equal to or smaller than e, we can safely delete any log file with $e < e_l$.

The next checkpoint is begun roughly 20 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.

5.3.1 Failure during checkpointing

This section provides a brief correctness analysis for what happens if there is a system crash when the checkpoint is being produced.

We first discuss what happens when there is a crash, but there is already a previous checkpoint installed in place.

If the system crashes before any checkpoint thread has had a chance to begin installing checkpoint files through atomic renames, the recovery system will replay the previous checkpoint. This is true because the checkpoint files have not been installed yet, and the checkpoint epoch also has not been installed. The system will be able to recover correctly.

If the system crashes while a checkpoint thread is installing the new checkpoint files, we may potentially have some checkpoint files from the previous checkpoint (C_{n-1}), and some checkpoint files from the current checkpoint (C_n). The recovery system will use some files from C_{n-1} and some files from C_n. This is correct because each checkpoint file is *valid*. Since each checkpoint thread waits before performing renaming, all of the checkpoint files from C_n contain only persistent values. Furthermore, atomic renames guarantee that the system will not end up with a partially overwritten or corrupted checkpoint file. Each checkpoint file is either a valid file from C_{n-1} or C_n. The log records from C_{n-1} to C_n have not been truncated, so no information from the overwritten C_{n-1} checkpoint files has been lost. The checkpoint epoch files `cepoch` has also not been modified. During recovery, the system will replay from $cepoch_{n-1}$. Thus, the system will be able to recover correctly.

If the system crashes during the `cepoch` file installation, the recovery program will use the old `cepoch` file to recover. This is correct because log records from C_{n-1} and C_n have not been truncated. The recovery program has to replay more log records than necessary, but the process is correct.

If the system crashes during log truncation, the recovery program will use the newest `cepoch` file to replay logs. It will use the recovered e_l to skip log files that are old and do not need to be replayed.

We now explore what happens when the system fails during the very first checkpoint. The existence of the `cepoch` file determines the end of the first checkpoint phase. The recovery process always checks for the `cepoch` file before trying to retrieve checkpoint and log files. If `cepoch` does not exist, the recovery process will simply recover from the beginning of the log files. This is correct because log truncation happens *after* `cepoch` has been synchronized. If `cepoch` does exist, then the situation is equivalent to what was described in the previous paragraph, where the system crashes during log truncation. Since the checkpoint files are valid and none of the necessary files have been truncated, the system will recover correctly.

5.4 Reduced checkpoint

The previous sections described the base design of checkpoint. In this section, we present the idea of a *reduced checkpoint*.

As metioned before, we would like the checkpoint to complete as fast as possible while affecting the runtime system as little as possible. If one tries to speed up the checkpoint too much, the runtime system will have lower throughput and longer latency due to interference with loggers. The solution lies in not checkpointing *unnecessary information*. Since the loggers are already logging information about the records being modified/inserted/deleted starting with the checkpoint epoch e_l, it is unnecessary for the checkpointers to also checkpoint such records. Instead, the checkpointer only needs to checkpoint the records that have epochs $e < e_l$.

If this checkpoint method is used, the wait time near the end of checkpointing is reduced because we only have to make sure that $e_l < e_p$, since we will only be recording the records that have epochs less than e_l.

The results of reduced checkpointing are shown in section 7.2.2.

Chapter 6

Recovery

This section describes how we recover from a failure. To recover the database we must first load the checkpoint into memory, then replay the log records. In both cases, we utilize the system's many cores to speed up recovery as much as possible.

6.1 Start of recovery

At the start of recovery, some metadata needs to be read from the disk. The recovery threads need two pieces of information – the checkpoint epoch, e_l (recovered from file `cepoch`), and the persistent epoch, e_p (recovered from file `pepoch`). For both checkpoint recovery and log recovery, any record encountered with an epoch e such that $e < e_l$ or $e > e_p$ should be skipped. It is safe to replay checkpoint or log records that satisfy $e < e_l$ because the value will be overwritten by values with larger epochs. It is *incorrect* to replay records that have $e > e_p$ because transactions with epochs greater than e_p were not necessarily persistent at the time of the crash. Group commit has not finished for epochs greater than e_p. If we process records for epochs after e_p we could not guarantee that the resulting database corresponded to a prefix of the serial order.

6.2 Recovering the checkpoint

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 there are m files for each table on every disk. Recovery is carried out by $n \cdot m$ threads. Each thread reads from one disk, and is responsible for reading and processing one file per table from that disk.

As described in §5.2.2, the checkpoint files are written to several directories for a single disk. The master replay thread is handed several directories that contain the checkpoint files, and tries to find files named `ckp_`ID. If it can find files matching those names, it will create new tables with the corresponding table IDs.

Processing of a file 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. Since a particular key/value pair only appears once in a checkpoint, there is no need to lock any record during checkpoint replay. The underlying Masstree [7] is able to handle concurrent non-conflicting tree operations efficiently.

6.3 Recovering the log

After all threads have finished their assigned checkpoint recovery tasks, the system moves on to log recovery. As mentioned in chapter 4, there was no attempt at organizing the log records at runtime (e.g. partitioning the log records based on what tables are being modified). In fact, it is likely that each log file will contain unrelated sets of modifications to various index trees. This situation is different than it was for the checkpoint, which was organized so that concurrent threads could work on disjoint partitions of the database.

We need a controlled method for replaying log records. However, since SiloR uses value logging, we do not have to order the log records by transaction IDs or be careful with which log records to replay first. The log records 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 matter.

We take advantage of this property to process the log in parallel, and to avoid unnecessary allocations, copies, and work. The log recovery threads use the e_l and e_p recovered in the section 6.1 to determine which log records to replay. All log records for transactions with epoch e such that $e < e_l$ or $e > e_p$ are ignored. The rest are replayed.

Before each log recovery thread starts, it must determine which log files it is responsible for. Each log recovery thread is in charge of a discrete set of log files. The recovery manager is in charge of creating a list of files for each log recovery thread. It looks for files with names matching `old_data.`e and `data.log`. For each disk we start up g log threads, where $g = \lceil N/n \rceil$. Since log recovery usually dominates recovery time due to the runtime system's high throughput, we use all the cores we can during this phase. Such a formula for g may cause the manager to spawn 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. For example, our current system has 3 disks and 32 cores, and we choose to use 33 threads instead of 30 threads because we can achieve better performance when SiloR uses 33 threads to recover.

The manager divides up the log files among the g threads for every disk. For each of the log files on a disk, the manager calls `lstat` to retrieve the file size. The log files are assigned to threads based on how many bytes each thread will have to process:

the next log file is assigned to the thread that has the minimum number of bytes to process so far. This is a greedy algorithm to ensure that all threads will process approximately the same number of bytes. We do not want straggler threads that have to process more bytes than the other threads.

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

Since log recovery does not process organized files, it is possible to have two concurrent threads replay different values for the same key. Therefore, we need to have concurrency control during log replay. During the log replay, the log thread first performs a search on the key it has. Each record has a lot of status bits, but for recovery purposes there are three bits that are necessary to know:

- the lock bit, indicating whether the record is locked
- the deleting bit, indicating whether this record is an absent record (Silo uses absent records to indicate logical deletes)
- the is_latest_version bit, indicating whether this record is the most recent version

After the thread finds a record, it tries to lock that record if possible, by using the lock bit. If the record is locked, the thread waits for the record to be unlocked. Once it has locked a record, it checks whether the record is the latest version. If it is not the latest version, the thread unlocks the old record and re-performs the search to find the latest record.

Once the thread has locked a record that is also the latest version, the thread checks whether the log contains a more recent value for the record than what it currently holds. If so, it tries to overwrite the existing record with the new value. This can only be done if the allocated space for the old tuple is enough to fit the latest value. If this is possible, the thread performs an overwrite and unlocks the record. If this is not possible, the thread will create a new record with the corresponding TID and insert that record into the tree. The new record needs to be locked and its is_latest_version bit set. After the insert is done, the old record's is_latest_version bit is cleared. Finally, the old record and the new record are both unlocked, and the old record is put into a queue for garbage collection later. If a thread happened to wait for the old record, it will see that the record is no longer the most recent version. The thread will perform search again to find the latest version, which has already been inserted into the tree.

If the log record indicates a delete, we cannot ignore it. We must replay the log record by using an absent record with the correct TID. This is necessary for correct behavior. Imagine two transactions T1 and T2 that modify record X. T1 comes before T2 and has a smaller TID. T1 updates X, while T2 removes X. During log replay, it is possible that T2 is replayed before T1. If there is no absent record inserted into the tree with T2's TID, the replay thread won't know that T2 exists and will insert T1's update into the tree. Therefore, deletes need to be recorded in the tree. If there is already a record in the tree, we simply set the deleting bit. Otherwise we insert a new absent record into the tree. The keys that have absent records are put into a separate list so that the values can be deleted from the tree after recovery is completed.

Finally, once all the threads are done with processing, records and keys that are not needed anymore will be garbage collected.

6.3.1 Backwards log processing

Value logging replay has the same result no matter what order files are processed. To more efficiently use CPU, we process the most recently written log files first. Since we know that `data.log` must contain the most recent entries, that file is always replayed first. The rest of the log files are ordered by their epochs – the file name `old_data.`e's e epoch. Larger epochs are replayed first.

When files are processed in strictly forward order, every log record will likely require overwriting some value in the tree. This is especially true for a skewed workload that tends to modify a subset of records. 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 comes from a later log file, is often newer than the log record. The log replay thread only needs to compare with the TID of the current value in the tree and skips the log record if the TID in the tree is larger.

6.4 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's 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, e.g., TIDs (or whatever is used for version numbers) must capture the dependencies, but need not capture anti-dependencies. SiloTIDs meet this requirement. And because TID comparison is a simple commutative test, log processing can take place in any order.

Chapter 7

Performance and Experiments

7.1 Experimental setup

All of our experiments were run on a single machine with four 8-core Intel Xeon E7-4830 processors clocked at 2.1GHz, yielding a total of 32 physical cores. Each core has a private 32KB L1 cache and a private 256KB L2 cache. The eight cores on a single processor share a 24MB L3 cache. The machine has 256GB of DRAM with 64GB of DRAM attached to each socket, and runs 64-bit Linux 3.2.0. For the YCSB tests, we include a NUMA-aware memory allocator. The TPC-C benchmark does not use the NUMA allocator. We do not have networked clients – all of the clients are local client threads.

We use three separate Fusion ioDrive2 flash drives for both logging and checkpointing. Each drive has a dedicated logger and a checkpointer. 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.

7.2 YCSB Experiments

To test the impact of checkpointing on runtime performance, we decided to use Yahoo Cloud Serving Benchmark (YCSB), a performance benchmark from Yahoo. We set the following parameters:

1. 70% single record read, and 30% single record write
2. 100 byte records
3. 400M keys, with an on-disk size of 46 GB
4. 28 worker threads

The original YCSB experiment uses 50% reads and 50% writes. We modified the percentage to be lower in order to not over-strain the disk IO from pure logging. 30% write is still a relatively aggressive write percentage for a workload.

7.2.1 Full checkpoint performance

We first present the performance of taking full checkpoints (as opposed to reduced checkpoints, described in 5.4).

Since we have a total of three Fusion IO disks, we use three checkpointer threads. Each thread is in charge of a separate partition of the each table in the database. Each checkpointer writes to a set of files on a single disk.

The throughput result can be seen in figure 7-1. In this graph we show the effects of the system taking two checkpoints. Each checkpoint period takes approximately 130 seconds.

Figure 7-1 shows the effects of two consecutive checkpoints. The gray sections show how long the checkpoint periods last. Figure 7-2 shows the latency graphs. We can see that both latency and throughput effects are more stable in the second checkpoint compared to the first checkpoint. We believe that the slight drop in performance in

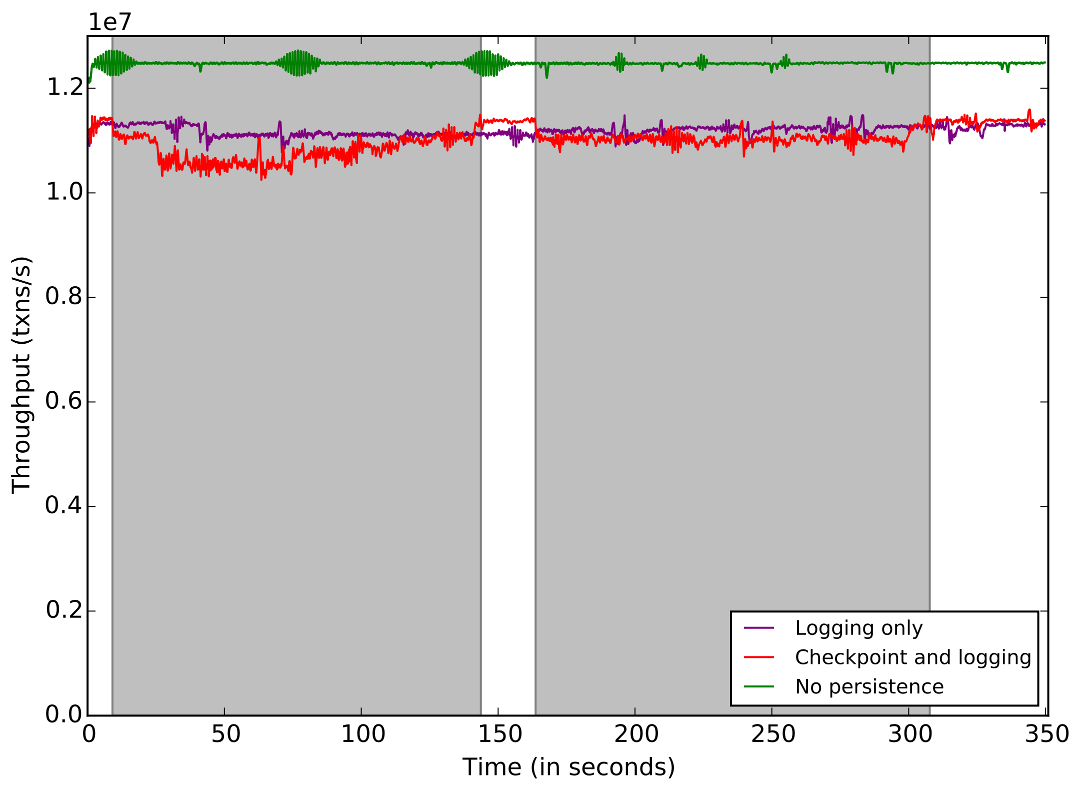


Figure 7-1: YCSB experiment throughput

the first checkpoint is a start up transient of the disks. Once the overall system is more stable, the checkpoint throughput and latency also reach a steady state.

The necessity of periodic `fsync`

Our initial implementation of parallel checkpoint consisted of only a single `fsync` for every open file descriptor at the end of the checkpoint. The results of this parallel checkpointing is shown in figure 7-3. What we found was that, without slowing down the checkpoint, throughput is greatly reduced. If the checkpointer runs as fast as possible, it will cause so much interference with the logger thread that is using the same disk that the interference causes back pressure on the worker threads. In order to control the amount of memory used for log buffers, we restrict the number of log buffers to a maximum number. If the loggers are behind on flushing out the

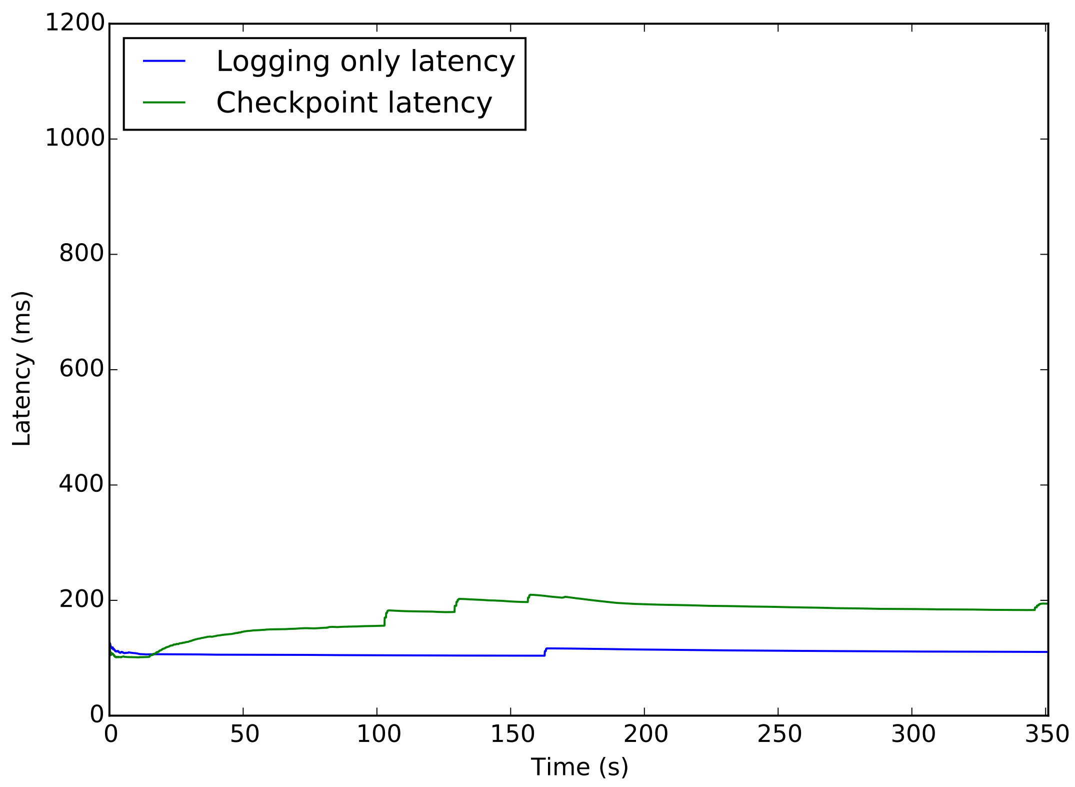


Figure 7-2: YCSB experiment latency

buffers to disk, then the worker threads will be stuck waiting for free buffers. In these experiments, we find that the loggers are often stuck in fsync for a long time, and thus the threads are always waiting for buffers.

To solve this problem, we decided to slow down the checkpoint by introducing periodic `usleep` calls. The effects of slowing down the checkpoint are shown in the last three graphs of 7-3. Clearly, if the checkpointer interferes with the loggers less, the throughput increases.

We later found that, even when the checkpoint was slowed down, average transaction latency was several seconds. This was counter-intuitive at first because the checkpointer did not seem to be competing with the logger threads very much. An experiment was performed that took out all of the `fsync` calls in the checkpointers. The latency and `fsync` time graphs (figure 7-4) show an experiment performed on

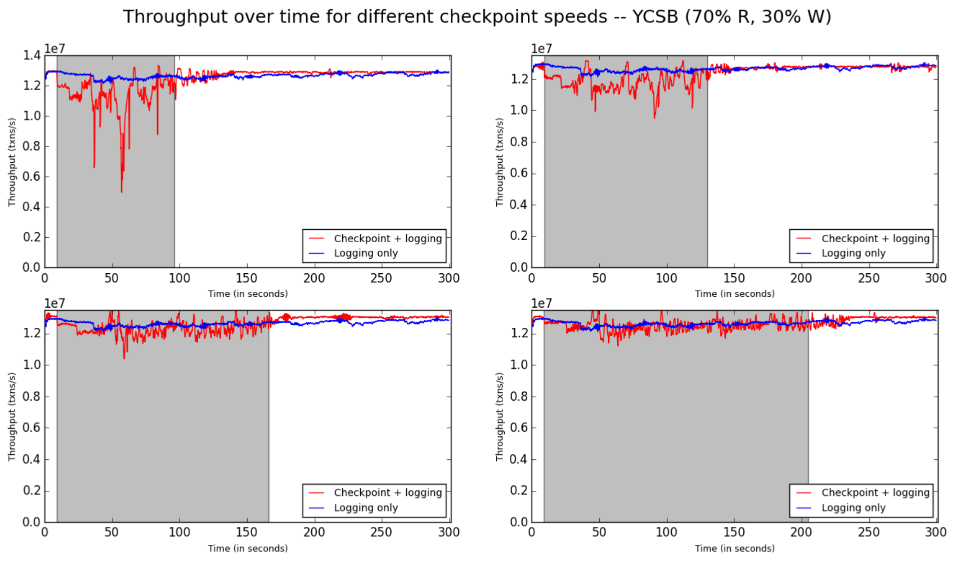
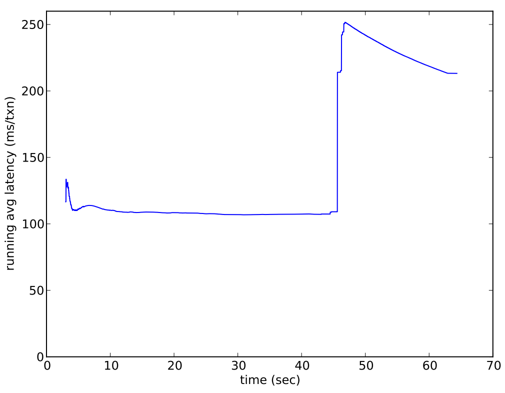


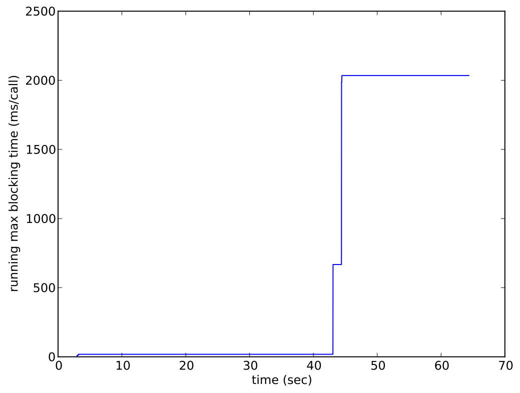
Figure 7-3: Throughput and latency of checkpoint on the YCSB benchmark. The first graph shows the effects on throughput when the checkpoint runs as fast as possible. The last three graphs show what happens when we lengthen the checkpoint period by introducing `usleep` calls.

a very small database. The checkpointer actually finishes checkpointing around 30 seconds into the experiment, but there is still a huge increase in latency *after* the `write` calls are finished. The latency increase corresponds to a very long `fsync` time for one logger. In this experiment, the system's long latency comes from one single spike in `fsync` time.

This puzzling result reveals some information about how the operating system deals with `fsync`. It seems that the operating system does not flush out the pages until when it *has to*. This means that without periodic `fsync` calls, the operating system accumulates pages in its buffer cache until the cache is full, then proceeds to flush out a huge number of pages from its buffer cache all at once. This bursty behavior caused the system to suddenly have a multi-second `fsync` call (see figure 7-4b), thereby suddenly increasing the latency. This is why we introduced additional fsync calls into the checkpoint threads. With those periodic `fsync` calls, transaction latency during checkpoints was greatly reduced to the reasonable values shown in figure 7-2.



(a) Bad latency



(b) Long `fsync` time

Figure 7-4: Results for a small database, with checkpointer's `fsync` turned off

7.2.2 Reduced checkpointing

The original design of the checkpointer was less efficient because we can take *reduced checkpoints* in SiloR. Reduced checkpoints are described in section 5.4. Figure 7-5 shows the throughput of the experiments, for no persistence, logging only persistence, and logging and checkpoint. Figure 7-6 shows a rolling average of the latency for logging only and logging and checkpoint.

We can see that the checkpoint period is further reduced by 20 seconds. This reduction comes from the fact that the checkpoint size has been reduced from 46 GB to 31 GB. This result makes sense for YCSB because the entire database is being uniformly randomly modified. For a benchmark that constantly overwrites the entire database at a fast pace, it will be easy to skip over records during a checkpoint because most of the encountered records will be "newly modified".

7.2.3 Effects of compression

One possible way to speed up checkpointing is by using compression. This method presents a trade off between CPU usage and disk IO. We implemented compression using the LZ4 compression algorithm. The YCSB benchmark updates the database

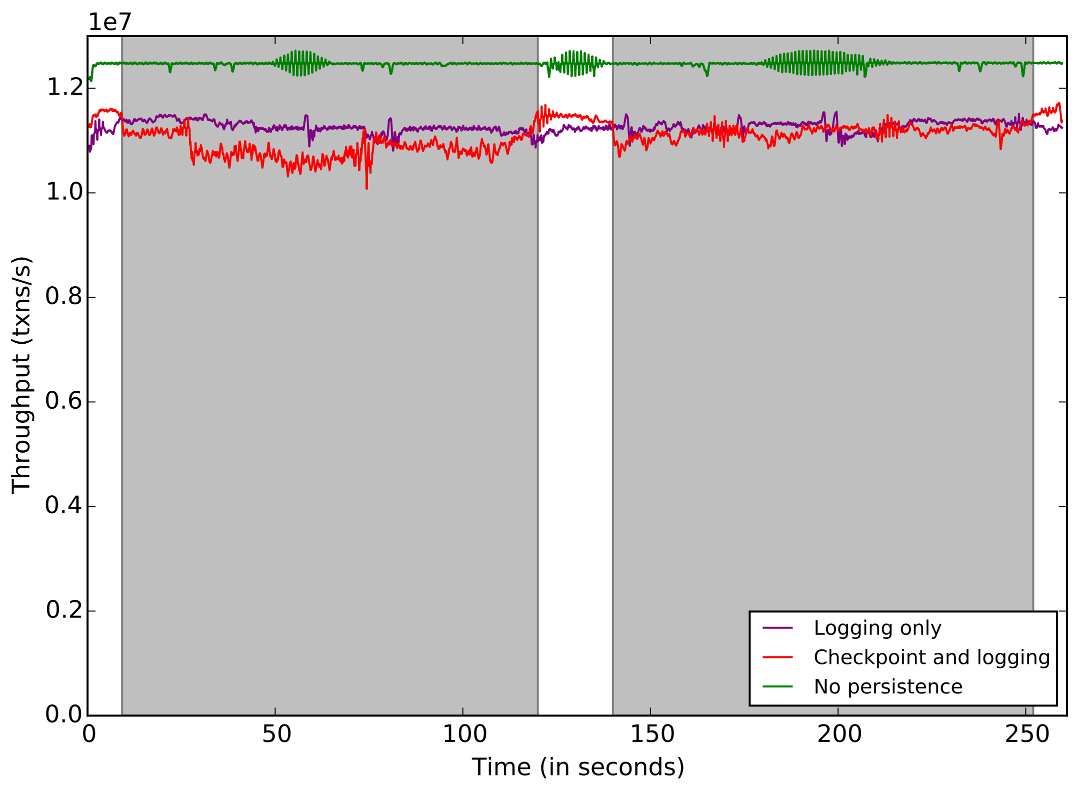


Figure 7-5: YCSB experiment with reduced checkpoint – throughput

using randomly constructed records, which results in a compression ratio of roughly 25%. However, our experiments showed no performance gains from using compression. Compression actually increased the checkpoint time by tens of seconds, and thus was not effective.

7.2.4 Recovery of YCSB

We recover YCSB using the method described in section 6. To find the maximum possible recovery time, we gather two checkpoint periods worth of log records and recover the entire database from that. Two checkpoint periods worth of logs roughly represents the maximum amount of log records we would ever have to handle because it is the worst case scenario: the previous checkpoint truncates logs before that checkpoint epoch C_{n-1}, and the system crashes right before the second checkpoint has a

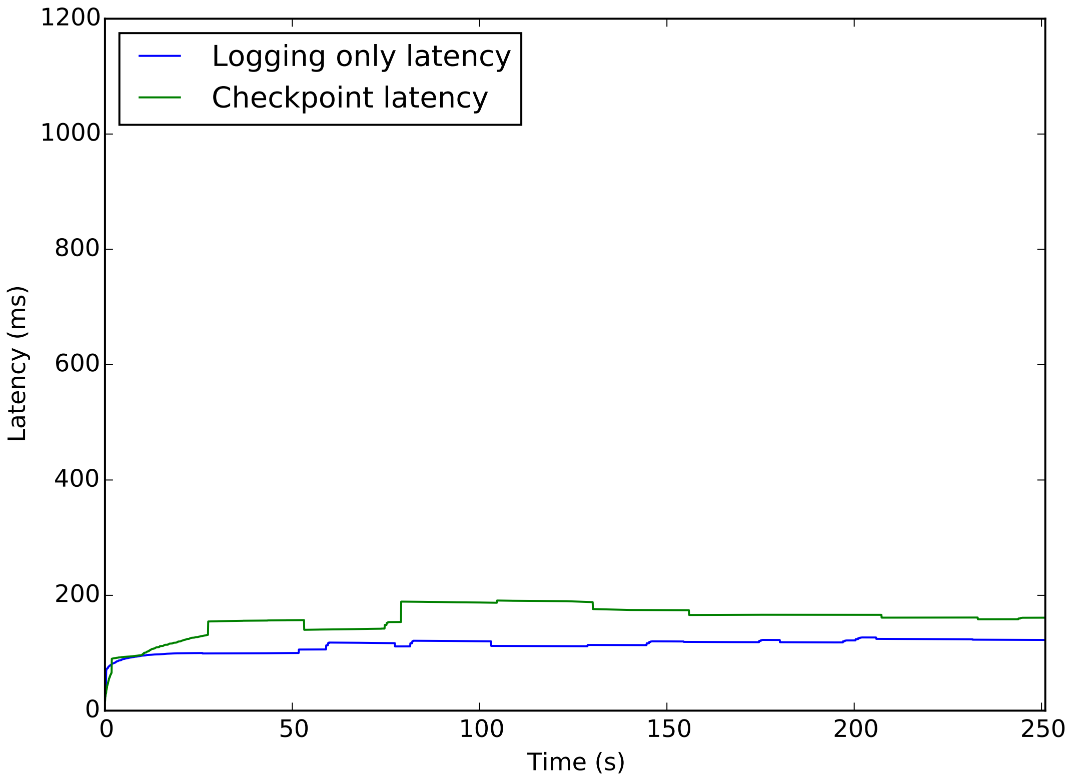


Figure 7-6: YCSB experiment with reduced checkpoint – latency

chance to truncate the logs and finish. Therefore, the amount of log records we have to recover is the log records accumulated from C_{n-1} to C_n, plus the entire second checkpoint period. This comes out to two checkpoint periods of logs, in addition to the time in between the checkpoints.

We performed recovery on the system from the experiment described in section 7.2.1. During the 350 seconds of the experiment, we accumulate two checkpoint periods of log records. The on-disk checkpoint size is 46.8 GB, while the total log size is 137 GB. The recovery program is able to recover the previous checkpoint in 40 seconds, and replay the logs in 109 seconds. Recall that our design allows us to take advantage of the parallelism in the system to recover the database; in our experiments, we use 4 threads per disk (total of 12 threads) to perform checkpoint restoration, and 11 threads per disk (total of 33 threads) for log recovery.

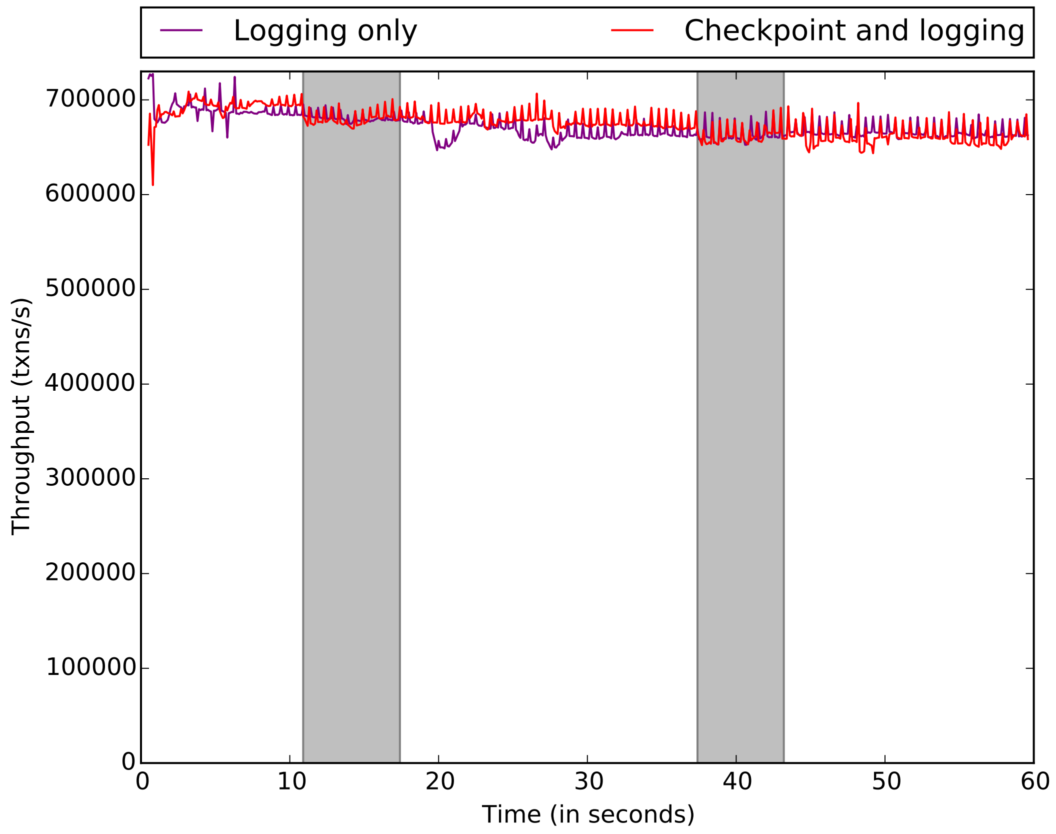


Figure 7-7: TPC-C results – throughput

We did the same experiment with the reduced checkpoint. As mentioned before, the total checkpoint image to recovery is 31 GB. We are able to recover 31 GB in 25 seconds. During the two checkpoint periods we accumulate 99 GB of logs, and that can be recovered in 84 seconds. Therefore, we are able to save roughly 40 seconds of recovery time by doing reduced checkpointing. Clearly, this is a worthwhile optimization, especially if the workload tends to modify a lot of data instead of having a small "hot" partition.

7.3 TPC-C

TPC-C is a standard online transaction processing benchmark. It has a very write-heavy workload and simulates customers submitting orders and orders getting pro-

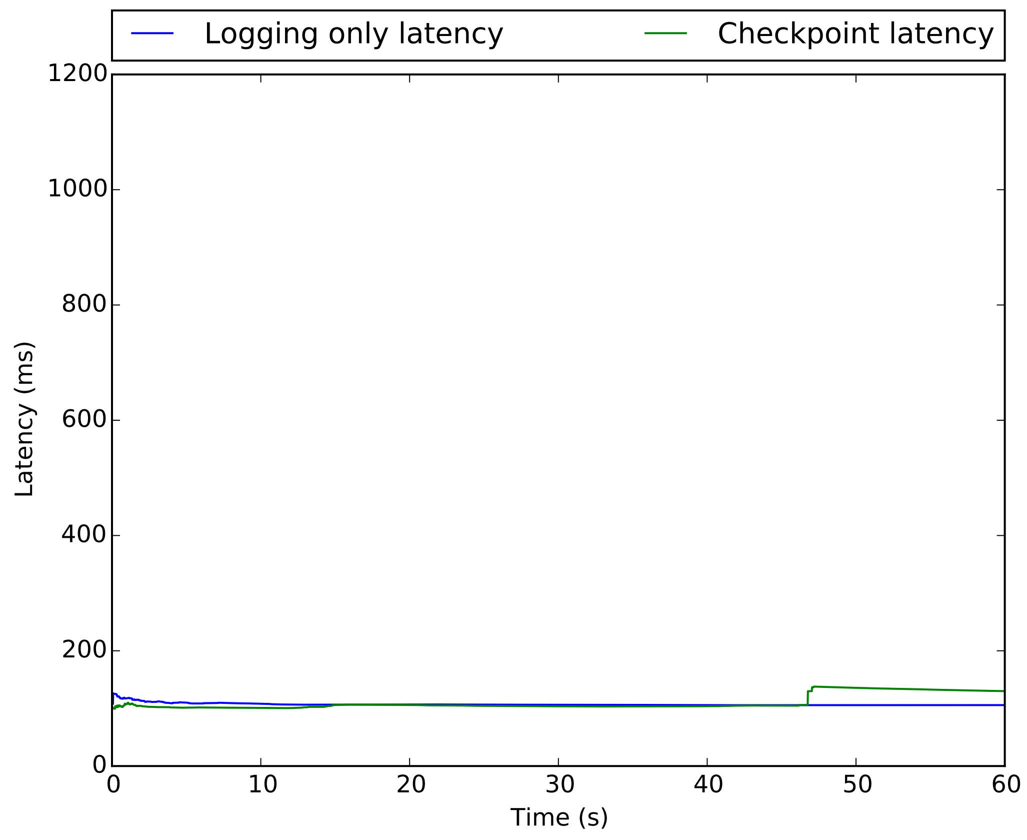


Figure 7-8: TPC-C results – latency

cessed. The TPC-C benchmark is different from YCSB because TPC-C is a more realistic workload, with areas of the database that are frequently updated and other areas less frequently updated. The workload is a lot more varied than the uniform random updates of YCSB.

However, a problem with TPC-C for Silo or any in-memory database system is that is that the database size keeps on growing without bound. This means that after a certain amount of processing the database size will exceed available memory. Therefore, we modify TPC-C by keeping its main write transaction `NewOrder`, and adding in a new transaction `FulfillOrder`. `FulfillOrder` acts as a replacement for `Delivery`, but performs more deletes to keep the database roughly the same size. Each `FulfillOrder` transaction starts by finding the most recent order executed by a random customer and removing the rows for that order from tables NewOrder, Order, and OrderLine. `FullfillOrder` transactions balance out the inserts performed by `NewOrder`, and the experiment runs 50% `NewOrder` and 50% `FulfillOrder`. We run with 24 threads instead of 28 threads for TPC-C because TPC-C's heavy write workload already uses significant disk IO.

7.3.1 Results

The results of modified TPC-C are shown in figure 7-7 and figure 7-8. We used the base checkpoint scheme for the experiment, but for a database that keeps on inserting larger keys and deleting keys, the base scheme already reduces the checkpoint size. The checkpoint period takes roughly 5 seconds. TPC-C recovery took 1.7s to recover the 1.17 GB of database, and 22 s to recover 18.4 GB of log records. The 18.4 GB of log records accumulated in 2 checkpoint periods, in addition to the delay between the two checkpoints.

Since the TPC-C database size is very small, we can extrapolate these numbers to calculate what would happen for a much larger database size. This means that for a database that is 50 times big (50 GB), the time to checkpoint will be be roughly $50s \cdot 5 + 20s = 270$ seconds. The total logs accumulated should be roughly

$$\frac{18.4GB}{50s} \cdot (250s \cdot 2 + 20s) \approx 191GB \tag{7.1}$$

The time to recover the logs should be

$$\frac{191GB}{(18.4GB/22s)} \approx 228s \tag{7.2}$$

The database will take roughly 50 seconds to recover. Thus, the total recovery time for a 50 GB database running TPC-C is 278 seconds, or approximately 4.6 minutes. These results are worse than that of YCSB because TPC-C is more write-intensive. In addition, this calculation is a conservative estimate. The real performance of a 50 GB database running TPC-C should be better.

Chapter 8

Conclusion and Future Work

This thesis presented SiloR, a system for maintaining persistence for a very fast large multi-core in-memory database (Silo). SiloR shows that a careful design of the logging and checkpointing systems can have low impact on runtime performance. In addition, SiloR's checkpoint and logging designs are able to exploit parallelism to its full extent for speedy recovery. SiloR is able to provide a durable database that has high throughput, low latency, and reasonable recovery time.

8.1 Future work

There are several ways the system can be improved. We first describe a few general ideas, then we will describe a promising idea in more depth.

When a checkpoint runs for the first time, it needs to write out all of the records. For subsequent checkpoints, it is sufficient to build upon the previous checkpoint by flushing out only data that was *changed*. Such a design is called *incremental checkpointing*. If a piece of data has not been modified since the last checkpoint, no data needs to be written. SIREN [5] uses pages to implement incremental checkpointing, but this is not efficient under a randomized workload. We would like to investigate a more sophisticated design that would work well under a wide range of workloads.

We have observed that much of the cost of recovering the log is key lookup: even when the record in the log isn't needed because its TID is smaller than what is in the database, we still need to do the lookup to find this out. It might be possible to speed up log recovery by removing these lookups, e.g., by using a hash table during recovery to avoid them.

Instead of writing to disk for maintaining persistence, we could explore replication. We could write the log to a backup server that will maintain a copy of the state of the database. The backup server will be constantly running "recovery" using the transmitted logs and applying those changes to its own database copy. While the primary is running normally, the backup can be used to process read-only queries. If the primary fails, the backup takes over as the new primary and serves client requests. If a node recovers, it obtains the current state from the current primary.

8.1.1 Rolling checkpoints

This section describes in detail what we believe to be the most promising future work – *rolling checkpoint*.

Building upon the base checkpoint design, where we take a *full checkpoint*, we can split that full checkpoint into several stages.

Imagine that the database is partitioned into k pieces. For example, the database consists of k tables $(T_1, T_2, \ldots)$ of approximately the same size. The idea is that we will break up the checkpoint into k pieces: first we write T_1, then T_2, etc. The corresponding stages are S_1, S_2, etc. We define a *full checkpoint period* that goes from e_L to e_H.

The full checkpoint C_1 as a whole starts at epoch e_{l_1}. We start writing T_1 at e_{l_1}, and this completes at epoch e_{h_1}. At this point we wait for $e_{h_1} \leq e_p$ (the persistence epoch) and then install the checkpoint files for T_1, plus an extra file that records e_{h_1} as the end epoch for T_1 and S_1. Then we write T_2 from e_{l_2} to e_{h_2}, and so on. Checkpoint C_1 is complete at e_{h_k}, and the full checkpoint period is $[e_L = e_{l_1}, e_H = e_{h_k}]$.

The next full checkpoint, C_2, will start immediately after the last stage of C_1 is done. C_2 starts checkpointing T_1 again. This stage will go from e'_{l_1} to e'_{h_1}. At the end of e'_{h_1}, we have a *rolling checkpoint* with checkpoint epoch interval $[e_L = e_{l_2}, e_H = e'_{h_1}]$. Thus, we are able to truncate the log records with epochs less than e_{l_2}. This shift will keep on happening as we checkpoint more stages.

The main benefit of the staged scheme is that we can perform log truncation after *each stage is completed*. This means that we will most likely end up with a much smaller log than the full checkpoint scheme. When we complete stage i of C_n, the full checkpoint after C_{n-1}, we can eliminate log files that only contain records for epochs before $e_{l_{i+1},n-1}$, the start epoch for stage $i+1$ completed in C_{n-1}. This is correct because the information in those records is either for T_i, which has just been copied again, or it is for later stages of C_n, but these modifications are captured either in C_n itself or in later log files.

The simple design works well if we have exactly k tables. We would like to generalize it to a database with any number of tables. Therefore, we would like to process *partitions* of a database, where each partition consists of several pre-defined *subranges* of tables. This method is possible but would have stricter requirements.

We now show a brief analysis of how this design performs with respect to full checkpoints.

We define a checkpoint period P, where P is the time it takes to checkpoint the entire database. Let d be the time between two consecutive full checkpoints.

- Worst case performance: in our current scheme the worst case gives us a log size based of $2P + d$. For the staged scheme, we instead have $P + (P/k) + d$, where k is the number of stages. This assumes that it takes roughly $\frac{1}{k}$ of the time for each stage.

- Average case performance: in our current scheme this is $P + \frac{(P+d)}{2}$. For the staged scheme it is $P + \frac{(P/k)+d)}{2}$.

As we increase the number of partitions, both the worst case performance and the average case performance for rolling checkpoint get closer to P. The performance of rolling checkpoints is bounded below by P, and above by $2P$. In the worst case scenario, if the table is extremely skewed and the partition is not divided up evenly, the rolling checkpoint will have the same performance as full checkpoints. However, on average, rolling checkpoints can reduce the checkpoint time by a lot, and a design needs to be developed so that the implementation achieves this ideal by dividing the work evenly into the stages.

Bibliography

[1] 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 *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 265–276. ACM, 2011.

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

[3] 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. 1: 1496–1499, 2008.

[4] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[5] A.-P. Liedes and A. Wolski. Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 99–99. IEEE, 2006.

[6] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery.

[7] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[8] 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 Transactions on Database Systems (TODS)*, 17 (1):94–162, 1992.

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

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

[11] D. Rosenkrantz. Dynamic database dumping. 1978.

[12] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. 1989.

[13] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. 2(1), Mar. 1990.

[14] 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 *VLDB*, 2007.

[15] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases.