Speedy Transactions in Multicore In-Memory Databases

Stephen Tu, Wenting Zheng, Eddie Kohler[†], Barbara Liskov, and Samuel Madden
MIT CSAIL and [†]Harvard University

Abstract

Silo is a new in-memory database that achieves excellent performance and scalability on modern multicore machines. Silo was designed from the ground up to use system memory and caches efficiently. For instance, it avoids all centralized contention points, including that of centralized transaction ID assignment. Silo's key contribution is a commit protocol based on optimistic concurrency control that provides serializability while avoiding *all* shared-memory writes for records that were only read. Though this might seem to complicate the enforcement of a serial order, correct logging and recovery is provided by linking periodically-updated *epochs* with the commit protocol. Silo provides the same guarantees as any serializable database without unnecessary scalability bottlenecks or much additional latency. Silo achieves almost 700,000 transactions per second on a standard TPC-C workload mix on a 32-core machine, as well as near-linear scalability. Considered per core, this is several times higher than previously reported results.

1 Introduction

Thanks to drastic increases in main memory sizes and processor core counts for server-class machines, modern high-end servers can have several terabytes of RAM and 80 or more cores. When used effectively, this is enough processing power and memory to handle data sets and computations that used to be spread across many disks and machines. However, harnassing this power is tricky; even single points of contention, like compare-and-swaps on a shared-memory word, can limit scalability.

This paper presents Silo, a new main-memory database that achieves excellent performance on multicore machines. We designed Silo from the ground up to use system memory and caches efficiently. We avoid all centralized contention points and make all synchronization scale with the data, allowing larger databases to support more concurrency.

Silo uses a Masstree-inspired tree structure for its underlying indexes. Masstree [23] is a fast concurrent B-tree-like structure optimized for multicore performance. But Masstree only supports non-serializable, single-key transactions, whereas any real database must support transactions that affect multiple keys and occur in some serial order. Our core result, the Silo commit protocol, is a minimal-contention serializable commit protocol that provides these properties.

Silo uses a variant of optimistic concurrency control (OCC) [18]. An OCC transaction tracks the records it reads and writes in thread-local storage. At commit time, after validating that no concurrent transaction's writes overlapped with its read set, the transaction installs all written records at once. If validation fails, the transaction aborts. This approach has several benefits for scalability. OCC writes to shared memory only at commit time, after the transaction's compute phase has completed; this short write period reduces contention. And thanks to the validation step, read-set records need not be locked. This matters because the memory writes required for read locks can induce contention [11].

Previous OCC implementations are not free of scaling bottlenecks, however, with a key reason being the requirement for tracking "anti-dependencies" (write-after-read conflicts). Consider a transaction t_1 that reads a record from the database, and a concurrent transaction t_2 that overwrites the value t_1 saw. A serializable system must order t_1 before t_2 even after a potential crash and recovery from persistent logs. To achieve this ordering, most systems require that t_1 communicate with t_2, such as by posting its read sets to shared memory or via a centrally-assigned, monotonically-increasing transaction ID [18, 19]. Some non-serializable systems can avoid this communication, but they suffer from anomalies like snapshot isolation's "write skew" [2].

Silo provides serializability while avoiding *all* shared-memory writes for read transactions. The commit protocol was carefully designed using memory fences to scalably produce results consistent with a serial order. This leaves the problem of correct recovery, which we solve using a form of *epoch-based group commit*. Time is divided into a series of short epochs. Even though transaction results always agree with a serial order, the system

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522713

does not explicitly know the serial order *except across epoch boundaries*: if t_1's epoch is before t_2's, then t_1 precedes t_2 in the serial order. The system logs transactions in units of whole epochs and releases results to clients at epoch boundaries. As a result, Silo provides the same guarantees as any serializable database without unnecessary scalability bottlenecks or much additional latency. Epochs have other benefits as well; for example, we use them to provide database *snapshots* that long-lived read-only transactions can use to reduce aborts.

On a single 32-core machine, Silo achieves roughly 700,000 transactions per second on the standard TPC-C benchmark for online transaction processing (OLTP). This is about 22,000 transactions per second per core. Per-core transaction throughput at 32 cores is 91% of that at 8 cores, a small drop indicating that Silo scales well. For context, the database literature for high-performance in-memory OLTP systems cites per-core TPC-C throughput at least several times lower than Silo's [16, 25, 27, 28], and benchmarks on our hardware with a commercial main-memory database system perform at most 3,000 transactions per second per core.

An important Silo design choice is its shared-memory store: any database worker can potentially access the whole database. Several recent main-memory database designs instead use *data partitioning*, in which database workers effectively own subsets of the data [25, 32]. Partitioning can shrink table sizes and avoids the expense of managing fine-grained locks, but works best when the query load matches the partitioning. To understand the tradeoffs, we built and evaluated a partitioned variant of Silo. Partitioning performs better for some workloads, but a shared-memory design wins when cross-partition transactions are frequent or partitions are overloaded.

Silo assumes a *one-shot* request model in which all parameters for each client request are available at the start, and requests always complete without further client interaction. This model is well-suited for OLTP workloads. If high-latency client communication were part of a transaction, the likelihood of abort due to concurrent updates would grow.

Our performance is higher than a full system would observe since our clients do not currently use the network. (In Masstree, network communication reduced throughput by 23% [23]; we would expect a similar reduction for key-value workloads, less for workloads with more computation-heavy transactions.) Nevertheless, our experiments show that Silo has very high performance, that transactions can be serialized without contention, and that cache-friendly design principles work well for shared-memory serializable databases.

2 Related work

A number of recent systems have proposed storage abstractions for main-memory and multicore systems. These can be broadly classified according to whether or not they provide transactional support.

2.1 Non-transactional systems

The non-transactional system most related to Silo is Masstree [23], an extremely scalable and high-throughput main-memory B^+-tree. Masstree uses techniques such as version validation instead of read locks and efficient fine-grained locking algorithms. It builds on several prior trees, including OLFIT [4], Bronson et al. [3], and B^{link}-trees [20], and adds new techniques, including a trie-like structure that speeds up key comparisons. Silo's underlying concurrent B^+-tree implementation was inspired by Masstree.

PALM [31] is another high-throughput B^+-tree structure designed for multicore systems. It uses a batching technique, extensive prefetching, and intra-core SIMD parallelism to provide extremely high throughput. PALM's techniques could potentially speed up Silo's tree operations.

The Bw-tree [21] is a high-throughput multiversion tree structure optimized for multicore flash storage. New versions of data are installed using delta records and compare-and-swaps on a mapping table; there are no locks or overwrites (we found both helpful for performance). Silo's data structures are designed for main memory, whereas many of the Bw-tree's structures are designed for flash. Like Silo, the Bw-tree uses RCU-style epochs for garbage collection; Silo's epochs also support scalable serializable logging and snapshots. LLAMA [22] adds support for transactional logging, but its logger is centralized.

2.2 Transactional systems

Silo uses optimistic concurrency control [10, 18], which has several advantages on multicore machines, including a relatively short period of contention. However, OCC and its variants (e.g., [1, 12, 34]) often induce contention in the commit phase, such as centralized transaction ID assignment or communication among all concurrently executing transactions.

Larson et al. [19] recently revisited the performance of locking and OCC-based multi-version concurrency control (MVCC) systems versus that of traditional single-copy locking systems in the context of multicore main-memory databases. Their OCC implementation exploits MVCC to avoid installing writes until commit time, and avoids many of the centralized critical sections present in classic OCC. These techniques form the basis for concurrency control in Hekaton [8], the main-memory component of SQL Server. However, their de-

sign lacks many of the multicore-specific optimizations of Silo. For example, it has a global critical section when assigning timestamps, and reads must perform non-local memory writes to update other transactions' dependency sets. It performs about 50% worse on simple key-value workloads than a single-copy locking system even under low levels of contention, whereas Silo's OCC-based implementation is within a few percent of a key-value system for small key-value workloads.

Several recent transactional systems for multicores have proposed partitioning as the primary mechanism for scalability. DORA [25] is a locking-based system that partitions data and locks among cores, eliminating long chains of lock waits on a centralized lock manager and increasing cache affinity. Though this does improve scalability, overall the performance gains are modest—about 20% in most cases—compared to a locking system. Additionally, in some cases, this partitioning can cause the system to perform worse than a conventional system when transactions touch many partitions.

PLP [26] is follow-on work to DORA. In PLP, the database is physically partitioned among many trees such that only a single thread manages a tree. The partitioning scheme is flexible, and thus requires maintaining a centralized routing table. As in DORA, running a transaction requires decomposing it into a graph of actions that each run against a single partition; this necessitates the use of rendezvous points, which are additional sources of contention. The authors only demonstrate a modest improvement over a two-phase locking (2PL) implementation.

H-Store [32] and its commercial successor VoltDB employ an extreme form of partitioning, treating each partition as a separate logical database even when partitions are collocated on the same physical node. Transactions local to a single partition run without locking at all, and multi-partition transactions are executed via the use of whole-partition locks. This makes single-partition transactions extremely fast, but creates additional scalability problems for multi-partition transactions. We compare Silo to a partitioned approach and confirm the intuition that this partitioning scheme is effective with few multi-partition transactions, but does not scale well in the presence of many such transactions.

Multimed [30] runs OLTP on a single multicore machine by running multiple database instances on separate cores in a replicated setup. A single master applies all writes and assigns a total order to all updates, which are then applied asynchronously at the read-only replicas. Multimed only enforces snapshot isolation, a weaker notion of consistency than serializability. Read scalability is achieved at the expense of keeping multiple copies of the data, and write-heavy workloads eventually bottleneck on the master.

In Silo, we eliminate the bottleneck of a centralized lock manager by co-locating locks with each record. VLL [29] also adopts this approach, but is not focused on optimizing for multicore performance.

Shore-MT [15], Jung et al. [17], and Horikawa [14] take traditional disk- and 2PL-based relational database systems and improve multicore scalability by removing centralized locking and latching bottlenecks. However, 2PL's long locking periods and requirement for read locking introduce inherent scalability concerns on multicore architectures.

Porobic et al. [28] perform a detailed performance analysis of shared-nothing versus shared-everything OLTP on a single multicore machine, and conclude that shared-nothing configurations are preferable due to the effects of non-uniform memory accesses (NUMA). Our results argue otherwise.

Snapshot transactions like Silo's have been implemented before, including in distributed transactional systems [7]. Our implementation offers very recent snapshots and tightly integrates with our epoch system.

2.3 Transactional memory

Silo's goals (fast transactions on an in-memory database) resemble those of a software transactional memory system (fast transactions on arbitrary memory). Recent STM systems, including TL2 [9], are based on optimistic concurrency control and maintain read and write sets similar to those in OCC databases. Some STM implementation techniques, such as read validation and collocation of lock bits and versions, resemble Silo's. (Many of these ideas predate STM.) Other STM implementation techniques are quite different, and overall Silo has more in common with databases than STMs. Our durability concerns are irrelevant in STMs; Silo's database structures are designed for efficient locking and concurrency, whereas STM deals with an arbitrary memory model.

STM would not be an appropriate implementation technique for Silo. Our transactions access many shared memory words with no transactionally relevant meaning, such as nodes in the interior of the tree. Unlike STM systems, Silo can distinguish relevant from irrelevant modifications. Ignoring irrelevant modifications is critical for avoiding unnecessary aborts. Finally, we know of no STM that beats efficient locking-based code.

3 Architecture

Silo is a relational database that provides tables of typed, named records. Its 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. One-shot requests are powerful; each consists of one

or more serializable database transactions that may include arbitrary application logic. We write one-shot requests in C++, with statements to read and manipulate the Silo database directly embedded. Of course a one-shot request could also be written in SQL (although we have not implemented that). Supporting only one-shot requests allows us to avoid the potential for stalls that is present when requests involve client interaction.

Silo tables are implemented as collections of index trees, including one primary tree and zero or more secondary trees per table (see Figure 1). Each record in a table is stored in a separately-allocated chunk of memory pointed to by the table's primary tree. To access a record by primary key, Silo walks the primary tree using that key. Primary keys must be unique, so if a table has no natural unique primary key, Silo will invent one. In secondary indexes, the index key maps to a secondary record that contains the relevant record's primary key(s). Thus, looking up a record by a secondary index requires two tree accesses. All keys are treated as strings. This organization is typical of databases.

Each index tree is stored in an ordered key-value structure based on Masstree [23]. Masstree read operations never write to shared memory; instead, readers coordinate with writers using version numbers and fence-based synchronization. Compared to other concurrent B-trees [4, 31], Masstree adopts some aspects of tries, which optimizes key comparisons. Each Masstree leaf contains information about a range of keys, but for keys in that range, the leaf may point either directly to a record or to a lower-level tree where the search can be continued. Although our implementation uses tree structures, our commit protocol is easily adaptable to other index structures such as hash tables.

Each one-shot request is dispatched to a single database worker thread, which carries out the request to completion (commit or abort) without blocking. We run one worker thread per physical core of the server machine, and have designed Silo to scale well on modern multicore machines with tens of cores. Because trees are stored in (shared) main memory, in Silo any worker can access the entire database.

Although the primary copy of data in Silo is in main memory, transactions are made durable via logging to stable storage. Results are not returned to users until they are durable.

Read-only transactions may, at the client's discretion, run on a recent consistent snapshot of the database instead of its current state. These *snapshot transactions* return slightly stale results, but never abort due to concurrent modification.

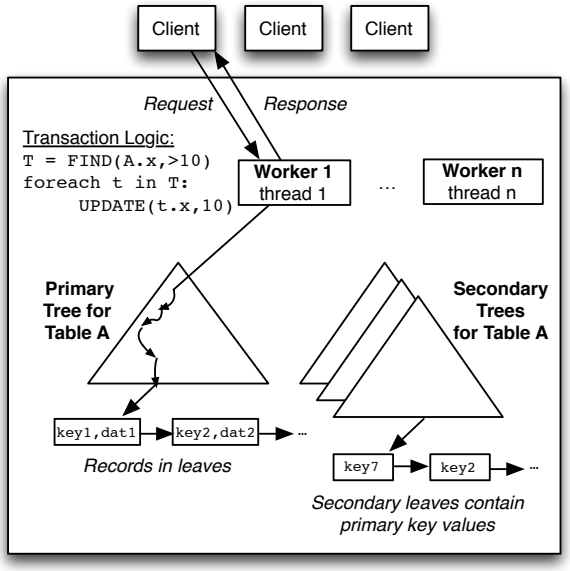

Figure 1: The architecture of Silo.

4 Design

This section describes how we execute transactions in Silo. Our key organizing principle is to eliminate unnecessary contention by reducing writes to shared memory. Our variant of OCC achieves serializability, even after recovery, using periodically-updated epochs; epoch boundaries form natural serialization points. Epochs also help make garbage collection efficient and enable snapshot transactions. Several design choices, such as transaction ID design, record overwriting, and range query support, simplify and speed up transaction execution further, and the decentralized durability subsystem also avoids contention. In the rest of this section we describe the fundamentals of our design (epochs, transaction IDs, and record layout), then present the commit protocol and explain how database operations are implemented. Later subsections describe garbage collection, snapshot transaction execution, and durability support.

4.1 Epochs

Silo is based on time periods called *epochs*, which are used to ensure serializable recovery, to remove garbage (e.g., due to deletes), and to provide read-only snapshots. Each epoch has an *epoch number*.

A *global* epoch number E is visible to all threads. A designated thread periodically advances E; other threads access E while committing transactions. E should advance frequently, since the epoch period affects transaction latency, but epoch change should be rare compared to transaction duration so that the value of E is generally cached. Our implementation updates E once every 40 ms; shorter epochs would also work. No locking is required to handle E.

Each worker w also maintains a *local* epoch number e_w. This can lag behind E while the worker computes, and is used to determine when it is safe to collect garbage. Silo requires that E and e_w never diverge too far: $E - e_w \leq 1$ for all w. This is not usually a problem since transactions are short, but if a worker does fall behind, the epoch-advancing thread delays its epoch update. Workers running very long transactions should periodically refresh their e_w values to ensure the system makes progress.

Snapshot transactions use additional epoch variables described below.

4.2 Transaction IDs

Silo concurrency control centers on *transaction IDs*, or *TIDs*, which identify transactions and record versions, serve as locks, and detect conflicts. Each record contains the TID of the transaction that most recently modified it.

TIDs are 64-bit integers. The high bits of each TID contain an epoch number, which equals the global epoch at the corresponding transaction's commit time. The middle bits distinguish transactions within the same epoch. The lower three bits are the *status bits* described below. We ignore wraparound, which is rare.

Unlike many systems, Silo assigns TIDs in a decentralized fashion. A worker chooses a transaction's TID only after verifying that the transaction can commit. At that point, it calculates the smallest number that is (a) larger than the TID of any record read or written by the transaction, (b) larger than the worker's most recently chosen TID, and (c) in the current global epoch. The result is written into each record modified by the transaction.

The TID order often reflects the serial order, but not always. Consider transactions t_1 and t_2 where t_1 happened first in the serial order. If t_1 wrote a tuple that t_2 observed (by reading it or overwriting it), then t_2's TID must be greater than t_1's, thanks to (a) above. However, TIDs do not reflect anti-dependencies (write-after-read conflicts). If t_1 merely observed a tuple that t_2 later overwrote, t_1's TID might be either less than or greater than t_2's! Nevertheless, the TIDs chosen by a worker increase monotonically and agree with the serial order, the TIDs assigned to a particular record increase monotonically and agree with the serial order, and the ordering of TIDs with different epochs agrees with the serial order.

The lower three bits of each TID word are status bits that are logically separate from the TID itself. Including these bits with the TIDs simplifies concurrency control; for example, Silo can update a record's version and unlock the record in one atomic step. The bits are a *lock* bit, a *latest-version* bit, and an *absent* bit. The lock bit protects record memory from concurrent updates; in database terms it is a *latch*, a short-term lock that protects memory structures. The latest-version bit is 1 when a record holds the latest data for the corresponding key. When a record is superseded (for instance, when an obsolete record is saved temporarily for snapshot transactions), the bit is turned off. Finally, the absent bit marks the record as equivalent to a nonexistent key; such records are created in our implementations of insert and remove.

We use "TID" to refer to a pure transaction ID and "TID word" for a TID plus status bits, including the lock.

4.3 Data layout

A record in Silo contains the following information:

- **A TID word** as above.
- **A previous-version pointer**. The pointer is null if there is no previous version. The previous version is used to support snapshot transactions (§4.9).
- **The record data**. When possible, the actual record data is stored in the same cache line as the record header, avoiding an additional memory fetch to access field values.

Committed transactions usually modify record data in place. This speeds up performance for short writes, mainly by reducing the memory allocation overhead for record objects. However, readers must then use a version validation protocol to ensure that they have read a consistent version of each record's data (§4.5).

Excluding data, records are 32 bytes on our system.

4.4 Commit protocol

We now describe how we run transactions. We first discuss transactions that only read and update existing keys. We describe how to handle inserts and deletes in §4.5 and range queries in §4.6.

As a worker runs a transaction, it maintains a *read-set* that identifies all records that were read, along with the TID of each record at the time it was accessed. For modified records, it maintains a *write-set* that stores the new state of the record (but *not* the previous TID). Records that were both read and modified occur in both the *read-set* and the *write-set*. In normal operation, all records in the *write-set* also occur in the *read-set*.

On transaction completion, a worker attempts to commit using the following protocol (see Figure 2).

In **Phase 1**, the worker examines all records in the transaction's *write-set* and locks each record by acquiring the record's lock bit. To avoid deadlocks, workers lock records in a global order. Any deterministic global order is fine; Silo uses the pointer addresses of records.

After all write locks are acquired, the worker takes a snapshot of the global epoch number using a single memory access. Fences are required to ensure that this read goes to main memory (rather than an out-of-date

```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
          or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

Figure 2: Commit protocol run at the end of every transaction.

cache), happens logically after all previous memory accesses, and happens logically before all following memory accesses. On x86 and other TSO (total store order) machines, however, these fences are compiler fences that do not affect compiled instructions; they just prevent the compiler from moving code aggressively. The snapshot of the global epoch number is the serialization point for transactions that commit.

In **Phase 2**, the worker examines all the records in the transaction's *read-set* (which may contain some records that were both read and written). If some record either has a different TID than that observed during execution, is no longer the latest version for its key, or is locked by a different transaction, the transaction releases its locks and aborts.

If the TIDs of all read records are unchanged, the transaction is allowed to commit, because we know that all its reads are consistent. The worker uses the snapshot of the global epoch number taken in Phase 1 to assign the transaction a TID as described in §4.2.

Finally, in **Phase 3**, the worker writes its modified records to the tree and updates their TIDs to the transaction ID computed in the previous phase. Each lock can be released immediately after its record is written. Silo must ensure that the new TID is visible as soon as the lock is released; this is simple since the TID and lock share a word and can be written atomically.

Serializability This protocol is serializable because (1) it locks all written records before validating the TIDs of read records, (2) it treats locked records as dirty and aborts on encountering them, and (3) the fences that close Phase 1 ensure that TID validation sees all concurrent updates. We argue its serializability by reduction to strict two-phase locking (S2PL): if our OCC validation protocol commits, then so would S2PL. To simplify the explanation, assume that the *write-set* is a subset of the *read-set*. S2PL acquires read locks on all records read (which, here, includes all records written) and releases them only after commit. Consider a record in the *read-set*. We verify in Phase 2 that the record's TID has not changed since its original access, and that the record is not locked by any other transaction. This means that S2PL would have been able to obtain a read lock on that record and hold it up to the commit point. For updated records, S2PL can be modeled as upgrading shared read locks to exclusive write locks at commit time. Our protocol obtains exclusive locks for all written records in Phase 1, then in Phase 2 verifies that this is equivalent to a read lock that was held since the first access and then upgraded. The fences that close Phase 1 ensure that the version checks in Phase 2 access records' most recent TID words.

Thread 1	**Thread 2**
$t_1 \leftarrow \text{read}(x)$	$t_2 \leftarrow \text{read}(y)$
$\text{write}(y \leftarrow t_1 + 1)$	$\text{write}(x \leftarrow t_2 + 1)$

Figure 3: A read-write conflict between transactions.

The protocol also ensures that epoch boundaries agree with the serial order. Specifically, committed transactions in earlier epochs never transitively depend on transactions in later epochs. This holds because the memory fences ensure that all workers load the latest version of E before read validation and after write locking. Placing the load before read validation ensures that committed transactions' *read-* and *node-set*s never contain data from later epochs; placing it after write locking ensures that all transactions in later epochs would observe at least the lock bits acquired in Phase 1. Thus, epochs obey both dependencies and anti-dependencies.

For example, consider two records x and y that both start out with value zero. Given the two transactions shown in Figure 3, the state $x = y = 1$ is not a valid serializable outcome. We illustrate why this final state is impossible in Silo. For it to occur, thread 1 must read $x = 0$, thread 2 must read $y = 0$, and both threads must commit. So assume that thread 1 reads $x = 0$ and commits, which means that thread 1's Phase 2 validation verified that x was unlocked and unchanged. Since thread 2 locks x as part of its Phase 1, we know then that thread 2's serialization point follows thread 1's. Thus, thread 2 will observe either thread 1's lock on y (which was acquired before thread 1's serialization point) or a new version

number for y. It will either set $y \leftarrow 2$ or abort.

4.5 Database operations

This section provides more detail about how we support different database operations.

Reads and writes When a transaction commits, we overwrite modified records if possible since this improves performance. If it is not possible (e.g., because the new record is larger), we create new storage to contain the new data, mark the old record as no longer being most recent, and modify the tree to point to the new version. Modification in place means that concurrent readers might not see consistent record data. The *version validation* protocol described below is used to detect this problem.

To modify record data during Phase 3 of the commit protocol, a worker while holding the lock (a) updates the record, (b) performs a memory fence, and (c) stores the TID and releases the lock. The consistency requirement is that if a concurrent reader sees a released lock, it must see both the new data and the new TID. Step (b) ensures that the new data is visible first (on TSO machines this is a compiler fence); Step (c) exposes the new TID and released lock atomically since the lock is located in the TID word.

To access record data during transaction execution (outside the commit protocol), a worker (a) reads the TID word, spinning until the lock is clear, (b) checks whether the record is the latest version, (c) reads the data, (d) performs a memory fence, and (e) checks the TID word again. If the record is not the latest version in step (b) or the TID word changes between steps (a) and (e), the worker must retry or abort.

Deletes Snapshot transactions require that deleted records stay in the tree: linked versions relevant for snapshots must remain accessible. A delete operation therefore marks its record as "absent" using the absent bit and registers the record for later garbage collection. Clients treat absent records like missing keys, but internally, absent records are treated like present records that must be validated on read. Since most absent records are registered for future garbage collection, Silo writes do not overwrite absent record data.

Inserts Phase 2 handles write-write conflicts by requiring transactions to first lock records. However, when the record does not exist, there is nothing to lock. To avoid this problem, we insert a new record for the insert request *before* starting the commit protocol.

An insert operation on key k works as follows. If k already maps to an non-absent record, the insert fails and the transaction is aborted. Otherwise, a new record r is constructed in the absent state and with TID 0, a mapping from $k \rightarrow r$ is added to the tree (using an *insert-if-absent* primitive), and r is added to both the *read-set* and the *write-set* as if a regular put occurred. The *insert-if-absent* primitive ensures that there is never more than one record for a given key; Phase 2's *read-set* validation ensures that no other transactions have superseded the placeholder.

A transaction that does an insert commits in the usual way. If the commit succeeds, the new record is overwritten with its proper value and the transaction's TID. If the commit fails, the commit protocol registers the absent record for future garbage collection.

4.6 Range queries and phantoms

Silo supports range queries that allow a transaction to access a range of the keys in a table. Range queries are complicated by the *phantom problem* [10]: if we scanned a particular range but only kept track of the records that were present during the scan, membership in the range could change without being detected by the protocol, violating serializability.

The typical solution to this problem in database systems is *next-key locking* [24]. Next-key locking, however, requires locking for reads, which goes against Silo's design philosophy.

Silo deals with this issue by taking advantage of the underlying B^+-tree's version number on each leaf node. The underlying B^+-tree guarantees that structural modifications to a tree node result in a version number change for all nodes involved. A scan on the interval $[a,b)$ therefore works as follows: in addition to registering all records in the interval in the *read-set*, we also maintain an additional set, called the *node-set*. We add the leaf nodes that overlap with the key space $[a,b)$ to the *node-set* along with the version numbers examined during the scan. Phase 2 checks that the version numbers of all tree nodes in the *node-set* have not changed, which ensures that no new keys have been added or removed within the ranges examined.

Phantoms are also possible due to lookups and deletes that fail because there is no entry for the key in the tree; in either case the transaction can commit only if the key is still not in the tree at commit time. In this case the node that would contain the missing key is added to the *node-set*.

The careful reader will have noticed an issue with insert operations, which can trigger structural modifications, and *node-set* tracking. Indeed, we need to distinguish between structural modifications caused by concurrent transactions (which must cause aborts) and modifications caused by the current transaction (which should *not*). We fix this problem as follows. Recall that the tree modifications for an insertion actually occur be-

fore commit time. For each tree node n that is affected by the insert (there can be more than one due to splits), let v_{old} be its version number before the insert, and v_{new} its version number afterwards. The insertion then updates the *node-set*: if n is in the *node-set* with version number v_{old}, it is changed to v_{new}, otherwise the transaction is aborted. This means only other concurrent modifications to n will cause an abort.

4.7 Secondary indexes

To Silo's commit protocol, secondary indexes are simply additional tables that map secondary keys to records containing primary keys. When a modification affects a secondary index, the index is explicitly changed via extra accesses in the code for the transaction that did the modification. These modifications will cause aborts in the usual way: a transaction that used a secondary index will abort if the record it accessed there has changed.

4.8 Garbage collection

Silo transactions generate two sources of garbage that must be reaped: B^+-tree nodes and database records. Rather than reference count these objects (which would require that all accesses write to shared memory), Silo leverages its epochs into an epoch-based reclamation scheme à la read-copy update (RCU) [11, 13].

When a worker generates garbage—for example, by deleting a record or by removing a B^+-tree node—it registers the garbage object and its *reclamation epoch* in a per-core list for that object type. The reclamation epoch is the epoch after which no thread could possibly access the object; once it is reached, we can free the object. This could happen either in a separate background task or in the workers themselves. We do it in the workers between requests, which reduces the number of helper threads and can avoid unnecessary data movement between cores.

Silo's snapshot transactions mean that different kinds of objects have different reclamation policies, though all are similar. The simplest policy is for B^+-tree nodes. Recall that every worker w maintains a local epoch number e_w that is set to E just before the start of each transaction. A tree node freed during a transaction is given reclamation epoch e_w. Thanks to the way epochs advance, no thread will ever access tree nodes freed during an epoch $e \leq \min e_w - 1$. The epoch-advancing thread periodically checks all e_w values and sets a global *tree reclamation epoch* to $\min e_w - 1$. Garbage tree nodes with smaller or equal reclamation epochs can safely be freed.

4.9 Snapshot transactions

We support running read-only transactions on recent-past snapshots by retaining additional versions for records. These versions form a *consistent* snapshot that contains all modifications of transactions up to some point in the serial order, and none from transactions after that point. Snapshots are used for running snapshot transactions (and could also be helpful for checkpointing). Managing snapshots has two tricky parts: ensuring that the snapshot is consistent and complete, and ensuring that its memory is eventually reclaimed.

We provide consistent snapshots using *snapshot epochs*. Snapshot epoch boundaries align with epoch boundaries, and thus are consistent points in the serial order. However, snapshot epochs advance more slowly than epochs. (We want to avoid frequent snapshot creation because snapshots are not free.) The snapshot epoch $\mathit{snap}(e)$ for epoch e equals $k \cdot \lfloor e/k \rfloor$; currently $k = 25$, so a new snapshot is taken about once a second.

The epoch-advancing thread periodically computes a global *snapshot epoch* $SE \leftarrow \mathit{snap}(E - k)$. Every worker w also maintains a local snapshot epoch se_w; at the start of each transaction, it sets $se_w \leftarrow SE$. Snapshot transactions use this value to find the right record versions. For record r, the relevant version is the most recent one with epoch $\leq se_w$. When a snapshot transaction completes it commits without checking; it never aborts, since the snapshot is consistent and is never modified.

Read/write transactions must not delete or overwrite record versions relevant for a snapshot. Consider a transaction committing in epoch E. When modifying or deleting a record r in Phase 3, the transaction compares $\mathit{snap}(\mathit{epoch}(r.\mathit{tid}))$ and $\mathit{snap}(E)$. If these values are the same, then it is safe to overwrite the record; the new version would supersede the old one in any snapshot. If the values differ, the old version must be preserved for current or future snapshot transactions, so the read/write transaction installs a new record whose *previous-version* pointer links to the old one. Snapshot transactions will find the old version via the link. (When possible, we actually copy the *old* version into new memory linked from the existing record, so as to avoid dirtying tree-node cache lines, but the semantics are the same.)

A record version should be freed once no snapshot transaction will access it. This is tied to snapshot epochs. When a transaction committing in epoch E allocates memory for a snapshot version, it registers that memory for reclamation with epoch $\mathit{snap}(E)$. The epoch-advancing thread periodically computes a *snapshot reclamation epoch* as $\min se_w - 1$. Snapshot versions with equal or smaller reclamation epochs can safely be freed. The reclamation process need not adjust any *previous-version* pointers: the dangling pointer to the old version will never be traversed, since any future snapshot transaction will prefer the newer version.

Deletions, however, require special handling. Deleted records should be unhooked from the tree, but this cannot happen right away: snapshot transactions must be able to find the linked older versions. The commit pro-

cess therefore creates an "absent" record whose status bits mark the value as deleted; the space for record data instead stores the relevant key. This absent record is registered for cleanup at reclamation epoch $snap(E)$. When the snapshot reclamation epoch reaches this value, the cleanup procedure modifies the tree to remove the reference to the record (which requires the key), and then registers the absent record for reclamation based on the tree reclamation epoch (the record cannot be freed right away since a concurrent transaction might be accessing it). However, if the absent record was itself superseded by a later insert, the tree should not be modified. The cleanup procedure checks whether the absent record is still the latest version; if not, it simply ignores the record. No further action is required, because such a record was marked for future reclamation by the inserting transaction.

4.10 Durability

Durability is a transitive property: a transaction is durable if all its modifications are recorded on durable storage and all transactions serialized before it are durable. Thus, durability requires the system to recover the effects of a set of transactions that form a prefix of the serial order. In Silo, epochs provide the key to recovering this prefix, just as they do for snapshot transactions. Since epoch boundaries are consistent with the serial order, Silo treats whole epochs as the durability commit units. The results of transactions in epoch e are not released to clients until all transactions with epochs $\leq e$ have been stored durably.

Durable storage could be obtained through replication, but our current implementation uses logging and local storage (disks). All transactions in epoch e are logged together. After a failure, the system examines the logs and finds the *durable epoch* D, which is the latest epoch whose transactions were all successfully logged. It then recovers all transactions with epochs $\leq D$, and *no more*. Not recovering more is crucial: although epochs as a whole are serially consistent, the serial order *within* an epoch is not recoverable from the information we log, so replaying a subset of an epoch's transactions might produce an inconsistent state. This also means the epoch period directly affects the average case latency required for a transaction to commit.

Logging in Silo is handled by a small number of logger threads, each of which is responsible for a disjoint subset of the workers. Each logger writes to a log file on a separate disk.

When a worker commits a transaction, it creates a log record consisting of the transaction's TID and the table/key/value information for all modified records. This log record is stored in a local memory buffer in disk format. When the buffer fills or a new epoch begins, the worker publishes its buffer to its corresponding logger using a per-worker queue, and then publishes its last committed TID by writing to a global variable $ctid_w$.

Loggers operate in a continuous loop. At the start of each iteration, the logger calculates $t = \min ctid_w$ for its workers. From this TID, it computes a *local durable epoch* $d = epoch(t) - 1$. All its workers have published all their transactions with epochs $\leq d$, so as far as this logger is concerned, epochs $\leq d$ are complete. It appends all the buffers plus a final record containing d to the end of a log file and waits for these writes to complete. (It never needs to examine the actual log buffer memory.) It then publishes d to a per-logger global variable d_l and returns the buffers back to the workers for recycling.

A thread periodically computes and publishes a *global durable epoch* $D = \min d_l$. All transactions with epochs $\leq D$ are known to have been durably logged. Workers can thus respond to clients whose transactions occurred in epochs $\leq D$.

Silo uses record-level redo logging exclusively, not undo logging or operation logging. Undo logging is not necessary for our system since we log after transactions commit; logging at record level simplifies recovery.

To recover, Silo would read the most recent d_l for each logger, compute $D = \min d_l$, and then replay the logs, ignoring entries for transactions whose TIDs are from epochs after D. Log records for the same record must be applied in TID order to ensure that the result equals the latest version, but replaying can otherwise be performed concurrently.

A full system would recover from a combination of logs and checkpoints to support log truncation. Checkpoints could take advantage of snapshots to avoid interfering with read/write transactions. Though checkpointing would reduce worker performance somewhat, it need not happen frequently. We evaluate common-case logging, but have not yet implemented full checkpointing or recovery.

5 Evaluation

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

- The cost of Silo's read/write set tracking for a simple key-value workload is low (§5.2).
- Silo scales as more cores become available, even when transactions are made persistent (§5.3).
- Silo's performance is more robust to workload changes compared to a partitioned data store, specficially as we increase the level of cross-partition contention and the skew of the workload (§5.4).
- Large read-only transactions benefit substantially from Silo's snapshot transactions (§5.5). This mech-

anism incurs only a small space overhead, even for a write heavy workload (§5.6).

We also show in §5.7 the relative importance of several of Silo's implementation techniques.

5.1 Experimental setup

All 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. We disabled hyperthreading on all CPUs; we found slightly worse results with hyperthreading enabled. The machine has 256GB of DRAM with 64GB of DRAM attached to each socket, and runs 64-bit Linux 3.2.0.

In all graphs, each point reported is the median of three consecutive runs, with the minimum and maximum values shown as error bars. In our experiments, we follow the direction of Masstree and size both internal nodes and leaf nodes of our B$^+$-tree to be roughly four cache lines (a cache line is 64 bytes on our machine), and use software prefetching when reading B$^+$-tree nodes.

We pay careful attention to memory allocation and thread affinity in our experiments. We use a custom memory allocator for both B$^+$-tree nodes and records. Our allocator takes advantage of 2MB "superpages" (a feature supported in recent Linux kernels) to reduce TLB pressure. We pin threads to cores so that the memory allocated by a particular thread resides on that thread's NUMA node. Before we run an experiment, we make sure to pre-fault our memory pools so that the scalability bottlenecks of Linux's virtual memory system [5] are not an issue in our benchmarks.

For experiments which run with persistence enabled, we use four logger threads and assign each logger a file residing on a separate device. Three loggers are assigned different Fusion IO ioDrive2 devices, and the fourth is assigned to six 7200RPM SATA drives in a RAID-5 configuration. Collectively, these devices provide enough bandwidth that writing to disk is not a bottleneck.

Our experiments do not use networked clients. We would expect networking to reduce our throughput somewhat; in Masstree, networked clients reduced throughput by 23% [23]. In our experiments, each thread combines a database worker with a workload generator. These threads run within the same process, and share Silo trees in the same address space. We run each experiment for 60 seconds. When durability is not an issue, our experiments use *MemSilo*, which is Silo with logging disabled.

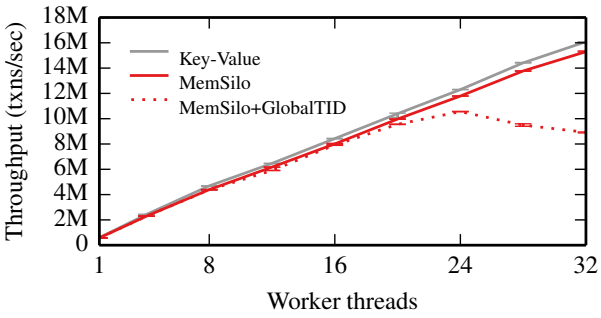

Figure 4: Overhead of *MemSilo* versus *Key-Value* when running a variant of the YCSB benchmark.

5.2 Overhead of small transactions

This section shows that Silo's read/write set tracking has low overhead by comparing to the underlying key-value store, which does not perform any tracking.

We evaluate two systems. The first system, *Key-Value*, is simply the concurrent B$^+$-tree underneath Silo. That is, *Key-Value* provides only single-key gets and puts. The second system is *MemSilo*. We ran a variant of YCSB workload mix A; YCSB [6] is Yahoo's popular key-value benchmark. Our variant differs in the following ways: (a) we fix the read/write ratio to 80/20 (instead of 50/50), (b) we change write operations to read-modify-writes, which in *MemSilo* happen in a single transaction, and (c) we shrink the size of records to 100 bytes (instead of 1000 bytes). These modifications prevent uninteresting overheads that affect both systems from hiding overheads that affect only *MemSilo*. More specifically, (a) prevents the memory allocator for new records from becoming the primary bottleneck, (b) creates a transaction that actually generates read-write conflicts, which stresses *MemSilo*'s protocol, and (c) prevents `memcpy` from becoming the primary bottleneck. Both transactions sample keys uniformly from the key space. We fix the tree size to contain 160M keys, and vary the number of workers performing transactions against the tree.

Figure 4 shows the results of running the YCSB benchmark on both systems. The overhead of *MemSilo* compared to *Key-Value* is negligible; *Key-Value* outperforms *MemSilo* by a maximum of 1.07×.

Globally generated TIDs Figure 4 also quantifies the benefits of designing Silo to avoid a single globally unique TID assigned at commit time. *MemSilo+GlobalTID* follows an identical commit protocol as Silo, except it generates TIDs from a single shared counter. This is similar to the critical section found in Larson et al. [19] Here we see a scalability collapse after 24 workers; *Key-Value* outperforms *MemSilo+GlobalTID* by 1.80× at 32 workers. This demonstrates the necessity of avoiding even a *single* global atomic instruction during the commit phase.

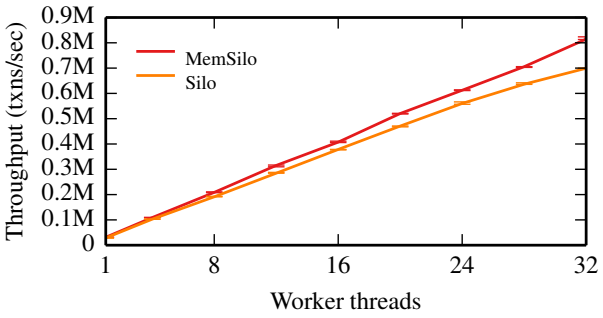

Figure 5: Throughput of Silo when running the TPC-C benchmark, including persistence.

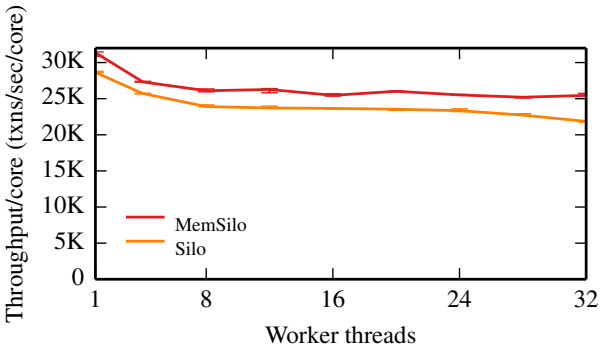

Figure 6: Per-core throughput of Silo when running the TPC-C benchmark.

5.3 Scalability and persistence

Our next experiment shows that Silo scales effectively as we increase the number of workers. We use the popular TPC-C benchmark [33], which models a retail operation and is a common benchmark for OLTP workloads. Transactions in TPC-C involve customers assigned to a set of districts within a *local* warehouse, placing orders in those districts. Most orders can be fulfilled entirely from the local warehouse, but a small fraction request a product from a *remote* warehouse. To run TPC-C on Silo, we assign all clients with the same local warehouse to the same thread. This models client affinity, and means memory accessed by a transaction usually resides on the same NUMA node as the worker. We size the database such that the number of warehouses equals the number of workers, so the database grows as more workers are introduced (in other words, we fix the contention ratio of the workload). We do not model client "think" time, and we run the standard workload mix involving all five transactions. To show how durability affects scalability, we run both *MemSilo* and *Silo*.

Figures 5 and 6 show the throughput of running TPC-C as we increase the number of workers (and thus warehouses) in both *MemSilo* and *Silo*. Scaling is close to linear for *MemSilo* up to 32 workers. The per-core throughput of *MemSilo* at 32 workers is 81% of the throughput at one worker, and 98% of the per-core throughput at

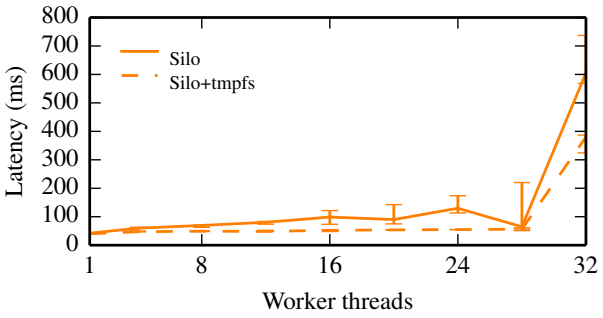

Figure 7: Silo transaction latency for TPC-C with logging to either durable storage or an in-memory file system.

8 workers. This drop happens because of several factors, including increasing database size and sharing of resources such as the L3 cache (particularly important from one to 8 threads) as well as actual contention. *Silo* performs slightly slower, and its scalability is degraded by logging past 28 workers, since worker threads and the four logger threads start to contend for cores.

We also ran experiments to separate the impact of Silo's logging subsystem from that of our durable storage devices. With Silo's loggers writing to an in-memory file system instead of stable storage (*Silo+tmpfs*), we observed at most a 1.03× gain in throughput over *Silo*. This argues that most of the loss in throughput when enabling durability is due to the overhead of transferring log records from worker threads to logger threads, rather than the overhead of writing to physical hardware (given sufficient hardware bandwidth).

Figure 7 shows transaction latency for this experiment (the latency of *MemSilo* is negligible and therefore not shown). For both *Silo+tmpfs* and *Silo*, we see a spike in latency around 28 workers due to worker/logger thread contention. The effect is more pronounced with *Silo*, illustrating the fact that latency is more sensitive to real hardware than throughput is.

Overall, we see that logging does not significantly degrade the throughput of Silo and incurs only a modest increase in latency. *MemSilo* outperforms *Silo* by only a maximum of 1.16× at 32 workers.

5.4 Comparison with Partitioned-Store

This section describes our evaluation of Silo versus a statically partitioned data store, which is a common configuration for running OLTP on a single shared-memory node [25, 30, 32]. Most of the time Silo is preferable.

Tables in TPC-C are typically partitioned on *warehouse-id*, such that each partition is responsible for the districts, customers, and stock levels for a particular warehouse. This is a natural partitioning, because as mentioned in §5.3, each transaction in TPC-C is centered around a single local warehouse.

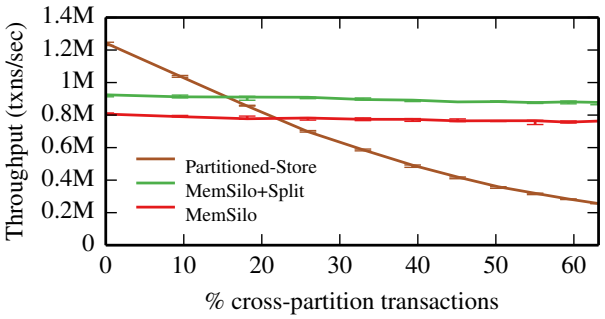

Figure 8: Performance of *Partitioned-Store* versus *MemSilo* as the percentage of cross-partition transactions is varied.

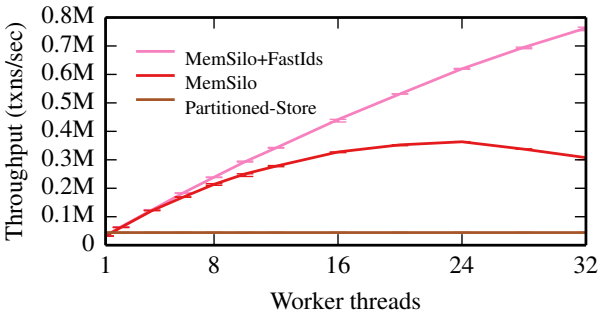

Figure 9: Performance of *Partitioned-Store* versus *MemSilo* as the number of workers processing the same size database is varied.

The design of *Partitioned-Store* is motivated by H-Store/VoltDB [32]. We physically partition the data by warehouse so that each partition has a separate set of B$^+$-trees for each table, and we associate each partition with a single worker. The workers are all in the same process to avoid IPC overhead. We then associate a global *partition lock* with each partition. Every transaction first obtains all partition locks (in sorted order). Once all locks are obtained, the transaction can proceed as in a single-threaded data store, without the need for additional validation. We assume that we have perfect knowledge of the partition locks needed by each transaction, so once the locks are obtained the transaction is guaranteed to commit and never has to acquire new locks. We implement these partition locks using spinlocks, and take extra precaution to allocate the memory for the locks on separate cache lines to prevent false sharing. When single-partition transactions are common, these spinlocks are cached by their local threads and obtaining them is inexpensive. We believe this scheme performs at least as well on a multicore machine as the cross-partition locking schemes described for H-Store. *Partitioned-Store* does not support snapshot transactions, so it incurs no overhead for maintaining multiple record versions. *Partitioned-Store* uses the same B$^+$-trees that *Key-Value* and Silo use, except we remove the concurrency control mechanisms in place in the B$^+$-tree. Additionally, we remove the concurrency control for record values (which *Key-Value* needed to ensure atomic reads/writes to a single key). *Partitioned-Store* does not implement durability.

We evaluate the costs and benefits of partitioning by comparing a partitioned version of Silo (*Partitioned-Store*) with Silo itself (*MemSilo*). We run two separate experiments: the first varies the percentage of cross-partition transactions in the workload, and the second varies the number of concurrent workers executing transactions against a fixed-size database.

Cross-partition transactions We first fix the database size and number of workers, and vary the percentage of cross-partition transactions.

As in our previous experiments on TPC-C, Silo executes transactions by associating each worker with a local warehouse and issuing queries against the shared B$^+$-trees. We focus on the most frequent transaction in TPC-C, the new-order transaction. Even though the new-order transaction is bound to a local warehouse, each transaction has some probability, which we vary, of writing to records in non-local warehouses (these records are in the stock table). Our benchmark explores the tradeoffs between *Partitioned-Store* and various versions of Silo as we increase the probability of a cross-partition transaction from zero to over 60 percent.

Figure 8 shows the throughput of running all new-order transactions on various versions of Silo and *Partitioned-Store* as we increase the number of cross-partition transactions. Both the number of warehouses and the number of workers are fixed at 28. The parameter varied is the probability that a *single* item is drawn from a remote warehouse. On the x-axis we plot the probability that any given transaction will touch *at least one* remote warehouse (each new-order transaction includes between 5 to 15 items, inclusive) and thus require grabbing more than one partition lock in *Partitioned-Store*.

The curves in Figure 8 show that *Partitioned-Store* is clearly the optimal solution for perfectly partitionable workloads. The advantages are two-fold: no concurrency control mechanisms are required, and there is better cache locality due to the partitioned trees being smaller. *Partitioned-Store* outperforms *MemSilo* by 1.54× at no cross-partition transactions. However, performance suffers as soon as cross-partition transactions are introduced. At roughly 20%, the throughput of *Partitioned-Store* drops below *MemSilo* and gets worse with increasing contention, whereas *MemSilo* maintains relatively steady throughput for the entire experiment. At roughly 60% cross-partition transactions, *MemSilo* outperforms *Partitioned-Store* by 2.98×. The results here can be un-

derstood as the tradeoff between coarse and fine-grained locking. While Silo's OCC protocol initially pays a non-trivial overhead for tracking record-level changes in low contention regimes, this work pays off as the contention increases.

To understand the overhead in more detail, we introduce *MemSilo+Split*, which extends *MemSilo* by physically splitting its tables the same way as *Partitioned-Store* does. This yields a 13% gain over *MemSilo*. The remaining difference is due to *Partitioned-Store* requiring no fine-grained concurrency control.

Skewed workloads This next experiment shows the impact of workload skew, or "hotspots," on performance in *Partitioned-Store* and Silo. We run a 100% new-order workload mix, fixing the database size to four warehouses in a single partition. We then vary the number of available workers, which simulates increased workload skew (more workers processing transactions over a fixed size database). Figure 9 shows the results of this experiment.

For *Partitioned-Store*, the throughput stays constant as we increase workers, because multiple workers cannot execute in parallel on a single partition (they serialize around the partition lock). For *MemSilo*, throughput increases, but not linearly, because of actual workload contention when running 100% new-order transactions. Specifically, a counter record per unique *(warehouse-id, district-id)* pair is used to generate new-order IDs. As we add more workers, we increase the number of read-write conflicts over this shared counter, which increases the number of aborted transactions. Note that this is not unique to OCC; in 2PL, conflicting workers would serialize around write locks held on the shared counter. In Figure 9, we see that *MemSilo* outperforms *Partitioned-Store* by a maximum of 8.20× at 24 workers.

Because the contention in this workload is a property of the workload itself and not so much the system, it is interesting to see how *MemSilo* responds if the contention is removed. We substantially reduce this contention in *MemSilo+FastIds* by generating IDs *outside* the new-order transaction. The new-order client request runs two transactions, where the first generates a unique ID and the second uses that ID. (This sacrifices the invariant that the new-order ID space is contiguous, since counters do not roll back on abort.) The result is throughput scaling nicely until 28 workers, when the next contention bottleneck in the workload appears. Overall *MemSilo+FastIds* outperforms *Partitioned-Store* by a maximum of 17.21× at 32 workers.

5.5 Effectiveness of snapshot transactions

In this section we show that Silo's snapshot transactions are effective for certain challenging workloads. Maintaining snapshots is not free (see §5.7); for some workloads where the frequency of large read-only transactions is low, such as the standard TPC-C workload, the benefits of snapshots do not outweigh their overhead. However, for other workloads involving many large read-only transactions over frequently updated records, such as the one evaluated in this section, we show that snapshots provide performance gains.

	Transactions/sec	**Aborts/sec**
MemSilo	200,252	2,570
MemSilo+NoSS	168,062	15,756

Figure 10: Evaluation of Silo's snapshot transactions on a modified TPC-C benchmark.

We again use the TPC-C benchmark, changing the setup as follows. We fix the number of warehouses to 8 and the number of workers to 16 (each warehouse is assigned to two workers). We run a transaction mix of 50% new-order and 50% stock-level; this is the largest of two read-only queries from TPC-C. On average, stock-level touches several hundred records in tables frequently updated by the new-order transaction, performing a nested-loop join between the order-line table and the stock table.

We measure the throughput of Silo under this load in two scenarios: one where we use Silo's snapshot transactions to execute the stock-level transaction at roughly one second in the past (*MemSilo*), and one where we execute stock-level as a regular transaction in the present (*MemSilo+NoSS*). In both scenarios, the new-order transaction executes as usual.

Figure 10 shows the results of this experiment. *MemSilo* outperforms *MemSilo+NoSS* by 1.19×. This is due to the larger number of aborts that occur when a large read-only transaction executes with concurrent modifications (recall that snapshot transactions never abort).

5.6 Space overhead of snapshots

This section shows that the space overhead of maintaining multiple versions of records to support snapshot transactions is low, even for update heavy workloads.

We run a variant of YCSB where every transaction is a read-modify-write operation on a single record. We pick this operation because each transaction has a very high probability of generating a new version of a record (since we size the tree to contain 160M keys). This stresses our garbage collector (and memory allocator), because at every snapshot epoch boundary a large number of old records need to be garbage collected. As in §5.2, we use 100 byte records and uniformly sample from the key space. We run *MemSilo* with 28 workers.

In this experiment, the database records initially occupied 19.5GB of memory. Throughout the entire run, this increased by only a maximum of 672.3MB, representing a 3.4% increase in memory for snapshots. This

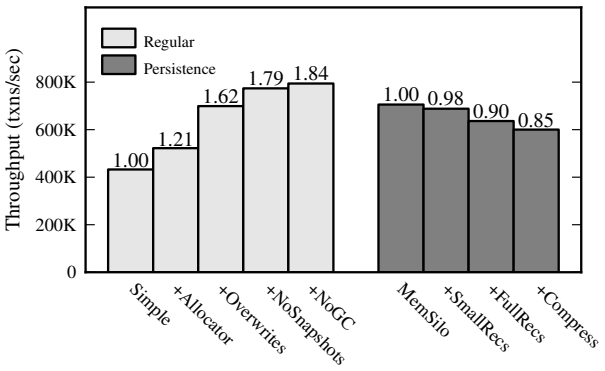

Figure 11: A factor analysis for Silo. Changes are added left to right for each group, and are cumulative.

shows that, despite having to keep old versions around, space overhead is reasonable and garbage collection is very effective at reaping old versions.

5.7 Factor analysis

To understand in greater detail the overheads and benefits of various aspects of Silo, we show a factor analysis in Figure 11 that highlights the key factors for performance. The workload here is the standard TPC-C mix running with 28 warehouses and 28 workers. We perform analysis in two groups of cumulative changes: the *Regular* group deals with changes that do not affect the persistence subsystem, and the *Persistence* group deals only with changes in the persistence layer.

In the *Regular* group, *Simple* refers to Silo running with no NUMA aware memory allocator and allocating a new record for each write. *+Allocator* adds the NUMA aware allocator described in §5.1. *+Overwrites* allows Silo to write record modifications in place when possible (and is equivalent to *MemSilo* as presented in Figure 5). *+NoSnapshots* disables keeping multiple versions of records for snapshots, and does not provide any snapshot query mechanism. *+NoGC* disables the garbage collector described in §4.8.

The two major takeaways from Figure 11 are that performing in-place updates is an important optimization for Silo and that the overhead of maintaining snapshots plus garbage collection is low.

For persistence analysis, we break down the following factors: *MemSilo* refers to running without logging. *+SmallRecs* enables logging, but only writes log records that are eight bytes long (containing a TID only, omitting record modifications); this illustrates an upper bound on performance for any logging scheme. *+FullRecs* is Silo running with persistence enabled (and is equivalent to *Silo* in Figure 5). *+Compress* uses LZ4 compression[1] to compress log records before writing to disk.

[1] http://code.google.com/p/lz4/

The takeaways from Figure 11 are that spending extra CPU cycles to reduce the amount of bytes written to disk surprisingly does not pay off for this TPC-C workload, and that the overhead of copying record modifications into logger buffers and then into persistent storage is low.

6 Conclusions

We have presented Silo, a new OCC-based serializable database storage engine designed for excellent performance at scale on large multicore machines. Silo's concurrency control protocol is optimized for multicore performance, avoiding global critical sections and non-local memory writes for read operations. Its use of epochs facilitates serializable recovery, garbage collection, and efficient read-only snapshot transactions. Our results show Silo's near-linear scalability on YCSB-A and TPC-C, very high transaction throughput on TPC-C, and low overheads relative to a non-transactional system. Together, these results show that transactional consistency and scalability can coexist at high performance levels in shared-memory databases.

Given the encouraging performance of Silo, we are investigating a number of areas for future work, designed to further improve its performance and augment its usability. These include further exploiting "soft partitioning" techniques to improve performance on highly partitionable workloads; fully implementing checkpointing, recovery, and replication; investigating how to balance load across workers most effectively, perhaps in a way that exploits core-to-data affinity; and integrating into a full-featured SQL engine.

Acknowledgements

We thank our shepherd Timothy Roscoe and the anonymous reviewers for their feedback. Eugene Wu provided helpful comments on an earlier draft of the paper. We are also very grateful to Yandong Mao, who generously ran benchmarking experiments on other systems. This work was supported by NSF Grants 1065219 and 0704424. Eddie Kohler's work was partially supported by a Sloan Research Fellowship and a Microsoft Research New Faculty Fellowship.

References

[1] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1), 1987.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 2005.

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

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

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Eurosys*, 2013.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[8] 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 *SIGMOD*, 2013.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[10] 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.

[11] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. URL http://www.cl.cam.ac.uk/users/kaf24/lockfree.html.

[12] T. Härder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2), 1984.

[13] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12), 2007.

[14] T. Horikawa. Latch-free data structures for DBMS: design, implementation, and evaluation. In *SIGMOD*, 2013.

[15] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, 2009.

[16] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.

[17] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, 2013.

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

[19] 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.

[20] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM TODS*, 6(4), 1981.

[21] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware. In *ICDE*, 2013.

[22] J. Levandoski, D. Lomet, and S. Sengupta. LLAMA: A cache/storage subsystem for modern hardware. *Proc. VLDB Endow.*, 6(10), 2013.

[23] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Eurosys*, 2012.

[24] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *VDLB*, 1990.

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

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

[27] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.

[28] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on hardware islands. *Proc. VLDB Endow.*, 5(11), 2012.

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

[30] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *Eurosys*, 2011.

[31] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endow.*, 4(11), 2011.

[32] 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.

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

[34] R. Unland. Optimistic concurrency control revisited. Technical report, University of Münster, Department of Information Systems, 1994.