ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads

Kangnyeon Kim[†] Tianzheng Wang[†] Ryan Johnson[‡] Ippokratis Pandis[*]

[†]University of Toronto
knkim,tzwang@cs.utoronto.ca

[‡]LogicBlox
ryan.johnson@logicblox.com

[*]Amazon Web Services
ippo@amazon.com

ABSTRACT

Large main memories and massively parallel processors have triggered not only a resurgence of high-performance transaction processing systems optimized for large main-memory and massively parallel processors, but also an increasing demand for processing heterogeneous workloads that include *read-mostly* transactions. Many modern transaction processing systems adopt a lightweight optimistic concurrency control (OCC) scheme to leverage its low overhead in low contention workloads. However, we observe that the lightweight OCC is not suitable for heterogeneous workloads, causing significant starvation of read-mostly transactions and overall performance degradation.

In this paper, we present ERMIA, a memory-optimized database system built from scratch to cater the need of handling heterogeneous workloads. ERMIA adopts snapshot isolation concurrency control to coordinate heterogeneous transactions and provides serializability when desired. Its physical layer supports the concurrency control schemes in a scalable way. Experimental results show that ERMIA delivers comparable or superior performance and near-linear scalability in a variety of workloads, compared to a recent lightweight OCC-based system. At the same time, ERMIA maintains high throughput on read-mostly transactions when the performance of the OCC-based system drops by orders of magnitude.

1. INTRODUCTION

Modern systems with massively parallel processors and large main memories have inspired a new breed of high-performance in-memory transaction processing systems [21, 22, 26, 29, 35, 45]. These systems leverage spacious main memory to fit the whole working set or even the whole database in DRAM with streamlined memory-friendly data structures. Multicore and multi-socket hardware optimizations further allow a much higher level of parallelism compared to conventional systems. With disk overheads and delays removed, transaction latencies drop precipitously and threads can usually execute transactions to completion without interruption. The result is a welcome reduction in contention at the logical level and less pressure on whatever the concurrency control (CC) scheme might be in place. A less welcome result is an increasing pressure for scalable data structures and algorithms to cope with the growing number of threads that concurrently execute transactions and need to communicate with each other [17].

Interactions at the logical level. Many designs exploit the reduction in the pressure on CC, by employing lightweight optimistic concurrency control (OCC) schemes, boosting even further the performance of these systems on suitable workloads [11, 23, 45]. But, as is usually the case, it appears that database workloads stand ready to absorb any and all concurrency gains the memory-optimized systems have to offer. In particular, there is high demand for database systems that can readily serve heterogeneous workloads, blending the gap between transaction and analytical processing [13, 25, 40]. These workloads often combine long *read-mostly* transactions for operational analytic with short and write-intensive transactions.

Lightweight OCC can be problematic in such workloads. First, its unfair (albeit lightweight) contention resolution, where a writer wins over a reader, leads to starvation of read-intensive transactions, as transactions have to abort if any portion of their read footprint is overwritten before they commit [48]. We believe that fairness among different types of transactions should be treated as a first-class citizen when serving heterogeneous workloads. Moreover, the lightweight OCC usually aborts loser transactions at commit time, wasting precious CPU cycles on long transactions that are destined to abort [1]. Consequently, heavyweight read-mostly transactions often lead to significant overall performance degradation as well. Therefore, going forward and as the industry shifts to heterogeneous workloads served by memory-optimized engines, it is vital for them to employ more effective CC schemes.

Interactions at the physical level. But it is not only the interaction at the logical level that should be central to the design of a memory-optimized database system. As commodity servers become increasingly parallel, many low-level issues (such as latching and thread scheduling) and design decisions—at the system architecture level—need to be revisited. The form of logging/recovery used, the storage management architecture, and scheduling policies for worker threads can impose drastic constraints on which forms of CC can be implemented at all, let alone efficiently. It can be difficult or impossible to adopt a different CC scheme without significant changes to the rest of the system. Thus, a physical layer should be designed with full awareness of consequences for CC and recovery, in addition to scalability efforts.

Desired properties. Considering the challenges, we argue that transaction processing systems should be designed with the following basic requirements:

- To provide robust and balanced concurrency control for the logical interactions over heterogeneous transactions.
- To address the physical interactions between threads in a scalable way and have a lightweight recovery methodology.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882905

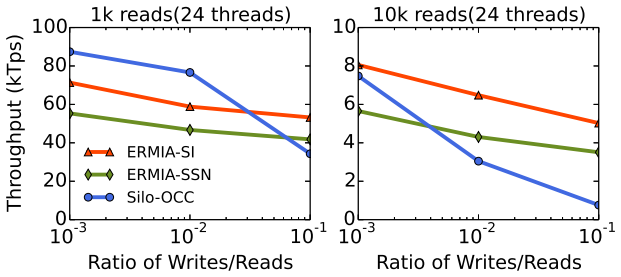

Figure 1: Performance of lightweight optimistic concurrency control and multi-version concurrency control at different transaction read-set sizes, as the ratio of writes increases. 1K reads (left); 10K reads (right).

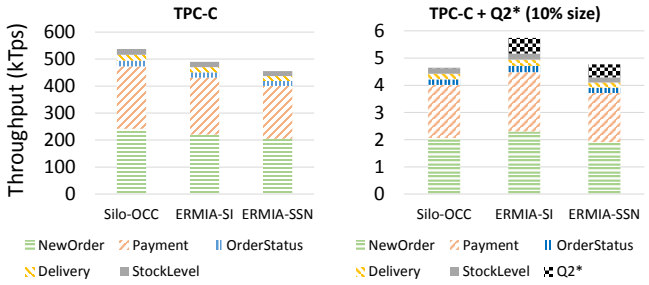

Figure 2: Breakdown of the commit rate of the TPC-C transactions (left) as well as when we add a read-intensive transaction (Q2*) to the mix (right). Silo achieves a very low commit rate, almost no progress, of the Q2* transaction.

1.1 ERMIA

In this paper, we present a novel memory- and multicore-optimized OLTP database, called ERMIA [1], to support the aforementioned desired properties. It employs a snapshot-isolation (SI) CC scheme that provides the following valuable features for heterogeneous workloads: (1) robustness against contention; (2) harmonious coexistence between readers and writers; and (3) early detection of doomed transactions to minimize the amount of wasted work. On top of the vanilla SI, ERMIA also provides serializability by adopting a recent proposal, called Serial Safety Net (SSN) [47].

We also present the design of a physical layer that interacts with the CC schemes efficiently and achieves multicore scalability in a memory-optimized design. The key components are (1) a scalable centralized log manager that provides global ordering; (2) latch-free indirection arrays which provide low overhead multi-versioning and efficient infrastructure for recovery; and (3) epoch managers that handle resource allocation and recycling.

As the detailed evaluation in Section 4 shows, ERMIA addresses inefficiencies of systems with lightweight OCC. In particular, Figure 1 demonstrates how the performance of Silo [45], a representative of the camp of memory-optimized systems with lightweight OCC, degrades as transactions have larger read footprint or when contention increases (experimental setup details in Section 4). As a comparison, Figure 1 shows the performance of ERMIA with snapshot isolation (ERMIA-SI) and serializable SI(ERMIA-SSN). Figure 1 shows that it just takes 0.1% or 1% of the touched records to be updates for the transaction throughput to drastically drop in Silo. The sharply decreasing curves of Silo in the figure indicate OCC's sensitivity to contention.

Further, Figure 2 demonstrates the unfairness of lightweight OCC schemes. The left graph shows the commit rate of each individual transaction of TPC-C under Silo as well as the ERMIA flavors. We can see that the commit rates are comparable under TPC-C. On the other hand, on the right side we add a read-mostly transaction in the mix, labeled at Q2*. We see that the commit rate of that read-intensive transaction starves in Silo, while for ERMIA it is quite high (the Q2* transaction is presented in Section 4.2). Also, given the drastic drop in overall TPS after we added Q2* transaction which takes more than 90% out of total cycles, Figure 2 also implies that Silo just wasted a huge number of cycles on aborted Q2* transactions while ERMIA spent most of its time on useful work.

1.2 Contributions and paper organization

[1] Stemmed from the two major components our system is built upon: ***E***poch-based ***R***esource ***M***anagement and ***I***ndirection ***A***rray.

In summary, this paper makes the following contributions:

- Highlights the mismatch between existing OCC mechanisms and heterogeneous workloads, and revisits snapshot isolation with cheap serializability guarantee as a solution.
- Presents a system architecture to efficiently support CC schemes in a scalable way with latch-free indirection arrays, scalable centralized logging, and epoch-based resource managers.
- Presents a comprehensive performance evaluation that studies the impact of CC and physical layer in various workloads, from the traditional OLTP benchmarks to heterogeneous workloads.

The rest of this paper is organized as follows. In Section 2 we lay out the design principles for memory-optimized engines to handle heterogeneous workloads on modern hardware. Section 3 discusses the detailed design of ERMIA. Section 4 compares the performance of ERMIA against an OCC-based memory-optimized system in a variety of transactional workloads. We discuss related work in Section 5 and conclude in Section 6.

2. DESIGN DIRECTIONS

In this section we discuss the design directions of database engines optimized for main-memory and multicores. We focus on three areas: the CC scheme that determines the interaction between concurrent transactions at the logical level; the mechanism that controls the interaction and communication among threads at the physical level; and recovery.

Shared-everything database. Some systems sidestep the issues of logical and physical contention entirely—along with the accompanying implementation complexity—by adopting physical partitioning and a single-threaded transaction execution model [21, 22]. This execution model introduces a different set of problems for mixed workloads and for workloads that are inherently difficult to partition. Given the developments in scaling-out the performance of distributed OLTP systems, especially for easy-to-partition workloads, e.g. [2, 8, 41], as well as for high availability and cost-effectiveness reasons, we predict that the successful architectures will combine scale-out solutions built on top of non-partitioning-based scale-up engines within each node. Therefore, we focus on the performance of non-partitioning-based memory-optimized engines within a single node.

Snapshot isolation and serializability. Broadly speaking, there are two camps of CC methods: the pessimistic, e.g. two-phase locking (2PL), and the optimistic (OCC). Past theory work [1] has shown that pessimistic methods are superior to optimistic ones under high contention. In practice, this result requires that pessimistic methods can be implemented with sufficiently low overhead relative

to their optimistic counterparts, which is not easy to achieve in practice. For example, a study of the SHORE storage manager reports roughly 25% overhead for locking-based pessimistic methods [14]. Having said that, typical memory-optimized engines that employ lightweight OCC and running on modern commodity servers, leave some room for exploration. There are different flavors of optimistic, or opportunistic, CC. Many recent systems adopt a lightweight read validation step at the end of the transaction, during pre-commit.

However, read validation is very opportunistic, and somewhat brittle, leaving the system vulnerable to workloads where writers are likely to overwrite readers (causing them to abort). The vulnerability against contention does not appear in the TPC-C benchmark where tiny transaction footprints and precipitous reduction in transaction latency offered by main-memory systems collectively minimize CC pressure (transaction contentions). Also, the unfair contention resolution of the lightweight OCC, a writer always wins over a reader, can be problematic in heterogeneous workloads. It is quite common that read-intensive analytic transactions are mixed with short, update-intensive transactions. The extreme favor for writers can cause massive aborts or even starvation of read-intensive transactions, consequently. Another problem is lazy transaction coordination policy that affects overall performance drastically. During forward processing, transactions keep all footprints locally and validate them only at the pre-commit time. Committed changes to a transaction's read set are not visible until the reader starts to validate its reads, wasting CPU cycles on long readers that are destined to abort. Regardless optimistic or pessimistic, the CC mechanism should not only have a low false positive rate detecting conflicts, but it should allow the system to detect doomed transactions as early as possible to minimize the amount of wasted work.

To recap, the following features are desirable for CC to orchestrate heterogeneous transactions: 1) robustness against contention, 2) balanced contention resolution over reader and writer and 3) early detection of abort conditions to minimize the amount of wasted work. Based on that, we believe that snapshot isolation based CC schemes satisfy all those requirements. Under snapshot isolation, reader-writer contention never happen by distributing transactions to multiple versions, bringing significantly higher throughput of read-intensive transactions. Also, it can recognize write-write conflicts as early as possible, possibly reducing the amount of wasted work on aborts.

Nevertheless, snapshot isolation still has remaining concerns. First of all, in presence of cyclic transaction dependency, SI can generate "write-skew" phenomenon which can violate serializable schedule. Generally SSI proposals in literature ensure serializability by tracking the "dangerous structure" that must exist in every serial dependency cycle under snapshot isolation. But their expensive tracking methods are usually prohibitively expensive for memory-optimized systems and tendency of aborting the "pivot" update transaction can starve writers [47], violating the fairness requirement. We argue that a cheap, fair certifier that can be overlaid on top of SI is the desired approach to serializability. We adopt such a certified, the Serial Safety Net [47] and describe it in Section 3. Also, efficient infrastructures in underlying physical layer are essential to make the SI-based CC schemes efficiently. We introduce physical layer supports in the following paragraphs.

Scalable centralized logging. Log managers are a well-known source of complexity and bottlenecks in a database system. The in-memory log buffer is a central point of communication that simplifies a variety of other tasks in the system, but which tends to kill scalability. Past work has attempted to optimize [19] or distribute [46] the log, with partial success and significant additional complexity. Systems such as H-Store [21] (and its commercial version, VoltDB) largely dispense with logging and rely instead on replication. These systems replace the logging bottleneck with a transaction dispatch bottleneck. Silo avoids the logging bottleneck by giving up the traditional total order in transactions. Avoiding total ordering is efficient but prevents the system from using any but the simplest of CC schemes—there is no practical way to implement repeatable read or snapshot isolation, for example.

We advocate a sweet spot between the extremes of fully coordinated logging (multiple synchronization points per transaction) and fully uncoordinated logging (no synchronization at all): latch-free logging with single synchronization point per transaction. A transaction with a reasonably small write footprint can acquire a totally ordered commit timestamp, and reserve all needed space in the log, using a single global atomic memory operation. While technically a potential bottleneck, our experimental results will prove that such a system can scale to *hundreds of thousands* of commits per second, while still preserving global ordering information that enables SI-based concurrency control schemes. We present this log manager in Section 3.

Latch-free indirection arrays. Transaction processing systems typically depend on their low-level storage manager component to mediate thread interactions at the physical level. The implementation of the storage manager is extremely tightly coupled to the CC scheme, making it difficult to modify or extend the CC schemes of legacy systems. Internal infrastructure matters terribly. It decides whether it is even possible to implement a particular CC scheme, and also which implementable schemes can be made practical. For example, the effort to enhance Postgres with serializable snapshot isolation (SSI) required a very large implementation effort, since the team had to integrate what it is essentially a lock manager with a purely multi-versioned system. Even then, the achieved performance borders on unusable, due to severe bottlenecks in multiple parts of the system (including a globally serialized pre-commit phase, latch contention in the new lock manager, and existing scalability problems in the log manager).

One promising technique that provides desirable properties for both SI-based CC schemes and physical contention is the notion of an *indirection array* [11, 39] for mapping logical object IDs to physical locations of records. Most of all, it is suitable for the physical implementation of CC for multi-versioned systems, as a single compare-and-swap (CAS) operation suffices to install a new version of an object without the use of heavyweight locking methods. Similarly, presence of an uncommitted version makes write-write conflicts easy to detect and manage.

Also, indirection can reduce the amount of logging required for an update in append-only systems. Update only requires installing a new physical pointer at the appropriate indirection array entry. Without the indirection, creating a new version requires updating every reference to that record, amplifying the amount of log records.

Moreover, it can reduce update pressure in (especially secondary) indexes; by storing a logical address (OID) of a tuple, indexes are isolated from update, as an OID of an updated tuple remains the same. Even though modern indexes leverage latch-free thread coordination for higher parallelism [28] [31], concurrent update still comes at an additional cost. Therefore, it is beneficial to offload the concurrent update pressure from the indexes to indirection arrays which have cheaper thread coordination. When it comes to secondary index maintenance, some systems maintain secondary indexes by mapping primary keys and secondary keys to avoid the update-propagation, however, it shifts burden to readers; secondary index access has to entail additional primary index probe.

Append-only storage. Append-only storage allows drastic simplifications of both I/O patterns and corner cases in the code. Com-

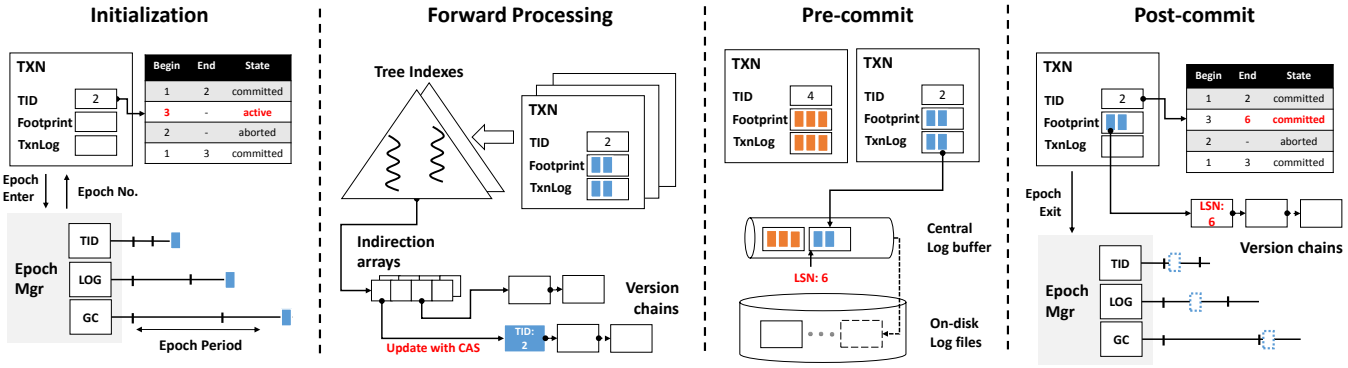

Figure 3: Architecture of ERMIA and transaction lifecycle.

bining a carefully designed log manager and indirection arrays produces a single-copy system where the log is a significant fraction of the database. Records graduate from the log to secondary storage only if they go a long time with no updates. The resulting system is also easier to recover, as both undo and redo are largely unnecessary—the log can be truncated at the first hole without losing any committed work.

Epoch-based resource management. Resource management is a key concern in any storage manager, and one of the most challenging aspects of it is ensuring that all threads in the system have a consistent view of the available resources, without imposing heavy per-access burdens. The infamous "ABA problem" [15] from the lock-free literature is one example of what can go wrong if the system does not maintain invariants about the presence and status of in-flight accesses. Epoch-based resource management schemes such as RCU [33] achieve this tracking at low cost by requiring only that readers inform the system whether they are *active* (possibly holding references to resources) or *quiescent* (definitely not holding any references to resources). These announcements can be made fairly infrequently, for example when a transaction commits or a thread goes idle. Resource reclamation then follows two phases: the system first makes the resource unreachable to new arrivals, but delays reclaiming it until all threads have quiesced at least once (thus guaranteeing that all thread-private references have died). Epoch-based resource management is especially powerful when combined with multi-versioning, as writers can coexist relatively peacefully with readers. Although epochs are traditionally fairly coarse-grained (e.g. hundreds of ms in Silo), we have implemented an epoch manager that is lightweight enough to use even at very fine time scales. As discussed in Section 3.4, ERMIA instantiates several epoch managers, all running at different time scales, to simplify all types of resource management in the system.

3. ERMIA

In this section, we start with an overview of ERMIA, and then describe its key pieces, with a focus on why we choose the design trade-offs we do.

3.1 Overview

ERMIA is designed around latch-free indirection arrays, epoch-based resource management and extremely efficient centralized logging. For indexing, it uses Masstree which is a concurrent and cache-efficient index structure, as Silo does [32]. We next briefly describe how ERMIA processes transactions using the components shown in Figure 3.

Initialization. When a transaction enters the system, it first joins three epoch-based resource managers: (1) the log manager, (2) transaction ID (TID) manager, and (3) the garbage collector. These resource managers then start to track the transaction to make sure that their resources (log buffers, TIDs and versions, respectively) are allocated/reclaimed correctly. Upon start, each transaction acquires a private log buffer, a TID and a begin timestamp—the current log sequence number (LSN). The transaction then calls in to the TID manager for initializing its transaction context and stores the begin timestamp in the context for CC interactions. The transaction is then ready to begin forward processing.

Forward processing. In ERMIA, transactions access the database through tree indexes. We associate each index/table with an indirection array. Different from traditional designs which give access to data in the leaf nodes, we store object IDs (OIDs) in the leaf level. Indexed by object IDs (OIDs), each array entry points to a chain of historic versions of the object (i.e., database record). Dictated by the underlying CC scheme, the transaction performs read/insert/update/delete operations on these structures (detailed in Section 3.2 and Section 3.6). To install a new version, the transaction stores its TID into the version's creation timestamp field. Other transactions who encounter the TID-stamped version have to follow the owner transaction's context with the TID to check if the version is visible. Each transaction accumulates the *descriptors* of its inserts and updates in the private log buffer to avoid log buffer contention. The garbage collector periodically goes over all indirection arrays to remove versions that are not needed by any transaction.

Pre-commit. We divide the commit process into two parts: pre-commit and post-commit. During pre-commit, the transaction first obtains a commit LSN from the log manager to fix its global order among all transactions and to reserve the space for storing its logs in the centralized log buffer. This is achieved by using a single atomic `fetch-and-add` instruction, which advances the current LSN by the size of the transaction's write footprint. Note that in most cases, this is the only global synchronization point in the log during a transaction life cycle. The transaction then goes through the underlying CC's commit protocol (detailed in Section 3.6), and follows the descriptors in its private log buffer to populate log records in its space reserved in the centralized log buffer. After this, the transaction switches its state to "committed"; all updates are visible to other transactions atomically after this point.

Post-commit. If a transaction survives pre-commit, it iterates versions in write-set, replacing its TID in the versions' creation timestamp field with its commit LSN, so that other transactions can directly check visibility on the LSN-stamped versions without accessing the owner transaction's context. Finally, the transaction

concludes by returning all resources and de-registering from epoch managers. If the transaction needs to abort, it changes its state to "aborted" and removes write-set versions from the version chains.

3.2 Indirection arrays

The indirection arrays used in ERMIA are linear arrays and similar to the ones proposed in the literature [11, 39]. All logical objects (database records) are identified by an OID that maps to a slot in an indirection array. The slot contains a physical pointer to data. The pointer may reference a chain of versions stored in memory, or stable storage (e.g., the durable log on disk). ERMIA leverages a latch-free singly linked list structure to provide cheap multi-versioning support. We next illustrate how insert, update, and read operations are handled with the structures.

Insert. The transaction first obtains a new OID from the OID manager and generates a new version. It then will store the new version's address to the corresponding slot in the table's indirection array. Note that this process is completely contention-free: it simply means writing to an element in an array of OIDs because no two threads will be allocated with the same new OID. After setting up the OID entry, the insert finishes by inserting to indexes using a key provided by the application.

Update/delete. To update a record, the transaction needs to create a new version out-of-place, and then follow the index and indirection array to reach the version chain. The new version is installed by an atomic `compare-and-swap` instruction against the corresponding OID slot in the indirection array, because the entry stores the address of the head version. An uncommitted head version acts as a write lock, so that write-write conflicts can be detected. Delete is treated as an update with tombstone marking. The garbage collector periodically scans the indirection arrays and removes any versions that are not needed (including deleted ones). Note that whether a transaction can update a record is dictated by the underlying CC scheme.

Read/scan. Once a transaction found an OID with a given key through index probe, it accesses the corresponding OID slot to fetch the first version in the version chain. While traversing the chain, the underlying CC scheme dictates read operations. It returns the appropriate version that should be read by a transaction based on the begin timestamp of the transaction and the version's creation timestamp. Scans are handled similarly, except that the transaction will first conduct a range query (instead of a point query) in the index to figure out which records (i.e., indirection array entries) to access.

Admittedly the use of another level of indirection will impose extra cache misses. But it is still promising in that database changes are largely absorbed by the indirection array; the updates are not propagated to other parts of database including indexes, as OIDs in the indexes remain same as before. Consequently, it can reduce the amount of log data and thread contentions in the indexes (usually concurrent trees).

3.3 Logging

The log manager is a pivotal component in most databases. It provides a centralized point of coordination that other pieces of the system build off of and depend on. Although its central nature can simplify recovery and provide transaction ordering for SI-based CC schemes, the log is also a notorious source of contention in many systems. ERMIA's log manager retains the benefits of a serial "history of the world" while largely avoiding the contention issues that normally accompany it. In particular, ERMIA's log manager features the following four useful features:

1. Communication in the log manager is extremely sparse. Most update transactions will issue just one global atomic fetch-and-add before committing, even in the presence of high contention and corner cases such as full log buffer or log file rotations.
2. Transactions maintain log records privately while in flight, and aggregate them into large blocks before inserting them into the centralized log.
3. Transactions can acquire their commit LSN before entering pre-commit, thus simplifying the latter significantly (as all committing transactions agree on their relative commit order).
4. Large object writes can be diverted to secondary storage, requiring only an indirect pointer in the actual log.

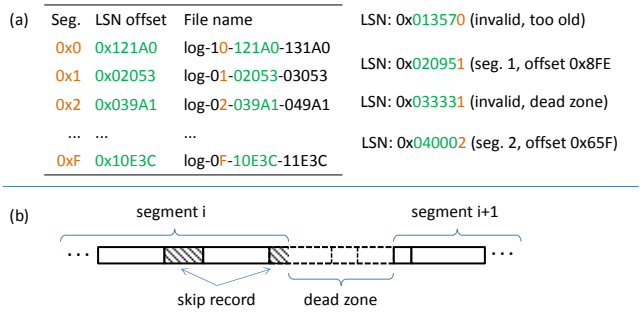

Figure 4: LSN space management.

The central feature of the log—*a single global atomic operation per insert*—rests on a key observation: while the LSN space must be monotonic, it need not be contiguous as long as sequence numbers can be translated efficiently to physical locations on disk. Therefore, each LSN consists of two parts: the higher-order bits identify an offset in a logical LSN space, while the lowest-order bits identify the physical log segment file the offset maps to. There are a fixed number of log segments in existence at any time (16 in our prototype), but segments may be arbitrarily large and are sized independently of each other.[2] Placing the segment number in low-order bits preserves the total of log offsets.

Figure 4 (a) illustrates how the system converts between LSN and physical file offsets. Each modulo segment number is assigned a physical log segment, which consists of a start and end offset, and the name of the file that holds the segment's log records. The file name is chosen so the segment table can be reconstructed easily at start up/recovery time, even if the current system's log segment size is different than that of the existing segments. Given the mapping table, LSN can be validated and converted to file offsets using a constant-time look up into the table.

Because we allow holes in the LSN space, acquiring space in the log is a two-step process. First, the thread claims a section of the LSN space by incrementing a globally shared log offset by the size of the requested allocation. Then, having acquired this block of logical LSN space, the transaction must validate it against various corner cases:

- Log segment full. If the block straddles the end of the current segment file, it cannot be used and a skip record is written in its place to "close" the segment. The thread competes to open the next segment file (see below) before requesting a new LSN.
- Between segments. Threads that acquire a log block that starts after the current segment file must compete to open the next segment. Blocks preceding the winner's block must be discarded as they do not correspond to a valid location on disk; losers retry.

[2] A system can thus scale log segment sizes upward to handle higher loads, and downward to conserve disk space.

- Log buffer full. The transaction retains its LSN but must wait for the corresponding buffer space to drain before using it.

Once a thread validates its LSN offset, it combines the segment and offset to form a valid LSN. A transaction can request log space at any time, for example to enter pre-commit with a valid commit LSN or to spill an oversized write footprint to disk; should it later abort, it simply writes a skip record.

Figure 4 (b) illustrates the various types of log blocks that might exist near a log segment boundary. Skip records in the middle of the segment come from aborted transactions (or are overflow blocks disguised as skip records). The skip record at the end of the segment "closes" the segment and points to the first valid block of the next segment. The "dead zone" between segments i and $i+1$ contains blocks that lost the race to open segment $i+1$; these do not map to any location on disk and should never be referenced.

Decoupling LSN offsets from log buffer and log file management simplifies the log protocol to the point that a single atomic operation suffices in the common case. There are only two times a transaction might require additional atomic operations: (a) Occasionally, a segment file will close and several unlucky transactions will race to open the next one. This case is very rare, as segment files can be hundreds of GB in size if desired. (b) A large write transaction may not be able to fit all of its log records in a single block, and so would be forced to write some number of overflow log blocks in a backward-linked list before committing. We expect this case to *reduce* contention, because a thread servicing large transactions—even with overflow blocks—will not visit the log nearly as often as a thread servicing short requests.

Because the log allocation scheme is fully non-blocking, there is no reliable way to know whether a thread has left. A thread with a reference to a segment that is about to be recycled could end up accessing invalid data (e.g. from segment $i+16$ rather than segment i). Therefore, we use an epoch manager to ensure that stragglers do not suffer the ABA problem when the system recycles modulo segment numbers, and that background cleaning has migrated live records to secondary storage before a segment is reclaimed (reassigning its modulo segment number effectively removes it from the log's "address space").

3.4 Epoch-based resource management

Epoch-based memory management–exemplified by RCU [33]—allows a system to track readers efficiently, without requiring readers to enter a critical section at every access. Updates and resource reclamation occur in the background once the system can prove no reader can hold a reference to the resource in question. Readers inform the system when they become *active* (intend to access managed resources soon) and *quiescent* (no longer hold any references to managed resources). These announcements are decoupled from actual resource accesses in order to amortize their cost, as long as they are made reasonably often. Ideal points in a database engine would include transaction commit or a worker thread going idle. Resource reclamation then follows two phases: the system first makes the resource unreachable to new arrivals, but delays reclaiming it until all threads have quiesced at least once (thus guaranteeing that all thread-private references have died). Once a resource is proven safe to reclaim, reclamation proceeds very flexibly: worker threads are assigned to reclaim the resources they freed and a daemon thread performs cleanup in the background.

We have developed a lightweight epoch management system that can track multiple time lines of differing granularities in parallel. A multi-transaction-scale epoch manager implements garbage collection of dead versions and deleted records, a medium-scale epoch manager implements read-copy-update (RCU) that manages physical memory and data structure usages [33], and a very short timescale epoch manager tracks transaction IDs , which we recycle aggressively (see details in Section 3.5).

The widespread and fine-grained use of epoch managers is enabled by a very lightweight design we developed for ERMIA. It has three especially useful characteristic. First, the protocol for a thread to report activation and quiescent points is lock-free and threads interact with the epoch manager through thread-private state they grant it access to. Thus, activating and quiescing a thread is inexpensive. Second, threads can announce conditional quiescent points: if the current epoch is not trying to close, a read to a single shared variable suffices. This allows highly active threads to announce quiescent points frequently (allowing for tighter resource management) with minimal overhead in the common case where the announcement is uninteresting. Third, and most importantly, ERMIA tracks three epochs at once, rather than the usual two.

In a traditional epoch management scheme, the "open" epoch accepts new arrivals, while a "closed" epoch will end as soon as the last straggler leaves. Unfortunately, this arrangement cannot differentiate between a straggler (a thread that has not yet quiesced in a long time) and a busy thread (one which quiesces often but is likely to be active at any given moment). Thus, when an epoch closes, most busy threads in the system will be flagged as stragglers—in addition to any true stragglers—and will be forced to participate in an expensive straggler protocol designed to deal with non-responsive threads. The ERMIA epoch manager, in contrast tracks a third epoch, situated between the other two, which we call "closing." When a new epoch begins, all currently active threads become part of the "closing" epoch but are otherwise ignored. Only when a third epoch begins does the "closing" epoch transition to "closed" and check for stragglers. Active threads will have quiesced and migrated to the current (open) epoch, leaving only true stragglers—if any—to participate in the straggler protocol.

Although the three-phase approach tracks more epochs, the worst-case duration of any epoch remains the same: it cannot be reclaimed until the last straggler leaves. In the common case where stragglers are rare, epochs simply run twice as often (to reflect the fact that threads have two epoch-durations to quiesce).

3.5 Transaction management

At its lowest level, the transaction manager is responsible to provide information about transactions that are currently running, or which recently ended, and this is partly achieved by allocating TIDs to all transactions and mapping those TIDs to the corresponding transaction's state. In our system, this is especially important because the CC scheme (see below) makes heavy use of TIDs to avoid corner cases.

Similar to the LSN allocation scheme the log manager uses, ERMIA's transaction manager assigns a TID to each transaction that combines an offset into a fixed-size table (where transaction state is held) with an epoch that identifies the transaction's "generation" (distinguishing it from other transactions that happened to use the same slot of the TID table).

Each entry in the TID table records the transaction's full TID (to identify the current owner), begin timestamp (an LSN), end timestamp (if one exists), and current status. The latter two fields are the most heavily used by the CC scheme, as transactions stamp records they create with their TID and only change that stamp to a commit timestamp during post-commit. During the cleanup period, other transactions will encounter TID-stamped versions and must call into the transaction manager to learn the true commit status/stamp. Such inquiries about a transaction's state can have three possible outcomes: (a) the transaction could still be in flight,

(b) the transaction has ended and the end stamp is returned, or (c) the supplied TID is invalid (from a previous generation). In the latter case, the caller should re-read the location that produced the TID—the transaction in question has finished post-commit and so the location is guaranteed to contain a proper commit stamp.

The TID table has limited capacity (currently 64k entries), and so the generation number will change frequently in a high-throughput environment. The TID manager thus tracks which TIDs are currently in use—allowing them to span any number of generations—and recycles a given slot only once it has been released. Because the system only handles a limited number of in-flight transactions at a time (far fewer than 64k), at most a small fraction of the TID table is occupied by slow transactions.

In order to serve TID inquiries, and handle allocation/deallocation of TIDs across multiple generations (all of which are quite frequent), we use only lock-free protocols, with an epoch manager, running at the time scale of a typical TID allocation—to detect and deal with stragglers who are delayed while trying to allocate a TID; this is only possible because the epoch manager is so lightweight. Without an epoch manager, a mutex would be required to prevent bad cases such as double allocations and thread inquiries returning inaccurate results.

3.6 Concurrency control

ERMIA's physical layer allows efficient implementations of a variety of CC schemes, including read-set validation and multi-version CC. In this paper, we focus on handling *read-mostly* workloads gracefully with SI-based CC schemes. Different components in the physical layer work together to support them efficiently; indirection arrays allow cheap (almost free) multi-versioning at an extremely low overhead; the log gives total ordering which is the key to implement snapshot-based CC mechanisms. To guarantee serializability, we adopt the Serial Safety Net (SSN) [47], a recent serializability certifier that can be overlaid on top of SI. We first give a brief overview of SI, and then explain in detail how SSN works and how we prevent phantoms.

3.6.1 Snapshot Isolation

Although not serializable [4], snapshot isolation (SI) is an ideal choice to start from: long, read-mostly transactions will have much higher chance to survive compared to lightweight OCC schemes.

Under SI, reads and writes never block each other. Reads are done by traversing the version chains: the reader transaction needs to perform visibility check directly on LSN-stamped versions by comparing its begin timestamp and the version's creation timestamp. If the version's creation timestamp predates the visitor's begin timestamp, the transaction will read it. Otherwise it moves on to the next version. In case the version's creation timestamp is a TID, the visitor has to access the owner transaction's context for visibility check. If the owner transaction is committed (but have not finished post-commit, i.e., it is still filling in the creation timestamp for new versions), and its commit timestamp predates the visitor's begin timestamp, the version is also visible. Otherwise the visitor has to move on to check the next version (which must be a committed version). Each update is done by installing a new version at the head of the chain. We forbid a transaction to update a record that has a committed head version later than its begin timestamp. We follow the "first-updater-wins" rule, so write-write conflicts are detected if the head version is uncommitted. The doomed updater will abort immediately to avoid dirty write, minimizing the amount of wasted work on aborts.

Since our SI scheme makes sure that dirty reads never happen and write-write conflicts have already been avoided on every update, if a transaction can successfully finish all of its updates, it enters pre-commit, following the process we have mentioned in earlier paragraphs to obtain a commit LSN, changes its status to "committed", and finishes its post-commit phase.

Algorithm 1 SSN commit protocol [47]

```
def ssn_commit(T):
  t.cstamp = next_timestamp() # begin pre-commit

  for v in t.writes: # finalize η(T)
    t.pstamp = max(t.pstamp, v.pstamp)

  # finalize π(T)
  t.sstamp = min(t.sstamp, t.cstamp)
  for v in t.reads:
    t.sstamp = min(t.sstamp, v.sstamp)

  if t.sstamp <= t.pstamp:
    t.abort()

  t.status = COMMITTED

  # update η(V)
  for v in t.reads:
    v.pstamp = max(v.pstamp, t.cstamp)

  for v in t.writes:
    v.prev.sstamp = t.sstamp # update π(V)
    # initialize new version
    v.cstamp = v.pstamp = t.cstamp
```

3.6.2 Serializability

Compared to OCC, SI can achieve better performance when handling heterogeneous workloads. However, it is not serializable. The existing state-of-the-art to make SI serializable is Serializable SI (SSI) [6], which tracks the "dangerous structure" that will cause possible non-serializable executions under SI. Despite its effectiveness, SSI could starve writers because it tends to abort the "pivot" (writer) transaction when there are concurrent readers in the dangerous structure. Instead, ERMIA adopts a recent proposal, the Serial Safety Net (SSN) [47], which provides both serializability and balanced reader/writer performance to guarantee serializability.

The Serial Safety Net. SSN is a cheap certifier that can be overlaid on top of any CC mechanism that prevents lost updates and dirty reads. The underlying CC scheme (e.g., SI) still dictates transaction access patterns; SSN only adds additional checks during transaction execution and at commit time to avoid non-serializable executions, which indicate a cycle in the corresponding dependency graph, where conflicting transactions represented by vertices are connected by dependency edges. Transactions are added to the graph once committed. To detect cycles, for each transaction T SSN maintains an easily-computed *priority stamp* $\pi(T)$ that summarizes all dependent transactions that committed *before*, but must be serialized *after* T. During transaction execution (i.e., when handling reads and writes) and at commit time, T will apply an *exclusion window* test against T's most-recent committed transaction U that conflicts with and committed before T. If U's commit timestamp is later than $\pi(T)$, then committing T might form a cycle in the dependency graph. Intuitively, SSN ensures that a "predecessor" (U) will not at the same time be a "successor" of T, which if committed might close a cycle in the dependency graph.

As noted in [47], the exclusion window test can be simplified as comparing a pair of timestamps: $\pi(T)$ and $\eta(T)$ which records T's most-recent predecessor: $\pi(T) \leq \eta(T)$ is forbidden. This observation leads to the commit protocol depicted in Algorithm 1. After getting a commit timestamp, the transaction will go over its

write set to finalize its $\eta(T)$. Note that record versions also maintain their π (sstamp) and η (pstamp) values, which are recorded by the corresponding transaction which overwrote or read the versions. Similarly, the transaction then examines its read set to finalize its `sstamp`. If the transaction survives the exclusion window test, it will proceed to update all read versions' `pstamp` with its commit timestamp. The commit protocol then finishes by updating overwritten versions' `sstamp` and initializing new versions with the commit timestamp.

Compared to SSI, SSN provides balanced performance for both readers and writers, and does not exaggerate the starvation of writers. Specifics about SSN vs. SSI are out of the scope of this paper; interested readers may refer to [47] for details.

Phantom protection. Although SSN prevents serialization dependency cycles, non-serializable executions can still arise due to *phantoms*, or insertion of new records into a range previously read by an in-flight transaction. ERMIA is amenable to various phantom protection strategies, such as hierarchical and key-range locking [24, 30]. We inherit Silo's tree-version validation strategy [45]. The basic idea is to track and verify tree node versions, taking advantage of the fact that any insertion into an index will change the version number of affected leaf nodes. As Silo does, ERMIA also maintains a *node set*, which maps from leaf nodes that fall into the range query to node versions. The node set is examined after pre-commit. If any node's version has changed, the transaction must abort to avoid a potential phantom. Interested readers may refer to [45] for details.

3.7 Recovery

Recovery in ERMIA is straightforward because the log contains only committed work; OID arrays are the only real source of complexity. The OID array objects are updated in place to avoid overloading the log, and are thus effectively volatile in-memory data structures. However, they are needed to find all other objects in the system (including themselves), so ERMIA employs a combination of fuzzy checkpointing and logical logging to maintain OID arrays properly across crashes and restarts. All OID arrays are periodically copied (non-atomically) and the disk address of each valid OID entry is dumped to secondary storage after recording a checkpoint-begin record in the log. A checkpoint-end record records the location of the fuzzy snapshot once the latter is durable. The location of the most recent checkpoint record is also recorded in the name of an empty checkpoint marker file in the file system. During recovery, the system decodes the checkpoint marker, restores the OID snapshots from the checkpoint, then rolls them forward by scanning the log after the checkpoint and replaying the allocator operations implied by insert and delete records. The replay process examines only log block headers which take less than 10% out of total space and does *not* replay the insertions or deletions themselves (which are safely stored in the log already). A similar process could potentially be applied to other internal data structures in the system, such as indexes, but we leave that exploration to future work. It is important to note that the process of restoring OID arrays is exactly the same when coming up from either a clean shutdown or a crash—the only difference is that a clean shutdown might have a more recent checkpoint available.

In summary, because the log is the database [3, 5], recovery only needs to rebuild the OID arrays in memory using sequential I/O; anti-caching [10] can take care of loading the actual data, though background pre-loading is highly recommended to minimize cold start effects.

4. EVALUATION

In this section, we evaluate the performance of ERMIA using various benchmarks including both traditional, update-heavy and emerging heterogeneous workloads. We compare the performance of ERMIA and Silo [45], a representative lightweight OCC implementation optimized for main-memory and multicore hardware. Through the experiments, we are interested in revealing and confirming the mismatch between lightweight OCC and heterogeneous workloads, and showing that ERMIA achieves its design goals of providing robustness and fairness, while maintaining high performance. Therefore, our experiments focus on two perspectives: (1) the capability to serve heterogeneous workloads (in particular, avoiding starvation of read-mostly transactions) and (2) scalability of the physical layer on massively parallel hardware.

4.1 Experimental setup

We run our experiments on a quad-socket server with four Intel Xeon E7-4807 processors (24 physical threads in total) and 64GB of RAM. All worker threads are pinned to a dedicated core to minimize context switch penalties and inter-socket communication costs. Log records are written to *tmpfs* asynchronously. We measure the performance of three systems with different CC schemes: Silo (Silo-OCC), ERMIA with SI (ERMIA-SI), and ERMIA with SSN (ERMIA-SSN). For Silo, read-only snapshots are enabled to handle read-only transactions. We also improved Silo's transaction abort handling and string manipulation mechanisms, which improves the throughput in TPC-C by ∼15% (in workloads where abort count is high, the impact of the optimization is higher than 15%). To achieve fair comparison, ERMIA uses the same benchmark code and phantom prevention mechanism as Silo's.

4.2 Benchmarks

We run a microbenchmark, TPC-C and TPC-E benchmarks on all system variants. For TPC benchmarks, we run both the original version described by the specifications [42, 43] and modified versions that contain synthesized read-mostly transactions. For each run, we load data from scratch on a pre-faulted memory pool and run the benchmark for 30 seconds. All runs are repeated for three times and we report the average numbers.

Microbenchmark. We use the the Stock table in TPC-C benchmark to build a synthesized workload. The microbenchmark consists of a single transaction that randomly picks a subset of the Stock table to read and a smaller fraction of it to update. The purpose is to create read-write conflicts. We have discussed the results from this benchmark in Section 1.

TPC-C. It is well-known that TPC-C is update-heavy with small transaction footprints. TPC-C is also a partitionable workload: conflicts can be reduced by partitioning and single-threading on local partitions. In such a environment, the only source of contention is cross-partition transactions. However, its impact is dampened by the small footprints, avoiding the majority of read-write conflicts. In this benchmark, we partition the database by warehouse ID and assign each worker thread a local warehouse, but 1% of the NewOrder transactions and 15% of Payment transactions are cross-partition.

TPC-C-hybrid. To simulate heterogeneous workloads, we adopt a modified version of the Query2 transaction (TPC-CH-Q2*) in TPC-CH [12] benchmark. The original version of Q2 lists suppliers in a certain region and their items having the lowest stock-level. We modify the query by enforcing it to pick a random region and update items in the stock table having lower quantity than a certain threshold. We vary the fraction of the Supplier table to be scanned by TPC-CH-Q2* to adjust transaction footprint size. The new transaction mix consists of 40% of NewOrder, 38% of Payment, 10%

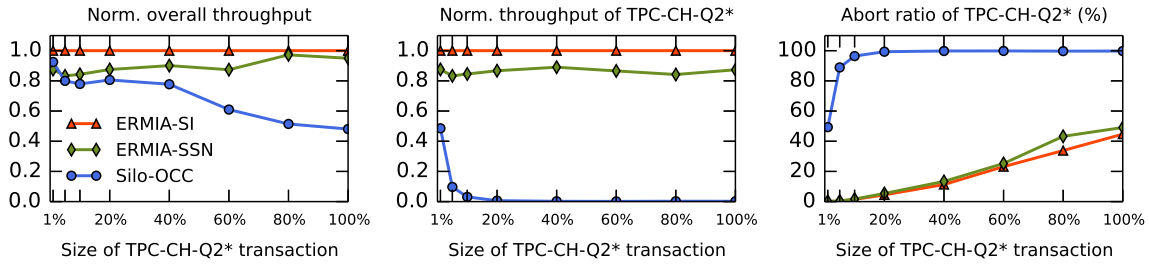

Figure 5: TPC-C-hybrid performance. Overall performance (left); TPC-CH-Q2* xct performance (middle), Abort ratio of TPC-CH-Q2* xct (right), varying size of TPC-CH-Q2* transaction. Performances are normalized to ERMIA-SI (see Table 1 for absolute numbers of overall TPS for ERMIA-SI).

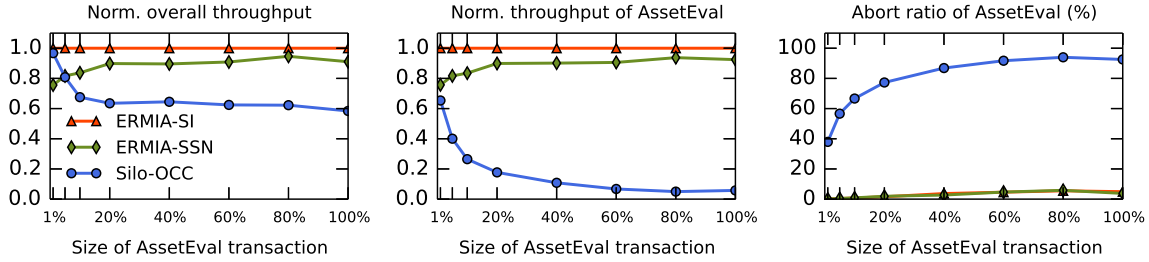

Figure 6: TPC-E-hybrid performance. Overall performance (left); AssetEval xct performance (middle), Abort ratio of AssetEval xct (right), varying size of AssetEval transaction. Performances are normalized to ERMIA-SI. (see Table 1 for absolute numbers of overall TPS for ERMIA-SI).

of TPC-CH-Q2*, and 4% of OrderStatus, StockLevel and Delivery each. The access pattern of TPC-CH-Q2* is determined by supplier ID, not by partitioning field (warehouse ID), thus, it is often cross-partition, interfering with local worker threads. Because of the majority of TPC-CH-Q2*'s access is in the Item and Stock tables, a TPC-CH-Q2* transaction will conflict frequently with NewOrder and other TPC-CH-Q2* transactions.

TPC-E. Although TPC-C has been dominantly used to evaluate OLTP systems, TPC-E is designed to be a more realistic OLTP benchmark with modern features. It models brokerage firm activities and has more sophisticated schema and transaction execution control [43]. It is also known for higher read-to-write ratio than TPC-C (∼10:1 vs. ∼2:1 [7]). The parameters we set for TPC-E experiments are 5000 customers, 500 scale factor and 10 initial trading days (the initial trading day parameter was limited by our machine's memory capacity).

TPC-E-hybrid. Similar to the changes we made to TPC-C, we also augment TPC-E with a new read-mostly transaction (AssetEval). AssetEval evaluates the aggregate assets for random customer accounts and inserts the results into a newly added AssetHistory table. The total asset of an account is computed by joining the HoldingSummary and LastTrade tables. The vast majority of contentions will occur between AssetEval and TradeResult transactions. To vary the degree of contention and the footprint size of AssetEval, we adjust the size of a customer account group in the CustomerAccount table to be scanned by AssetEval. The revised workload mix is as follows: BrokerVolume (4.9%), CustomerPosition (8%), MarketFeed (1%), MarketWatch (13%), SecurityDetail (14%), TradeLookup (8%), TradeOrder (10.1%), TradeResult (10%), TradeStatus (9%), TradeUpdate (2%) and AssetEval (20%).

4.3 Impact of CC schemes

	1%	5%	10%	20%	40%	60%	80%	100%
TPC-C-hybrid	70,319	12,938	5,698	2,435	1,173	868	750	647
TPC-E-hybrid	37,151	7,675	4,298	2,142	1,090	705	524	429

Table 1: Overall TPS of ERMIA-SI in TPC-C-hybrid and TPC-E-hybrid over varying sizes of read-mostly transactions.

We explore how ERMIA and Silo react to heterogeneous workloads with the different CC schemes they employ. We first examine the throughput of read-mostly transactions in all the three CC schemes. We extract performance numbers of TPC-CH-Q2* and AssetEval transactions from TPC-C-hybrid and TPC-E-hybrid benchmarks, respectively, and normalize them to the numbers of ERMIA-SI. As shown in Figure 5 (middle), Silo-OCC only produces half TPC-CH-Q2* commits in ERMIA, even at the smallest footprint size. As we increase the size of TPC-CH-Q2* transaction, Silo-OCC quickly collapses, converging to nearly zero throughput for this transaction. Starting from 40%, Silo-OCC produces two orders of magnitudes lower commits than ERMIA. Figure 6 (middle) shows a relatively modest curve for Silo-OCC because TPC-E-hybrid has less contention than TPC-C-hybrid. Nevertheless, Silo-OCC becomes quickly vulnerable in high contention scenarios and the throughput of AssetEval quickly drops to unacceptable level, the same as in TPC-C-hybrid.

Figure 5 (right) and Figure 6 (right) show the percentage of aborted TPC-CH-Q2* and AssetEval transactions out of their number of total executions. We find strong correlation between abort ratio and the throughput of read-mostly transactions. In TPC-E-hybrid, 40% of AssetEval transactions are aborted by Silo-OCC at the smallest footprint size and it keeps increasing as the size of AssetEval grows. TPC-C-hybrid shows a much sharper increase of abort ratio under Silo-OCC due to frequent conflicts with NewOrder.

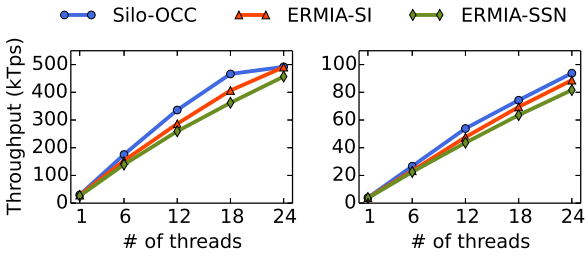

Figure 7: Throughput when running TPC-C (left); TPC-E (right). ERMIA achieves near-linear scalability over 24 cores and comparable peak performance to Silo.

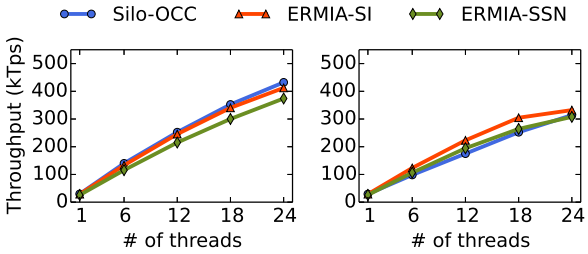

Figure 8: Throughput when running TPC-C with uniformly random access (left); TPC-C with 80-20 access skew (right). Due to robust CC schemes, ERMIA is less sensitive to contention.

ERMIA generally commits most read-intensive transactions. Note that ERMIA's abort ratio also increases when TPC-CH-Q2*'s footprint size passes the 40% mark. Since read-write conflicts never happen under SI, the majority of the aborts in ERMIA-SI came from write-write conflicts between TPC-CH-Q2* transactions, which are inevitable under any CC scheme that prevents dirty reads and lost updates. SSN adds slightly more aborts on top of it, but the number of the additional aborts due to serializability violation is marginal. This verifies that contention imposed heavy pressure on the CC and lightweight OCC is not a good match for heterogeneous workloads because of unacceptable performance penalty for read-mostly transactions due to unfair contention resolution. Read-mostly transactions are unlikely to commit as a result of extreme favor for short, update-intensive transactions. Meanwhile, ERMIA effectively protects read-intensive transactions from updaters with SI.

Figure 5 (left) and Figure 6 (left) show the normalized overall throughput when varying the size of TPC-CH-Q2* and AssetEval transactions. ERMIA-SI performs the best at all transaction sizes and ERMIA-SSN pays an additional cost for serializability guarantee; this comes from the trade-off between strong consistency and performance. ERMIA maintains superior overall performance while committing most heavyweight, read-mostly transactions, while Silo-OCC rarely commits them. Note that higher commits for read-mostly transactions do not significantly contribute to overall throughput due to their limited fractions out of workload mix. The figure also highlights OCC's reader starvation and lazy communication problems. As shown in Figure 6 (left), the performance gap between ERMIA and Silo-OCC closes when we hit the 5% mark for AssetEval footprint size. After this point, Silo-OCC becomes worse than ERMIA's. A similar trend is found in Figure 5 (left), with an earlier break-even point at 1% to 5%. These trends

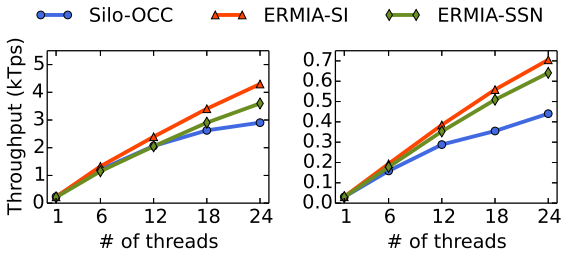

Figure 9: Throughput when running TPC-E-hybrid with 10% AssetEval transaction (left); with 60% AssetEval transaction (right). CC pressure deteriorates scalability of Silo in heterogeneous workloads.

show that Silo-OCC's growing loss on read-mostly transactions affects overall performance.

Another interesting finding is that Silo-OCC is still slower than ERMIA despite its negligible commits from read-mostly transactions. If the underlying CC had detected read-mostly transactions that are destined to abort in a timely manner, it should have skipped the uncomfortable transactions and switched to other (much shorter) transactions as early as possible, reaping more commits from other transactions. Unlike SI and SSN, the lightweight OCC schemes cannot detect conflicts until commit time. Hence, heavyweight read-mostly transactions often lead to unnecessary domination of CPUs, instead of giving more CPU cycles to other transactions that are more likely to commit. On the other hand, ERMIA can detect write-write conflicts on every update, giving some transactions early-out and avoids most read-write conflicts. This is evidenced by Figure 5 (left and right): starting from the 20% mark on the x-axis, the increasing performance gap between ERMIA and Silo does not indicate Silo-OCC's sudden slowdown, but rather shows that ERMIA minimized the amount of wasted work on aborted TPC-CH-Q2* transactions, by capturing write-write conflicts early and switching to other transactions. This result proves that OCC's validate-at-commit scheme can have a huge impact on overall performance, wasting a tremendous amount of CPU cycles.

4.4 Scalability

In this section, we measure scalability of all systems in diverse workloads, from original TPC benchmarks to mixed workloads we introduced. We start from original TPC benchmarks where lightweight OCC shines, as those workloads impose little pressure on the OCC scheme.

We measure overall throughput in TPC-C and TPC-E, varying the number of worker threads. Figure 7 shows that ERMIA achieves comparable peak performance and scalability trends to Silo-OCC in both TPC-C and TPC-E. Silo-OCC shows the best performance by taking advantages of its low-overhead CC and physical layer when little CC pressure exists. On ERMIA, indirection arrays brought more cache-misses and SSN paid additional serializability guarantee cost. In TPC-E, Silo-OCC avoided most of read-write conflicts by forwarding read-only transactions to read-only snapshots although TPC-E is more contentious than TPC-C (when read-only snapshots are turned off, Silo-OCC delivers similar performance to ERMIA-SSN).

To investigate the impact of skew-induced contentions on scalability, we enforce workers to pick up target partition randomly in TPC-C, following uniform and 80-20 random distribution, each time transactions start. In Figure 8, we can find that growing access skew

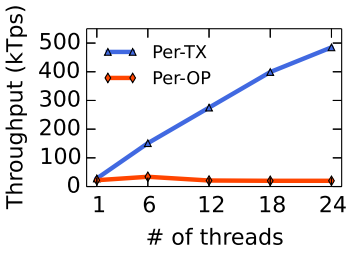

Figure 10: Throughput of ERMIA-SI with per-transaction logging and per-operation logging when running TPC-C.

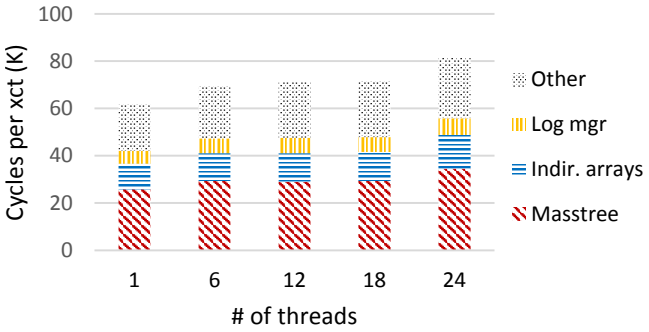

Figure 11: Cycle breakdown of the various components of ERMIA-SI when running an increasing number of threads run TPC-C.

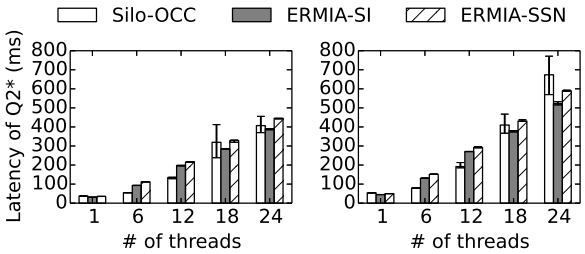

Figure 12: Latency of TPC-CH-Q2* at 60% size (left); 80% size (right), varying the number of threads. Under Silo-OCC, latency fluctuates with larger transaction size and higher concurrency.

suppresses Silo-OCC's scalability more than ERMIA; Silo-OCC is dragged down to performance of ERMIA-SI with uniform random access and to performance of ERMIA-SSN performance with highly skewed access.

We also study the impact of heavy read-mostly transaction on scalability in heterogeneous workloads. Figure 9 shows scalability at different sizes, 10% and 60%, of AssetEval transaction in TPC-E-hybrid, increasing the number of concurrent workers. Overwhelmed by CC-pressure, Silo-OCC does not achieve linear scalability and it becomes worse as we run larger read-mostly transactions. ERMIA benefits from its robust CC scheme and scalable storage manager. This figure shows that not only the scalability of underlying physical layer, but also CC scheme performance dictates overall performance.

Figure 10 shows efficiency of ERMIA's log manager. Unlike the WAL protocol, ERMIA log manager has only single round-trip to centralized log buffer at pre-commit to reduce contention. As comparison, we emulate the traditional way of logging by enforcing log buffer round-trip for every single update operation. As shown in the figure, the impact of per-transaction logging is drastic in memory-optimized systems. The per-operation logging did not scale at all, even without log buffer latching (single atomic instruction was used to reserve log buffer space).

Figure 11 illustrates CPU cycle breakdown per transaction of ERMIA-SI in TPC-C. With growing parallelism, the critical components maintain quite steady overhead. It indicates that the building blocks of ERMIA are generally scalable over 24 cores. As shown in the figure, Masstree takes the biggest bottleneck (41%) out of total cycles. At the expense of robust CC schemes, ERMIA pays 16% overhead as indirection costs which mainly came from additional last-level cache misses (32% of cache misses occurred in the indirection arrays). The overhead of log manager consistently takes 8 9% at all points; it proves that fully-decentralized logging and giving up total ordering are not necessarily required to achieve scalability over 24 cores. Although we ran the experiments on a 24-core machine, we do not expect that log manager's overhead will skyrocket over 32 cores. The cost of epoch-based resource managers are negligible (less than 1%). Resource management does not suffer from thread contentions at all.

4.5 Latency analysis

In this section, we perform latency analysis. Figure 12 shows latency of TPC-CH-Q2* over varying number of threads with 60% and 80% transaction size. We calculated average latency of the transaction within a run and presented the median value out of three runs in the figure. The maximum and minimum values are represented with error bars. Note that latency of TPC-CH-Q2* transaction is proportional to the number of threads in TPC-C-hybrid workload because the Stock table which is heavily accessed by the transaction grows proportionally to scale factor, and we give the same number of scale factor with the number of threads.

Let us take a look on latency variance first. Both ERMIA-SI and ERMIA-SSN have negligible variance in all cases. Under Silo-OCC, the transaction latency is consistent in general, however, in the cases where the transaction is large and the number of concurrent threads are high, we can observe latency fluctuation (noticeable variance tends to appear when the transaction takes more than 200ms). Also, latency increases more quickly than ERMIA with increasing parallelism, as shown in Figure 12. Considering growing size of the transaction, ERMIA delivers consistent latency regardless of concurrent threads indeed.

One of the main reasons for the results is read-write contention. During commit protocol, lightweight OCC grabs write locks on their write set first, validates read set and releases the write locks after installation of the write set. Meanwhile, readers facing the locked tuples have to wait for them to be released. Since both larger transaction size and higher concurrency possibly trigger more read-write contention, they often lead to larger latency variance and performance burden. In the worst case, heavy read-mostly transactions can hold write locks for a long time until read validation on their huge footprint is completed, blocking other transactions to proceed upon read-write conflict. Fundamentally, it is a problem of single-versioned database and might not be a serious problem when serving short transactions only. However, in the presence of heavy read-mostly transactions, it could bring huge latency variance and slowdown, as confirmed by Figure 12.

Another factor that possibly affects latency in Silo is the use of spinlock in coordinating read-write contention. Spinlock generally

does not consider fairness over waiters; some unlucky readers could wait excessively due to the random nature of spinlock. We believe MCS lock could be a promising alternative for its fairness and multicore friendliness [34]. In addition, we also found that Masstree relies on spinlock for concurrent access coordination, which partly causes latency increases.

5. RELATED WORK

In terms of concurrency control, one of the most important studies has been [1]. This modeling study shows that if the overhead of pessimistic two-phase locking can be comparable to the overhead of optimistic methods then the pessimistic one is superior. The same study shows that it is beneficial to abort transactions that are going to abort as soon as possible. That is corroborated by other studies as well, e.g. [38]. We follow the findings, trying to detect conflicts early.

Many of the memory-optimized systems adopt lightweight optimistic concurrency control schemes that are suitable only for a small fraction of transactional workloads. Silo [45], Hekaton [11] and Foedus [23] employ lightweight optimistic concurrency control schemes that perform validations at pre-commit. As a comparison with ERMIA, Silo performs in place updates: under normal circumstances the system maintains only a single committed version of an object, in addition to some number of uncommitted private copies (only one of which can be installed). In order to support large read-only transactions, a heavyweight copy-on-write snapshot mechanism must be invoked. These snapshots are too expensive to use with small transactions, and unusable by transactions that perform any writes. Similarly, Microsoft Hekaton [11] employs similar multi-versioning CC [26]. It is technically multi-versioned, but the snapshot-plus-read-validation CC scheme it uses means that older versions become unusable to update transactions as soon as any overwrite commits. Foedus keeps logically-equivalent snapshots in NVRAM for OLAP queries. For all practical purposes, these systems are multi-versioned only for read-only transactions.

H-Store (and its commercial version, VoltDB) is a characteristic partitioning-based system [21]. H-Store physically partitions each database to as many instances as the number of available processors, and each processor executes each transaction in serial order without interruption. Performance issues raise when the system has to execute multi-site transactions that touch data from two or more separate database instances. Lots of work has been put in the area, including low overhead concurrency control mechanisms [20], deterministic transaction execution to avoid two-phase commit overhead [41], and partitioning advisors that help to co-locate data that are frequently accessed in the same transactions, thereby reducing the frequency of multi-site transactions, e.g. [9, 37, 44].

For scaling up, many proposals have focused on exploiting multicore parallelism. DORA [35] employs logical partitioning and PLP [36] extends the data-oriented execution principle, by employing physiological partitioning. Under PLP, the logical partitioning is reflected at the root level of the B+tree indexes that now are essentially multi-rooted. Both DORA and PLP use Shore-MT's codebase [18], which is a scalable but disk-optimized storage manager. Hence, their performance lacks in comparison with the memory-optimized proposals. Additionally, even though only logical, there is a certain overhead in the performance due to the partitioning mechanism employed. Aether is an holistic approach for scalable logging, minimizing centralized log buffer contention [16]. It combined several techniques such as early lock release, log buffer pipelining and consolidation buffer. Wang et al. [46] proposed distributed logging with non-volatile memory to improve logging performance even further.

HANA [40] and Hyrise [13] are OLAP-OLTP hybrid engines with more focus on analytic side. Commonly, they append delta records to row-store partitions and gradually transform the delta to columnar compressed main tables. Krueger et al. suggested an efficient merge operation between the delta partitions and main tables, exploiting multicore parallelism as well [25]. Hyper [22] follows H-Store's single-threaded execution principle. It exploits "copy-on-write" functionality to generate snapshots for read-only queries. To scale up to multi-cores, they employ the hardware transactional memory capabilities of the latest generation of processors [27].

The indirection array, which is central to ERMIA's design, is a well-known technique, for example presented in [39]. It is worth mentioning that Hekaton also uses a technique similar to the indirection map. ERMIA relies on Masstree for indexing. Masstree [31] is a trie-based index structure, designed with lock-free technique for concurrent thread coordination. In Hekaton, Bw-tree [28] exploits indirection map and delta records to achieve lock-free design.

6. CONCLUSION

In this paper, we underlined the mismatch between the lightweight OCC currently in vogue with memory-optimized systems and emerging heterogeneous workloads that include read-mostly transactions. To address the challenge, we proposed ERMIA, a memory-optimized transaction processing system built from scratch, to accommodate heterogeneous workloads. It provides robust and balanced CC schemes to orchestrate heterogeneous transactions. Also its physical layer supports the CC schemes efficiently and achieves near-linear scalability over parallel processors with scalable centralized log manager, latch-free indirection arrays and epoch-based resource managers. Experimental results confirm that ERMIA can maintain high performance for long read-mostly transactions, while achieving comparable or superior overall performance and (near-linear) scalability in various transactional workloads.

7. REFERENCES

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM TODS*, 12(4), 1987.

[2] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with ramp transactions. In *SIGMOD*, 2014.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *NSDI*, 2012.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[5] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, January 2011.

[6] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, 2008.

[7] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD Record*, 39, 2010.

[8] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3, 2010.

[10] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *PLVDB*, 2013.

[11] 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.

[12] F. Funke, A. Kemper, and T. Neumann. Benchmarking hybrid OLTP&OLAP database systems. In *BTW*, 2011.

[13] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE—A main memory hybrid storage engine. *PVLDB*, 4, 2010.

[14] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[16] R. Johnson et al. Aether: a scalable approach to logging. *PVLDB*, 3, 2010.

[17] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *The VLDB Journal*, 2013.

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

[19] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3, 2010.

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

[21] 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. *PVLDB*, 2008.

[22] A. Kemper and T. Neumann. HyPer – a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[23] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, 2015.

[24] H. Kimura, G. Graefe, and H. Kuno. Efficient locking techniques for databases on modern hardware. In *ADMS*, 2012.

[25] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1), 2011.

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

[27] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

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

[29] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *CIDR*, 2015.

[30] D. B. Lomet. Key range locking strategies for improved concurrency. In *VLDB*, 1993.

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

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

[33] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, 1998.

[34] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1), 1991.

[35] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1), 2010.

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

[37] A. Pavlo, E. P. C. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2), 2011.

[38] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PLVDB*, 5(12), 2012.

[39] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making updates disk-I/O friendly using SSDs. *PVLDB*, 6, 2013.

[40] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD*, 2012.

[41] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3, 2010.

[42] TPC. TPC benchmark C (OLTP) standard specification, revision 5.11, 2010. Available at http://www.tpc.org/tpcc.

[43] TPC. TPC benchmark E standard specification, revision 1.12.0, 2010. Available at http://www.tpc.org/tpce.

[44] K. Q. Tran, J. F. Naughton, B. Sundarmurthy, and D. Tsirogiannis. JECB: A join-extension, code-based approach to OLTP data partitioning. In *SIGMOD*, 2014.

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

[46] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 2014.

[47] T. Wang, R. Johnson, A. Fekete, and I. Pandis. The Serial Safety Net: Efficient concurrency control on modern hardware. In *DaMoN*, 2015.

[48] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB.*, 8(3), 2014.