ERMIA: Fast and robust memory-optimized OLTP

University of Toronto Technical Report (CSRG-623)

Kangnyeon Kim Tianzheng Wang Ryan Johnson Ippokratis Pandis*

University of Toronto
knkim,tzwang,ryan.johnson@cs.utoronto.ca

*Cloudera
ippokratis@cloudera.com

ABSTRACT

Large main memories and massively parallel processors have triggered not only a resurgence of new high-performance transaction processing systems, but also the evolving of heterogeneous read-mostly transactions. However, many of these systems adopt lightweight optimistic concurrency control schemes that are only suitable for short, write-intensive workloads, similar to the TPC-C benchmark. By analyzing the desired features of main-memory optimized database systems, we argue that it is the system architecture that largely dictates the concurrency control scheme a system can employ, hence the type of workloads that can be gracefully handled. Therefore, it becomes difficult, if not impossible, to adopt a different concurrency control scheme to robustly handle various workloads without significant changes to the rest of the system.

We believe that transaction processing systems should be designed from the ground up with four basic requirements: to not heavily rely on partitioning; to provide flexible and robust concurrency control for the logical interactions between transactions; to address the physical interactions between threads in a scalable way; and to have a clear recovery methodology. We report on the design and prototype implementation of a system, called ERMIA, from the ground up to support the requirements we lay out. Our evaluation shows how the resulting architecture achieves these goals without unnecessary sacrifices to performance in other areas. In particular, we show that ERMIA achieves comparable performance with a recent high-performing transaction processing system in TPC-C, which is the workload most of the recent systems are optimized for. On the other hand, for a modified TPC-E workload with an extra read-mostly transaction, ERMIA achieves up to 40% higher performance.

1. INTRODUCTION

Modern systems with massively parallel processors and large main memories have inspired a new breed of high-performance, memory-optimized transaction processing systems [16, 26, 17, 20, 22, 37]. These systems leverage spacious main memory to fit the whole working set in DRAM with streamlined, memory-friendly data structures. Further, multicore and multi-socket hardware optimizations 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 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 [12].

Interactions at the logical level. Many designs exploit the reduction in the pressure on CC, by employing very optimistic and lightweight schemes, boosting even further the performance of these systems on suitable workloads. 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 database workloads, blending the gap between transaction and analytical processing. This trend is at least partly enabled by the improved concurrency and reduced contention offered by memory-optimized systems [9]. Mixed workloads have two significant impacts on CC, however. First, the write/read ratio decreases from 1:2 (e.g. TPC-C) to 1:10 or less (e.g. TPC-E [4, 34]), usually *by increasing the number of reads as the number of writes remains stable*. Second, workloads frequently include some fraction of large transactions that are *read-mostly rather than read-only*—a trend reflected in the TPC-E benchmark. Unfortunately, both of these workload properties result in larger effective CC footprints, putting pressure on the CC scheme. Therefore, going forward and as the industry shifts to heterogeneous workloads served by memory-optimized engines, it is vital for them to employ effective and robust CC schemes.

We observe that the CC schemes currently in vogue with memory-optimized systems are not robust under contention, particularly when short write-intensive transactions coexist with longer *read-mostly* transactions. The two main families of approaches can be loosely classified as two-phase locking (2PL) [10] and optimistic concurrency control (OCC) [19]. 2PL is common in traditional disk-oriented systems, and is often criticized because of high overheads, its policy of blocking transactions (leading to deadlocks and other scheduling problems), and a tendency to "lock up" (performance crash) once the aggregate transactional footprint grows too large, a state quickly attained when heavy read-mostly transactions enter the system. OCC, on the other hand, never blocks readers—and may not even block writers—thus avoiding most scheduling issues. Although they differ in details, the rising generation of memory-optimized systems almost universally adopts a form of OCC that is effectively single-versioned, with read footprint validation at pre-commit. Two systems that characteristically employ this type of OCC are Microsoft's Hekaton [20] and Silo [37]. This type of ap-

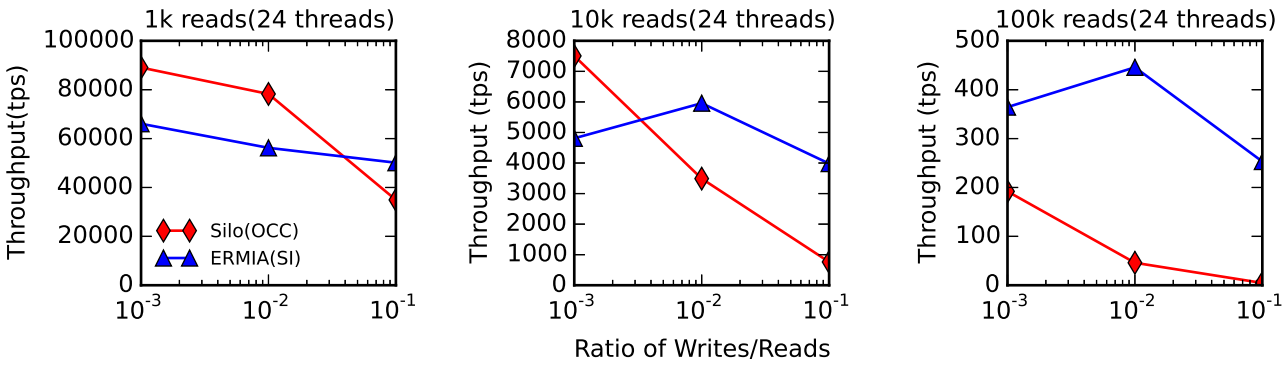

Figure 1: Performance of lightweight optimistic concurrency control and multi-version concurrency control, as the ratio of writes increases.

proach suffers badly in highly concurrent workloads [39] because transactions must abort if any portion of their read footprint is overwritten before they commit. In Figure 1 we demonstrate how the performance of Silo, a representative of the camp of transaction processing engines with lightweight OCC whose code is publicly available, degrades as transactions have larger read footprint or when contention increases. As a comparison, we also show the performance of ERMIA with snapshot isolation CC. (experimental setup details in Section 4.) 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.

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 transaction processing engine. As commodity server hardware becomes increasingly parallel many of the low-level issues (latching, thread scheduling, etc..) and design decisions—at the architecture level—need to be revisited. The form of logging 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. For example, it was reported in [29] that the implementation effort required to add support for serializable snapshot isolation (SSI) in Postgres was very high, due to a lack of supporting infrastructure in the system. The point is not that such design choices should be avoided, but rather that they should be made only with a full awareness of the consequences for concurrency control.

Physical partitioning. 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 [16, 17]. This execution model introduces a different set 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. [5, 2, 33], as well as for high availability and cost-effectiveness reasons, we predict that the successful architectures will combine scale-out solutions build 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.

In Section 2 we lay out the design principles that we believe are critical for transaction processing engines in the environment of highly-parallel servers with ample main memory. Next, on Section 3, we present *ERMIA*, a memory-optimized transaction processing architecture that carefully combines several techniques—epoch-based resource management, indirection arrays [31], and a specially-designed log manager—to enable robust CC, scalable thread interactions, and easy recovery. Section 4 compares the performance of an ERMIA prototype against a representative of the new breed of memory-optimized shared-everything transaction processing systems, and shows how the resulting architecture achieves its goals without necessarily sacrificing performance in other areas.

2. DESIGN DIRECTIONS

In this section we briefly discuss desired properties of transaction processing system architectures. We focus on three areas: the concurrency control mechanism that determines the interaction between concurrent transactions at the logical level; the mechanism that controls the interaction/-communication of threads at the physical level; and recovery. As we already argued, we are aiming for a single-node design that achieves scale-up with minimal physical partitioning.

Append-only storage. Append-only storage allows drastic simplifications of both I/O patterns and corner cases in the code. Combining a carefully designed log manager and indirection arrays (below) 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.

Logging. Recovery should be centerpiece in transaction processing system designs. Log managers are a well-known source of complexity and bottlenecks in a database system. Many of the recent proposals do not provide a thorough design for recovery, in stark contrast to more venerable logging protocols such as ARIES [25].

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 [14] or distribute [38] the log, with partial success and significant additional complexity. Systems such as H-Store [16] (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, and transactions *cannot see their own writes without sacrificing performance*.

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). 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, previous work has shown that such a system can scale to a few *millions* of commits per second [37], while still preserving ordering information that enables advanced concurrency control schemes. By way of comparison, the current world record in TPC-E is not quite 9Ktps. We present this log manager in Section 3.

Concurrency control. 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 [11]; roughly speaking, that means that a lightweight OCC scheme would likely outperform it until its abort rate rises past 25%.

Having said that, typical memory-optimized engines that employ lightweight OCC and running on modern commodity servers, leave some room for exploration. First, a contentious workload can easily drive abort rates to 25% or higher, opening the door for heavier but more robust schemes. Second, modern systems achieve extraordinarily high throughput for short-running, partition-able transactions with predictable footprints, and it might be worth trading 25% of peak performance for more robust behavior across a wider spectrum of transactional and mixed workloads. In other words, it is easier to tolerate losing 15-20% of peak performance if net throughput is still several hundreds of thousands of transactions per second.

There are different flavor of optimistic, or opportunistic, CC. Many recent systems adopt a lightweight read validation step at the end of the transaction, during pre-commit. 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). Further, these optimistic schemes make no guarantee that it is safe to retry the transaction immediately after pre-commit fails, an issue that performance evaluations typically downplay by not retrying failed transactions at all. Ideally, a good CC scheme would provide a "safe restart" property [29] where the system guarantees that a transaction will not fail twice because of the same conflict. 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.

At least to our knowledge, there is no (publicly available) system that is both fast enough and has the appropriate internal infrastructure to support the implementation of robust CC schemes. The 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.[1] 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). Many of the design decisions we outline in Section 3 were specifically taken in order to allow more flexibility in implementing multiple CC schemes without facing such severe performance trade-offs.

Physical layer. 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.

One promising technique that provides desirable properties for both CC and physical contention is the notion of an *indirection array* [31, 8] for mapping logical object IDs to physical locations. For example, an indirection array reduces the amount of logging required for updates in append-only systems, because updating a record only requires installing a new physical pointer at the appropriate logical array entry; without the indirection, creating a new version requires updating every reference to that record. Updating leaf entries of tree-based secondary indexes with large nodes can be especially expensive, as all nodes on the path from root to leaf must be rewritten. In addition to the additional I/O burden, frequent root node updates lead to severe physical contention and—depending on the CC implementation—false positive CC aborts. Write-optimized data structures such as LSM trees [32] alleviate the I/O bottleneck but shift significant burden to readers, who must now reconcile multiple datasets when performing range scans.

As a comparison, Silo does not employ indirection arrays. Instead, it 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, Hekaton is technically multi-versioned, but the

[1] Once all this groundwork was done, extending Postgres to other CCs is relatively easy.

snapshot-plus-read-validation CC scheme it uses means that older versions become unusable to update transactions as soon as any overwrite commits. For all practical purposes, both systems are multi-versioned only for read-only transactions.

Indirection arrays are 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. Similarly, presence of an uncommitted version makes write-write conflicts easy to detect and manage. Indirection also eases the reclamation of obsolete versions in the background, avoiding interference with foreground transaction processing.

With indirection arrays, even anti-caching [7] is largely simplified: in the vanilla anti-caching algorithm all the secondary indexes have to be updated whenever a record is evicted from memory, with some considerable overhead. The indirection array gives a convenient place to replace an in-memory pointer with an on-disk pointer, and so in some sense acts as a lightweight buffer pool.

There are also low-level reasons, detailed in Section 3.5, for using indirection arrays. For example, space management becomes easier, and they admit a convenient analog to cache-friendly compact index structures such as CSB+Trees [30]. These benefits do not justify adopting indirection, but they become convenient to use once the indirection layer is in place for other reasons.

Epoch-based resource management. Resource management is a key concern in any storage manager, and one of the most challenging aspects of resource management 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 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 [24] 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 3.3, 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

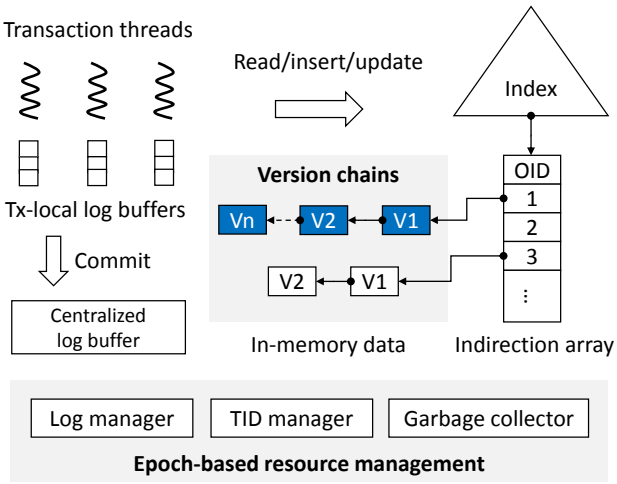

Figure 2: Architecture of ERMIA.

ERMIA is designed around epoch-based resource management and extremely efficient, centralized logging. Figure 2 shows the major components in ERMIA and how they interact with each other. The log manager, TID manager, and garbage collector are three major entities that use the epoch-based resource management machinery. To avoid log buffer contention, each transaction acquires its private log buffer from the log manager. Upon commit, the transaction will fix its global order in the log via a single compare-and-swap instruction and flush all its log records in one go (see 3.2 for details). Although both log buffers and TIDs are managed by the epoch-based resource management mechanism, they use separate epoch duration and can track stragglers efficiently. In ERMIA, transaction threads access the database through indexes. Different from traditional tree structures which give access to data in the leaf level, in each index we embed an indirection array, which can provide easy implementation of snapshot isolation and garbage collection. Indexed by object IDs (OIDs), each array entry points to a chain of historic version for the object (tuple). The garbage collector periodically goes over all version chains and remove superseded versions that are not needed by any transactions.

3.2 Logging

The log manager is a pivotal component in most database engines. It provides a centralized point of coordination that other pieces of the system build off of and depend on. Because of its central nature, 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. Useful features of the log include:

1. Communication in the log manager is extremely sparse. Most update transactions will issue just one global atomic compare-and-swap 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 log.

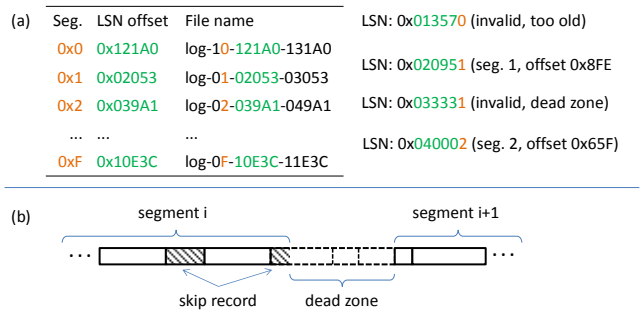

Figure 3: ERMIA's log manager.

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.

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 log sequence number (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 3(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 validates 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.
- 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 3(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").

As a final note, we observe that, for many objects in the system, their physical location is the log record that created their latest version. We therefore extend the LSN with a notion of "address spaces" that identify whether the offset points to the log, secondary storage, or memory; the indirection array stores these augmented LSN to very cleanly implement anti-caching [7].

3.3 Epoch-based resource management

Epoch-based memory management–exemplified by RCU [24]—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)

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

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 [24], and a very short timescale epoch manager tracks transaction IDs (TIDs), which we recycle aggressively (see details in Section 3.4).

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).

3.4 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), start 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.

3.5 Indirection arrays

The indirection arrays used in ERMIA are similar to the ones proposed in the literature [31, 8]. All logical objects are identified by an object ID (OID) that maps to a slot in an OID array that contains the physical pointer to data. The pointer may reference disk, or a chain of versions stored in memory. As with Hekaton, uncommitted versions are never written to disk; but unlike Hekaton, we dispense with delta records (too expensive to apply) and use pure copy-on-write. New versions can be installed by an atomic compare-and-swap operation, and an uncommitted record at the head of

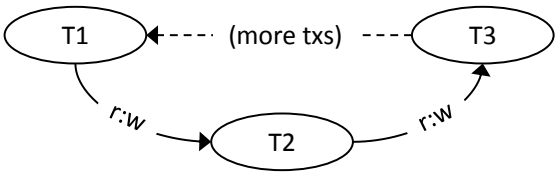

Figure 4: The "dangerous structure" that must exist in every serial dependency cycle under snapshot isolation. Here, T1 and T2 read versions that are overwritten by T2 and T3, respectively, and T3 commits before both T1 and T2.

the chain constitutes a write lock for CC schemes that care to track write-write conflicts (as most do).

3.6 Concurrency control

ERMIA has been designed from the ground up to allow efficient implementations of a variety of CC mechanisms, including Silo/Hekaton flavored read-set validation and snapshot isolation. Moreover, it allows efficient implementations of non-trivial CC schemes (other than simplistic read-set validation) to handle heterogeneous workloads gracefully and make them serializable (e.g., complex financial transactions represented by TPC-E [35]). Different components in ERMIA work together to make this possible: indirection arrays allow cheap (almost free) multi-versioning; at an extremely low overhead, the log gives total commit ordering, which is the key to implement snapshot isolation; the transaction manager can also help determine a version's age easily, making resource management straightforward.

Depending on the target workload, ERMIA can use read set validation, two-phase commit, snapshot isolation, or even serializable snapshot isolation (SSI) [3]. We focus on handling *read-mostly* workloads gracefully while maintaining comparable performance for short, update-intensive workloads. Although not serializable, snapshot isolation is an ideal choice to start from: long, read-mostly transactions will have much higher chance to survive when compared to read-set validation, and ERMIA can provide SI at virtually zero additional cost. When desired, ERMIA uses SSI to guarantee serializability. In particular, our variant of SSI takes advantage of ERMIA's design to eliminate the high overheads and communication bottlenecks that plague all existing SSI implementations the authors are aware of. The following paragraphs briefly highlight the SSI variant we developed for ERMIA.

Stamp-based tracking. SSI ensures serializability by tracking the "dangerous structure" that must exist in every serial dependency cycle under snapshot isolation. As shown in Figure 4 [3], transaction T1 or T2 must abort if a read-write dependency exists and T3 committed first. Since ERMIA provides global ordering through the log, and commit timestamps are available during pre-commit, tracking and detecting the dangerous structure becomes straightforward and multiple transactions are allowed to make pre-commit progress simultaneously. In contrast, many SSI implementations allow only one transaction at a time to enter pre-commit, and fully decentralized OCC schemes cannot easily provide SI because the lack of a global commit order makes it expensive or even impossible to determine which of two transactions precedes the other.

The basic tracking for SSI is facilitated by maintaining three timestamps in each transaction: `s0`, `s1`, and `s2`.[3] `s0` is the transaction's own commit time. `s1` is the "successor stamp," and records the smallest `s0` of any transaction that overwrote any version in T's read footprint. Similarly, `s2` records the smallest `s1` of any transaction that overwrote T's read footprint. Finally, each version maintains an `rstamp` that records the latest commit timestamp of any reader.

Whenever transaction T performs a read, it checks whether `s2` is set on that version. If so, T is T1 of a dangerous structure where T3 committed first; T2 already committed, so T1 must abort to prevent a potential cycle. T next checks whether `s1` is set. If so, T is a potential T2 of a cycle (the "pivot") and we prefer to abort it if T has overwritten any in-flight readers: doing so provides the "safe restart" property discussed earlier. If no such reader exists, T is allowed to continue, but records the smallest `s1` it encounters in order to check for late-arriving readers during pre-commit. T also remembers the smallest `s0` of any version it has read. At pre-commit, T checks again whether it is the "pivot" of any dangerous structure involving an in-flight T1, aborting if so. Otherwise, T can enter post-commit and finalize stamps in each version it created and overwritten. Versions created by T will have T's commit timestamp as its s0, while versions overwritten by T will have T's commit timestamp as its `s1`, and the remembered `s1` as its `s2`. Finally, versions read by T will have T's commit timestamps its `rstamp` if it is smaller than T's commit timestamp. There is no need to update in-flight readers because of the pivot test already performed. We next describe how the commit protocol works.

Commit protocol. We divide the commit process into two parts: pre-commit and post-commit. Pre-commit under SSI involves three major steps: (1) obtain a commit stamp; (2) examine the read set to learn the smallest `s1`, as shown in the first half of Algorithm 3.6; (3) examine overwritten versions by iterating the write set to update T's rstamp. As shown in the second half of Algorithm 3.6, during step 3, T must abort if it has as valid `s1` and it has overwritten any in-flight readers or any version with `rstamp` larger than `s1` (i.e., T will be the T2 in a dangerous structure if allowed to commit). Crucially, these pre-commit checks can be performed in parallel by multiple transactions, because all transactions in pre-commit have valid commit stamps and therefore agree on which of members of a potential dangerous structure attempted to commit first. If the transaction survived pre-commit, version stamps will be updated. Note that these version stamps (except the commit stamp) are only needed at run time; the versions stored in the log and on disk do not require SSI checks because no in-flight transactions at the time of a crash could have committed. The above commit protocol largely simplifies SSI checks using ERMIA's infrastructure.

In the rest of this section we describe major optimizations that further improve SSI's performance, followed by a description of the phantom protection mechanism we use.

Lightweight reader tracking. As described in Algorithm 3.6, the writer needs to track all readers of versions it overwrites, and verify readers' status at pre-commit. To achieve this, one might maintain a linked list of reader TIDs

[3] These timestamps are copied to the versions that transaction creates during post-commit, allowing TID table entries to be reclaimed in a timely manner, but other strategies are possible as well.

Algorithm 1 ERMIA SSI commit protocol.

```
1: Input: Committing transaction T
2: T.cstamp = log.next_commit_stamp()
3: /* Get the smallest s1 from all reads. */
4: ts1 = 0;
5: for each read version v : do
6:    T_s = overwriter transaction of v
7:    if T_s exists and T_s.cstamp < cstamp then
8:       Spin until T_s is resolved
9:       if T_s is committed then
10:         T.s1 = min(T_s.cstamp, T.s1)
11:   else
12:      T.s1 = min(T_s.cstamp, v.s1)
13:
14: /* Check writes. */
15: for each written version w : do
16:   if w is an insert then
17:      continue
18:   v = the overwritten version
19:   for each reader transaction T_r : do
20:      if T_r is active then
21:         T.abort()
22:      T.rstamp = max(T.rstamp, w.rstamp)
23:      if T.rstamp ≥ T.s1 then
24:         T.abort()
```

in each version, or in a system-wide lock manager of sorts (Postgresql SSI follows the latter approach). Readers add themselves to the list when they read the version and unlink when they commit or abort. However, maintaining such a list—whether in version headers or a lock manager—increases the effective size of a version significantly and causes significant extra cache misses, especially for small versions that are frequently read by multiple transactions (a common case for main-memory systems with massively parallel hardware). Since ERMIA executes each transaction from beginning to end using a single thread, without changing transaction context, we employ an optimization to embed reader information compactly in versions. The system maintains a small table with one entry per worker thread, with each entry assigned to one worker thread at start-up. The worker thread stores the TID of any transaction it is currently executing in this table. The reader list for a given version then shrinks to a bitmap, with a set bit indicating that the corresponding worker thread has read the version. An overwriting transaction can thus easily discover all readers of a version, by reading thread TIDs for all set bits during pre-commit.

Read optimization. Maintaining read sets is a major performance overhead not only for SSI, but also for OCC schemes that guarantee serializability. Long reader transactions are especially expensive to track, and existing optimizations usually handle only a limited class of read-only transactions [29], while providing no benefit for *read-mostly* transactions. To address this issue, we observe that most tuples in a database tend to be "old", i.e., not recently updated. Long, read-mostly transactions tend to access a large number of these old tuples. Based on this observation, we optimize reads by setting an age threshold such that any transaction reading an "old" version can commit without verifying that read. In detail, a version's age is determined by subtracting its creation timestamp from the accessing transaction's begin timestamp. A version is determined to be "old" if its age is larger than a predefined threshold, and will not be inserted to the read set. This greatly reduces tracking overhead, but the reader can no longer perform complete SSI checks during pre-commit. Instead, the responsibility for checking those versions passes to any transaction that overwrites an old tuple: the writer must assume that the old tuple does, in fact, have readers, and will abort if that assumption makes it the pivot in a dangerous structure.

The above method can generate false positives, because the writer must conservatively assume a reader exists, when this may not be true. To reduce such false positives, we employ a `bstamp` field in each old version to record the largest reader's begin timestamp (which the reader maintains at the time of the read). The writer will only abort if at least one reader found in worker thread TID list has a begin timestamp older than the version's `bstamp`; this extra check eliminates false positives for records that have not been read recently.

Phantom protection. Although SSI prevents serialization dependency cycles, serialization failures can also 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 [18, 23]. Since ERMIA's current implementation is a heavily modified version of Silo [37], we inherit the latter's tree-version validation strategy. The validation is lightweight, if conservative, and using it also provides a more fair comparison in the evaluations that follow. 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. In addition to maintaining the read set, same as Silo, 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 [37] for more 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. Although OID arrays are normal objects, which are themselves reached through a master OID array, they can occupy hundreds of MB or more and are thus far too large for copy-on-write to be practical at each record insertion or deletion. Instead, 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 a 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 for-

ward 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 (typically under 10% a block) 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, recovery only needs to rebuild the OID arrays in memory using sequential I/O; anti-caching will 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 compare the performance of ERMIA and Silo, a representative of the lightweight OCC camp, from two perspectives: (1) the impacts of concurrency control scheme on performance and (2) scalability of the underlying storage manager. The purpose of our evaluation is two-fold. First, we show that although recent OCC proposals for main-memory OLTP perform extremely well, they are only suitable for a small fraction of transaction workloads—short update intensive transactions—primarily due to limitations imposed by their (tightly-integrated) concurrency control mechanism. Second, we demonstrate that ERMIA is able to offer robust concurrency control, maintaining high performance even under contentious workloads without sacrificing features. At the same time, ERMIA is also able to match the performance of specialized OCC solutions when running their favorite workloads. We also show that because of carefully orchestrated communications and interactions in both the physical and logical levels, ERMIA scales well under various types of workloads.

4.1 Experimental setup

We used a 4-socket server with 6-core Intel Xeon E7-4807 processors, for a total of 24 physical cores and 48 hyperthreads. The machine has 64GB of RAM. All worker threads were pinned to a dedicated core and NUMA node to minimize context switch penalty and inter-socket communication costs. Log records are written to tmpfs asynchronously. We measure the performance three systems: Silo, ERMIA with SI (ERMIA-SI), and ERMIA with SSI (ERMIA-SSI). Next we introduce the benchmarks.

4.2 Benchmarks

We run a microbenchmark, TPC-C and TPC-E benchmarks on all variants. In each run, we load data from scratch and run the benchmark for 40 seconds.

Microbenchmark. We use the Stock table of TPC-C database (24 warehouses) for the microbenchmark transactions. The Stock table has 100K*Warehouses (2.4M) records. Each transaction randomly picks a subset of records to read and a smaller fraction of those to update.

TPCE-original. TPC-E is a new standard OLTP benchmark. Although TPC-C has been dominantly used to evaluate OLTP systems performance for the past few decades, TPC-E was designed as a more realistic OLTP benchmark with modern features. TPC-E models brokerage firm activities and has more sophisticated schema model and transaction execution control[35]. It is also known for higher read to write ratio than TPC-C.

TPCE-hybrid. Since TPC-E is not a heterogeneous workload, we introduce a new read-mostly analytic transaction that evaluates aggregate assets of a random customer account group and inserts the analytic activity log into *analytic_history* table. Total assets of an account are computed by joining *holding_summary* and *last_trade* tables. The vast majority of contentions will occur between the analytic transaction and *trade-result* transaction. 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). The followings are our new workload mix with the analytic transaction: broker-volume (4.9%), customer-position (8%), market-feed (1%), market-watch (13%), security-detail (14%), trade-lookup (8%), trade-order (10.1%), trade-result (10%), trade-status (9%), trade-update (2%) and analytic (20%).

TPCC-fixed. TPC-C is well-known for update-heavy workload and small transaction footprints. Also, it is well-partitionable workload; conflicts can be reduced even further by partitioning and single-threadinging on local partitions, creating a nearly ideal environment for lightweight OCC schemes. In the partitioned TPC-C, the only source of contention is multi-partition transaction, however, its impact on CC is dampened by small transaction footprints, avoiding the majority of read-write conflicts. In TPCC-fixed, entire database is partitioned, all worker threads are given a local warehouse at benchmarking initialization, and does not change the partition binding during runtime. The fraction of cross partition transaction is 1%.

TPCC-random. In TPCC-random, we enforce worker threads to pick a partition randomly during runtime, following non-uniform distribution. The purpose of this modification is to create a reasonable amount of contention in the workload.

4.3 TPC-E

We first explore how ERMIA and Silo react to contention in heterogeneous workload. Figure 5 (left) shows the normalized throughput with 24 worker threads, varying contention degree when we run TPC-E with the analytic transaction (TPCE-hybrid). To vary the degree of contention, we adjust the size of a customer account group to be scanned by the analytic transaction from 1% to 60%. At the lowest contention degree(1%), ERMIA-SI and Silo deliver similar performance. However, SILO starts to lag behind ERMIA-SI from 5% contention degree and the performance gap between them increases. Especially, at 60% contention degree, SILO's throughput is only 59% of ERMIA-SI throughput. If the analytic query takes more than 20% in the workload mix, the performance gap will be even larger. ERMIA-SSI also does not collapse by contention. Even though it falls behind Silo by 26% at 5% contention degree, it starts to close the gap quickly and catches up with Silo at 20% contention degree. However, ERMIA-SSI still suffers from serializability checking cost and larger cache footprint for transaction dependency tracking. The main cause for the result is that the contention in the workload imposed heavy pressure on the OCC protocol; OCC enforced analytic queries to abort even if a small fraction of read-set is invalidated by

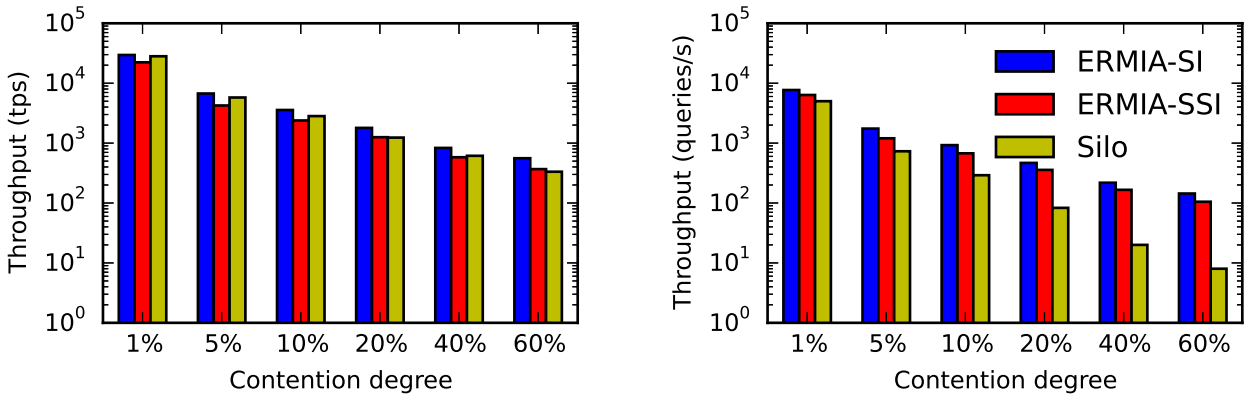

Figure 5: Normalized throughput (left); normalized analytic transaction throughput (right) of TPCE-hybrid, varying contention degree.

updaters. Meanwhile, ERMIA endured the contention by protecting the queries from updaters effectively with snapshot isolation and distributed the contentions across multiple versions. we now take a closer look on the throughput of the analytic transactions to see how well both systems support heterogeneous workloads. Figure 5 (right) illustrates normalized query throughput from the previous experiment. Even at the 1% contention degree, Silo only commits 65% as many queries as ERMIA-SI. As query size increases, Silo's queries become more vulnerable to updaters and query throughput drops sharply. These massive query aborts affected the overall throughput in the previous experiment, and would have an even larger impact if aborted queries were retried until successful rather than dropped. In recap, Silo's OCC scheme favors update transactions and transactions with large read footprints can starve in contentious workloads. Meanwhile ERMIA provides balanced query/-transaction performance. This experiment shows that it is discouraging to serve emerging heterogeneous workload with OCC, even if overall throughput looks reasonable.

In Figure 6 (left), we fix contention degree to 20% scan range and see scalability trends, increasing the number of workers. Overwhelmed by contention, Silo does not achieve linear scalability. ERMIA benefits from its robust CC scheme and its storage manager retains linear scalability. This figure shows that not only the scalability of underlying physical layer, but also CC scheme performance dictates overall performance. We also performed the same experiment in the original TPC-E, without the analytic transaction. As illustrated in Figure 6 (right), both ERMIA-SI and Silo achieve linear scalability over 24 cores. Silo does not suffer from contention, since TPCE-original has insignificant contention. ERMIA-SSI delivers 83% of ERMIA-SI performance with 24 threads due to the serializability cost.

4.4 TPC-C

We also run normal TPC-C, where lightweight OCC shines. This experiment focuses on evaluating the storage manager's natural scalability, as TPC-C imposes little pressure on the CC scheme.

We measure throughput of all systems in TPCC-fixed, varying the number of worker threads. Figure 7 (right) shows that all systems scale reasonably over 24 cores. ERMIA falls behind Silo by 10% and 20%, with SI and SSI respectively, at the peak performance for the following reasons. First, OCC did not suppress throughput due to lack of contention. Second, Silo benefits from TPC-C's extremely small cache footprint (L2 resident). Unlike Silo, ERMIA maintains multi-versions in the indirection array, resulting in additional cache miss costs during indirection array traversals to find the desired version. This was invisible in the TPC-E because its large cache footprint already overwhelmed the processor caches. Thus, we can characterize the case where Silo outperforms ERMIA in workloads where 1) contention is rare and 2) transaction footprint is small enough to take advantage of processor caches; TPC-C is the compelling example of such workload. ERMIA-SSI had an additional 10% cost to guarantee serializability.

We now run TPCC-random and see how the previous results change. As shown in the Figure 7 (right), we can see that Silo is more sensitive to contention than ERMIA. Compared to TPCC-fixed, the throughput of Silo decreased by approximately 30%, while ERMIA lost only 15% lower performance, catching up with Silo.

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. [29]. We follow the findings, trying to detect conflicts early.

The indirection map, which is central to ERMIA's design, is a well-known technique, for example presented in [31].

Many of the memory-optimized systems adopt lightweight optimistic concurrency control schemes that are suitable only for a small fraction of transactional workloads. The designs can be categorized in three categories: non-partitioning- and partitioning-based systems and clustered solutions.

Silo's [37] employs a light-weight optimistic concurrency control that performs validations at pre-commit. That, as

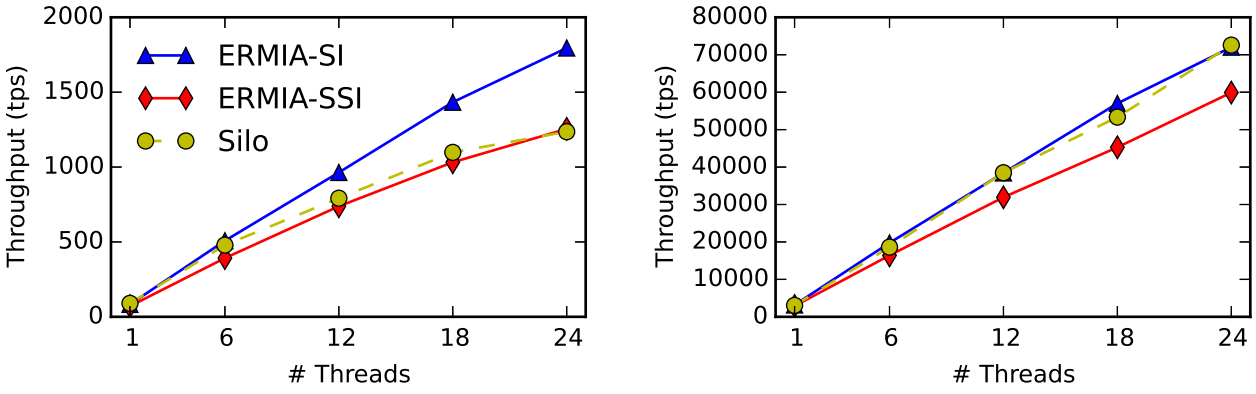

Figure 6: Throughput when running TPCE-hybrid (left); TPCE-original (right).

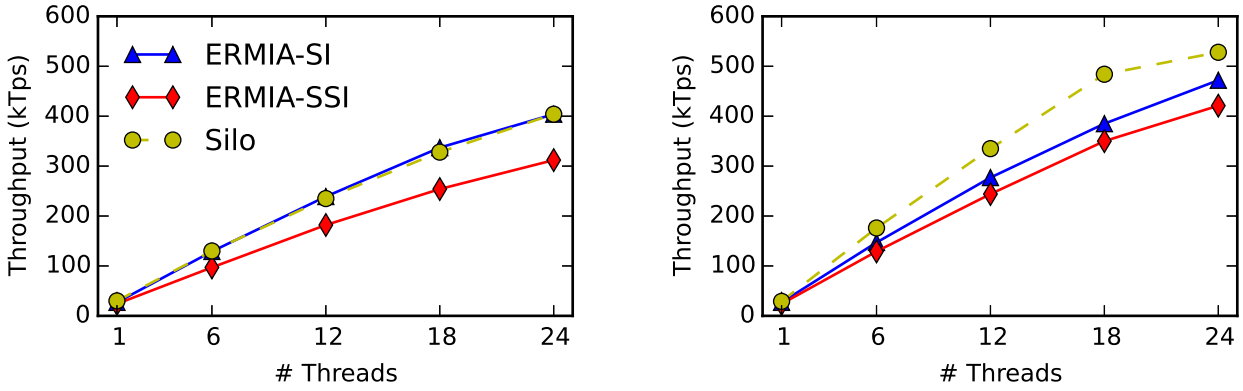

Figure 7: Throughput when running TPCC-random (left); TPCC-fixed (right).

we showed in Section 4, performs well only in a limited set of workloads. Microsoft Hekaton [8] employs similar multi-versioning CC [20]. It is worth mentioning that Hekaton also uses a technique similar to the indirection map, which we also use.

H-Store (and its commercial version, VoltDB) is a characteristic partitioning-based system [16]. 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. Problems raise when the system has to execute mutli-site transactions, 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 [15], but also 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. [6, 28, 36]. Hyper [17] follows H-Store's single-threaded execution principle. To scale up to multi-cores they employ the hardware transactional memory capabilities of the latest generation of processors [21].

DORA [26] employs logical partitioning PLP [27] 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. PLP is Both DORA and PLP use Shore-MT's codebase [13], which is a scalable but disk-optimized storage manager with significantly bloated code-base. 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.

In addition to the work on scaling up the performance of transaction processing systems in mutlicore and multisocket environment, there has been also lots of interest on scaling out. Those scale out systems, such as Google's Spanner [5], RAMP [2] and Calvin [33], emphasize the weakness of the partitioning-based camp. Because the easy to partition workloads and databases would have already been partitioned of different physical nodes. Therefore whatever data are assigned to a single node it would be quite difficult to further partition. Hence the need of scalable multicore and multisocket transaction processing system designs.

6. CONCLUSION

In this paper we underlined the weaknesses of recent OCC-based transaction processing system proposals, and presented a novel system that does not rely on partitioning, provides robust concurrency control even for mixed workloads, addresses the physical interactions between threads in a scalable way and has a clear recovery methodology. We believe that this is a significant step toward efficiently processing mixed transactional workloads.

7. REFERENCES

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM TODS*, 12(4):609–654, 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. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, 2008.

[4] 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.

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

[6] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3:48–57, 2010.

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

[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] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

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

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

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

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

[15] E. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010.

[16] 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.

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

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

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

[20] 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.

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

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

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

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

[25] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.

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

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

[28] A. Pavlo, E. P. C. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2):85–96, 2011.

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

[30] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.

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

[32] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *SIGMOD*, 2012.

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

[34] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP benchmarks - the obsolete, the ubiquitous, the unexplored. In *EDBT*, 2013.

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

[36] 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.

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

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

[39] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. Technical report, MIT CSAIL, 2014.