Cicada: Dependably Fast Multi-Core In-Memory Transactions

Hyeontaek Lim
Carnegie Mellon University
hl@cs.cmu.edu

Michael Kaminsky
Intel Labs
michael.e.kaminsky@intel.com

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

ABSTRACT

Multi-core in-memory databases promise high-speed online transaction processing. However, the performance of individual designs suffers when the workload characteristics miss their small sweet spot of a desired contention level, read-write ratio, record size, processing rate, and so forth.

Cicada is a single-node multi-core in-memory transactional database with serializability. To provide high performance under diverse workloads, Cicada reduces overhead and contention at several levels of the system by leveraging optimistic and multi-version concurrency control schemes and multiple loosely synchronized clocks while mitigating their drawbacks. On the TPC-C and YCSB benchmarks, Cicada outperforms Silo, TicToc, FOEDUS, MOCC, two-phase locking, Hekaton, and ERMIA in most scenarios, achieving up to 3X higher throughput than the next fastest design. It handles up to 2.07 M TPC-C transactions per second and 56.5 M YCSB transactions per second, and scans up to 356 M records per second on a single 28-core machine.

1. INTRODUCTION

Multi-core in-memory transactional systems promise significant performance gains over disk-based systems, but recent proposals often fail to deliver consistently high performance outside of a narrow spectrum of workload characteristics, suffering low performance under high contention [43] and limited scalability with multiple cores [63]. Several proposed solutions to address these problems use hard partitioning [29, 30, 56] or batching [16, 43, 50], incurring high latency and/or requiring specific forms of transaction submission and execution, which limits their applicability.

This paper presents *Cicada*, a multi-core in-memory database system with fast serializable concurrency control that is *optimistic*, *multi-version*, and *multi-clock*. Worker threads execute transactions speculatively without eagerly writing to the shared memory (optimistic); use a certain version among multiple versions of records (multi-version); and maintain per-thread clocks (multi-clock).

Cicada's design reduces the overhead and contention of transaction processing at several levels of the system. Its *optimistic multi-version* design reduces both memory-access-level interference and transaction-level conflicts between concurrent transactions. Scalable timestamp allocation using *distributed clocks that are loosely synchronized with other clocks* enables tens of millions of transactions per second on a single machine, which was infeasible in prior fully-featured multi-version databases. *Best-effort inlining* and *rapid garbage collection* make memory access efficient, enabling this multi-version design to share the low overhead of single-version schemes that shine under low contention. *Contention regulation* performs globally coordinated backoff to mitigate the cost of frequent aborts that cause performance collapse on prior optimistic schemes under contention.

Lessons from designing Cicada include:

- Single-node systems can leverage distributed systems techniques to scale well on multi-core CPUs.
- Multi-version designs, despite their complexity, can be more efficient than single-version designs.
- Optimistic execution performs much better under contention by using new techniques to mitigate the overhead from aborts.

We evaluate Cicada using the TPC-C [54] and YCSB [8] benchmarks against prior state-of-the-art in-memory database designs including both single-version schemes—Silo [55], TicToc [64], FOEDUS [32], MOCC [57], and two-phase locking [3, 15]—and multi-version schemes—Hekaton [12, 35] and ERMIA [31].

Our experiments show that Cicada performs better than, or at least equally to, compared systems. Cicada achieves 3X and 1.37X higher throughput on contended TPC-C and YCSB workloads, respectively, than the next fastest scheme. Under uncontended YCSB scenarios, Cicada achieves up to 69.2% higher throughput than other systems. On uncontended TPC-C, Cicada is at least 5.54% faster than others using the same benchmark implementation, and it is at most 11.1% slower than FOEDUS and MOCC that employed TPC-C-specific optimizations, including index bypassing and vertical partitioning, which we chose not to port to Cicada and other systems to maintain the generality of our experiments. Cicada consistently outperforms any other multi-version scheme and two-phase locking. On a single 28-core machine, Cicada processes 2.07 M TPC-C transactions per second and 56.5 M YCSB transactions per second, and scans up to 356 M records per second.

Section 2 examines modern in-memory concurrency control schemes. Section 3 presents the design of Cicada. Section 4 evaluates Cicada against prior state-of-the-art concurrency control schemes and performs factor analysis.

2. TODAY'S MULTI-CORE IN-MEMORY CONCURRENCY CONTROL SCHEMES

This section describes modern concurrency control schemes, and identifies their strengths and weaknesses, with a short summary of Cicada's design components that address these weaknesses.

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

ACM ISBN 978-1-4503-4197-4/17/05.

DOI: http://dx.doi.org/10.1145/3035918.3064015

2.1 Optimistic Concurrency Control

Many high-speed in-memory databases designed for multi-core systems, including Silo [55], FOEDUS [32], MOCC [57], BCC [65], and TicToc [64], use a variant of optimistic concurrency control (OCC) [33]. OCC assumes that conflicts between transactions are rare: It executes the main part of transactions without locking records, which reduces locking overhead. It has three phases: The transaction reads records from the shared memory and performs all writes to a local, private copy of the records (the "*read phase*"). Later, the transaction performs a series of checks (the "*validation phase*") to ensure consistency. After successful validation, the OCC system commits the transaction by making the changes usable by other transactions (the "*write phase*"). Because of its optimistic assumption, OCC performs best under low contention.

Recent OCC designs are *single-version concurrency control* (1VCC) with *in-place updates* ("OCC-1V-in-place") that make transaction execution lightweight and keep garbage collection overhead low [55, 64, 65]. 1VCC keeps a single committed copy of each record, and all read-write transactions use that copy. 1VCC often supports *read-only snapshots* that provide a consistent view of slightly stale data for read-only transactions [55, 65]. With in-place updates, a write to an existing record overwrites the old value without allocating new memory for the new value in the write phase. This approach differs from the original OCC, which exchanges a pointer to the record data to perform a write [33], requiring garbage collection of the old value.

Strengths: OCC's lock-free read phase permits more concurrency than pessimistic schemes such as two-phase locking (2PL) [15]. A reader of a record is blocked briefly while a writer of the same record is being validated, and writers are never blocked by readers. Particularly in main-memory databases, staging uncommitted changes in local memory is beneficial because it reduces cache misses caused by concurrent transactions writing to the same record. The low overhead of OCC-1V-in-place makes it a favored design choice by state-of-the-art schemes for multi-core in-memory databases; these schemes have high performance under low contention [55].

Weaknesses: OCC suffers from both well-known and less-known problems:

(1) *Frequent aborts* under high contention degrade OCC's performance. While OCC's lock-free execution provides good concurrency on multi-core CPUs, it risks many aborts by executing transactions too optimistically. Furthermore, 1VCC limits the ability to avoid conflicts between transactions because it can serve only the latest version of records.

Aborts can be expensive. Executing an aborted transaction consumes local CPU cycles. It can also slow down other threads by *reading* a memory location being written, repeatedly invalidating the involved threads' L1 and L2 cacheline. Ironically, OCC's lightweightness can aggravate this low-level contention by retrying aborted transactions too rapidly.

Current solutions to reduce the effect of frequent aborts are limited. TicToc relaxes the limitation of 1VCC with more flexible transaction ordering [64], but this technique still disallows accessing an earlier version of the record if the record has been already updated by a concurrent transaction. Read-only snapshots are not a complete solution either; they cannot be used in read-write transactions because serializability protocols designed for 1VCC do not permit such multi-version access. Snapshot support can reduce throughput by about 10.5% [55], which compromises the low-overhead advantage of 1VCC. The staleness of snapshots in 1VCC is typically high because snapshots are generated at a coarse-grained interval (e.g., 1 second [55]), which further limits their applicability.

(2) *Extra reads* are a less-known, but important source of performance overhead in OCC-1V-in-place. In-place updates can cause temporary inconsistency to the record data because transactions in the read phase do not lock records, allowing concurrent writes. A reader may see record data that is partially overwritten, observe different data for repeated reads, or even access an invalid memory location for variable-length data, which resembles inconsistency issues in transactional memory [20]. To handle the potential inconsistency within the record data, modern OCC schemes make a local copy of the record and verify the consistency of the copy before exposing it to the application [55, 64]. However, creating a consistent local copy incurs extra read(s) of the record before the application actually consumes it; the cost increases for larger records.

(3) *Index contention* may occur in the OCC designs that write to global indexes before entering the write phase. Upon creation of a new record, several OCC designs [32, 55] insert a new index entry to the table's indexes as well; the new record is locked to prevent concurrent transactions from reading it before it is committed. This *early index update* ensures that the new record satisfies unique-key invariants [55] and also simplifies making the index change visible to the current transaction. However, early index updates can create contention at indexes by frequently modifying their internal data structure even for the transactions that are eventually aborted. Furthermore, concurrent transactions that attempt to read the new record may be blocked for an extended period of time if the transaction that created the new record has a long read phase. In other words, index updates in many OCC designs neglect the OCC's principle of avoiding global writes during the read phase, experiencing common performance penalties of record updates in non-OCC designs.

OCC's weaknesses are largely attributable to the use of 1VCC. OCC is helpful in reducing cross-core communication, which is important for high-speed in-memory databases. The high abort rate of OCC is acceptable if it translates into significantly reduced memory/cache-level contention and the cost of aborts can be minimized. The extra reads problem is specific to OCC-1V-in-place. The cost of index contention can be reduced if an OCC design can avoid early index updates.

Cicada approach: *Contention regulation* automatically prevents excessive transaction restarts to increase commit throughput. *Optimistic multi-version* reduces the cost of aborts and avoids extra reads, achieving low index contention by allowing index updates to be deferred until validating the transaction.

2.2 Multi-Version Concurrency Control

Multi-version concurrency control (MVCC) [3] is a popular design choice for today's on-disk databases [5, 48]. While MVCC is less dominant for in-memory databases, recent research has led to several new in-memory MVCC schemes including Hekaton [12, 35], HyPer [44], Bohm [16], Deuteronomy [38, 39], and ERMIA [31]. MVCC reduces conflicts between transactions by using multiple copies (versions) of a record; a transaction can use an earlier version of a record even after the record has been updated by a concurrent writer. MVCC is an effective design for read-intensive workloads [31, 44].

MVCC uses a *timestamp* to determine which version of records to serve. A transaction is assigned a timestamp when it begins. A version has a *write timestamp*, which indicates when the version becomes valid, and a *read timestamp*, which specifies when the version becomes invalid or until when it must remain valid. The transaction compares its timestamp against versions' timestamps to find and use *visible* versions. The timestamps of versions use either the transaction's initial timestamp or a separate timestamp allocated at commit time.

Strengths: MVCC reduces transaction conflicts. MVCC can con-

High-level component	Contention level Low	Contention level High	Operation intensity Read	Operation intensity Write	Record size Small	Record size Large	High-speed
Optimistic multi-version	▽	▲	▲	▽	▽	▲	▽
Loosely synchronized clocks	–	–	–	–	–	–	▲
Best-effort inlining	▲	–	▲	–	▲	–	–
Rapid garbage collection	▲	▲	–	▲	▲	▲	–
Contention regulation	–	▲	▲	▲	–	–	–
Full Cicada system	▲	▲	▲	▲	▲	▲	▲

Performance Impact
▲ positive
▽ negative

Table 1: High-level components of Cicada and their performance impact for different workload characteristics.

tinue to process a transaction accessing a record even when the record has been updated by a concurrent transaction, whereas 1VCC experiences many conflicts under contended and/or long transactions [35].

Weaknesses: MVCC's main weaknesses are the high overhead of its transaction processing and data storage, and the contention during the timestamp allocation:

(1) *Computation and storage overhead of searching and storing multi-version records* can make MVCC require more CPU cycles and memory than 1VCC. In-memory databases amplify the effect of this computation and space overhead because they are expected to provide orders of magnitude higher throughput than disk-based databases, making CPU cycles precious; it is also harder to increase memory size than disk size, making the space overhead harder to tolerate. Most MVCC schemes use *indirection* to search versions in a list or an array [12, 31, 35, 39], which can become particularly expensive if it causes a cache miss and the workload's working set does not fit in CPU cache. A recent MVCC proposal [44] eliminates indirection for latest version access by performing in-place updates of the latest version; however, this design recreates the extra read problem of OCC-1V-in-place.

(2) *Large footprint* is caused by touching more memory than 1VCC to search versions and manage multi-version records. A larger working set reduces cache hit ratios, degrading the performance of in-memory transaction processing. Frequent garbage collection can keep the footprint small, but garbage collection must be efficient to avoid incurring high overhead.

(3) *Writes to the shared memory* are performed in most MVCC schemes during their main transaction execution [12, 35, 39, 44]. Such a design can harm multi-core performance [55].

(4) *A bottleneck at timestamp allocation* limits the scalability of most MVCC schemes. They use a straightforward centralized approach to allocate timestamps, wherein worker threads atomically increment a shared counter [12, 16, 35]. Because of the high cost of using atomic operations on a single memory location [10], the throughput of schemes using a shared counter is limited to a few million transactions per second, *whether or not the workload is contended*. This rate is an order of magnitude lower than the maximum throughput of fast 1VCC schemes [63, 64]. Future many-core CPUs may aggravate this scalability limit of prior MVCC schemes [57, 63].

The weaknesses of MVCC have been only partly addressed in modern MVCC designs [12, 16, 35, 44]. Existing MVCC schemes report 20–45% lower throughput than 1VCC under low contention [35, 44]. Even under contended workloads, most MVCC schemes have not shown a consistent performance advantage over 1VCC due to their high baseline overhead [63, 64].

Cicada approach: *Best-effort inlining* reduces indirection by inlining read-mostly records' version. *Rapid garbage collection* keeps a working set small by quickly reclaiming old versions. *Optimistic multi-version* avoids shared memory writes before transaction validation. *Loosely synchronized clocks* eliminate the timestamp allocation bottleneck.

2.3 Constrained Parallel Execution

Hard-partitioned databases, such as H-Store [29], VoltDB [56], and an early version of HyPer [30], divide the dataset into per-core partitions, giving each core exclusive access to its own partitions. These systems process transactions serially within a partition, avoiding the need for any concurrency control overhead. They excel under easily-partitionable workloads, but their performance rapidly degrades when more transactions cross partition boundaries [55].

Recent designs including Doppel [43], Bohm [16], Orthrus [50], and IC3 [58] regulate the parallelism of contended data access based on static and dynamic analysis on data access patterns. They reduce the cost of shared memory access to contended memory locations on multi-core CPUs and the rate of spurious aborts without statically dividing the main dataset. However, they suffer high latency caused by coarse-grained batching and/or require a pre-analysis step that forces submitting and executing a whole transaction at once.

Cicada approach: Cicada requires no batching or pre-analysis.

2.4 Hardware Transactional Memory

Hardware transactional memory (HTM) [23] provides a low-overhead tool to detect conflicts. The latest commercial processors support a version of HTM called restricted transactional memory (RTM) [27]. HTM-based designs improve transaction processing speed [37, 59, 60, 61]. Although HTM is promising for both uncontended and contended workloads, we focus on general concurrency control schemes that are applicable to a broader range of systems. We believe that leveraging HTM and designing general concurrency control schemes are not conflicting goals, considering that proposed HTM-based designs are founded on conventional schemes such as timestamp ordering [3, 6], OCC, and 2PL.

Cicada approach: Cicada is a non-HTM design.

3. DESIGN

Cicada is a multi-core in-memory database with serializability, featuring three key design aspects: *optimistic*, *multi-version*, and *multi-clock*. Cicada provides high, robust performance under low and high contention, on read-intensive and write-intensive workloads, on small and large records, and for high-speed workloads that execute tens of millions of transactions per second.

Table 1 summarizes Cicada's high-level design components, as well as the workload characteristics that each addresses. *Optimistic multi-version execution*, an optimistic timestamp ordering scheme [34] designed for multi-core in-memory OLTP, reduces both memory access–level and transaction-level interference at records accessed by concurrent transactions, avoiding in-place updates to eliminate extra reads, and aborting potentially conflicting transactions early during the read phase before creating garbage. *Loosely synchronized clocks* maintains per-thread clocks for scalable timestamp allocation, using non-blocking one-sided synchronization. *Best-effort inlining* embeds a small version alongside the record metadata to avoid indirection without creating a contention point. *Rapid garbage collection* reclaims stale data frequently and concurrently to ensure small footprint that improves cache hit ratios. Although Cicada can exhibit high abort rates, *contention regulation* performs

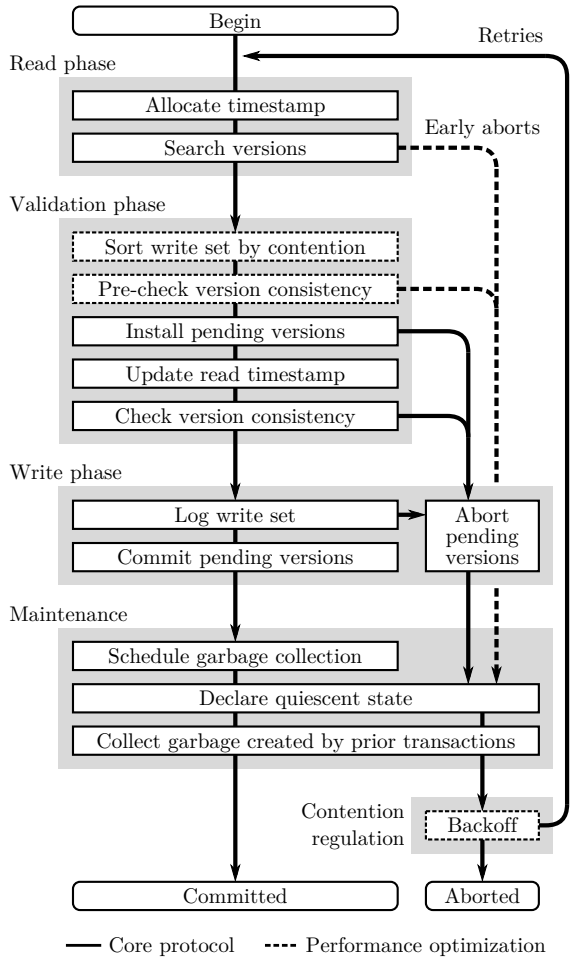


Figure 1: The workflow of Cicada.

a novel globally coordinated backoff scheme that limits the performance penalty of aborts and maximizes system-wide throughput.

Cicada exposes its functionality through a simple C++ interface. This interface can be used directly by an application or via a high-level wrapper that supports rich languages such as SQL via interpretation or native code generation [12, 44], possibly interacting with remote clients.

As depicted in Figure 1, Cicada's transaction processing adopts OCC's main phases of read, validation, and write, and adds maintenance and contention regulation.

In the read phase, a transaction begins with a newly allocated timestamp (§3.1) and runs the application logic that requests record access; Cicada searches for a particular version of the requested record by comparing the timestamp of the transaction and versions (§3.2). Because Cicada avoids in-place updates, record reads directly use the shared version of the record without creating a local version. Record writes and inserts use thread-local versions to store new record data throughout the read phase. Cicada keeps track of record reads, writes, and inserts as the read, write, and insert set of the transaction.

After the read phase, Cicada validates the consistency of the read and write sets and ensures serializability by using the transaction timestamp and version timestamps (§3.4) with performance optimizations (§3.5). The validation phase makes new changes in the write set reachable, which creates garbage that may not be collected immediately because other threads might be accessing the data.

In the write phase, if validation has succeeded, Cicada logs and commits the changes (§3.7). Otherwise, it deallocates any immediately reclaimable items.

Garbage is discovered and collected during maintenance (§3.8); Cicada frequently and concurrently reclaims stale multi-version data within tens of microseconds.

For an abort, Cicada performs randomized backoff using global coordinated maximum backoff time (§3.9), and retries the aborted transaction as needed.

3.1 Multi-Clock Timestamp Allocation

Cicada assigns a transaction a timestamp at its beginning. The timestamp is used to decide which version of records to use in the transaction. This timestamp also determines the serialized order of committed transactions in Cicada.

Cicada maintains *loosely synchronized software clocks* to generate timestamps. The multi-clock design eliminates a traditional performance bottleneck of timestamp allocation in MVCC on multi-core CPUs [63]. It also avoids relying on synchronized hardware clocks [4] because hardware virtualization can make hardware clocks unstable upon live migration, and tightly synchronized clocks may become expensive to implement on future many-core CPUs [63]. Instead, Cicada adopts distributed clocks that have been used in distributed transactions [1, 9, 40, 66].

Each worker thread holds a 64-bit local clock. A clock is incremented right before the thread allocates a timestamp. The increment amount is the locally measured elapsed time since the last clock increment.

Elapsed time is susceptible to the aforementioned noise as well. Our implementation uses the Time Stamp Counter [26] to measure the elapsed time on each core, which provides no guarantees on continuous clock increments and multi-socket clock synchronization. However, this measurement done within a local core remains scalable, so limiting the minimum and maximum clock increment—e.g., $(0, 1\text{ hour}]$—is sufficient to prevent excessive clock changes and help one-sided synchronization compensate for measurement error by making all clocks "catch up" with the fastest clock.

Timestamps are generated by combining three factors: the current local clock, a clock boost, and the thread ID. The clock boost is a per-thread quantity that is temporarily granted to a thread upon an abort; the thread ID serves as a tie-breaker. An adjusted clock is obtained by adding the current local clock and a clock boost, and making the sum larger than the last adjusted clock. A new 64-bit timestamp is generated by taking the low-order 56 bits of the adjusted clock and appending the 8-bit thread ID.

Each thread remembers two timestamps. (`thread.wts`) stores the timestamp generated by the above procedure. (`thread.rts`) stores `min_wts` minus 1, where `min_wts` is the minimum of (`thread.wts`) for all threads, updated by a leader thread periodically (§3.8). `min_rts` is also calculated similarly to (`thread.rts`), and is used for safe garbage collection. A read-write transaction uses (`thread.wts`) as its timestamp. A read-only transaction uses (`thread.rts`) instead, and does not track or validate the read set; it always sees consistent data because concurrent or future read-write transactions' timestamp is no earlier than `min_wts` and thus is later than (`thread.rts`).

Cicada tolerates loose synchronization of clocks. Its protocol (§3.4) does not assume that ordered timestamps are also ordered in physical time; it only requires that timestamps are unique and each thread's timestamps monotonically increase, which is accomplished by using the thread ID suffix and monotonically incrementing the clock.

However, loose synchronization can still harm performance. A thread using a too early timestamp is likely to fail to write to a

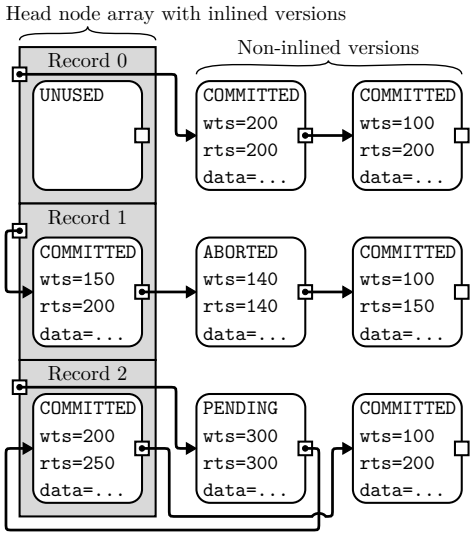


Figure 2: Multi-version data structures in Cicada.

contended record. To correct clock skew, Cicada uses long-lasting and short-lived mechanisms:

(1) *One-sided synchronization* opportunistically adjusts the local clock by peeking at a remote clock without blocking remote transaction processing. A worker thread periodically (every 100 µs in our implementation) chooses a remote thread in a round-robin way. It reads the remote clock and compensates for the small latency in the cache coherency protocol. If the compensated remote clock is faster than the current local clock, it becomes a new local clock.

This one-sided synchronization protocol is conservative. It can correct a slow local clock, but it cannot adjust a fast local clock because it cannot determine if the local clock is indeed fast or the remote clock was incremented a while ago. However, this scheme is effective because *all* threads *frequently* synchronize their clock with each other. Each thread has a chance of reading a fresh remote clock right after it has been incremented, and all slow clocks eventually catch up to fast clocks.

(2) *Temporary clock boosting* provides short-term correction. When an abort occurs due to a conflict, Cicada sets the clock boost to a fixed quantity that is larger than the possible residual skew after one-sided synchronization (1 µs in our implementation). The boost is reset to zero upon a commit.

Cicada may run for a long duration. Its 64-bit clocks and timestamps *wrap around*. To handle wraparounds, Cicada reinserts a version with a very early timestamp as a new version with the latest timestamp and identical record data.

The overhead of reinsertions is small. Reinsertions are infrequent: if a clock is incremented at 5 GHz, and is further truncated to 52 bits to accommodate up to 4096 threads (12 bits for the thread ID), a wraparound occurs every 10 days. Cicada only has to reinsert data with expiring timestamps incrementally over several days, excluding recently updated data, which makes the cost negligible in in-memory databases. Read-only transactions are unaffected as usual.

For logging, it can be desirable to have a total order of timestamps. A logger can use *extended timestamps* that include an *era*, which stores the number of wraparounds.

By default, Cicada's transactions do not guarantee *external consistency* [1, 19] across multiple threads. Cicada permits committing a transaction with an earlier timestamp after committing another transaction with a later timestamp when using multiple threads, similarly to data-driven timestamp-ordering schemes [64]. In other words, a serial schedule of transactions determined by their timestamps in Cicada may not match the commit order in physical time. Although such a mismatch is rarely a problem in practice because dependent transactions can force strict ordering by accessing the same record, it may be desirable to force stricter consistency even when access sets may be disjoint. For external consistency, Cicada can postpone notifying the application of a successful commit until `min_wts` becomes larger than the committed transaction's timestamp; this can add about 100 µs of extra latency, whose precise amount depends on how quickly clocks increment, but Cicada can still process other pending transactions during the delay. If only *causal consistency* [47] is required, Cicada can increment a local clock to make the timestamp for a new transaction larger than the maximum timestamp of its preceding transactions; the local clock adjustment is instant because Cicada's multi-clock does not require the increment of the local clock to be the same as that of the real-time clock and its one-sided synchronization corrects the clock drift.

3.2 Multi-Version Execution

Cicada implements relational tables using an expandable array of records. The array uses 2-level paging with a fixed page size (2 MiB in our implementation). Each record can be located by its array index (record ID).

Cicada organizes the versions of a record as a singly-linked list, as shown in Figure 2. Each version list starts with a *head* node stored in the array, followed by *version* nodes. The head may contain an *inlined version* (§3.3). A version contains (1) a write timestamp (`wts`) that is the timestamp of the transaction that has created this version; (2) a read timestamp (`rts`) that indicates the maximum timestamp of (possibly) committed transactions that read this version; and (3) the record data. The version also has (4) commit status (`status`) that indicates the validity of this version; and (5) allocation information including the NUMA node ID for NUMA-aware allocation and the version size (not shown). The version list is sorted by `wts` of the versions, forming latest-to-earliest order from the head.

A version becomes *reachable* once it has been *installed* into the version list by a writer during its validation phase (§3.4). All fields except `rts` and `status` are immutable. `rts` can be updated concurrently by any reader during its validation. `status` is initially `PENDING` when the version has been installed, and becomes either `COMMITTED` or `ABORTED` by the writer in its write phase. `UNUSED` indicates an inlined version that is not in use. Deleting a record installs a zero-length version whose status becomes `DELETED` when committed, which makes garbage collection to reclaim the record ID for future record allocation.

A transaction with a timestamp (`tx.ts`) accessing a record scans the version list of the record from latest to earliest order to find a version to use. It ignores any later version `v` if (`v.wts`) $>$ (`tx.ts`). Otherwise, it checks (`v.status`). For `PENDING`, it spin-waits until the status is changed. For `ABORTED`, it ignores this version and proceeds to an earlier version. For `COMMITTED`, it stops searching and chooses the version; we refer to this version as the version *visible* to the transaction.

The blocking behavior regarding `PENDING` versions is based on several observations. Blocking is short because a version remains `PENDING` for only a short period of time during validation. A pending version is likely to become `COMMITTED` instead of `ABORTED` because it has been installed only after the writer passes early consistency checks; thus, speculatively ignoring this pending version risks an abort. A pending version can still be aborted, creating cascaded aborts if it is speculatively used. Therefore, Cicada spin-waits, unlike prior MVCC designs making speculative decisions [35, 39].

During version search, as an important performance optimization, Cicada may perform an *early abort* of the current transaction that is

likely to be aborted. For a write using the visible version v, it checks whether (v.rts) $\leq$ (tx.ts) because the validation protocol will abort the transaction otherwise. For a read-modify-write (RMW), Cicada applies a *write-latest-version-only* rule. It aborts the current transaction if there is a version v′ such that (v′.wts) $>$ (tx.ts) and is either COMMITTED or PENDING because that later version likely aborts this transaction.

Cicada supports *read-own-writes*, which serves existing thread-local versions when a transaction accesses the same records again [45]. It provides consistency within a transaction by not losing earlier writes even if the application fails to reuse the pointer to the local version. Cicada finds earlier local writes using a lightweight thread-local hash table. A hash table entry is indexed by the table and record ID. The entry contains a pointer to the metadata that can locate a local version of the record. If the application can ensure the reuse of local versions and/or no multiple accesses for the same record, it can instruct Cicada to bypass the duplicate access check.

3.3 Best-Effort Inlining

Cicada uses *best-effort inlining* to reduce the indirection cost of multi-version execution while avoiding creating overhead and contention.

A transaction *attempts* to use the preallocated space for the inlined version in the head. It decides whether to use inlining when a write access to a record is requested; if the inlined version is UNUSED, it attempts to take ownership of the inlined version using an atomic compare-and-swap (CAS) operation on the status field to change it to PENDING. If the CAS succeeds, it uses the inlined version to store new record data; otherwise, it falls back to dynamic allocation of a non-inlined version. Inlining is applied only to small records (up to 216 bytes of record data inlined in our implementation—4 cachelines per head node including overhead) because inlining large records has diminishing returns, and large head nodes complicate memory management of the head node array.

Figure 2 illustrates inlining. Record 0 is not using inlining; the pointer of the head points to a non-inlined version. Record 1 uses inlining for the latest version, which can save a cache miss for transactions accessing the record. Record 2 shows that the inlined version does not have to be the latest version; the inlined version behaves the same as if it were non-inlined, except that it is simply marked as UNUSED when deallocated.

Cicada may *promote* a non-inlined version to make it inlined. The conditions are (1) a transaction reads a non-inlined version v as the visible version; (2) the version is early enough: (v.wts) $<$ min_rts; and (3) the inlined version is currently UNUSED. If so, Cicada automatically upgrades the read access to an RMW access, which attempts to write an inlined version with the same record data. The promotion may fail if there is a concurrent write to the record, but condition (2) makes this case rare. As a result, even if the latest version of a record is non-inlined, it eventually becomes inlined.

To avoid creating a contention point at the inlined version, promotion only optimizes infrequently- or never-changing read-intensive records. If a record is frequently written, promoting a version of such a record will incur unnecessary write overhead. If the record is never read, promotion does not provide a performance benefit; Cicada still can perform promotion for never-used records by scanning tables occasionally, simply to save space.

3.4 Serializable Multi-Version Validation

Cicada's validation protocol ensures that the execution of a committed transaction appears to occur atomically at the transaction's timestamp (tx.ts). As a result, any schedule for committed transactions in Cicada is equivalent to the serial schedule that executes the committed transactions in their timestamp order; Appendix A provides the formal proof. This multi-version protocol permits a transaction to freely read and write non-latest versions of records unless it violates serializability.

Validation has three required steps: (1) *Pending version installation*: It installs PENDING versions by inserting them into the version list of the records in the write set; the installation uses an atomic compare-and-swap operation to keep versions ordered by wts in the version list. (2) *Read timestamp update*: It updates, if necessary, the read timestamp of every version v in the read set to ensure (v.rts) $\geq$ (tx.ts) using an atomic compare-and-swap operation. (3) *Version consistency check:* It verifies that (a) every previously visible version v of the records in the read set is the currently visible version to the transaction, and (b) every currently visible version v of the records in the write set satisfies (v.rts) $\leq$ (tx.ts).

The pending version installation step blocks concurrent transactions that share the same visible version and have a higher timestamp than (tx.ts). If a concurrent transaction using the same visible version has a lower timestamp than (tx.ts), it may proceed to install its own pending version, aborting either this transaction itself or that concurrent transaction. Similar to early aborts, this step aborts the current transaction if the current visible version v fails to satisfy (v.rts) $\leq$ (tx.ts).

The read timestamp update step serves to notify other transactions that this version was read "as late as" (tx.ts).

The version consistency check step ensures (a) that no other transactions have written any new version that changes the visibility of the versions read by this transaction, and (b) that this transaction does not commit a too early version that would invalidate the consistency of already committed transactions relying on the constant visibility of the versions read at their timestamp. Note that the latter check uses the *currently* visible version to increase the concurrency of write-only (not RMW) operations that do not depend on the previous record data.

After successful validation, Cicada provides a customizable logger with the transaction timestamp and read, write, and insert set. If logging fails, the logger can abort the transaction; the logger can also ignore the logging failure and retry logging later if the application allows realizing the durability of a transaction after it has been committed [55].

Committing versions in the write phase simply changes their status from PENDING to COMMITTED.

A rollback upon an abort changes each pending version's status to ABORTED *only if* the version has been already installed. Otherwise, the pending version is deallocated for immediate reuse, without experiencing the ABA problem [42]. Similarly, any reserved record ID for a new record is freed and becomes available for reuse.

Note that the read timestamp update step is fast because its write is *conditional*. A read timestamp remains unchanged if it is already later than (tx.ts). Our 28-core testbed (§4.3) running multi-clock (§3.1) achieves 2.3 billion read timestamp updates per second on a single record. By comparison, it performs only 55 million unconditional atomic fetch-and-adds per second on a single record.

3.5 Optimizations for Efficient Validation

Cicada achieves efficient validation with small footprint using several performance optimizations:

(1) *Sorting the write set by contention* is performed before any validation steps to reduce the footprint of subsequent steps upon an abort. Cicada first calculates the approximate contention level of the records in the write set by using wts of their latest version (the first version in the version list); a larger wts of a record implies that this record is likely more contended than others. It sorts the

write set in descending order of approximate contention level, and the validation steps using the write set follow this order. *Partial sorting* is sufficient because the contention level is relevant only to highly contended records. Partial sorting costs $\mathcal{O}(n \log k)$ to sort top-k items in the write set with total n records ($k = 8$ in our implementation).

This sorting realizes *contention-aware validation*, allowing Cicada to detect conflicts early before installing many pending versions (which will become garbage upon an abort) and touching a large amount of memory. This optimization is impossible or costly in many OCC schemes: Silo, TicToc, FOEDUS, and MOCC must sort the entire write set in a globally consistent order (e.g., memory address or primary key of records) to avoid deadlocks during the locking step of their validation phase; such a design does not allow flexible locking order and costs $\mathcal{O}(n \log n)$ for full sorting. Cicada does not have this limitation because it has deadlock freedom: the pending version installation is prioritized by transaction timestamps, which avoids a dependency cycle.

(2) *Early version consistency check* is performed after sorting the write set. This is identical to the version consistency check of the core validation protocol, detecting most aborts before installing versions that would become garbage. This technique is inspired by TicToc's preemptive aborts [64].

These two optimizations can add unnecessary overhead under low contention because they do not improve the performance of uncontended workloads. Each thread adaptively omits both steps if the recent transactions have been committed (5 in a row in our implementation).

(3) *Incremental version search* reduces the cost of repeated version search. The pending version installation and version consistency check steps navigate version lists, which is redundant with the version search done in the read phase. Such repeated full version searches become particularly expensive for contended records because each search must traverse newly-inserted versions that are not in the local CPU cache. To reduce the cost of repeated search, the initial version search in the read phase remembers `later_version` whose `wts` is immediately later than (`tx.ts`). `later_version` is updated whenever a new version qualified as `later_version` is discovered during subsequent version search. Because the version list is sorted by `wts` in descending order, any new version that can abort the current transaction is guaranteed to appear after `later_version` in the version list. Thus, repeated version search can safely resume from `later_version`.

3.6 Indexing

Cicada decouples indexing from the main transactional storage. All indexes, including primary indexes, are separate data structures from their main table. An index stores 64-bit record IDs as values and does not store actual record data or raw pointers. Cicada's main indexing scheme, *multi-version indexes*, tackles two major problems: avoiding phantoms and reducing index contention.

Avoiding phantoms is required for serializability [12]. The problem arises when new records that a transaction could use appear after the transaction has been already committed. Phantoms may occur if the system only validates access to individual records and overlooks index data that affect index search (e.g., range scans).

Cicada's multi-version indexes use a variant of index node validation [55]. Each index node maintains both a write timestamp and a read timestamp in the same way that it maintains table records. For a range query, index nodes whose key range intersects with the query's key range are included in the read set of the transaction. For a point query for an absent key, the index node that could include the key is added to the read set. For a key insert or removal, modified index nodes are included in the read and write sets of the transaction. The validation of index node accesses precludes phantoms by detecting index node changes prior to the validation and preventing future index node changes using an earlier timestamp (§3.4).[1]

The multi-version indexes leverage Cicada's multi-version execution. They use plain Cicada tables as a memory pool; an index node is stored as a record, and pointers to index nodes and to indexed records use their record IDs. This scheme resembles concurrent index designs using software transactional memory [24, 51].

Low index contention is one of the beneficial side effects of unifying Cicada's transaction processing and index validation. Unlike many modern OCC designs that modify index data structures during the *read* phase of the transaction [32, 55], Cicada's multi-version indexes defer index updates until validating the transaction by keeping index node writes in thread-local memory; we reuse Cicada's read-own-writes support for table records to enable a transaction to read back its own index updates. Consequently, index update attempts by the transactions that are eventually aborted create little extra contention because they never modify global index data.

We retain support for *single-version indexes* backed by a conventional concurrent index data structure. They are more lightweight than multi-version indexes by making fewer memory accesses, but single-version indexes without deferred index updates often experience high index contention.

3.7 Durability and Recovery

Cicada supports durability and recovery using parallel value logging and checkpointing. We believe that Cicada can also support fast transaction-consistent checkpointing [49] by leveraging its multi-version design, which we leave as future work. We provide a design sketch of scalable durability and recovery. These techniques are well-known and documented [67]; here, we describe how one would implement them in the context of Cicada.

Redo logs are created by logger threads, each of which services one or more worker threads on the same NUMA node. After a worker validates a transaction, it sends a new log record to its logger; the record contains the write timestamp and data of newly installed versions in the transaction's write and insert set. The logger appends the record to its per-thread redo log file. The worker then marks the new versions `COMMITTED`, allowing validation of dependent transactions. For traditional block devices and networked replicas, loggers can amortize write latency by using group commit [22, 67]. On byte-addressable non-volatile memory, workers can exploit low-latency writes [7, 13, 25] and avoid communication overhead by directly performing logging without group commit and separate loggers.

Checkpoints are generated regularly by checkpointer threads in the background. Checkpointing virtually partitions each table. For each record in a partition, checkpointers store the latest committed version in per-thread checkpoint files. This process happens asynchronously without taking locks. For safe memory access, checkpointers participate in maintaining `min_rts` (§3.1); they frequently update (`thread.rts`) to avoid hindering `min_rts` increments.

Upon recovery, multiple recovery threads replay versions in redo logs and the last successful checkpoint in descending write timestamp order. A version is installed unless a version with a later write timestamp already exists in the memory for the same record; each record keeps only the latest version. Record deletion (§3.2)

[1] As a common performance optimization, Cicada does not validate read-only accesses to internal index nodes because validating leaf index nodes that are responsible for holding index key-value pairs suffices for phantom avoidance. Temporary structural inconsistencies are mitigated by using sibling pointers, similarly to $\mathrm{B}^{\mathrm{link}}$-tree [36].

is executed only after finishing all replays. If a DELETED version is applied when it is encountered, a non-DELETED version with an earlier timestamp can recreate the deleted record later in recovery, violating the durability of the record deletion. After completing replays, the system initializes clocks so that new timestamps are later than any replayed version's write timestamp.

Space management: Each per-thread redo log is chunked to a certain size (e.g., 1 MiB). The system records current min_wts in the beginning of checkpointing. Upon checkpoint creation, it purges old checkpoint and redo log files whose latest write timestamp is earlier than the recorded min_wts.

3.8 Rapid Garbage Collection

Cicada's garbage collection rapidly reclaims memory to maintain a small footprint. Garbage is collected frequently and concurrently in the cooperative maintenance.

Frequent garbage collection is important for Cicada to approach 1VCC's lightweightness. Garbage collection in prior databases is typically infrequent—e.g., every tens of ms. However, such infrequent garbage collection would inflate the working set significantly in MVCC. For example, suppose that it takes 80 ms to reclaim newly-created garbage (twice the 40-ms epoch length of Silo [55] using epoch-based reclamation (EBR) [18]). If each transaction creates 1 KiB of stale records (8×2 cachelines) and runs at 3.5 M transactions per second (a write-intensive, uniform YCSB scenario that writes eight 100 byte records per transaction in §4), the working set is 80 ms $\times$ 1 KiB $\times$ 3.5 M/s = 287 MB. This is far beyond today's CPU cache size; even if it fits in future caches, filling so much cache space with garbage lowers the cache efficiency.

For frequent garbage collection, Cicada uses a variant of EBR and quiescent-state-based reclamation (QSBR) [21]. It detects reclaimable versions using fine-grained timestamps, unlike other designs that use coarser-grained epochs for garbage collection [31, 38, 55, 64].

As the first step of the maintenance, a thread records the metadata of new versions committed by the last transaction. Committing a version v turns its earlier versions for the same record into garbage when the earlier versions are not visible to current and future transactions. For each v, the thread enqueues an item containing a pointer to v and a copy of (v.wts) into the local garbage collection queue.

The thread then declares a quiescent state by setting a per-thread flag regularly (every 10 µs in our implementation). If a leader thread sees that every flag has been set, it resets all flags and monotonically updates min_wts and min_rts, which store the global minimum of (thread.wts) and (thread.rts) for all threads.

After the quiescence, a thread inspects its local garbage collection queue. It checks the front queue item to see if (v.wts) $<$ min_rts. If so, it can safely reclaim the earlier versions of v for the same records because all current and future transactions use v or later versions. Otherwise, it stops checking; the check will fail anyway for the subsequent queue items because (v.wts) of each item monotonically increases within the queue.

Concurrent garbage collection allows multiple threads to reclaim the versions of different records as well as of the same record. Cicada maintains a small *per-record* data structure containing a *garbage collection lock* and the *minimum write timestamp* (record.min_wts), separate from the main record metadata (the head), which is prefetched while creating a new garbage collection item for the record. The thread performs garbage collection for a committed version v if (a) acquiring the garbage collection lock succeeds and (b) (v.wts) $>$ (record.min_wts). If condition (a) fails, this garbage collection item is discarded to avoid excessive garbage collection attempts on contended records. Condition (b) ensures that the pointer to v is not dangling. The thread detaches the rest of the version list from v, updates (record.min_wts), and releases the lock, making the record available for concurrent garbage collection. Finally, the thread returns the versions in the detached version list to its local memory pool.

3.9 Contention Regulation

OCC schemes are vulnerable to contention [43]. They execute transactions with minimal coordination between threads, which makes it hard to detect conflicts early.

Contention wastes local and global system resources. An aborted transaction consumes local CPU cycles for its execution. Even though OCC avoids making shared memory writes in the read phase, reading contended data can cause contention by invalidating another thread's cacheline. The memory management overhead of MVCC can further increase the cost of aborts. Although Cicada's early aborts and early version consistency check steps avoid installing versions until a transaction is likely to be committed, a few aborts reach the pending version installation step and creates garbage.

Backoff is a common mechanism to reduce contention. A thread sleeps for a certain duration after it aborts a transaction. The duration of sleep varies by backoff schemes. For example, the DBx1000 framework [11] chooses a random duration between 0 and the maximum backoff duration of 100 µs for an aborted transaction, and each thread allows at most 10 transactions' backoff to overlap.

Backoff schemes based on local information are often suboptimal. Although they can reduce interactions between threads, they tend to be overly conservative under high abort rates. Different workloads running on different systems have different optimal backoff time; some scenarios may favor the maximum backoff time of a few µs for the highest performance, even though the abort rate may remain high, while the others may benefit from longer backoff time and lower abort rates.

Cicada regulates contention with randomized backoff using *globally coordinated* maximum backoff time. A leader thread updates the maximum backoff time as part of its maintenance. The leader uses *hill climbing* to incrementally find the optimal maximum backoff time that maximizes the throughput of committed transactions. Each worker thread tracks the number of locally committed transactions. Periodically (5 ms in our implementation), the leader aggregates the number of committed transactions across all threads to obtain the throughput. It then calculates the changes of the throughput and the maximum backoff time between the second-to-last period and the last period. If the gradient (the throughput change divided by the maximum backoff time change) is positive, it increases the maximum backoff time by a fixed amount (0.5 µs in our implementation); if the gradient is negative, it decreases the maximum backoff time by the same amount. If the gradient is zero or undefined (no change in the maximum backoff time), it chooses a direction at random.

4. EVALUATION

This section compares the performance of Cicada and modern in-memory database designs, and examines the contribution of Cicada's components to its performance.

We show that (1) Cicada consistently achieves high performance under low and high contention, on read-intensive and write-intensive workloads, on small and large records, and for high-speed workloads; and (2) Cicada's components are crucial to its performance.

4.1 Compared Systems

We compare Cicada with seven serializable in-memory concurrency control schemes: Silo [55], TicToc [64], FOEDUS [32],

MOCC [57], 2PL no-wait [3], Hekaton [12, 35], and ERMIA SI+SSN [31]. *Silo* is an OCC-1V-in-place scheme using epoch-based group commit. *TicToc* is an OCC-1V-in-place scheme using flexible data-driven timestamp allocation to improve the performance of contended workloads over Silo. *FOEDUS* is an OCC-1V-in-place scheme designed to scale on many-core systems; *MOCC* mixes FOEDUS's OCC and locking to reduce the cost of access to contended records. *2PL no-wait* is two-phase locking that avoids deadlock by aborting the current transaction upon locking failure and uses 1VCC and in-place updates. *Hekaton* is an MVCC scheme that forms the basis of a production DBMS. *ERMIA* is an MVCC design that improves fairness among read-mostly and write-intensive transactions; we use its SI+SSN variant for serializability.

Implementation: We implement Cicada in C++. We use the reference implementation of Silo [52], TicToc [11], FOEDUS and MOCC [17], and ERMIA [14]. For Hekaton and 2PL no-wait, we use implementations available in DBx1000 [11, 63]. We also use DBx1000's reimplementation of Silo, denoted as *Silo′*. Silo′ is often faster than the reference Silo because Silo′ uses DBx1000's backoff scheme (§3.9), and its hash index for unordered index queries.

Optimization: We optimized the implementation of compared concurrency control schemes to improve their performance on our testbed. (1) For Silo, we changed its page pool to allocate hugepages directly via `mmap()` with `MAP_HUGETLB`. (2) For 1VCC schemes in DBx1000 (Silo′, TicToc, and 2PL), we collocated the record data and its concurrency control metadata on the same cacheline to reduce indirection as in the original Silo. The measured performance of compared schemes on our testbed is higher than their published results because of the more efficient memory access.

Missing features: The original DBx1000 lacks (1) an ordered index with phantom avoidance, (2) a NUMA-aware small object allocator, and (3) dynamic record creation and deletion. We modified DBx1000 to use Masstree [41] as an ordered index and ported Silo's phantom avoidance scheme.[2] We also apply Silo's RCU implementation to provide existing DBx1000 schemes with efficient object allocation and dynamic record management.

FOEDUS and MOCC require at least 4 threads on our testbed. We simply omit experiments using fewer threads on these systems.

4.2 Workloads

Experiments use two standard workloads: TPC-C and YCSB. *TPC-C* [54] is a benchmark for online transaction processing (OLTP) databases. TPC-C has a configurable number of *warehouses*. A worker thread mostly interacts with its local warehouses, but approximately 10% of `NewOrder` and 15% of `Payment` transactions access a remote warehouse. *YCSB* [8] is a benchmark commonly used for key-value store evaluation, and also adopted in transactional database evaluation by accessing multiple records in a single transaction. YCSB supports different workload characteristics via configurable parameters: the number of *requests per transaction*, the ratio of *reads* to all read and read-modify-write (RMW) requests, and the *skew* factor of the Zipf distribution for requested keys. Each read or read-modify-write request chooses a random key based on the key distribution, and reads or updates the corresponding record, performing a simple calculation with the field data. A scan picks a random key in the same way and reads a certain number of records using subsequent keys.

Implementation: Cicada shares benchmark implementations with existing concurrency control schemes in DBx1000 via a thin wrapper that allows Cicada to be used as a concurrency control scheme within DBx1000. That is, Cicada and existing schemes in DBx1000 share the benchmark code, but have separate data storage and transaction processing engines. Cicada's DBx1000 compatibility allows direct comparisons with existing DBx1000 schemes.

We implemented missing benchmarks for compared systems. (1) The original DBx1000 implemented *TPC-C-NP*, a subset of TPC-C that uses only `NewOrder` and `Payment` transactions; we implemented the full TPC-C for DBx1000. (2) Silo, FOEDUS, MOCC, and ERMIA only supported a uniform key distribution on YCSB; we ported DBx1000's YCSB benchmark for these implementations to support Zipf key distributions.

Optimization: The TPC-C implementation of existing systems employs different TPC-C-specific optimizations. (1) Silo uses read-only transactions for `OrderStatus` and `StockLevel`. (2) DBx1000 systems use hash indexes for the tables that do not require range queries. (3) Silo and ERMIA remember the last seen `NO_O_ID` in `Delivery` within the benchmark client and never reuse `NO_O_ID` even when a transaction fails to process a corresponding entry. (4) FOEDUS and MOCC accelerate transactions with index bypassing and vertical partitioning.

We retain these optimizations in existing TPC-C implementations to respect the original authors' performance tuning effort. Cicada's TPC-C uses only the first two optimizations: (1) Cicada provides low-latency read-only transactions at almost no cost; and (2) DBx1000-compatible schemes share the same TPC-C implementation, including index types. However, (3) Cicada avoids externally storing any user data that could reduce aborts; and (4) Cicada always uses indexing and no vertical partitioning to reflect the cost of index searches and contended record accesses more precisely. Consequently, the TPC-C implementation for Cicada and other DBx1000-compatible schemes is one of the least optimized implementations among compared systems, which helps maintain the generality of our experiments using this TPC-C implementation.

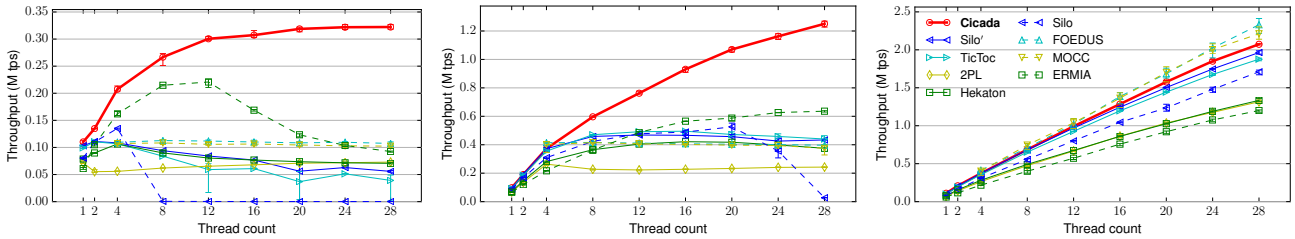
Configuration: Our experiments encompass diverse benchmark configurations that appear in literature [8, 50, 55, 58, 61, 64]. Contended TPC-C uses 1 warehouse and 4 warehouses. Uncontended TPC-C uses a number of warehouses equal to worker threads; 28-warehouse TPC-C uses about 10 GB of user data (i.e., excluding indexes and storage overheads) by the end of a run. The YCSB benchmark uses 10 M records of 100 bytes by default, which corresponds to 1 GB of user data; we include experiments using up to 2000 bytes per record, which is 20 GB of user data. The number of records in each scan is chosen uniformly at random in $[1, 100]$. We refer to the read ratio of 95% (5% RMW) as *read-intensive* and 50% (50% RMW) as *write-intensive*. We use combinations of read-/write-intensiveness, the Zipf skew of the key distribution, and the number of requests per transaction. Several benchmark configurations, such as TPC-C with many warehouses and YCSB with small skew, have working sets far larger than our testbed's CPU cache size, resulting in 42.8 GB/s of peak memory bandwidth use.

4.3 Testbed and Measurement

Experiments use a single server equipped with two Intel® Xeon® E5-2697 v3 CPUs (each with 14 cores and 35 MiB last level cache) and 128 GiB of DRAM. The experiments pin threads to different cores and use NUMA-aware memory allocation with hugepages.

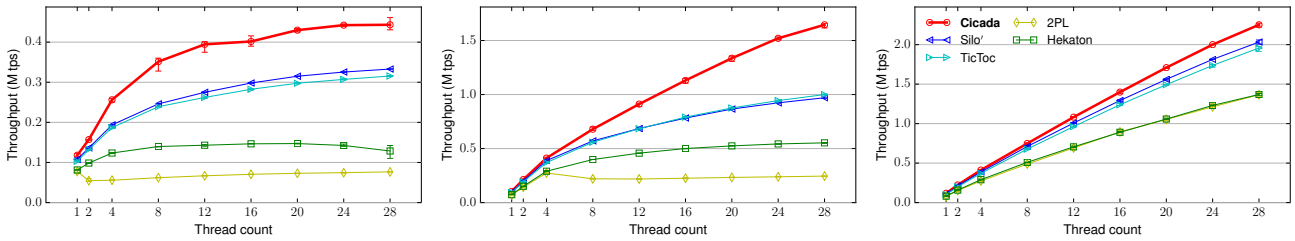
All processing is done in memory; persistent logging and remote clients are disabled. We expect that logging will lower the through-

[2] TicToc does not provide the detailed design and implementation of phantom avoidance [64]. TicToc's authors have indicated [62] that complete phantom avoidance would need to maintain a read timestamp on each index node in the same way as normal records in addition to performing Silo's phantom avoidance. Our experiments using TicToc without index read timestamps have faster execution speed and fewer aborts than with the full phantom avoidance, which is sufficient to provide the upper bound of TicToc's performance.



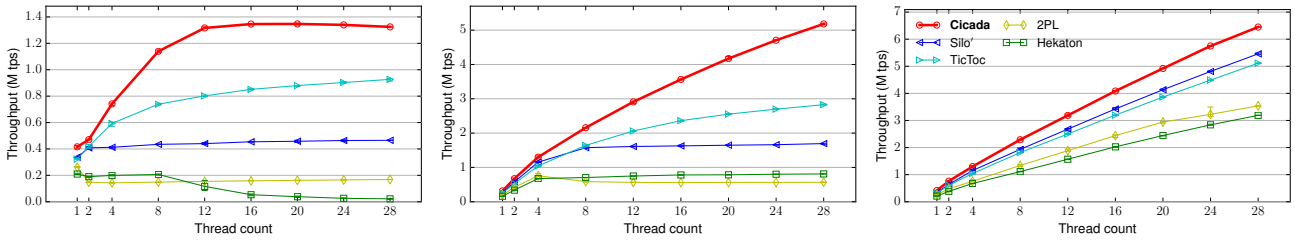
(a) 1 warehouse. **(b)** 4 warehouses. **(c)** Warehouse count = thread count.

Figure 3: TPC-C (full mix) throughput; with phantom avoidance.



(a) 1 warehouse. **(b)** 4 warehouses. **(c)** Warehouse count = thread count.

Figure 4: TPC-C (full mix) throughput; with deferred index updates and no phantom avoidance. DBx1000-compatible schemes only.



(a) 1 warehouse. **(b)** 4 warehouses. **(c)** Warehouse count = thread count.

Figure 5: TPC-C-NP (`NewOrder` and `Payment` only) throughput. DBx1000-compatible schemes only.

put of all schemes (e.g., by 17% [44]). Remote clients will add extra overhead; efficient I/O stacks [2, 28, 46] can minimize this overhead.

Each experiment is run 5 times. Every data point in graphs show an error bar that indicates the minimum and maximum, whose difference is typically small. The measurement begins after an initial ramp-up period. The experiments enforce fairness among transactions by retrying aborted transactions without starving transactions that make frequent conflicts (e.g., `Delivery` in TPC-C). For consistency across benchmarks, `Throughput` (the y-axis) measures the number of all *committed* transactions per second (tps). The throughput of committed `NewOrder` is 45% of the total committed throughput in TPC-C results.

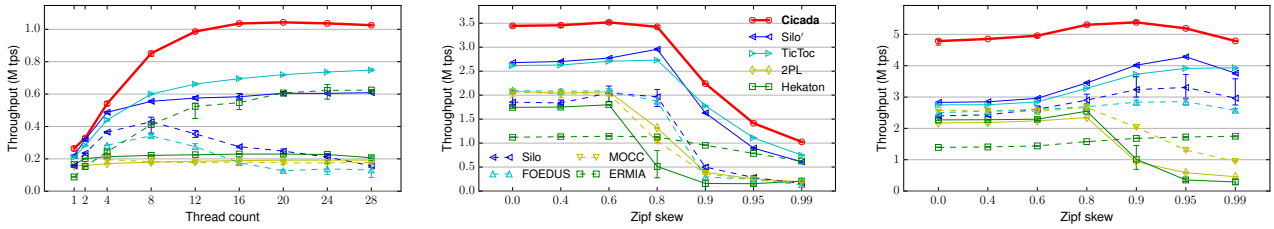
4.4 TPC-C Experiments

Under high contention: Figures 3a and 3b examine the effect of contention using TPC-C with 1 and 4 warehouses, respectively, using up to 28 threads. All schemes except Cicada scale poorly because they suffer frequent conflicts at record and index updates. Only ERMIA has comparable 1-warehouse performance to Cicada's using 12 threads, but its throughput collapses with more threads because ERMIA attempts excessive parallel data access to contended records; Cicada's contention regulation automatically avoids such excessive access. Cicada achieves up to 3X higher throughput than the next fastest design.

Note that Cicada's throughput continues to scale up to 12 threads for 1-warehouse TPC-C and 28 threads for 4-warehouse TPC-C. This may appear to violate TPC-C's inherent concurrency limit: a `Payment` transaction writes to a per-warehouse record in the `WAREHOUSE` table, which disallows concurrent `Payment` transactions greater than the warehouse count. However, Cicada benefits from more threads because (1) `WAREHOUSE` is modified by `Payment` only, and (2) `Payment` is only a single transaction type in TPC-C. Cicada's multi-version execution allows running other transaction types concurrently because they only read `WAREHOUSE` or do not access it at all. `Payment` is typically cheaper to process than `NewOrder`, `Deliver`, and `StockLevel`. A bottleneck by the concurrency limit of `WAREHOUSE` becomes apparent when Cicada uses many threads enough to have active `Payment` for every warehouse.

To scale under contention, however, a system must have efficient index updates even with high abort rates. Cicada's multi-version indexes keep any updates entirely in local memory before the transaction is committed, reducing the cost of index updates. In contrast, other systems suffer index contention; they perform early index updates before transaction validation, which causes their phantom avoidance to frequently abort concurrent transactions that accessed the modified nodes.

Figures 4a and 4b use the same warehouse configuration as above, with systems modified to reduce the cost of index updates. Systems defer index updates until all record accesses are validated and also omit phantom avoidance. Since this modification requires extensive code changes, we only compare DBx1000-compatible schemes. Cicada uses a single-version index based on Masstree without phantom avoidance as in other systems to show that Cicada



(a) Write-intensive, skew of 0.99. **(b)** Write-intensive, 28 threads. **(c)** Read-intensive, 28 threads.

Figure 6: YCSB throughput using 16 requests per transaction.

maintains high performance when using conventional single-version indexes. With cheaper index updates, most systems, notably Silo′ and TicToc, show better scalability, confirming that their high index update cost under frequent aborts is a major (but not the only) bottleneck. However, Cicada still outperforms compared systems. Cicada's multi-version execution and contention regulation make transaction processing efficient by reducing high-level (transaction) and low-level (memory access) contention. Cicada's 1-warehouse throughput is 33.2% higher than Silo′, reaching 443 k tps, which is comparable to that of a modern concurrency control scheme that is based on static analysis [58], even though Cicada uses no analysis.

Under low contention: Figure 3c shows the throughput of uncontended TPC-C using the same number of warehouses as threads. TPC-C is write-intensive, so MVCC creates and reclaims many versions, making this scenario favor 1VCC schemes. ERMIA and Hekaton's throughput is lower than others, due to MVCC overhead. In contrast, Cicada's 28-thread throughput is 2.07 M tps. This performance is up to 11.1% lower than FOEDUS and MOCC that use extra TPC-C-specific optimizations that Cicada and DBx1000 schemes do not implement. Using the same TPC-C implementation on DBx1000, Cicada is 5.54% faster than Silo′; Cicada's high speed under low contention is possible by best-effort inlining, rapid garbage collection, and no extra read cost, which enable efficient memory access despite maintaining multiple versions. In particular, Cicada's inlining is applied to its own hash index, reducing the number of random memory accesses required for each index search.

Figure 4c uses lightweight index updates under low contention. Cicada maintains up to 10.7% higher throughput than other systems and scales linearly with more core count.

TPC-C-NP: Figure 5 shows the TPC-C-NP (`NewOrder` and `Payment` only) performance on DBx1000. The result is largely similar to TPC-C with deferred index updates and no phantom avoidance in Figure 4, but Cicada shows an even larger performance gain because relatively short and prone-to-conflict transactions of TPC-C-NP make the benefit of Cicada's efficient transaction execution and conflict resolution more visible.

4.5 YCSB Experiments

Figure 6a compares the performance of fast OCC schemes under contended YCSB using 16 requests per transaction, 50% read/50% RMW, and Zipf skew of 0.99. Cicada's multi-version design and contention regulation help maintain its high performance: Cicada's throughput is higher than others when using the same thread count; with 28 threads, Cicada's throughput is 37.1% higher than TicToc's. Cicada's performance decreases only slightly when many threads execute transactions; FOEDUS and MOCC reach their peak throughput at 8 threads and experience performance degradations with more threads, indicating that their contention regulation is less effective than Cicada's.

Figure 6b plots the throughput of systems on write-intensive YCSB with 28 threads and variable skew to examine the effect of the write-intensiveness and the skew of the key popularity distribution separately. All schemes experience lower throughput with higher Zipf skew of the key popularity distribution because of increased conflicts on a few popular keys. Cicada maintains higher throughput than compared systems under both low and high skew. With low skew where the working set is large and has little access locality, Cicada's best-effort inlining keeps the number of random memory accesses low; in particular, Cicada's inlining makes multi-version hash indexes competitive with conventional hash indexes used in compared systems. With high skew, the benefit of best-effort inlining diminishes because the working set is small, causing fewer last level cache misses. However, Cicada's multi-version execution and contention regulation helps sustain high performance under contention.

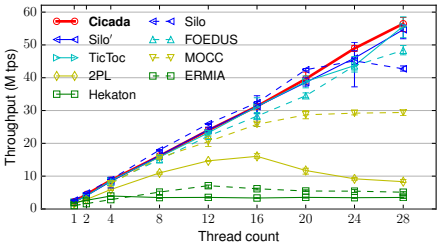


Figure 7: Read-intensive YCSB. 1 request per transaction, skew of 0.99.

Figure 6c shows the read-intensive YCSB performance with 28 threads and variable skew. Systems achieve slightly higher performance with moderate skew (e.g., 0.8) due to locality of memory reads, but they eventually show lower performance with high skew (e.g., 0.99) due to more frequent conflicts at record reads and updates. Cicada again maintains the highest throughput across all contention levels, achieving up to 69.2% higher throughput than the next best. The performance gap is larger than on write-intensive YCSB because random memory access time, which Cicada's best-effort inlining saves, constitutes for a larger fraction of execution time on this read-intensive YCSB. Cicada also does not suffer from the cost of extra reads that slow OCC-1V-in-place schemes including Silo and TicToc.

4.6 Factor Analysis

Multi-clock (§3.1): As shown on Figure 7, Cicada processes 56.5 M tps for a YCSB workload using 1 request per transaction, 95% read/5% RMW, Zipf skew of 0.99, and 28 threads. This result makes Cicada one of the fastest in-memory MVCC designs; other MVCC designs such as Hekaton that use centralized timestamping reach only 3.50–5.10 M tps for 28 threads. Modifying Cicada to use traditional timestamp allocation with atomic fetch-and-add on a shared counter drops throughput to 6.22 M tps, showing that multi-clock timestamp allocation is crucial for Cicada to handle high-speed transactions.

Multi-version execution (§3.2): Cicada maintains high throughput

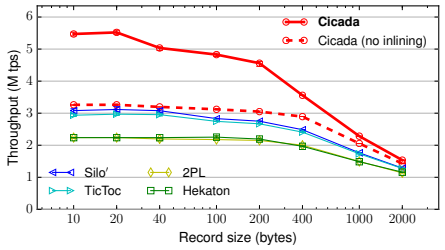


Figure 8: Read-intensive YCSB throughput with 16 requests per transaction, no skew, and varying record size. DBx1000-compatible schemes only.

No-wait	No-latest	No-sort	No-precheck
−13.2%	−4.9%	−12.7%	−9.5%

Table 2: Throughput differences when disabling a performance optimization on contended YCSB.

under contended workloads using multi-version execution. The fine-grained versioning also ensures that Cicada's snapshots have low staleness. For read-only TPC-C transactions (`OrderStatus` and `StockLevel`), Cicada's average and 99.9-th percentile staleness is 117 µs and 724 µs. This staleness is orders of magnitude smaller than that of previous epoch-based snapshots, which reaches tens to hundreds of ms; for example, Silo has 0.5 second staleness on average because a snapshot is taken every second [55].

Best-effort inlining (§3.3): Figure 8 plots the throughput of read-intensive YCSB using 16 requests per transaction, a uniform key popularity, and 28 threads. Best-effort inlining improves throughput under this uncontended workload especially for small record size because the indirection accounts for a large fraction of the processing cost for small records, but the benefit gradually diminishes with large records. However, even for large records, inlining boosts performance because the node size of Cicada's multi-version hash index is 24 bytes, which is within the size limit of inlining (216 bytes in the implementation).

Best-effort inlining also accelerates scans. On read-intensive YCSB with skew of 0.99 and 28 threads, scans are executed as read-only transactions. Each RMW updates a single record. Cicada scans 356 M records per second (rps) with inlining, but only 203 M rps without inlining.

Performance optimizations (§3.2, §3.5): Table 2 summarizes the throughput difference on contended YCSB using 16 requests per transaction, 50% read/50% RMW, skew of 0.99, and 28 threads, when omitting each of the performance optimizations. `No-wait` speculatively ignores `PENDING` versions as in Hekaton. `No-latest` disables the write-latest-version-only rule. `No-sort` skips the contention-aware write set sorting. `No-precheck` skips the early version consistency check. By omitting any single optimization, Cicada achieves 4.9–13.2% lower throughput.

Rapid garbage collection (§3.8): Figure 9 shows the TPC-C throughput of Cicada using 28 threads with different minimum intervals of quiescence. Performing garbage collection only every 100 ms drops throughput by 36.0% for 28-warehouse TPC-C, 27.2% for 4-warehouse TPC-C, and 2.2% for 1-warehouse TPC-C vs. the default 10 µs garbage collection interval; rapid garbage collection is most beneficial for high-throughput workloads that create more garbage and inflate the working set. The infrequent garbage collection causes high space overhead (the total version count / the total record count − 100%) of 9.89%. In contrast, rapid garbage collection in Cicada limits the space overhead to 1.83%.

Contention regulation (§3.9): Figure 10 illustrates the effectiveness of Cicada's contention regulation, using TPC-C with 4 warehouses (top), TPC-C-NP with 4 warehouses (middle), and write-intensive YCSB with 1 request per transaction and skew of 0.99

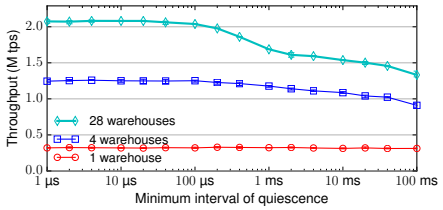


Figure 9: TPC-C throughput with different minimum intervals of quiescence for garbage collection.

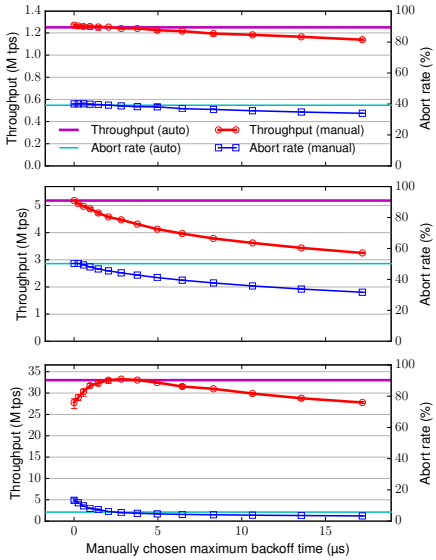
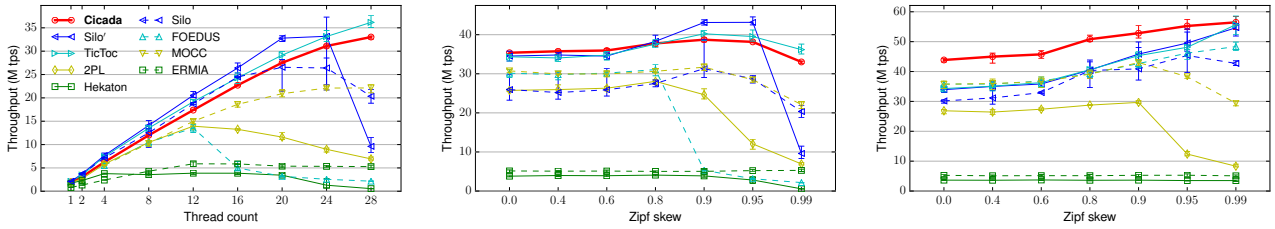


Figure 10: Throughput and abort rate of contended TPC-C (top), TPC-C-NP (middle), and YCSB (bottom) using contention regulation (`auto`) and fixed backoff settings (`manual`).

(bottom). All experiments use 28 threads. Horizontal lines plot the throughput and abort time (the ratio of the time spent on aborted transaction processing and backoff to the total processing time) when using maximum backoff time globally coordinated by contention regulation. Curves with markers use manually chosen maximum backoff time instead. The graphs show that the optimal maximum backoff time is workload dependent, and it can be desirable to use small (often zero) maximum backoff time for the best throughput even though the abort rate may stay high. Cicada achieves nearly optimal performance under both workloads. Our experiments also showed the unique maximum throughput without local maxima, making contention regulation reliable. This result suggests that high abort rates are not necessarily harmful; inexpensive aborts can make high abort rates acceptable and achieve high throughput.

5. CONCLUSION

Cicada is a transactional in-memory database that uses a variety of innovations to achieve high performance under diverse workloads. It borrows from distributed system designs to unshackle its transaction ordering from being bottlenecked by a centralized clock; uses multi-version execution to provide the benefits of optimistic concurrency control without its major drawbacks; uses best-effort inlining and rapid garbage collection to blend the high concurrency of multi-version concurrency control with the low overhead of single-version designs; and uses a novel low-overhead global backoff coordination scheme to optimize its throughput under contention. Using these techniques, Cicada outperforms prior state-of-the-art in-memory databases under most workloads, and roughly matches their performance on the remainder.



(a) Write-intensive, skew of 0.99. **(b)** Write-intensive, 28 threads. **(c)** Read-intensive, 28 threads.

Figure 11: YCSB throughput using 1 request per transaction. Included as part of Appendix B.

APPENDIX

A. PROOF OF SERIALIZABILITY

We prove the serializability of Cicada's protocol. For brevity, we assume no timestamp wraparounds by using extended timestamps (§3.1); using either timestamp format makes no difference to the schedule of transactions in Cicada. We consider read-write transactions and omit proofs regarding record inserts and deletes.

Definition 1. The *visible version* of a record for a transaction is a version whose write timestamp is the latest (highest) among all committed versions of the record and is earlier (smaller) than the transaction's timestamp.

LEMMA 1. *All transactions have a unique timestamp.*

PROOF. Each thread monotonically increases its local clock and never reuses the same clock for timestamp allocation. Timestamps have the thread ID as a unique suffix, which guarantees that all timestamps are unique. □

LEMMA 2. *A version of a record that is read by a committed transaction is the visible version of the record in the serial schedule.*

PROOF. Let a committed transaction be $\texttt{tx}$, and the committed version of a record read by $\texttt{tx}$ be $\texttt{v}$.

Assume that there exists a committed transaction $\texttt{tx}'$ that commits $\texttt{v}'$ such that $(\texttt{v.wts}) < (\texttt{v}'\texttt{.wts}) < (\texttt{tx.ts})$. $\texttt{v}'$ instead of $\texttt{v}$ would become the visible version to $\texttt{tx}$.

If $\texttt{tx}'$ has installed $\texttt{v}'$ before $\texttt{tx}$ passes the version consistency step, $\texttt{tx}$ is blocked in the version consistency check step while $\texttt{v}'$ is PENDING. If $\texttt{v}'$ becomes COMMITTED, $\texttt{tx}$ sees $\texttt{v}'$ as the currently visible version and is aborted, which is impossible because $\texttt{tx}$ is committed. If $\texttt{v}'$ becomes ABORTED, it is a contradiction to the assumption that $\texttt{tx}'$ is committed. Thus, $\texttt{tx}'$ must install $\texttt{v}'$ after $\texttt{tx}$ passes the version consistency check step.

Recall the order of validation steps. $\texttt{tx}$ performs the read timestamp update step before the version consistency check step. The read timestamp update step for $\texttt{tx}$ ensures $(\texttt{tx.rts}) \leq (\texttt{v.rts})$. $\texttt{tx}'$ performs the version consistency check step after installing $\texttt{v}'$.

(Case 1) Suppose $\texttt{tx}'$ reads $\texttt{v}$. $\texttt{tx}'$ observes $(\texttt{tx}'\texttt{.ts}) = (\texttt{v}'\texttt{.wts}) < (\texttt{v.rts})$. Thus, $\texttt{tx}'$ is aborted by failing the version consistency check step, which is a contradiction to the assumption that $\texttt{tx}'$ is committed.

(Case 2) Suppose $\texttt{tx}'$ reads a committed version $\texttt{v}''$ that is earlier than $\texttt{v}$. $\texttt{tx}$ already passed the version consistency step by observing $\texttt{v}$, so $\texttt{tx}'$ also observes $\texttt{v}$, which makes $\texttt{tx}'$ fail the version consistency check step because $\texttt{v}$, not $\texttt{v}''$, is the current visible version. This again makes a contraction to the assumption that $\texttt{tx}'$ is committed.

(Case 3) Suppose $\texttt{tx}'$ reads a committed version $\texttt{v}''$ that is later than $\texttt{v}$. We substitute $\texttt{tx}$ and $\texttt{tx}'$ with $\texttt{tx}'$ and $\texttt{tx}''$. Reapplying this lemma reaches Case 1 or Case 2 in finite steps, precluding the existence of $\texttt{v}''$ if $\texttt{tx}'$ is committed. Consequently, this makes a contradiction to the assumption that $\texttt{tx}'$ is committed.

Therefore, no such $\texttt{tx}'$ exists. $\texttt{v}$ is the visible version to $\texttt{tx}$. □

THEOREM 1. *Any schedule for committed transactions in Cicada is equivalent to the serial schedule that executes the committed transactions in their timestamp order.*

PROOF. A committed transaction creates at most one version for a record. By Lemma 1, each version's write timestamp following the transaction's timestamp is unique within a record. With Lemma 2, every committed transaction reads the uniquely determined visible version of the record as it would in the serial schedule. Therefore, any schedule of committed transactions in Cicada is equivalent to the serial schedule. □

B. ADDITIONAL EVALUATION

Figure 11 plots the YCSB throughput using similar configurations as Figure 6, but with 1 request per transaction. By making each transaction tiny, we examine the worst-case overhead of transaction initialization and finalization. As an MVCC design, Cicada has more out-of-transaction processing, such as timestamp allocation and garbage collection, than 1VCC designs such as Silo and TicToc. The overhead is most apparent on write-intensive YCSB as shown in Figures 11a and 11b because Cicada must reclaim many newly-created versions. Furthermore, Cicada loses its advantage of avoiding extra read costs that is more significant under read-intensive workloads. However, the overhead is not excessively high; Cicada achieves 8.7% lower throughput than TicToc on the write-intensive YCSB with 28 threads and skew of 0.99. Cicada's read-intensive YCSB performance is similar to or higher than Silo′ and TicToc.

Cicada supports an optimization that we do not use in any of our experiments because no other systems implement it: reading a single record without using a transaction. Cicada allows directly locating and reading versions. Because the record data is always consistent in Cicada, it is unnecessary to lock the shared version or create a local version. This optimization can accelerate workloads with many single-record reads (e.g., TATP [53]), mitigating the initialization and finalization overhead with tiny transactions.

C. ACKNOWLEDGMENTS

This work was supported by funding from National Science Foundation under awards CNS-1345305 and CCF-1535821, and Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS). We thank Sol Boucher, Anuj Kalia, Andrew Pavlo, Xiangyao Yu, and anonymous reviewers for their feedback. We appreciate Tianzheng Wang and Kangnyeon Kim for helping us run their systems.

D. REFERENCES

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[2] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[3] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), June 1981.

[4] S. Boyd-Wickizer. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, 2013.

[5] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[6] M. J. Carey and M. Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases*, 1984.

[7] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[8] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX OSDI*. USENIX, 2012.

[10] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[11] DBx1000: A single node OLTP database management system. https://github.com/yxymit/DBx1000, 2016.

[12] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013.

[13] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[14] ERMIA: CC for modern main-memory OLTP systems. https://github.com/ermia-db/ermia, 2016.

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

[16] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endowment*, 8(11), July 2015.

[17] FOEDUS: Fast optimistic engine for data unification services. https://github.com/hkimura/foedus_code, 2016.

[18] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2003.

[19] D. K. Gifford. Information storage in a decentralized computer system. Technical Report CSL-81-8, Xerox Palo Alto Research Centers, 1983.

[20] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[21] T. E. Hart. *Comparative Performance of Memory Reclamation Strategies for Lock-Free and Concurrently-Readable Data Structures*. PhD thesis, University of Toronto, 2005.

[22] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, 1989.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[24] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[25] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *Proc. VLDB Endowment*, 8(4), Dec. 2014.

[26] Intel 64 and IA-32 Architectures Developer's Manual: Vol. 2B. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf, 2016.

[27] Intel Xeon Processor E5-2600 v4 Product Family. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e5-brief.pdf, 2016.

[28] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[29] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endowment*, 1(2), Aug. 2008.

[30] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, 2011.

[31] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[32] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[33] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[34] M.-Y. Lai and W. K. Wilkinson. Distributed transaction management in JASMIN. In *Proceedings of the 10th International Conference on Very Large Data Bases*, 1984.

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

[36] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, Dec. 1981.

[37] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*, 2014.

[38] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in Deuteronomy. Conference on Innovative Data Systems Research (CIDR 2015), 2015.

[39] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in Deuteronomy. *Proc. VLDB Endowment*, 8(13), Sept. 2015.

[40] B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, 1991.

[41] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.

[42] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, 2002.

[43] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[44] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[45] C. U. Orji, L. Lilien, and J. Hyziak. A performance analysis of an optimistic and a basic timestamp-ordering concurrency control algorithms for centralized database systems. In *Proceedings of the 4th International Conference on Data Engineering*, Feb. 1988.

[46] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[47] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malô, France, Oct. 1997.

[48] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endowment*, 5(12), Aug. 2012.

[49] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[50] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core OLTP under high contention. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[51] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, 1995.

[52] Silo: Multicore in-memory storage engine. https://github.com/stephentu/silo, 2013.

[53] Telecommunication application transaction processing (TATP) benchmark. http://tatpbenchmark.sourceforge.net/, 2011.

[54] TPC benchmark C. http://www.tpc.org/tpcc/, 2010.

[55] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[56] VoltDB, the NewSQL database for high velocity applications. https://voltdb.com/, 2016.

[57] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endowment*, 10(2), Oct. 2016.

[58] Z. Wang, Y. Cui, H. Yi, S. Mu, H. Chen, and J. Li. Unleashing parallelism in multi-core databases via dependency tracking. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[59] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[60] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[61] Y. Wu and K.-L. Tan. Scalable in-memory transaction processing with HTM. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, June 2016.

[62] Private communication with Xiangyao Yu, Aug. 2016.

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

[64] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. TicToc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[65] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang. BCC: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proc. VLDB Endowment*, 9(6), Jan. 2016.

[66] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[67] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Oct. 2014.