TicToc: Time Traveling Optimistic Concurrency Control

Xiangyao Yu
CSAIL MIT
yxy@csail.mit.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Daniel Sanchez
CSAIL MIT
sanchez@csail.mit.edu

Srinivas Devadas
CSAIL MIT
devadas@csail.mit.edu

ABSTRACT

Concurrency control for on-line transaction processing (OLTP) database management systems (DBMSs) is a nasty game. Achieving higher performance on emerging many-core systems is difficult. Previous research has shown that timestamp management is the key scalability bottleneck in concurrency control algorithms. This prevents the system from scaling to large numbers of cores.

In this paper we present TicToc, a new optimistic concurrency control algorithm that avoids the scalability and concurrency bottlenecks of prior T/O schemes. TicToc relies on a novel and provably correct data-driven timestamp management protocol. Instead of assigning timestamps to transactions, this protocol assigns read and write timestamps to data items and uses them to lazily compute a valid commit timestamp for each transaction. TicToc removes the need for centralized timestamp allocation, and commits transactions that would be aborted by conventional T/O schemes. We implemented TicToc along with four other concurrency control algorithms in an in-memory, shared-everything OLTP DBMS and compared their performance on different workloads. Our results show that TicToc achieves up to 92% better throughput while reducing the abort rate by 3.3× over these previous algorithms.

1. INTRODUCTION

Multi-core systems are now pervasive, as parallelism has become the main approach to increase system performance. Systems with tens to hundreds of CPU cores are already on the market [2, 12], and thousand-core chips will be available in the near future [10]. Conventional DBMSs, however, still scale poorly beyond a few cores. The key bottleneck of on-line transaction processing (OLTP) DBMSs is their concurrency control algorithm. Prior work has shown that common concurrency control algorithms suffer from both fundamental and artificial scalability bottlenecks [34, 37, 29]. Although recent work ameliorates some artificial bottlenecks [20, 21, 24, 28, 32, 35], fundamental bottlenecks remain.

Ideally, concurrency control schemes should restrict the inherent parallelism in transactional workloads as little as possible, while incurring small management overhead that scales well with the number of cores. Most of the recently-proposed concurrency control schemes are based on *timestamp ordering* (T/O) [5]. T/O schemes assign a unique, monotonically-increasing timestamp to each transaction. The DBMS then uses these timestamps to process conflicting operations in the proper order. The two most common variants of T/O are multi-version concurrency control (MVCC) [30] and optimistic concurrency control (OCC) [22].

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2882935

T/O algorithms are popular because they allow significant concurrency, but suffer from a fundamental scalability bottleneck: timestamp allocation. Using a shared-memory counter to produce timestamps limits T/O schemes to a few million transactions per second, orders of magnitude lower than modern OLTP workload requirements [35, 37]. Prior work has proposed hardware and software techniques to increase timestamp allocation throughput, but both approaches have serious limitations. On the hardware side, centralized asynchronous counters [37], remote atomic memory operations [2, 18], and fully-synchronized clocks [19] alleviate the timestamp allocation bottleneck, but they are challenging to implement and are not available in current systems. On the software side, coarse-grained timestamp epochs with group commit [35] reduces the frequency of timestamp allocations, but still limits concurrency in common scenarios as we show later.

In this paper we present **TicToc**, a new concurrency control algorithm that achieves higher concurrency than state-of-the-art T/O schemes and completely eliminates the timestamp allocation bottleneck. The key contribution of TicToc is a technique that we call *data-driven timestamp management*: instead of assigning timestamps to each transaction independently of the data it accesses, TicToc embeds the necessary timestamp information in each tuple to enable each transaction to compute a valid commit timestamp after it has run, right before it commits. This approach has two benefits. First, each transaction infers its timestamp from metadata associated to each tuple it reads or writes. No centralized timestamp allocator exists, and concurrent transactions accessing disjoint data do not communicate, eliminating the timestamp allocation bottleneck. Second, by determining timestamps *lazily* at commit time, TicToc finds a logical-time order that enforces serializability even among transactions that overlap in physical time and would cause aborts in other T/O-based protocols. In essence, TicToc allows commit timestamps to move forward in time to uncover more concurrency than existing schemes without violating serializability.

We present a high-performance, OCC-based implementation of TicToc, and prove that it enforces serializability. We also design several optimizations that further improve TicToc's scalability. Finally, we compare TicToc with four other modern concurrency control schemes in the **DBx1000** main-memory DBMS [1], using two different OLTP workloads on a multi-socket, 40-core system. Our results show that TicToc achieves up to 92% higher throughput than prior algorithms under a variety of workload conditions.

2. BACKGROUND

OLTP DBMSs support the part of an application that interacts with end users. End users send requests to the application to perform some function (e.g., post a comment, purchase an item). The application processes these requests and then executes transactions in the DBMS to read or write to the database. A *transaction* in the context of one of these systems is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher-level function [15].

A *concurrency control scheme* is the protocol that a DBMS uses to interleave the operations of simultaneous transactions in such a way to provide the illusion that each transaction is running exclusively on the database. There are two classes of concurrency control algorithms [5]: two-phase locking and timestamp ordering.

Two-phase locking (2PL) was the first method proven to ensure correct execution of concurrent DBMS transactions [6, 13]. Under this scheme, transactions have to acquire locks for a particular element in the database before they are allowed to execute a read or write operation on that element. 2PL is considered a pessimistic approach because it assumes that transactions will conflict and thus they need to acquire locks to avoid this problem. If a transaction is unable to acquire a lock for an element, then it is forced to wait until the lock becomes available. If this waiting is uncontrolled, then the DBMS can incur deadlocks. Thus, a major difference among 2PL variants is their deadlock-avoidance strategy.

Timestamp ordering (T/O) concurrency control schemes generate a serialization order of transactions a priori based on monotonically increasing timestamps. The DBMS uses these timestamps to process conflicting operations in the proper order (e.g., read and write operations on the same element, or two separate write operations on the same element) [5].

Although 2PL has been widely adopted in traditional DBMSs (e.g., IBM DB2, Microsoft SQL Server, MySQL), the contention introduced by locks severely hurts performance in today's many-core systems [37]. Almost every OLTP DBMS released in the last decade that we are aware of, including all but a few of the NewSQL DBMSs [3], uses a T/O-based concurrency control scheme.

We next discuss T/O algorithms in further detail to understand their key bottlenecks. We then discuss state-of-the-art algorithms in Section 2.2. This will provide the necessary background for our presentation of the TicToc algorithm in Section 3.

2.1 Timestamp Allocation

The execution of transactions must obey certain ordering constraints. In the strictest isolation level (i.e., serializable), the execution schedule must be equivalent to a schedule where all the transactions are executed sequentially. In a T/O-based concurrency control algorithm, this serial order is expressed using timestamps. Each transaction is assigned a unique and monotonically increasing timestamp as the serial order that is used for conflict detection. Multi-versioning (MVCC) and optimistic (OCC) concurrency control algorithms are both timestamp based.

In traditional T/O-based algorithms, a centralized timestamp allocator assigns a unique timestamp to each transaction. A common way to implement the allocator is through an *atomic add* instruction that increments a global counter for each new transaction. This approach, however, is only able to generate less than 5 million instructions per second on a modern multi-core system due to the long latency incurred by the CPU's cache coherence protocol [11, 35]. As such, most state-of-the-art T/O-based algorithms suffer from the timestamp allocation bottleneck [37].

Hardware support can alleviate the timestamp allocation bottleneck. For example, Tilera processors support remote atomic operations [14] that can increment the timestamp counter without incurring extra cache coherence traffic [2, 18]. In practice, this achieves 100 million timestamps per second [11]. A second option is to add a special hardware timestamp counter on the multi-core chip (which does not exist in any CPUs today). This approach is able to allocate one billion timestamps per second according to simulations [37]. The third option is to produce timestamps using a clock that is synchronized across all cores. In fact, small-scale Intel systems have a synchronized clock [19]. However, fully-synchronized clocks are challenging to maintain in large-scale, multi-socket systems, which use different clock sources that drift over time. Keeping these clocks completely synchronized would require either an unreasonable amount of communication for adjustments, or a global clock source with an impractically low frequency.

All the hardware solutions described here require some hardware support that either does not exist in any CPUs or only exists in a subset of the CPUs today. Even for those CPU architectures that do have this support, it is not guaranteed that the support will still exist in the future considering that its cost increases with the number of cores. Moreover, even if this hardware support exists in all processors, a T/O-based concurrency control algorithm may still achieve suboptimal performance. In these algorithms, timestamps are statically assigned to transactions and the assignment does not depend on the data access pattern. At runtime, the actual dependency between two transactions may not agree with the assigned timestamp order and thus transactions may be unnecessarily aborted, which hurts performance (see example in Section 3.1).

2.2 Optimistic Concurrency Control

The original OCC algorithm was proposed in 1981 [22], but it has only recently been adopted in high-performance OLTP DBMSs. By contrast, MVCC, the other T/O-based algorithm, has been used in DBMSs for several decades (e.g., Oracle, Postgres).

Under OCC, the DBMS executes a transaction in three phases: *read*, *validation*, and *write*. In the *read phase*, the transaction performs read and write operations to tuples without blocking. The DBMS maintains a separate private workspace for each transaction that contains its read set and write set. All of a transaction's modifications are written to this workspace and are only visible to itself. When the transaction finishes execution, it enters the *validation phase*, where the OCC scheme checks whether the transaction conflicts with any other active transaction. If there are no conflicts, the transaction enters the *write phase* where the DBMS propagates the changes in the transaction's write set to the database and makes them visible to other transactions.

Many algorithms have been proposed to refine and improve the original OCC algorithm [8, 17, 26, 31]. The first OCC derivatives from the 1980s dealt with improving transaction validation for single-threaded systems with limited memory [17, 31].

One OCC-based protocol that bears some similarity to our proposed TicToc algorithm is the *dynamic timestamp allocation* (DTA) approach [4, 7]. Instead of assigning a specific timestamp to a transaction, DTA allows the transaction to take a range of timestamps and adjusts the commit timestamp during the validation phase. This approach was revisited in the 1990s for "real-time" DBMSs to give higher priority in the validation phase to certain transactions [23, 25]. Similar to TicToc, DTA-based OCC can reduce the number of aborts compared to a traditional OCC. The key difference is that DTA only assigns timestamps to transactions and not to tuples. As a result, DTA-based OCC requires the DBMS to use a global critical section for coordination among concurrently-validating transactions. This is a major scalability bottleneck on multi-core and multi-socket systems [37].

Silo is a state-of-the-art OCC algorithm that achieves high throughput by avoiding bottlenecks caused by global locks or timestamp allocation [35]. In Silo, a global timestamp (called an *epoch*) is allocated at coarse time granularity (every 40 ms) and is used to indicate the serial order among transactions. Within an epoch, transaction IDs are used to identify data versions as well as to detect conflicts. These IDs, however, do not reflect the relative order among transactions. This is because they only capture read-after-write dependencies, but not write-after-read dependencies (anti-dependencies). Silo is still able to enforce serializable execution, but only able to exploit a limited amount of parallelism.

To tackle the above issues, we now present a new OCC variant that uses decentralized data-driven timestamp management.

3. THE TICTOC ALGORITHM

Like other T/O-based algorithms, TicToc uses timestamps to indicate the serial order of the transactions. But unlike these previous approaches, it does not assign timestamps to transactions using a centralized allocator. Instead, a transaction's timestamp is calculated *lazily* at its commit time in a *distributed* manner based on the tuples it accesses. There are two key advantages of this timestamp management policy. First, its distributed nature avoids all of the bottlenecks inherent in timestamp allocation [37], making the algorithm highly scalable. Second, laziness makes it possible for the DBMS to exploit more parallelism in the workload, thereby reducing aborts and improving performance.

3.1 Lazy Timestamp Management

To see why lazy timestamp management can reduce conflicts and improve performance, we consider the following example involving two concurrent transactions, A and B, and two tuples, x and y. The transactions invoke the following sequence of operations:

1. A read(x)
2. B write(x)
3. B commits
4. A write(y)

This interleaving of operations does not violate serializability because transaction A can be ordered before B in the serial order. But A cannot commit after B in the serial order because the version of x read by A has already been modified by B.

Traditional OCC algorithms assign timestamps to transactions statically, essentially agreeing on a fixed sequential schedule for concurrent transactions. This eases conflict detection, but limits concurrency. In this example, if transaction A is assigned a lower timestamp than transaction B, then A can commit since the interleaving of operations is consistent with timestamp order. However, if transaction A is assigned a higher timestamp than transaction B, A must eventually abort since committing it would violate the schedule imposed by timestamp order.

By contrast, TicToc does not allocate timestamps statically, so it does not restrict the set of potential orderings. It instead calculates the timestamp of each transaction lazily at the transaction's commit time by inspecting the tuples it accessed. In our example, when transaction A reaches its commit point, TicToc calculates the commit timestamp using the versions of the tuple x and y it actually read/wrote rather than the latest version in the database right now. And since the version of tuple x read by A is older than the one written by B, A will be ordered before B and can commit.

To encode the serialization information in the tuples, each data version in TicToc has a valid range [4] of timestamps bounded by the write timestamp (*wts*) and the read timestamp (*rts*). Specifically, a particular version is created at timestamp *wts* and is valid

Algorithm 1: Read Phase

```
Data: read set $RS$, tuple $t$
1 $r = RS.get\_new\_entry()$
2 $r.tuple = t$
  # Atomically load wts, rts, and value
3 < r.value = t.value, r.wts = t.wts, r.rts = t.rts >
```

until timestamp *rts*. A version read by a transaction is valid if and only if that transaction's commit timestamp is in between the version's *wts* and *rts*. And a write by a transaction is valid if and only if the transaction's commit timestamp is greater than the *rts* of the previous version. Formally, the following invariant must hold for transaction T to commit:

$$\begin{aligned} &\exists\ commit\_ts, \\ &\quad (\forall v \in \{versions\ read\ by\ T\}, v.wts \leq commit\_ts \leq v.rts) \\ &\quad\quad \wedge (\forall v \in \{versions\ written\ by\ T\}, v.rts < commit\_ts) \end{aligned} \quad (1)$$

This policy leads to serializable execution because all the reads and writes within a transaction occur at the same timestamp. A read always returns the version valid at that timestamp and a write is ordered after all the reads to older versions of the same tuple.

3.2 Protocol Specification

Like standard OCC algorithms, each transaction in TicToc accesses the database without acquiring locks during normal operation. This is known as the *read phase*. When the transaction invokes the commit operation, the protocol then takes the transaction through the *validation phase* to check whether it should be allowed to commit. If it does, then it enters the *write phase* where the transaction's changes are applied to the shared database.

We now discuss these phases in further detail.

3.2.1 Read Phase

The DBMS maintains a separate read set and write set of tuples for each transaction. During this phase, accessed tuples are copied to the read set and modified tuples are written to the write set, which is only visible to the current transaction. Each entry in the read or write set is encoded as $\{tuple, data, wts, rts\}$, where $tuple$ is a pointer to the tuple in the database, $data$ is the data value of the tuple, and *wts* and *rts* are the timestamps copied from the tuple when it was accessed by the transaction. For a read set entry, TicToc maintains the invariant that the version is valid from *wts* to *rts* in timestamp order.

Algorithm 1 shows the procedure for a tuple access request in the read phase. The pointer to the tuple is stored in the read or write set depending on the request type. The data value and read and write timestamps are also recorded. Note that the value and timestamps must be loaded atomically to guarantee that the value matches the timestamps. We explain in Section 3.6 how to efficiently perform this operation in a lock-free manner.

3.2.2 Validation Phase

In the validation phase, TicToc uses the timestamps stored in the transaction's read and write sets to compute its commit timestamp. Then, the algorithm checks whether the tuples in the transaction's read set are valid based on this commit timestamp.

The first step for this validation, shown in Algorithm 2, is to lock all the tuples in the transaction's write set in their primary key order to prevent other transactions from updating the rows concurrently. Using this fixed locking order guarantees that there are no deadlocks with other transactions committing at the same time. This technique is also used in other OCC algorithms (e.g., Silo [35]).

Algorithm 2: Validation Phase

```
Data: read set RS, write set WS
# Step 1 – Lock Write Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
$commit\_ts = 0$
for e in $WS \cup RS$ do
    if e in WS then
        $commit\_ts = max(commit\_ts, e.tuple.rts + 1)$
    else
        $commit\_ts = max(commit\_ts, e.wts)$
    end
end
# Step 3 – Validate the Read Set
for r in RS do
    if r.rts < commit_ts then
        # Begin atomic section
        if $r.wts \neq r.tuple.wts$ or ($r.tuple.rts \leq commit\_ts$ and
        isLocked(r.tuple) and r.tuple not in W) then
            abort()
        else
            r.tuple.rts = max(commit_ts, r.tuple.rts)
        end
        # End atomic section
    end
end
```

The second step in the validation phase is to compute the transaction's commit timestamp from the timestamps stored within each tuple entry in its read/write sets. As discussed in Section 3.1, for a tuple in the read set but not in the write set, the commit timestamp should be no less than its *wts* since the tuple would have a different version before this timestamp. For a tuple in the transaction's write set, however, the commit timestamp needs to be no less than its current *rts* $+ 1$ since the previous version was valid till *rts*.

In the last step, the algorithm validates the tuples in the transaction's read set. If the transaction's *commit_ts* is less than or equal to the *rts* of the read set entry, then the invariant *wts* $\leq$ *commit_ts* $\leq$ *rts* holds. This means that the tuple version read by the transaction is valid at *commit_ts*, and thus no further action is required. If the entry's *rts* is less than *commit_ts*, however, it is not clear whether the local value is still valid or not at *commit_ts*. It is possible that another transaction has modified the tuple at a logical time between the local *rts* and *commit_ts*, which means the transaction has to abort. Otherwise, if no other transaction has modified the tuple, *rts* can be extended to be greater than or equal to *commit_ts*, making the version valid at *commit_ts*.

Specifically, the local *wts* is first compared to the latest *wts*. If they are different, the tuple has already been modified by another transaction and thus it is not possible to extend the *rts* of the local version. If *wts* matches, but the tuple is already locked by a different transaction (i.e., the tuple is locked but it is not in the transaction's write set), it is not possible to extend the *rts* either. If the *rts* is extensible or if the version is already valid at *commit_ts*, the *rts* of the tuple can be extended to at least *commit_ts*. Note that the whole process must be done atomically to prevent interference from other transactions. The DBMS also does not need to validate tuples that are only in the write set since they are already protected by the locks acquired at the beginning of the validation phase.

In TicToc, there is no centralized contention point during transaction execution. The locks and atomic sections protect only the tuples that a transaction touches. In Section 5, we present optimizations to further reduce the contention caused by these operations.

Algorithm 3: Write Phase

```
Data: write set WS, commit timestamp commit_ts
for w in WS do
    write(w.tuple.value, w.value)
    $w.tuple.wts = w.tuple.rts = commit\_ts$
    unlock(w.tuple)
end
```

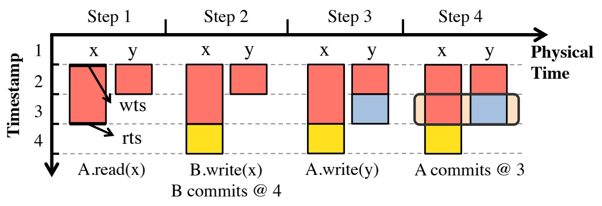


Figure 1: An example of two transactions executing using TicToc.

3.2.3 Write Phase

Finally, if all of the tuples that the transaction accessed pass validation, then the transaction enters the write phase. As shown in Algorithm 3, in this phase the transaction's write set is written to the database. For each tuple in the transaction's write set, the DBMS sets their *wts* and *rts* to *commit_ts*, indicating that it is a new version. All locks that were acquired during the validation phase are then released, making the changes visible to all other transactions.

3.3 Example

We now revisit the example in Section 3.1 and explain how TicToc is able to commit both transactions A and B even though previous OCC algorithms could not. Fig. 1 shows a step-by-step diagram. In this example, one operation occurs in each physical step. The *wts* and *rts* for tuples x and y are encoded as the start and end point of the vertical bands.

Step 1: Transaction A reads tuple x. The current version of x and its timestamps (*wts* $= 2$ and *rts* $= 3$) are stored in A's read set.

Step 2: Transaction B writes to tuple x and commits at timestamp 4. The version of x will be overwritten and both the *wts* and *rts* of the new version will become 4.

Step 3: Transaction A writes to tuple y. Since the previous version of y has *rts* $= 2$, the new version can be written at timestamp 3. At this point, the new version of y is only stored in the write set of transaction A and is not visible to other transactions yet.

Step 4: Transaction A enters the validation phase. According to Algorithm 2, the commit timestamp should be the maximum of the read tuple's *wts* and write tuple's *rts* $+1$, which is timestamp 3 in this example. Then, transaction A validates the read set by checking whether the version of tuple x it read is valid at timestamp 3. Since transaction A's version of tuple x is valid at timestamp 2 and 3, it passes the validation phase and commits.

Note that when the DBMS validates transaction A, it is not even aware that tuple x has been modified by another transaction. This is different from existing OCC algorithms (including Hekaton [24] and Silo [35]) which always recheck the tuples in the read set. These algorithms would abort transaction A in this example because tuple x has already been modified since transaction A last read it.

3.4 Spurious Aborts

A transaction may not always be able to validate its read set during the validation phase, which leads to aborts that may seem spurious. For example, if tuple y in Fig. 1 was originally valid from timestamps 1 to 4, then transaction A's *commit_ts* has to be 5. And since x's *rts* cannot be extended to 5, A has to abort. In this case, A aborts not because of the timestamps and not data values.

In general, the reason that these aborts occur is because other transactions violate serializability. For the example above, imagine that there exists a transaction C reading tuple x and y after B commits but before A commits. C is able to commit at timestamp 4 as it observes B's write to x and the original value of y. C will extend the *rts* of y to 4. This means that A cannot commit now without violating serializability because there is a dependency cycle between A, B and C[1]. Note that when A enters the validation phase, it does not know that C exists or that A would form a dependency cycle with other committed transactions.

Note that in TicToc a transaction aborts only if a tuple it reads or writes is overwritten by another transaction that enters the validation phase first. So only concurrent transactions (i.e., one starts before the other commits) can cause aborts. If a transaction commits, then all transactions that start after it will observe its changes.

3.5 Discussion

Beyond scalability and increased concurrency, TicToc's protocol has two other distinguishing features. Foremost is that the transaction's logical commit timestamp order may not agree with the physical commit time order. In the example from shown in Fig. 1, transaction A commits physically after transaction B, but its commit timestamp is less than transaction B's commit timestamp. This means that A precedes B in the serial schedule. This also indicates that TicToc is not *order-preserving* serializable, since the serial order may not be the commit order.

Another feature of TicToc is that logical timestamps grow more slowly than the number of committed transactions. Moreover, the rate at which the logical timestamp advances is an indicator of the contention level in the workload. This is because different transactions may commit with the same logical timestamp. Such a scenario is possible if two transactions have no conflicts with each other, or if one transaction reads a version modified by the other transaction. At one extreme, if all transactions are read-only and thus there is no contention, all transactions will have the same commit timestamp. At the other extreme, if all the transactions write to the same tuple, each commit would increase the tuple's *wts* by one, and the logical timestamp would increase at the same rate as the number of committed transactions. Since most OLTP workloads have some contention, the DBMS's logical timestamps will increase more slowly than the number of committed transactions; the higher the contention, the faster logical timestamps advance. We will show this in Section 6.5.

3.6 Implementation

As shown in Algorithms 1 and 2, both the read and validation phases require the DBMS to atomically read or write tuples' timestamps. But implementing these atomic sections using locks would degrade performance. To avoid this problem, TicToc adopts an optimization from Silo [35] to encode a lock bit and a tuple's *wts* and *rts* into a single 64-bit word (`TS_word`) of the following form:

[1] $A<B$ due to write-after-read on x, $B<C$ due to read-after-write on x, and $C<A$ due to write-after-read on y.

Algorithm 4: Atomically Load Tuple Data and Timestamps

```
Data: read set entry $r$, tuple $t$
1 do
2     $v1 = t.read\_ts\_word()$
3     $read(r.data, t.data)$
4     $v2 = t.read\_ts\_word()$
5 while $v1 \neq v2$ or $v1.lock\_bit == 1$;
6 $r.wts = v1.wts$
7 $r.rts = v1.wts + v1.delta$
```

`TS_word[63]`: Lock bit (1 bit).
`TS_word[62:48]`: *delta* = *rts* − *wts* (15 bits).
`TS_word[47:0]`: *wts* (48 bits).

The highest-order bit is used as the lock bit. *wts* is stored as a 48-bit counter. To handle *wts* overflows, which happens at most once every several weeks for the most active workloads, the tuples in the database are periodically loaded in the background to reset their *wts*. This process is infrequent and can be performed concurrently with normal transactions, so its overhead is negligible.

Algorithm 4 shows the lock-free implementation of atomically loading the data and timestamps for the tuple read operation from Algorithm 1. `TS_word` is loaded twice, before and after loading the data. If these two `TS_word` instances are the same and both have the lock bit unset, then the data value must not have changed and is still consistent with the timestamps. Otherwise, the process is repeated until both timestamps are consistent. There are no writes to shared memory during this process. To avoid starvation, one could revert to more heavy-weight latching if this check repeatedly fails.

Similarly, Algorithm 5 shows the steps to atomically extend a tuple's *rts* in TicToc's validation phase (Algorithm 2). Recall that this operation is called if *commit_ts* is greater than the local *rts*; the DBMS makes the local version valid at *commit_ts* by extending the *rts* of the tuple. The first part of the algorithm is the same as explained in Section 3.2.2; validation fails if the tuple's *rts* cannot possibly be extended to *commit_ts*.

Since we only encode *delta* in 15 bits in `TS_word`, it may overflow if *rts* and *wts* grow far apart. If an overflow occurs, we also increase *wts* to keep *delta* within 15 bits without affecting the correctness of TicToc. Intuitively, this can be considered a dummy write to the tuple at the new *wts* with the same data. Inserting such a dummy write does not affect serializability. Increasing *wts*, however, may increase the number of aborts since another transaction may consider the version as being changed while it has not actually changed. This effect is more problematic the fewer bits *delta* uses. Although not shown in the paper, our experiments indicate that 15 bits is enough for the overflow effect to be negligible.

Finally, the new *wts* and *delta* are written to a new `TS_word` and atomically applied to the tuple. The DBMS uses an atomic *compare-and-swap* instruction to make sure that the `TS_word` has not been modified by other transactions simultaneously.

Scanning tuples in a database may miss tuples being inserted because they are not observed by the scanning transaction. Standard techniques for solving this problem include using locks in indexes [31] or rescanning the tuples during the validation phase [24]. Both techniques incur significant performance overhead. This can be avoided by running the transactions at lower isolation levels (e.g., snapshot isolation) if the application allows it. We believe that it is possible to apply the data-driven timestamp management concept used in TicToc to order-preserving indexes to avoid this phantom anomaly for serializable transactions. This exploration is outside of the scope of this paper and is left for future work.

Algorithm 5: Read-Set Validation

```
Data: read set entry $r$, write set $W$, commit timestamp commit_ts
1  do
2      success = true
3      v2 = v1 = r.tuple.read_ts_word()
4      if r.wts ≠ v1.wts or (v1.rts ≤ commit_ts and isLocked(r.tuple)) and r.tuple not in W then
5          Abort()
6      end
       # Extend the rts of the tuple
7      if v1.rts ≤ commit_ts then
           # Handle delta overflow
8          delta = commit_ts − v1.wts
9          shift = delta − delta ∧ 0x7fff
10         v2.wts = v2.wts + shift
11         v2.delta = delta − shift
           # Set the new TS word
12         success = compare_and_swap(r.tuple.ts_word, v1, v2)
13     end
14 while not success;
```

3.7 Logging and Durability

The TicToc algorithm can support logging and crash recovery in a similar way as traditional concurrency control algorithms. The DBMS can use the canonical ARIES approach if there is only a single log file [27]. But ARIES cannot provide the bandwidth required in today's multi-core systems [39]. Implementing arbitrarily scalable logging is out of the scope of this paper and is left for future work. In this section, we briefly discuss one idea of implementing parallel logging with multiple log files on TicToc.

Parallel logging has been studied in other DBMSs [36, 39]. The basic idea is to perform logging in batches. All transactions in a previous batch must be ordered before any transaction in a later batch, but the relative ordering among transactions within the same batch can be arbitrary. For each batch, logs are written to multiple files in parallel. A batch is considered durable only after all the logs within that batch have been written to files.

In TicToc, the batching scheme requires that transactions in a later batch must have commit timestamps greater than transactions in a previous batch. This can be achieved by setting a minimum commit timestamp for transactions belonging to the new batch. To start a new batch, each worker thread should coordinate to compute the minimum commit timestamp that is greater than the commit timestamps of all transactions in previous batches. Each transaction in the new batch has a commit timestamp greater than this minimum timestamp. Setting a minimum timestamp does not affect the correctness of TicToc since timestamps only increase in this process, and transactions are properly ordered based on their timestamps. The performance of this parallel logging scheme should be the same as with other concurrency control algorithms [36, 39].

4. PROOF OF CORRECTNESS

In this section, we prove that the TicToc algorithm is able to correctly enforce serializability.

4.1 Proof Idea

To prove that a schedule is serializable in TicToc, we need to show that the schedule is equivalent to another schedule where all the transactions are executed serially. Previous T/O concurrency control algorithms use the transaction's unique timestamps to determine the serial order. In TicToc, however, the commit timestamp is derived from the accessed tuples and no global coordination takes place, and thus two transactions may commit with the same timestamp. Therefore, transactions cannot be fully ordered based on their commit timestamps alone.

Our proof instead uses a combination of the timestamp and physical time orders [38]. A transaction's logical commit time is its *commit timestamp*; its *physical commit time* is the physical time between a transaction's validation phase and write phase. We define the following specific serial order.

DEFINITION 1 (SERIAL ORDER). *Using $<_s$, $<_{ts}$ and $<_{ps}$ to indicate serial order, commit timestamp order, and physical commit time order, respectively, the serial order between transaction A and B is defined as follows:*

$$A <_s B \triangleq A <_{ts} B \vee (A =_{ts} B \wedge A \leq_{pt} B)$$

The serial order defined in Definition 1 is a total order among all the transactions. Transaction A is ordered before transaction B if A has a smaller commit timestamp or if they have the same commit timestamp but A commits before B in physical time. If A and B both have the same logical and physical commit time, then they can have arbitrary serial order.

The goal of the correctness proof is summarized as follows:

THEOREM 1. *Any schedule in TicToc is equivalent to the serial schedule defined in Definition 1.*

To prove this, we show that the dependencies in the actual schedule are maintained as the dependencies in the equivalent serial schedule. Specifically, a read in the actual schedule always returns the value of the last store in the serial schedule. We also prove that transactions having the same commit timestamp and physical commit time do not have conflicts.

4.2 Formal Proofs

We first prove a useful lemma that will be used to prove subsequent Lemmas 2 and 3.

LEMMA 1. *Transactions writing to the same tuple must have different commit timestamps.*

PROOF. According to Algorithms 2 and 3, a tuple is locked while being written, therefore only one transaction can write to that tuple at any time. According to line 3 in Algorithm 3, both *wts* and *rts* of the modified tuple become its commit timestamp.

According to line 7 in Algorithm 2, if another transaction writes to the same tuple at a later time, its commit timestamp must be strictly greater than the tuple's current *rts*. Since *rts* never decreases in the TicToc algorithm, the commit timestamp of the later transaction must be greater than the commit timestamp of the earlier transaction. Therefore, transactions writing to the same tuple must have different commit timestamps. □

As discussed in the previous section, two requirements are needed to prove Theorem 1. First, transactions that have the same commit timestamp and physical time must not conflict. Second, a read always returns the latest write in the serial order. We now prove these two requirements for TicToc:

LEMMA 2. *Transactions that commit at the same timestamp and physical time do not conflict with each other.*

PROOF. According to Lemma 1, write-write conflicting transactions must have different commit timestamps. Therefore, we only need to show that all read-write or write-read conflicting transactions commit at different logical or physical time.

Consider a pair of transactions committing at the same physical time. One reads a tuple and the other writes to the same tuple. Then, the commit timestamp of the reading transaction must be less than or equal to the tuple's current *rts*. And the commit timestamp of the writing transaction must be greater than the tuple's current *rts*. Therefore, they have different commit timestamps. □

LEMMA 3. *A read operation from a committed transaction returns the value of the latest write to the tuple in the serial schedule.*

PROOF. We first prove that if a committed transaction's read observes another transaction's write, then the reading transaction must be ordered after the writing transaction in the serial schedule.

A tuple's *wts* is always updated together with its value, and the *wts* is always the commit timestamp of the transaction that writes the value. Line 9 in Algorithm 2 states that if another transaction reads the value, then its commit timestamp must be greater than or equal to *wts*. If the commit timestamp equals *wts*, then the reading transaction still commits after the writing transaction in physical time because the writing transaction only makes its writes globally visible after its physical commit time. By Definition 1, the reading transaction is always ordered after the writing transaction in the serial schedule.

We next show that the write observed by the following read is the latest write in the serial schedule. In other words, if the writing transaction has timestamp t_1 and the reading transaction has timestamp t_2, no other write to the same tuple happens at timestamp t, such that $t_1 \leq t \leq t_2$.

According to Algorithm 2, when the reading transaction commits, it can observe a consistent view of the tuple's `TS_word` with *wts* and *rts*, where $t_1 = wts$ and $t_2 \leq rts$. This implies that so far in physical time, no write to the same tuple has happened between t_1 and t_2 in logical time because otherwise the *wts* of the tuple would be greater than t_1. No such write can happen in the future either because all future writes will have timestamps greater the tuple's *rts* and thus greater than t_2. □

PROOF OF THEOREM 1. According to Lemma 2, transactions with the same commit timestamp and physical commit time do not conflict. Thus, all serial orders among them are equivalent.

According to Lemma 3, for transactions with different commit timestamps or physical commit times, a read in a transaction always returns the latest write in the serial schedule. According to Lemma 1, only one such latest write can exist so there is no ambiguity. Then, for each transaction executed in the actual schedule, all the values it observes are identical to the values it would observe in the serial schedule. Hence, the two schedules are equivalent. □

5. OPTIMIZATIONS

The TicToc algorithm as presented so far achieves good performance when tested on a multi-socket system (Section 6). There are, however, still places in the validation phase that may create unnecessary contention and thus hurt performance. For example, locking the transaction's write set during the validation phase may cause thrashing for write-intensive benchmarks.

In this section, we discuss several optimizations that we developed for TicToc to minimize contention. We also discuss how TicToc works for weaker isolation levels for those applications that do not need strong serializability.

5.1 No-Wait Locking in Validation Phase

In TicToc's validation phase (Algorithm 2), tuples in a transaction's write set are locked following the primary key order. This is essentially the same process as a 2PL protocol where a transaction may wait if the next lock that it needs to acquire is not immediately available. Waiting for locks, however, may create thrashing problems at high core counts, even if locks are acquired in primary key order [37]. Thrashing happens when a transaction already holds locks and waits for the next lock. The locks it already holds, however, may block other transactions.

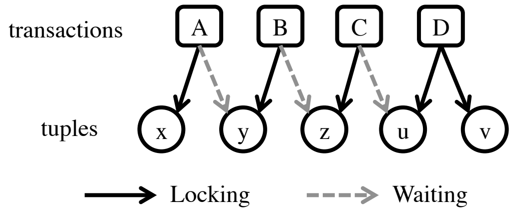


Figure 2: An example of lock thrashing in a 2PL protocol.

Consider a pathological case shown in Fig. 2 where each transaction tries to lock two tuples. Transaction D has already acquired both of the locks that it needs, while transactions A, B, and C are waiting for locks held by other transactions. When transaction D commits, C is able to acquire the lock and make forward progress. Transactions A and B, however, still need to wait. The end result is that the four transactions are validated sequentially.

Note that, in this particular example, it is actually possible to abort transactions A and C so that B and D can acquire the locks and run in parallel. After they finish, A and C can also run in parallel. This schedule only takes half the execution time compared to the pathological schedule, but it requires an additional deadlock detection thread to quickly identify these scenarios.

A better approach to avoid the thrashing problem is to use a 2PL variant based on non-waiting deadlock prevention in TicToc's validation phase [5]. This protocol optimization, which we refer to as **no-wait**, is like running a mini concurrency control algorithm inside of the TicToc algorithm. With no-wait, if a transaction fails to acquire a lock for a tuple in its write set during the validation phase, the validation is immediately aborted (releasing any locks) and then TicToc restarts the validation phase. The transaction sleeps for a short period (1 μs) before retrying to reduce restarts. Our experiments show that the algorithm's performance is not overly sensitive to the length of this sleep time as long as it is not too large.

The no-wait optimization minimizes the blocking and allows more transactions to validate simultaneously. In the example in Fig. 2, if no-wait is used, then A or B may run in parallel with D.

5.2 Preemptive Aborts

The first step of the validation phase (Algorithm 2) locks the tuples in the transaction's write set before it examines the read set. If a transaction ends up aborting because read set validation fails, then this locking potentially blocked other transactions unnecessarily. We observe that for some transactions the decision to abort can actually be made before locking the write set tuples. We call this optimization **preemptive abort**. Since all the serialization information is already stored in the tuples' timestamps, it can be used to make early abort decisions, thereby reducing contention.

To better understand this, consider a transaction with one tuple in its read set. This transaction will fail the validation phase if this tuple's local *rts* is less than the transaction's commit timestamp and its local *wts* does not match the tuple's latest *wts*. A tuple's latest *wts* can be atomically read from the `TS_word` of the tuple. The transaction's commit timestamp, however, cannot be accurately determined before the write set is locked because a tuple's *rts* in the write set might be changed by a different transaction. The key observation here is that we just need to find an approximate commit

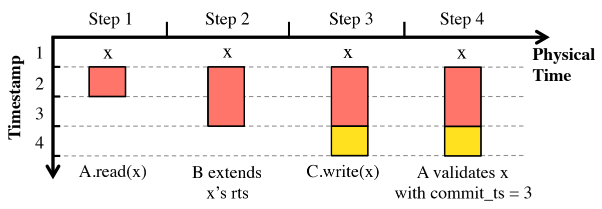


Figure 3: Using a tuple's timestamp history to avoid aborting.

timestamp. Thus, this optimization achieves some early aborts, but does not catch all the transactions that will fail read set validation.

We compute the approximate commit timestamp using the local *wts* and *rts* in the read and write sets. For each tuple in the read set, the approximate commit timestamp is no less than the tuple's local *wts*; for each tuple in the write set, the approximate commit timestamp is no less than the tuple's local *rts* $+1$. Note that the actual commit timestamp is no less than our approximation, because the latest timestamps in the tuples cannot be less than the local timestamps. Once an approximate commit timestamp is determined, it is used to determine if the transaction should be preemptively aborted.

5.3 Timestamp History

TicToc always aborts a transaction if its local *rts* of a tuple is less than *commit_ts* and the local *wts* does not match the latest *wts*. There are some cases, however, where the latest *wts* is greater than *commit_ts* and the local version is still valid at *commit_ts*. Such transactions can actually commit without violating serializability.

Fig. 3 shows such an example. Transaction A first reads tuple x with *wts* $= 2$ and *rts* $= 2$. Later, tuple x's *rts* is extended to timestamp 3 due to the validation of transaction B. Then, tuple x is modified by transaction C and thus the latest *wts* and *rts* both become 4. Finally, transaction A enters the validation phase and validates tuple x at *commit_ts* $= 3$ (not 2 because transaction A accessed other tuples not shown in the figure). At this point, transaction A only has the local timestamps of x (*wts* $= 2$ and *rts* $= 2$) and knows that the local version is valid at timestamp 2, but does not know if it is still valid at timestamp 3. From transaction A's perspective, it is possible that the local version has been extended to timestamp 3 by some other transaction; it is also possible, however, that some other transaction did a write that is only valid at timestamp 3. Based on all the information transaction A has, these two situations are indistinguishable.

To prevent these unnecessary aborts, we can extend TicToc to maintain a history of each tuple's *wts* rather than just one scalar value. When a new version is created for a tuple, the *wts* of the old version is stored in a history buffer. The value of the old version does not need to be stored since transactions in TicToc always read the latest data version. Therefore, the storage overhead of this optimization in TicToc is smaller than that of MVCC. In our implementation, the history buffer is a per-tuple array that keeps a fixed number of the most recent *wts*'s, and thus the DBMS does not have to perform garbage collection.

During a transaction's validation phase, if a read tuple's local *rts* is less than *commit_ts* and the *wts* does not match the latest *wts*, then the DBMS checks if the *wts* matches any version in the tuple's history buffer. If so, the valid range of that version is from the local *wts* to the next *wts* in the history buffer. If *commit_ts* falls within that range, the tuple can still be validated.

5.4 Lower Isolation Levels

The TicToc algorithm described in Section 3 provides serializable isolation, which is the strictest isolation level in ANSI SQL. With minimal changes, TicToc can also support lower isolation levels for those applications that are willing to sacrifice isolation guarantees in favor of better performance and lower abort rate.

Snapshot Isolation: This level mandates that all of a transaction's reads see a consistent snapshot of the database, and that the transaction will commit only if it does not conflict with any concurrent updates made since that snapshot. In other words, all the read operations should happen at the same timestamp (*commit_rts*) and all the write operations should happen at a potentially later timestamp (*commit_wts*), and the written tuples are not modified between *commit_rts* and *commit_wts*.

To support snapshots, instead of using a single *commit_ts* and verifying that all reads and writes are valid at this timestamp, two commit timestamps are used, one for reads (*commit_rts*) and one for writes (*commit_wts*). The algorithm verifies that all reads are valid at *commit_rts* and all writes are valid at *commit_wts*. It also guarantees that, before the transaction writes to a tuple, its previous *wts* is less than or equal to *commit_rts*. All of these can be implemented with minor changes to Algorithm 2.

Repeatable Reads: With this weaker isolation level, a transaction's reads do not need to happen at the same timestamp even though writes should still have the same commit timestamp. This means there is no need to verify the read set in the validation phase. For a tuple read and updated by the same transaction, however, the DBMS still needs to guarantee that no other updates happened to that tuple since the transaction last read the value.

6. EXPERIMENTAL EVALUATION

We now present our evaluation of the TicToc algorithm. For these experiments, we use the DBx1000 OLTP DBMS [1]. This is a multi-threaded, shared-everything system that stores all data in DRAM in a row-oriented manner with hash table indexes.

DBx1000 uses worker threads (one per core) that invoke transactions from a fixed-length queue. Each transaction contains program logic intermixed with query invocations. Queries are executed serially by the transaction's worker thread as they are encountered in the program logic. Transaction statistics, such as throughput and abort rates, are collected after the system achieves a steady state during the warm-up period. The abort rate is calculated as the total number of aborts divided by the total number of transaction attempts (both committed and aborted transactions).

DBx1000 includes a pluggable lock manager that supports different concurrency control schemes. This allows us to compare five approaches all within the same system:

TICTOC:	Time traveling OCC with all optimizations
SILO:	Silo OCC [35]
HEKATON:	Hekaton MVCC [24]
DL_DETECT:	2PL with deadlock detection
NO_WAIT:	2PL with non-waiting deadlock prevention

We deployed DBx1000 on a 40-core machine with four Intel Xeon E7-4850 CPUs and 128 GB of DRAM. Each core supports two hardware threads, for a total of 80 threads. The experiments with more than 40 threads (shaded areas in the throughput graphs) use multiple threads per core, and thus may scale sub-linearly due to contention. To minimize memory latency, we use `numactl` to ensure each thread allocates memory from its own socket.

6.1 Workloads

We next describe the two benchmarks that we implemented in the DBx1000 DBMS for this analysis.

TPC-C: This workload is the current industry standard to eval-

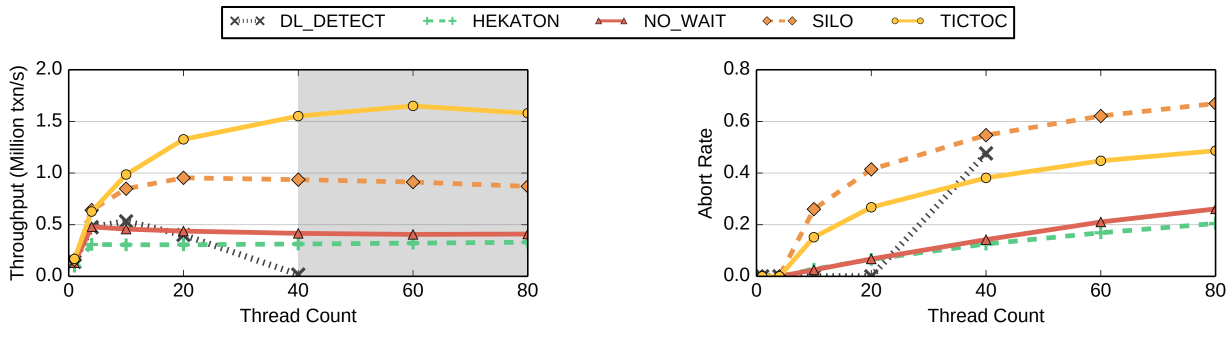


Figure 4: TPC-C (4 Warehouses) – Scalability of different concurrency control algorithms on TPC-C workload with 4 warehouses.

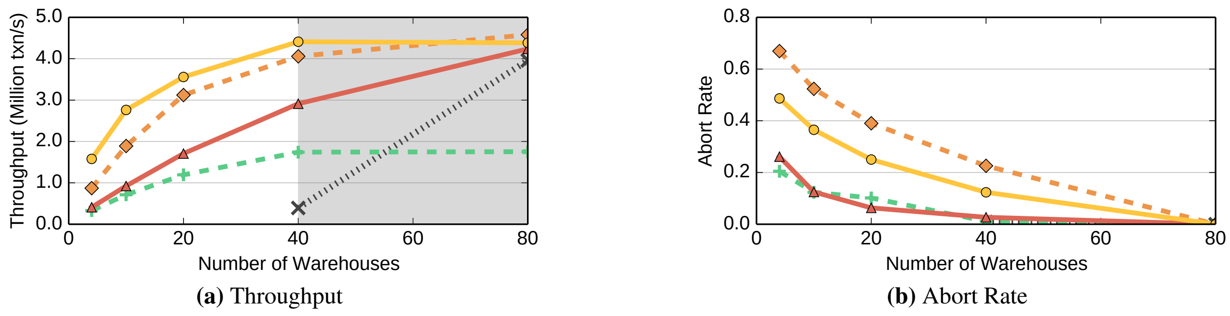


Figure 5: TPC-C (Variable Warehouses) – Scalability of different concurrency control algorithms on TPC-C when sweeping the number of warehouses. The number of worker threads in DBx1000 is fixed at 80.

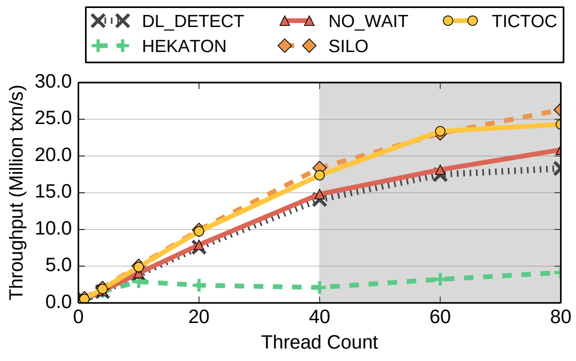


Figure 6: YCSB (Read-Only) – Results for a read-only YCSB workload for the different concurrency control schemes and the *atomic add* timestamp allocator.

uate OLTP systems [33]. It consists of nine tables that simulate a warehouse-centric order processing application. Only two (`Payment` and `NewOrder`) out of the five transactions in TPC-C are modeled in our simulation, with the workload comprised of 50% of each type. These two make up 88% of the default TPC-C mix and are the most interesting in terms of complexity for our evaluation.

YCSB: The Yahoo! Cloud Serving Benchmark is representative of large-scale on-line services [9]. Each query accesses a single random tuple based on a Zipfian distribution with a parameter ($theta$) that controls the contention level in the benchmark [16]. We evaluate three different variations of this workload:

1. **Read-Only:** Two read queries per transaction and a uniform access distribution ($theta$=0).
2. **Medium Contention:** 16 queries per transaction (90% reads and 10% writes) with a hotspot of 10% tuples that are accessed by ∼60% of all queries ($theta$=0.8).
3. **High Contention:** 16 queries per transaction (50% reads and 50% writes) with a hotspot of 10% tuples that are accessed by ∼75% of all queries ($theta$=0.9).

For all of the YCSB experiments in this paper, we used a ∼10 GB database containing a single table with 10 million records. Each tuple has a single primary key column and then 10 additional columns each with 100 bytes of randomly generated string data.

6.2 TPC-C Results

We first analyze the performance of all the concurrency control algorithms on the TPC-C benchmark.

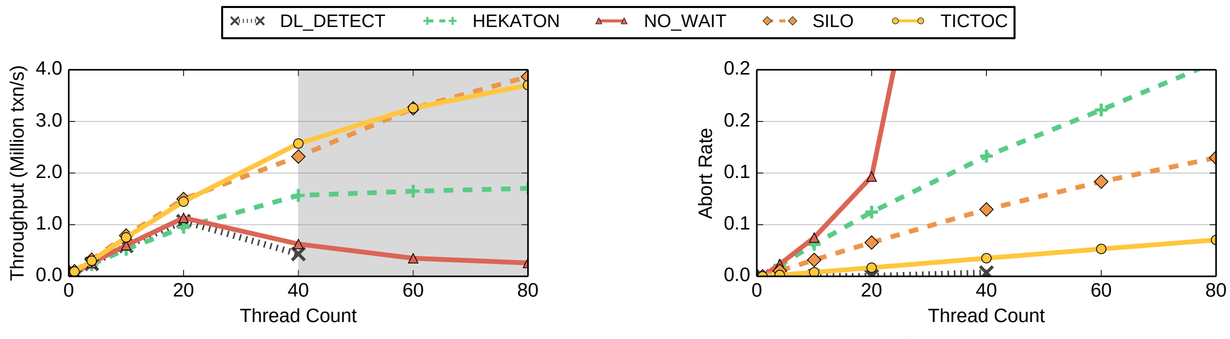
The number of warehouses in TPC-C determines both the size of the database and the amount of concurrency. Each warehouse adds ∼100 MB to the database. The warehouse is the root entity for almost all of the tables in the database. We follow the TPC-C specification where ∼10% of the `NewOrder` transactions and ∼15% of the `Payment` transactions access a "remote" warehouse.

We first run TPC-C with four warehouses, as this is an example of a database that has a lot of contention. We then run an experiment where we fix the number of threads and scale the number of warehouses in the database. This measures how well the algorithms scale when the workload has more parallelism opportunities.

6.2.1 4 Warehouses

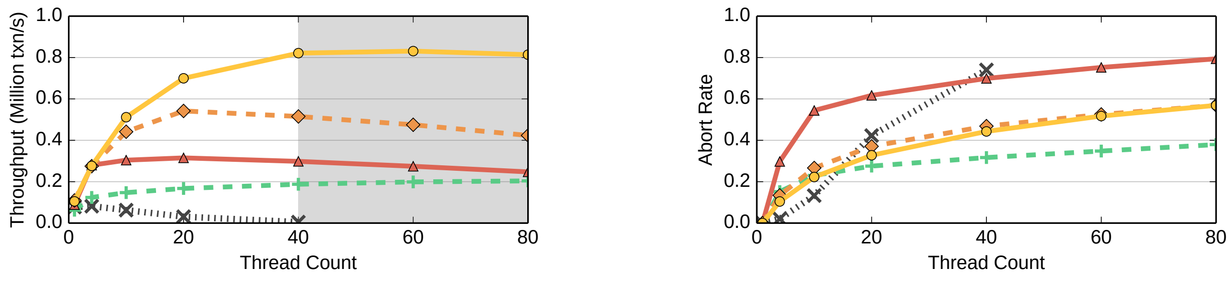
The results in Fig. 4 show that the performance improvements of additional threads are limited by contention on the `WAREHOUSE` table. Each `Payment` transaction updates a per-warehouse tuple in this table and each `NewOrder` transaction reads that tuple. Since there are only four such tuples in the entire database, they become the bottleneck of the whole system.

The `Payment` transaction is simpler faster than `NewOrder` transactions. In SILO, when the `NewOrder` transaction enters the validation phase, it is likely that a `Payment` transaction has already modified the tuple in the `WAREHOUSE` table. Therefore, SILO (like other



(a) Throughput **(b)** Abort Rate

Figure 7: YCSB (Medium Contention) – Results for a read-write YCSB workload with medium contention. Note that DL_DETECT is only measured up to 40 threads.



(a) Throughput **(b)** Abort Rate

Figure 8: YCSB (High Contention) – Results for a read-write YCSB workload with high contention. Note that DL_DETECT is only measured up to 40 threads.

traditional OCCs) frequently aborts these NewOrder transactions.

In TICTOC, the NewOrder transaction would also see that the WAREHOUSE tuple has been modified. But most of the time the transaction can find a common timestamp that satisfies all the tuples it accesses and thus is able to commit. As shown in Fig. 4, TICTOC achieves 1.8× better throughput than SILO while reducing its abort rate by 27%. We attribute this to TICTOC's ability to achieve better parallelism by dynamically selecting the commit timestamp.

The figure also shows that DL_DETECT has the worst scalability of all the algorithms. This is because DL_DETECT suffers from the thrashing problem discussed in Section 5.1. Thrashing occurs because a transaction waits to acquire new locks while holding other locks, which cause other transactions to block and form a convoy. NO_WAIT performs better than DL_DETECT as it avoids this thrashing problem by not waiting for locks. HEKATON also supports non-blocking reads since a transaction can always access a previous version and transactions that perform conflicting writes are immediately aborted. Since rolling back an aborted transaction in an in-memory DBMS is a relatively fast operation, these increased aborts do not significantly hurt performance. But NO_WAIT still performs worse than TICTOC and SILO due to the usage of locks. Similarly, HEKATON is slower because of the overhead of maintaining multiple versions.

6.2.2 *Variable Warehouses*

As we increase the number of warehouses while fixing the number of worker threads, the contention in the system will decrease. In Fig. 5 the number of warehouses is swept from 4 to 80 but the number of worker threads is fixed to 80.

When the number of warehouses is small and contention is high, TICTOC performs consistently better than SILO for the same reason as in Section 6.2.1. As the number of warehouses grows, parallelism in TPC-C becomes plentiful, so the advantage of TICTOC over SILO decreases and eventually disappears at 80 warehouses. With respect to the scheme's measured abort rate, shown in Fig. 5b, TICTOC has consistently fewer aborts than SILO for fewer than 80 warehouses because it is able to adjust timestamps to commit transactions that SILO aborts.

6.3 YCSB Results

We now compare TicToc to other concurrency control schemes under different YCSB scenarios.

6.3.1 *Read-Only*

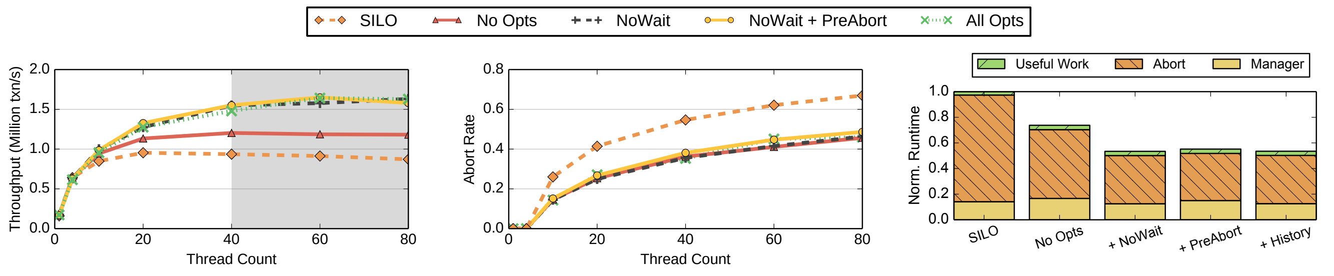
We executed a YCSB workload comprising read-only transactions with a uniform access distribution. This provides a baseline for each concurrency control scheme before we explore more complex workload arrangements.

The results in Fig. 6 show that all of the algorithms except for HEKATON scale almost linearly up to 40 threads. Beyond that point, scaling is sub-linear as the threads executing on the same physical core contend for pipeline resources. TICTOC and SILO achieve better absolute performance than the other algorithms because they do not have locking overheads. HEKATON is limited by its centralized timestamp allocation component. It uses a single *atomic add* instruction on a global counter, which causes threads accessing the counter from different cores to incur cache coherence traffic on the chip. In our 4-socket system, this limits HEKATON to ∼5 million timestamps per second.

6.3.2 *Medium Contention*

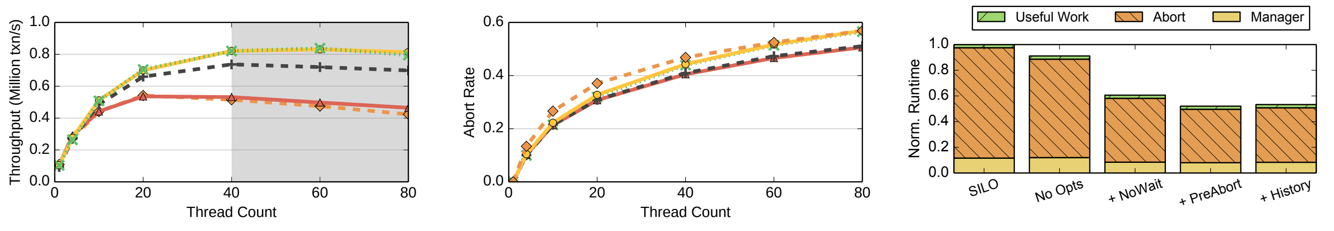
In a read-only workload, transactions do not conflict with each other and thus any algorithm without artificial bottlenecks should scale. For workloads with some contention, however, the ways that the algorithms handle conflicts affect the DBMS's performance.

Fig. 7 shows the throughput and abort rate of the medium contention YCSB workload. The results in Fig. 7a show that SILO and



(a) Throughput **(b)** Abort Rate **(c)** Execution Time Breakdown (80 threads)

Figure 9: TicToc Optimizations (TPC-C) – Throughput measurements of TicToc using the different optimizations from Section 5 for TPCC with 4 warehouses.



(a) Throughput **(b)** Abort Rate **(c)** Execution Time Breakdown (80 threads)

Figure 10: TicToc Optimizations (YCSB) – Throughput measurements of TicToc using the different optimizations from Section 5 for a high contention read-write YCSB workload.

TICTOC both scale well and achieve similar throughput. But the graph in Fig. 7b shows that TICTOC has a ~3.3× lower abort rate than SILO. This is due to TICTOC's data-driven timestamp management, as transactions can commit at the proper timestamp that is not necessarily the largest timestamp so far.

The throughput measurements show that DL_DETECT again has the worst scalability of all the algorithms due to lock trashing. Since NO_WAIT does better since transactions can get immediately restarted when there is a deadlock. HEKATON performs better than the 2PL schemes since multiple versions allows more read operations to succeed (since they can access older versions) which leads to fewer transaction aborts. But this adds overhead that causes HEKATON to perform worse than TICTOC and SILO.

6.3.3 High Contention

We now compare the algorithms on a YCSB workload with high contention. Here, conflicts are more frequent and the workload has lower inherent parallelism, which stresses the DBMS and allows us to more easily identify the main bottlenecks in each algorithm.

As expected, the results in Fig. 8 show that all algorithms are less scalable than in the medium contention workload. We note, however, that the performance difference between TICTOC and SILO is more prominent. As we discuss next, this performance gain comes from the optimizations we presented in Section 5.

Both TICTOC and SILO have similar abort rates in Fig. 8b under high contention. The timestamp management policy in TICTOC does not reduce the abort rate because the workload is too write-intensive and the contention level is so high that both algorithms have similar behaviors in terms of aborting transactions.

6.4 TicToc Optimizations

We now evaluate the optimizations we presented in Section 5 to determine their individual effect on TicToc's overall performance. To do this, we run DBx1000 multiple times using TicToc but enable the optimizations one-at-a-time. We use four different configurations in this experiment:

1. **No Opts:** TicToc without any optimizations.
2. **NoWait:** TicToc with no-wait locking described in Section 5.1.
3. **NoWait + PreAbort:** TicToc with no-wait locking and preemptive aborts from Section 5.2.
4. **All Opts:** TicToc with no-wait locking, preemptive aborts, and timestamp history from Section 5.3.

Recall that this last configuration is the default setting for TicToc in all of the other experiments in this paper. We also include the performance measurements for SILO from Fig. 8a for comparison.

The results in Fig. 9 show the performance, abort rate, and time breakdown (at 80 cores) for the TPC-C workload with four warehouses. At 80 threads, TICTOC without optimizations achieves 35.6% higher throughput than SILO, and has a 32% lower abort rate. This gain comes from the greater parallelism exploited by the TICTOC's timestamp management policy. Using the no-wait optimization for locking transactions' write sets provides another 38% performance gain at 80 threads while not affecting the abort rate. In Fig. 9c, we see that the gains of basic TICTOC over SILO mainly come from reducing the abort rate. Optimizations do not reduce TICTOC's abort rate further, but they do reduce the amount of time wasted in aborting transactions. These optimizations effectively make each abort take a shorter amount of time.

Fig. 10 shows the same experiments on high contention YCSB workload. Here we see similar performance improvement as in TPC-C but it comes from different aspects of TICTOC. TICTOC without any optimizations only performs 10% better than SILO, and most of the performance improvement comes from the no-wait and preemptive abort optimizations. In contrast to Fig. 9, using preemptive aborts provides a larger performance gain in YCSB. This is partly because in YCSB each transaction locks more tuples during the validation phase. Preemptive aborts alleviate the contention caused by these locks.

A key finding is that the timestamp history optimization does not provide any measurable performance gain in either workload. This was initially surprising to us, but upon further investigation we are convinced that this is indeed correct. In a way, TICTOC without this optimization already stores multiple versions of a tuple in each transaction's private workspace. This means that each transaction

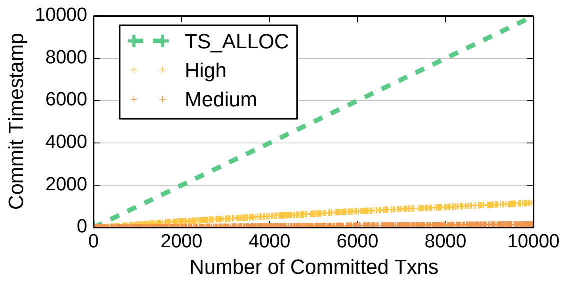


Figure 11: Logical Time Analysis – Comparison of the growth rate of the timestamps in TICTOC versus TS_ALLOC.

	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	0.43	1.55	0.63	2.32	2.57
SI	–	1.78	–	–	2.69
RR	0.72	1.88	1.89	2.45	2.69

(a) Throughput (Million txn/s)

	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	0.35%	11.6%	63.2%	6.47%	1.76%
SI	–	1.96%	–	–	1.54%
RR	0.10%	1.94%	9.9%	0.71%	0.72%

(b) Abort Rate

Table 1: Isolation Levels (Medium Contention) – Performance measurements for the concurrency control schemes running YCSB under different isolation levels with 40 threads.

can commit in parallel using its own version. Although in theory timestamp history can enable more concurrency, in practice there is no clear performance benefit for the workloads we evaluated.

6.5 Logical Time Analysis

We analyze how TicToc's commit timestamps grow over time under different levels of contention. Ideally, we would like timestamps to grow slowly over time relative to the total number of committed transactions, indicating that synchronization among transactions is relatively infrequent. For this experiment, we execute the medium and high contention YCSB workloads from Section 6.3 and track the values of the transactions' commit timestamps over time. We also include TS_ALLOC as a baseline where each transaction is assigned a unique timestamp using an atomic add instruction. This is representative of the timestamp growth rate in other T/O-based algorithms, such as HEKATON.

Fig. 11 shows the relationship between logical timestamps and the number of committed transactions for the three configurations. With the TS_ALLOC protocol, the number of committed transactions and logical timestamps increase at the same rate. In TicToc, however, logical timestamps increase at a slower rate: 64× and 10× slower for low and high contention levels in YCSB, respectively. What is interesting about these measurements is that the rate of growth of logical timestamps indicates the inherent level of parallelism in a workload that can be exploited by TicToc. In the high contention workload, for example, this ratio is 10×. This corroborates our results in Fig. 8a that show the DBMS was only able to achieve 7.7× better throughput from running multiple threads in the high contention YCSB workload.

6.6 Isolation Levels

All of the experiments so far have used serializable isolation. Serializable is the strictest isolation level and thus usually has less concurrency opportunities than lower isolation levels. We now compare the DBMS's performance when transactions execute under snapshot isolation (SI) and repeatable read isolation (RR) levels versus the default serializable isolation (SR). All five algorithms support the RR level. For SI, we are only able to test the TICTOC and HEKATON algorithms. This is because supporting SI requires the DBMS to maintain multiple versions for each tuple, and thus this requires significant changes to the other algorithms. We use the medium- and high-contention YCSB workloads from Section 6.3.

The medium-contention YCSB results are shown in Table 1. For this setting, the workload has enough parallelism and thus all the optimistic T/O-based algorithms only see small improvements when running at a lower isolation level (4.7% for TICTOC and 5.6% for SILO), whereas for the pessimistic 2PL algorithms the improvement is more pronounced (67.4% for DL_DETECT and 200.0% for NO_WAIT). HEKATON only has a 21.3% improvement from SR to RR. The abort rate measurements in Table 1b show that the lower isolation levels achieve lower abort rates because there are fewer conflicts between transactions. As expected, all the algorithms have the fewest number of aborted transactions under RR since it is the most relaxed isolation level.

	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	0.005	0.18	0.30	0.52	0.82
SI	–	0.23	–	–	0.90
RR	0.010	0.23	0.35	0.80	1.04

(a) Throughput (Million txn/s)

	DL_DETECT	HEKATON	NO_WAIT	SILO	TICTOC
SR	74.0%	34.4%	69.9%	46.8%	44.3%
SI	–	30.9%	–	–	40.1%
RR	74.3%	30.4%	71.3%	42.3%	39.7%

(b) Abort Rate

Table 2: Isolation Levels (High Contention) – Performance measurements for the concurrency control schemes running YCSB under different isolation levels with 40 threads.

The high-contention YCSB results are shown in Table 2. Lower isolation levels have better performance than serializable isolation. Again, the throughput of the RR isolation level is slightly better than SI's. In general, for this workload setting we found that different isolation levels do not cause large reductions in abort rates due to the significant amount of contention on hotspot tuples.

7. CONCLUSION

In this paper we have presented TicToc, a new OCC-based concurrency control algorithm that eliminates the need for centralized timestamp allocation. TicToc instead uses a novel *data-driven timestamp management* approach that decouples logical timestamps and physical time by deriving transaction commit timestamps from data items. This enables an OLTP DBMS to exploit more parallelism than other OCC-based algorithms. We have also presented several optimizations that leverage this timestamp management policy to further improve TicToc's performance. Our evaluation results show that, compared to another state-of-the-art OCC algorithm, TicToc achieves up to 92% higher throughput while reducing transaction abort rates by up to 3.3× under different workload conditions.

ACKNOWLEDGMENTS

This research was funded (in part) by the Intel Science and Technology Center for Big Data and the U.S. National Science Foundation (CCF-1438955, CCF-1438967).

For questions or comments about this paper, please call the CMU Database Hotline at `+1-844-88-CMUDB`**.**

8. REFERENCES

[1] DBx1000. https://github.com/yxymit/DBx1000.

[2] Tile-gx family of multicore processors. http://www.tilera.com.

[3] M. Aslett. How will the database incumbents respond to NoSQL and NewSQL? The 451 Group, April 2011.

[4] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic timestamp allocation for transactions in database systems. In *2nd Int. Symp. on Distributed Databases*, pages 9–20, 1982.

[5] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[6] P. A. Bernstein, D. Shipman, and W. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979.

[7] C. Boksenbaum, M. Cart, J. Ferrié, and J.-F. Pons. Concurrent certifications by intervals of timestamps in distributed database systems. *Software Engineering, IEEE Transactions on*, (4):409–419, 1987.

[8] M. J. Carey. Improving the performance of an optimistic concurrency control algorithm through timestamps and versions. *Software Engineering, IEEE Transactions on*, SE-13(6):746–751, June 1987.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC'10*, pages 143–154.

[10] W. J. Dally. GPU Computing: To Exascale and Beyond. In *Supercomputing '10, Plenary Talk*, 2010.

[11] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Symposium on Operating Systems Principles*, pages 33–48, 2013.

[12] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *Proc. of the High Performance Extreme Computing Conference*, 2013.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *CACM*, 19(11):624–633, 1976.

[14] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer: Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.*, 100(2), 1983.

[15] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154, 1981.

[16] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. SIGMOD, pages 243–252, 1994.

[17] T. Härder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, Nov. 1984.

[18] H. Hoffmann, D. Wentzlaff, and A. Agarwal. Remote store programming. In *High Performance Embedded Architectures and Compilers*, pages 3–17. Springer, 2010.

[19] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, 17.14.1 Invariant TSC, 2015.

[20] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. EDBT, pages 24–35, 2009.

[21] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. SIGMOD '15, pages 691–706, 2015.

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

[23] K.-W. Lam, K.-Y. Lam, and S.-L. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Real-Time Technology and Applications Symposium*, pages 174–179. IEEE, 1995.

[24] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 5(4):298–309, Dec. 2011.

[25] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Real-Time Systems Symposium*, pages 66–75. IEEE, 1993.

[26] D. A. Menascé and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[28] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939, September 2010.

[29] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *Proc. VLDB Endow.*, 5:1447–1458, July 2012.

[30] D. P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, 1978.

[31] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. VLDB '83, pages 81–88, 1983.

[32] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[33] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0), June 2007.

[34] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, Mar. 1998.

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

[36] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[37] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. volume 8, pages 209–220. VLDB Endowment, 2014.

[38] X. Yu and S. Devadas. TARDIS: timestamp based coherence algorithm for distributed shared memory. In *International Conference on Parallel Architectures and Compilation Techniques*, 2015.

[39] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 465–477. USENIX Association, 2014.

APPENDIX: DYNAMIC TIMESTAMP ALLOCATION

As discussed in Section 2.2, the original DTA technique [4] was designed to resolve and detect deadlocks using dependency graph analysis. It was first used in deadlock detection and prevention for 2PL algorithms. In DTA, timestamps are associated with each transaction and are compared when two transactions conflict with each other. Timestamps are not associated with tuples in DTA like they are in TicToc.

DTA was then adapted to OCC algorithms [7] and was later studied in the context of real-time DBMSs [23, 25]. In these designs, DTA was used to detect conflicts and determine the relative ordering among transactions. Similar to TicToc, DTA OCC also reorders transactions in logical time, which may not agree with physical time. However, since timestamps are only associated with transactions but not tuples, the DBMS has to maintain a dependency graph of all active transactions. This dependency graph is updated whenever there is a conflicting access. All the DTA OCC algorithms proposed in the literature have this critical section that becomes a serious performance bottleneck as thread count increases.

We modeled DTA OCC in our DBx1000 system and compared it against TicToc. Our implementation of DTA OCC is idealized since we modeled an empty critical section where a full-blown DTA OCC requires considerable logic in the critical section. As a result, our implementation provides an upper bound on the performance of DTA OCC. For this experiment, we used the same medium-contention YCSB workload that we use in Section 6.3.2. The results in Fig. 12 show that, as expected, DTA OCC fails to scale beyond 20 cores due to this critical section bottleneck. This matches previous studies that have shown that algorithms with shared global resources are unable to scale beyond 16–32 active threads [37].

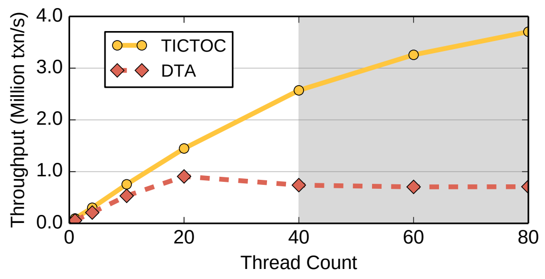


Figure 12: Dynamic Timestamp Allocation – Performance comparison of the DTA OCC and TicToc algorithms for the YCSB workload with medium contention.