Locking Timestamps versus Locking Objects

Marcos K. Aguilera
VMware Research

Tudor David[*]
IBM Research, Zurich

Rachid Guerraoui
EPFL

Junxiong Wang
EPFL

ABSTRACT

We present *multiversion timestamp locking (MVTL)*, a new genre of multiversion concurrency control algorithms for serializable transactions. The key idea behind MVTL is simple: lock individual timestamps instead of locking objects. After presenting a generic MVTL algorithm, we demonstrate MVTL's expressiveness: we give several simple MVTL algorithms that address limitations of current multiversion schemes, by committing transactions that previous schemes would abort, by avoiding the problems of serial aborts or ghost aborts, and by offering a way to prioritize transactions that should not be aborted. We give evidence that, in practice, MVTL-based algorithms can outperform alternative concurrency control schemes.

ACM Reference Format:

Marcos K. Aguilera, Tudor David, Rachid Guerraoui, and Junxiong Wang. 2018. Locking Timestamps versus Locking Objects. In *PODC '18: ACM Symposium on Principles of Distributed Computing, July 23–27, 2018, Egham, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3212734.3212742

1 INTRODUCTION

Serializable transactions are a powerful paradigm available in many computing systems, such as transactional memory, database systems, and key-value storage systems. To ensure serializability, transactions require a scheme for concurrency control to handle any negative consequences of transaction interleaving.

The literature on concurrency control is rich [6, 26], and a particularly appealing class of algorithms is called *multiversion concurrency control* [5]. Briefly, these algorithms keep a *history* of each object, containing many versions of the data with associated timestamps. This history gives the system a *choice* of which version to use when an object is accessed. This choice permits more transactions to execute concurrently without blocking or aborting. For example, in some multiversion algorithms [6, 8], read-only transactions can execute without ever blocking or aborting, and update transactions can concurrently update the same object. Enabling more concurrency has become particularly important with the proliferation of multi-core and large-scale systems. Multiversion algorithms have wide application: they are used in database systems both commercial and academic [10, 19, 26, 27], and recent work has applied them to key-value storage systems and transactional memory (e.g., [11, 15–17, 22, 23]). In this paper, we do not restrict ourselves to particular applications, but rather study multiversion algorithms in their broadest scope.

There are three main genres of multiversion algorithms: *lock based*, *timestamp ordering*, and *serialization graph based* [6]. Lock-based algorithms (e.g., MV2PL [6]) acquire locks to avoid the ill-effects of concurrency; these algorithms are very simple. Timestamp ordering algorithms (e.g., MVTO [6]) assign a timestamp to each transaction, and then serialize transactions by timestamp; these algorithms permit read-only transactions to execute without ever aborting. Serialization graph algorithms (e.g., MVSGT [26]) detect cycles in the serialization graph to prevent a violation of serializability; these algorithms permit higher levels of concurrency than the alternatives.

Despite their many benefits, all types of multiversion algorithms have limitations. Lock-based algorithms significantly restrict the degree of concurrency. Timestamp ordering algorithms are susceptible to aborts, including *serial aborts*—aborts in serial executions—and *ghost aborts*—aborts caused by a conflict with a transaction that already aborted. Serialization graph algorithms are complex and incur significant computation overheads [3, 16, 20].

In this paper, we introduce a new genre of multiversion algorithms, called *multiversion timestamp locking* or MVTL in short. MVTL is based on a simple idea: use locks as in lock-based algorithms, but lock individual timestamps of objects, rather than entire objects at a time. A transaction is allowed to commit if it can find at least one timestamp that it managed to lock across all its objects. Intuitively, MVTL performs well because it uses locks with fine granularity: not only individual objects have separate locks, but individual timestamps within objects have their own locks. Locking at fine granularity increases parallelism and decreases blocking and aborting, as the system can explore many serialization points for each transaction.

Conceptually, MVTL keeps a lock state for each object and each timestamp, which amounts to an infinitely large lock state. However, in practice we can reduce the lock state significantly using interval compression, so that each object holds just a few lock intervals, and this state can be subsequently discarded when the associated versions are purged.

To precisely define MVTL, we give a generic algorithm (§4) that has several nondeterministic choices, such as what timestamps each operation tries to lock, and how locks are acquired (wait or give up on blocked locks). We prove that these choices do not affect safety: the generic algorithm is correct irrespective of them. However, the choices are crucial for performance.

We then propose several specific algorithms that specialize the generic MVTL algorithm by fixing these choices to obtain different benefits (§5). These algorithms are simple and address some important drawbacks of existing multiversion algorithms, such as serial aborts, ghost aborts, the lack of a priority scheme for transactions, and more. We also show that pessimistic and timestamp ordering

[*] Work done while the author was at VMware Research and EPFL.

PODC '18, July 23–27, 2018, Egham, United Kingdom

ACM ISBN 978-1-4503-5795-1/18/07…$15.00
https://doi.org/10.1145/3212734.3212742

algorithms can be seen as special cases of MVTL. Thus, in a precise sense, MVTL unifies these algorithms.

Next, we discuss some practical considerations around MVTL, such as how to compress the lock state (§6). We separate out these considerations because they are orthogonal to the concepts underlying the MVTL algorithm. However, they are important to using MVTL in practice.

Then, we show how to extend the basic MVTL algorithm to distributed transactions in a message-passing system (§7). We believe MVTL is particularly relevant in this setting as it can be quite communication efficient.

We implement an MVTL-based algorithm, and experimentally compare its behavior with multiversion and lock-based alternatives (§8). The results indicate significant advantages of MVTL in read-write workloads, and no disadvantages under read-only workloads.

To summarize, the contributions of this paper are the following:

- We propose a new genre of multiversion algorithms for transactions, called multiversion timestamp locking (MVTL), which is based on the idea of locking timestamps.
- We give several MVTL algorithms, which address various limitations of current multiversion algorithms.
- We show that MVTL generalizes both multiversion timestamp ordering and pessimistic multiversion algorithms.
- We discuss practical considerations for implementing MVTL, including techniques to compress the lock state.
- We describe a version of MVTL for distributed transactions.
- We implement an MVTL-based algorithm and demonstrate its advantages over alternatives.

The main contribution is conceptual in nature: locking individual timestamps is a new way to approach multiversion algorithms. The specific MVTL algorithms we present are simple and just scratch the surface; the investigation of additional MVTL algorithms is an exciting direction for future work. Also interesting is to implement MVTL in other types of transactional systems, such as software transactional memory, transactional key-value storage systems, transactional object systems, and database systems. While the fundamental MVTL algorithms we present are system agnostic, the details of how these algorithms can be best implemented depend on the system and deserve further study.

Due to space limitations, details of our contribution are omitted, including proofs and pseudo-code of some algorithms. They are given in the full paper [1].

2 MODEL

We consider a standard model for a multi-threaded concurrent system [13]. The system has processes that communicate via atomic shared memory. The system is asynchronous: there are no bounds on the relative speed of processes. We assume the existence of a discrete global clock, and processes may or may not have access to the global clock. More precisely, processes may have local clocks that match the global clock ("synchronized clocks") or that are within a known bound ϵ of the global clock ("ϵ-synchronized clocks").

We consider algorithms that implement a transactional storage system. Such a system maintains a set of objects and allows processes to manipulate the objects using transactions. Each object has a unique key (identifier) and, by abuse of language, we refer to the object and its key interchangeably. The system supports four operations with their usual semantics: $\textsc{begin}(tx)$ starts a transaction tx, $\textsc{commit}(tx)$ tries to commit tx and returns a success indication, $\textsc{read}(tx, k)$ reads key k within tx, and $\textsc{write}(tx, k, v)$ writes v to k within tx. Transactions are dynamic: their read and write operations can depend on the results of prior operations in the transaction.

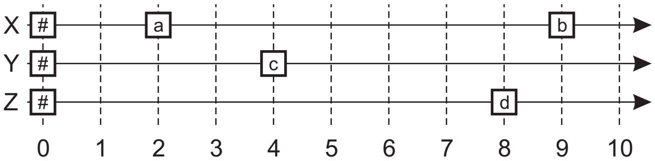
Our correctness condition is *multiversion view serializability*, a form of serializability well-suited for multiversion algorithms. Roughly, this condition requires every multiversion schedule of the algorithm to be equivalent to a serial monoversion schedule [6, 26].

Some of our results refer to a *workload*, which specify the transactional work submitted to the system. More precisely, a workload is a sequence of operations indexed by the transaction they belong to, where each operation is *read*(k), *write*(k, v), or *commit*. We use workloads to study how different protocols react to the same inputs.

3 OVERVIEW

After recalling multiversion concurrency control, we introduce the notion of *timestamp locking* and explain how it addresses weaknesses of existing multiversion algorithms.

Multiversion concurrency control and the MVTO+ algorithm. A well-known genre of multiversion algorithms is multiversion timestamp ordering. Its basic idea is to assign a timestamp to each transaction and then use the timestamp to determine (a) what version the transaction reads from, (b) what version it writes to, and (c) the serialization order of transactions. This idea can lead to several slightly different algorithms. To focus the discussion, here we present a concrete algorithm denoted MVTO+, which is identical to the MVTO algorithm in [6] but with an improvement: it avoids cascading aborts by not reading uncommitted data. For each object, MVTO+ keeps many versions and a timestamp for each version. It is useful to think of each object as an evolving timeline with values. Each transaction tx has a unique timestamp t; when tx reads an object, it obtains the version of the object with the largest timestamp before t. When tx writes an object, tx does not immediately produce a new version but instead it stores the written value in a temporary area for the transaction. Upon commit, tx takes each written value in the temporary area and produces a new version with timestamp t.



For example, the figure above depicts three objects X, Y, and Z. Each object has an initial version denoted #. In addition, X has two other versions with data a and b and timestamps 2 and 9; Y has data c with timestamp 4; and Z has data d with timestamp 8. Suppose a transaction tx is assigned timestamp 6. If tx reads X, it obtains a—the largest version with a timestamp before 6. Similarly, if tx reads Y, it obtains c. If tx writes e to Z and commits, then Z gets a new version with data e and timestamp 6.

Ultimately, transactions are serialized by the order of their timestamps. A key implication is that, after tx reads X and obtains a, another transaction should not produce a version of X with a timestamp between 2 and 6. To prevent this behavior, MVTO+ keeps a *read-timestamp* for each version: this is the largest timestamp with which the version was read by a transaction. In the example, after *tx* reads X and obtains a, the read-timestamp of a becomes 6 (if it was not already larger than 6).

Timestamp locking. We look at MVTO+ slightly differently, using our new notion of *timestamp locking*. This notion allows us to generalize MVTO+ into our new MVTL algorithm. Rather than read-timestamps, we can think that each object has several locks, one for each timestamp. When *tx* reads X, rather than updating the read-timestamp of a to 6, we can think that *tx* obtains a read-lock on each timestamp between 3 and 6. When another transaction wishes to write a version with timestamp, say 5, it must obtain the write-lock on that timestamp. But the read-locks by *tx* prevent this from happening, as required by MVTO+. We can now see the read-timestamp of a as simply a compact representation of the fact that there are read-locks between 3 and 6.

Thinking about timestamp locks has several advantages over read-timestamps. First, with read-timestamps, it is not clear what should happen if *tx* aborts: should the read-timestamp of a be updated to its previous value? But what is the previous value if several other transactions read a concurrently? This is a hard question, and MVTO+ avoids it altogether by taking an unnecessarily conservative approach: when *tx* aborts, it leaves the read-timestamp of a at 6. We show that this choice leads to ghost aborts. In contrast, timestamp locks provide a better alternative: if *tx* aborts, its read-locks are removed but the read-locks of other transactions remain.

Second, with timestamp locks, there is no reason that a transaction should be restricted to obtaining write-locks on just one timestamp, or obtaining read-locks on a range that ends with the transaction's timestamp. Permitting more choices allows the system to avoid serial aborts, as we explain later.

These advantages are captured by our MVTL algorithm, which we now briefly summarize. With MVTL, when a transaction wishes to read an object, it selects a version of the object to read and obtains read-locks on one or more timestamps adjacent to and immediately following that version. To write an object, the transaction obtains write-locks on one or more timestamps anywhere. To commit, the transaction must find a single common timestamp that is read-locked or write-locked across all objects read or written by the transaction, respectively. If such a timestamp exists, the transaction commits; otherwise, it aborts.

The exact timestamps that are locked by reads and writes depend on a *locking policy*. The algorithm remains correct for any locking policy, but a poorly chosen policy causes many aborts because there is no common locked timestamp. We present some simple but interesting algorithms using various locking policies, each with its own advantages.

4 GENERIC MVTL ALGORITHM

We now present our generic MVTL algorithm in detail. We start with some basic concepts (§4.1), explain a simple lock extension we use (§4.2), and cover the main algorithm (§4.3). We present a centralized version of MVTL designed for a single server. We later describe a distributed version of MVTL intended for distributed transactions. Some practical considerations for implementing MVTL, including how locks and data can be compacted, are discussed in §6.

4.1 Preamble

The system keeps many versions of data in an array $\mathit{Values}[k, t]$ where k is a key and t is a timestamp. To ensure processes pick distinct timestamps, we add a process id to a timestamp; thus, a timestamp is a pair (v, p) ordered lexicographically, where v is a number. There is a smallest timestamp denoted 0, and a special initial value denoted #, such that initially $\mathit{Values}[k, 0] = \#$ for every k. It is also useful to assume that initially $\mathit{Values}[k, t] = \perp$ for every k and $t \neq 0$, representing an empty storage system.

4.2 Freezable locks

In the MVTL algorithm, each version of a key is a *write-once object*—an object initially set to $\perp$ that may change its state at most once. We define a simple variation of readers-writer locks, which we call freezable locks, which are appropriate for such objects and we use them in MVTL. A freezable lock is similar to a readers-writer lock, except that a lock holder can freeze the lock to indicate that it will never release it. Freezing is useful because it tells other processes that they should not wait to acquire the lock; we use this feature in several specialized MVTL algorithms. If a lock holder does not freeze a lock, it is expected to release it eventually.

We apply freezable locks to write-once objects as follows. A process acquires the lock in write mode if it intends to write the object. The process may ultimately fail to write if the transaction aborts, in which case it releases the lock; but if the transaction commits, the process freezes its lock to ensure other processes will not try to write the object again. Similarly, a process acquires the lock in read mode to read the object and it freezes the lock in case of a commit; if the object was not written (its state is $\perp$), this prevents other processes from writing to it, sealing its fate.

4.3 Algorithm

Algorithm 1 shows the main code of the generic MVTL algorithm. For clarity, we assume that the code in lines 17–19 is executed atomically, but we later remove this assumption (§6). To write a value into key k, a transaction obtains zero or more write-locks on timestamps for that key (function WRITE-LOCKS in line 4). Intuitively, a write-lock on a timestamp t for key k allows the transaction to commit with timestamp t as far as accesses to k are concerned. After getting the locks, the transaction remembers the key and value; the write is not visible to other transactions until the transaction commits.

To read a key, a transaction gets zero or more read-locks on timestamps for that key (function READ-LOCKS in line 7), with the requirement that these timestamps form a contiguous interval that starts immediately after the version that the read returns. For instance, if $[tr{+}1, te]$ denotes the read-locked timestamps, then the read must return the value committed with timestamp *tr*. This requirement is necessary for serializability: intuitively, the read locks permit the transaction to commit with any timestamp $t \in [tr{+}1, te]$ after having read v, by preventing other transactions from writing a different value with a timestamp between *tr* and *te*. After locking, the transaction remembers k and *tr;* knowledge of k is necessary

Algorithm 1 The generic MVTL algorithm (part 1/2): main code

```
1:  function BEGIN($tx$)
2:      $tx.readset \leftarrow \emptyset$; $tx.writeset \leftarrow \emptyset$; $tx.committs \leftarrow \bot$
3:  function WRITE($tx$, $k$, $v$)                      ▹ write $v$ to $k$ in transaction $tx$
4:      WRITE-LOCKS($tx$, $k$)              ▹ write lock some subset of timestamps
5:      add $(k, v)$ to $tx.writeset$         ▹ remember key and value we wrote
6:  function READ($tx$, $k$)                              ▹ read $k$ in transaction $tx$
7:      $tr \leftarrow$ READ-LOCKS($tx$, $k$) ▹ read lock some interval $[tr{+}1, \ldots]$ with
                                           $Values[k, tr] \neq \bot$
8:      if $tr = \bot$ then return $\bot$                                 ▹ read failed
9:      add $(k, tr)$ to $tx.readset$       ▹ remember key and version we read
10:     return $Values[k, tr]$                           ▹ return committed value
11: function COMMIT($tx$)                            ▹ try to commit transaction $tx$
12:     COMMIT-LOCKS($tx$)                    ▹ locks to acquire at commit time
13:     $T \leftarrow \{t : \forall k \in tx.readset.keys$, $tx$ has a lock on $(k, t)$ and
                $\forall k \in tx.writeset.keys$, $tx$ has a write-lock on $(k, t)\}$
                                        ▹ try to find a locked timestamp for $tx$
14:     if $T = \emptyset$ then mark $tx$ as aborted
15:     else
16:         $tx.committs \leftarrow$ COMMIT-TS($T$)      ▹ pick some timestamp in $T$
17:         for $(k, v) \in tx.writeset$ do
18:             freeze write-lock for $tx$ on $(k, tx.committs)$   ▹ freeze locks
19:             $Values[k, tx.committs] \leftarrow v$        ▹ expose committed value
20:         mark $tx$ as committed
21:     if COMMIT-GC($tx$) then GC($tx$)                        ▹ invoke gc or not
22: function GC($tx$)                ▹ garbage collect locks of $tx$ after it ended
23:     if $tx$ committed then
24:         for $(k, tr) \in tx.readset$ do
25:             freeze read-locks for $tx$ on $[tr{+}1, tx.committs]$
26:     release all unfrozen read- and write-locks for $tx$
```

Algorithm 2 The generic MVTL algorithm (part 2/2): policy

```
function WRITE-LOCKS($tx$, $k$)
    acquire write-locks for $tx$ on $(k, T)$ for some set $T$
function READ-LOCKS($tx$, $k$)                          ▹ returns a timestamp or $\bot$
    acquire read-locks for $tx$ on $(k, T)$ for some $T = [tr{+}1, \ldots]$ where
                  $Values[k, tr] \neq \bot$
    either return $tr$ or return $\bot$
function COMMIT-LOCKS($tx$)
    acquire read- or write-locks for $tx$ on some keys and timestamps
function COMMIT-TS(T) return some $t \in T$
function COMMIT-GC($tx$) either return $true$ or return $false$
```

to commit, and knowledge of both k and tr is needed to garbage collect the locks of the transaction.

To commit, a transaction gets zero or more additional locks (function COMMIT-LOCKS in line 12) and tries to find a commit timestamp t that is write-locked for every k in the write-set, and that is read- or write-locked for every k in the read-set. (A key in the read-set may be write-locked because the transaction read the key and then wrote it.) If there are many such timestamps, the transaction picks one (function COMMIT-TS in line 16). The transaction then freezes write-locks on that timestamp and records the written values so that they can be seen by other transactions. As an optional step (as determined by calling COMMIT-GC in line 21), the transaction may garbage collect the locks it holds. Doing so freezes the read locks between the version read and the commit timestamp, and releases all other locks. If the algorithm skips garbage collection on commit, garbage collection can be invoked any time later in the background; this is not shown in the code.

The algorithm depends on a policy of what locks to acquire, how to pick one of many possible commit timestamps, and whether to garbage collect during commit; these choices can depend on the transaction and other considerations. The choices are determined by the functions that we mentioned above: WRITE-LOCKS, READ-LOCKS, COMMIT-LOCKS, COMMIT-TS, and COMMIT-GC. The generic MVTL algorithm uses a generic policy that makes these choices nondeterministically (Algorithm 2). For example, to obtain write locks, the generic policy nondeterministically picks a set T of timestamps to lock. To obtain read locks, the policy picks an interval of timestamps starting immediately after a committed version.

We prove that the generic MVTL algorithm is correct with its nondeterministic choices. Naturally, this correctness carries over to any specialization that fixes the nondeterministic choices. These specializations lead to different algorithms (§5).

Some policies of the generic algorithm may cause deadlocks, where a process waits forever to acquire a lock. In such cases, standard techniques for deadlock detection can be used to abort the required transactions (e.g., cycle detection in the wait-for graph, timeout, etc). In the full paper, we show the following:

THEOREM 1. *The generic MVTL algorithm (Algorithms 1 and 2) ensures serializability.*

5 SIMPLE MVTL ALGORITHMS

We now give several simple algorithms that are instances of the generic MVTL algorithm, each with a different benefit. To specify these algorithms, we specialize the generic policy of MVTL (Algorithm 2). Due to limited space, we omit proofs and the detailed pseudo-code of some algorithms; they are provided in the full paper.

5.1 The preferential algorithm

Roughly speaking, our preferential algorithm, denoted MVTL-Pref, works with multiple timestamps for each transaction, where one of the timestamps is preferential. The algorithm tries to commit a transaction using its preferential timestamp, but if doing so would abort, it tries one of the other timestamps. To ensure viability of the other timestamps, the algorithm locks them as necessary during the execution.

More precisely, MVTL-Pref is parameterized by a function $A(t)$ that takes the transaction's preferential timestamp and returns a non-empty set of alternative timestamps different from t. $A(t)$ is a choice of the user of the algorithm. For example, $A(t) = \{t{-}10, t{+}10\}$ indicates that $t{-}10$ and $t{+}10$ are the alternative timestamps for a transaction with preferential timestamp t. The preferential timestamp itself comes from a clock, as in other timestamp-based protocols.

We assume that clock timestamps are unique (e.g., by appending the process id to each timestamp t) and that $A(t)$ also produces unique timestamps (e.g., by using the process id in t for each timestamp in $A(t)$).

When executing a read on a key k, the algorithm determines a version to return based on the preferential timestamp, and then read-locks contiguous timestamps of k to cover as many alternative timestamps as possible. When executing a write to key k, the algorithm obtains no locks; rather, locks are acquired at commit time, as follows. If the algorithm cannot obtain a write-lock for the preferential timestamp for each written key, it tries one of the alternative timestamps. If it manages to obtain read- and write-locks for all read and written objects at one of the timestamps, the transaction commits; otherwise it aborts.

Algorithm 3 The MVTL-Pref algorithm

```
function INITIALIZATION(tx)
    $tx.PrefTS \leftarrow clock()$
    $tx.PossTS \leftarrow \{tx.PrefTS\} \cup A(tx.PrefTS)$
                                        ▷ possible timestamps for tx
function WRITE-LOCKS(tx, k) return ▷ lock write-set only on commit
function READ-LOCKS(tx, k)
    repeat
        $tr \leftarrow \max\{t : t < tx.PrefTS$ and $Values[k, t] \neq \bot\}$
                                        ▷ candidate value to read
        $tmax \leftarrow \max\{t \in tx.PossTS :$
                no timestamps in $[tr+1, t]$ are write frozen$\}$
        for $t \leftarrow tr+1$ to $tmax$ do   ▷ read-lock $[tr+1, tx.TS]$ if possible
            try to acquire read-lock for tx on $(k, t)$, waiting
                if timestamp is write-locked but not frozen
            if found frozen write-lock then
                release read-locks acquired above; break ▷ exit "for" loop
    until found no frozen locks in the for loop
    $tx.PossTS \leftarrow tx.PossTS \cap [tr, tmax]$   ▷ update possible timestamps
    return tr
function COMMIT-LOCKS(tx)
    for $t \in tx.PossTS$ do   ▷ Find a good timestamp. Loop order: first
                               tx.PrefTS then arbitrary for PossTS
        gotlocks ← true
        for $(k, tr) \in tx.writeset$ do
            try to write-lock for tx on $(k, t)$, without waiting if a
                timestamp is read-locked
            if write-lock not acquired then
                gotlocks ← false        ▷ this timestamp will not work
                release all write locks for tx
                break                   ▷ exit inner "for" loop
        if gotlocks then break ▷ found a timestamp for which we can
                                 get write locks; exit outer "for" loop
    if gotlocks then $tx.TS \leftarrow t$   ▷ found good timestamp
    else $tx.TS \leftarrow \bot$            ▷ no good timestamps
function COMMIT-TS(T) return tx.TS
function COMMIT-GC(tx) return false
```

The pseudo-code of MVTL-Pref is given in Algorithm 3. We can show that if we choose the alternative timestamps $A(t)$ to be smaller than the preferential timestamps t, then the resulting MVTL-Pref algorithm aborts strictly fewer workloads compared to MVTO+. More precisely, we have the following:

Theorem 2. *Suppose that $\forall t' \in A(t), t' < t$. (a) If a workload W produces no abort under MVTO+, then W produces no abort under MVTL-Pref. (b) There are infinitely many workloads that produce no aborts under MVTL-Pref but produce aborts under MVTO+.*

5.2 The prioritizer algorithm

Multiversion timestamp ordering provides no way for critical transactions to be prioritized over normal transactions. We explain how MVTL can do that, by using a policy that gives more locks to critical transactions. There are many ways to do that, but the simplest one is as follows. Normal transactions obtain their locks as in multiversion timestamp ordering using synchronized clocks, while critical transactions try to acquire all locks as in pessimistic concurrency control except that critical transactions do not block waiting for any of its locks. Both types of transactions garbage collect on commit. We give the detailed pseudo-code of the algorithm in the full paper.

Theorem 3. *In the MVTL-Prio algorithm, transactions labeled critical are never aborted by transactions labeled normal.*

Given that high-priority transactions behave similarly to pessimistic concurrency control, they can cause deadlocks. However, transactions with normal priority behave identically to those in MVTO+, and thus never cause deadlocks.

5.3 The ϵ-clock algorithm

Multiversion timestamp ordering uses clocks to obtain its timestamps, but if clocks are not synchronized or monotonic[1], it is susceptible to *serial aborts*—aborts that occur in an execution that is completely serial. This is a concern in modern multicore machines that do not guarantee that clocks across cores are perfectly synchronized. For example, T_2 gets timestamp 2, reads an object X, and commits. Afterwards, T_1 gets a smaller timestamp 1, writes X, and tries to commit. This will cause T_1 to abort since the read-timestamp of X at version 0 is 2. This is the schedule:

$$\begin{array}{lllll} T_2: & R(X) & C & & \\ T_1: & & & W(X) & A \end{array}$$

Here, time flows to the right and each line shows the operations of a transaction. R, W, C, and A indicate a read, write, commit, and abort; and X is the key. Thus, this schedule has two transactions T_1 and T_2, where T_2 reads X and commits, and then T_1 writes X and aborts.

The MVTL-ϵ-clock algorithm, which we now introduce, avoids serial aborts when used with ϵ-synchronized clocks. Briefly, when it starts, a transaction reads the clock, obtains a time t, and for each read and write tries to lock the interval $[t-\epsilon, t+\epsilon]$. At the end, it commits at the smallest common timestamp it locked for every accessed object. Before completing the commit, the transaction runs garbage collection. Algorithm 4 shows the detailed pseudo-code.

In a sequential execution, it is possible to show that *tx* picks a commit timestamp that is at most t, and thus it releases the lock on higher timestamps. As a result, the next transaction in the sequence will always have its own real time in the intersection of locked timestamps, and therefore does not abort.

Theorem 4. *The MVTL-ϵ-clock algorithm is not susceptible to serial aborts when clocks are ϵ-synchronized.*

[1] A monotonic clock is one that ensures that it returns a higher timestamp if it is queried later in time. Monotonic clocks and time-synchronized clocks are equivalent for the purposes of this discussion.

Algorithm 4 The MVTL-ϵ-clock algorithm

```
function INITIALIZATION(tx)
    now ← clock()
    tx.TS ← [now − ε, now + ε]
function WRITE-LOCKS(tx, k)
    try to write-locks for tx on (k, tx.TS), waiting
        if a timestamp is read- or write-locked but not frozen
    tx.TS ← write-locks that tx could acquire
function READ-LOCKS(tx, k)
    if tx.TS = ∅ then return ⊥
    m ← max tx.TS
    repeat
        tr ← max{t : t < m and Values[k, t] ≠ ⊥}
        for t = tr+1 to m do ▹ read-lock interval [tr+1, m] if possible
            try to acquire read-lock for tx on (k, t), waiting
                if timestamp is write-locked but not frozen
            if found frozen write-lock then
                release read-locks acquired above; break ▹ exit "for" loop
    until found no frozen locks in the for loop
    tx.TS ← tx.TS ∩ [tr+1, m]
    return tr
function COMMIT-LOCKS(tx) return
function COMMIT-TS(T) return min T
function COMMIT-GC(tx) return true
```

5.4 Existing algorithms as special cases

We now show that MVTL generalizes two popular transactional algorithms, MVTO+ and pessimistic concurrency control. More precisely, we give two algorithms MVTL-TO and MVTL-Pessimistic, which specialize MVTL and behave exactly like MVTO+ and pessimistic concurrency control, respectively.

In MVTL-TO, each transaction obtains a timestamp t from a clock when the transaction starts. Writes do not lock anything, reads try to lock $[tr{+}1, t]$ (waiting for unfrozen locks) where tr is the largest timestamp before t for which $Values[k, tr] \neq \perp$, and commits lock t for each object in the transaction's write-set. Garbage collection is not invoked on commit. We give the detailed pseudo-code of the algorithm in the full paper.

THEOREM 5. *The MVTL-TO algorithm behaves as the MVTO+ algorithm.*

Pessimistic concurrency control locks objects before accessing them, to prevent conflicting operations from executing concurrently. To emulate pessimistic concurrency, the MVTL-Pessimistic algorithm works as follows. Writes acquire write locks on all timestamps (blocking), while reads acquire read-locks on all timestamps in $[tr{+}1, \infty]$ (blocking). Garbage collection is invoked on commit.

THEOREM 6. *The MVTL-Pessimistic algorithm behaves as the pessimistic concurrency control algorithm.*

5.5 The ghostbuster algorithm

Under multiversion timestamp ordering, a transaction may abort and later create a conflict with another transaction, causing it to abort. For example, suppose that T_1 starts with timestamp 1, T_2 starts with timestamp 2, and T_3 starts with timestamp 3. Then T_3 reads X and commits, T_2 reads Y, writes X, and tries to commit with its timestamp 2, but T_2 aborts because T_3 read X with timestamp 3. Next T_1 writes Y and tries to commit but aborts due to the read by T_2. This is a ghost abort, because the write of T_1 has a conflict with a transaction T_2 that had aborted before the write of T_1 started. This is the schedule:[2]

T_3 :	$R(X)$	C					
T_2 :			$R(Y)$	$W(X)$	A		
T_1 :						$W(Y)$	A

We define ghost aborts precisely in the full paper.

While multiversion timestamp ordering has ghost aborts, MVTL-Ghostbuster can avoid that. MVTL-Ghostbuster is a simple modification to the MVTL-TO algorithm (§5.4): when a transaction commits, it performs garbage collection. This ensures that transactions that abort do not leave behind locks that cause ghost aborts. We thus have the following:

THEOREM 7. *The MVTL-Ghostbuster algorithm is not susceptible to ghost aborts.*

6 PRACTICAL CONSIDERATIONS

Reducing lock state space. When we presented the generic MVTL algorithm, we defined a lock for each timestamp and object, which amounts to an infinite lock state space. We did not include mechanisms to compress this information, because they are orthogonal to the essence of the algorithm. However, a practical implementation should compress the lock state. To do so, we observe that MVTL algorithms usually acquire and release locks on a small number of points or contiguous intervals (this is true for all MVTL algorithms in this paper). Rather than keeping a lock state for each timestamp, an implementation can keep a single lock state for an entire interval. In the algorithms we presented, each object holds at most one lock interval per committed transaction. We evaluate the amount of lock state in §8.4.5. Furthermore, this state can be discarded when the associated version of the object is purged, as we discuss next.

Purging versions. By nature, a multiversion algorithm keeps multiple versions of each object. Doing so is feasible as storage prices fall. Disk systems such as database systems already use multiversion algorithms, but even memory systems can do so now. Nevertheless, multiversion algorithms need a way to purge old versions so that each object holds few versions—possibly just one after write activity on the object quiesces. We now explain how this can be done in MVTL. This is easy: at any time, the system can purge any version older than the latest committed one, without affecting the correctness of the algorithm. Transactions that need purged versions will abort, so in practice we purge versions older than a time limit chosen based on the duration of the longest expected transactions. In some MVTL algorithms, there is a lower bound on the timestamps that a transaction locks (e.g., ϵ-clock algorithm); we can purge versions with timestamps below the bound except the last one before the bound, without causing any side-effects. We evaluate the effectiveness and cost of garbage collection in §8.4.5.

Removing the atomic block. Algorithm 1 has an atomic block in lines 17–19, to avoid partially exposing the writes of a committing transaction when we assign to the array $Values[k, t]$. We can remove

[2] Here, transactions get a timestamp before their first operation, but one can construct a more complex schedule with the same problem even if transactions get a timestamp at the first operation.

this atomic block by (1) first storing a special value in $Values[k, t]$ for all timestamps in the for loop, (2) then storing the actual value v for all timestamps in the loop, and (3) having other processes wait if they read *Values* and see the special value.

7 DISTRIBUTED MVTL ALGORITHM

The MVTL algorithm of §4 works in a single machine, where multiple threads share lock state and data versions. We now explain how to extend that algorithm to distributed transactions, where clients in different machines execute transactions over the same data set. More precisely, the system consists of a set of clients who want to execute transactions, and a set of storage servers who keep the data, where clients and servers can be in different machines connected by a network. The data is partitioned across the servers by its key, and clients know how to find the server responsible for a key (e.g., by hashing the key or using a configuration map).

The basic idea of the distributed algorithm is that servers hold the state that is shared across clients: locks and data versions. Clients contact the servers to execute the steps of the algorithm in §4 that involve this state. More precisely, the server responsible for a key k keeps all versions and locks for k. A client contacts that server when it wishes to read k, create a new version for k, or manipulate k's lock state (obtain, freeze, or release locks on timestamps).

The system is subject to failures that may disrupt the algorithm. A failed client may leave write locks in an unfrozen state indefinitely, causing other transactions to block. A failed server can similarly cause indefinite waiting from clients.

To address these problems, we associate a *commitment* object with each transaction. This object solves consensus on the outcome of a transaction, which can be "abort" or "commit with a timestamp t", ensuring that clients and servers all agree on the outcome. The details of the algorithm are given in the full paper.

8 EXPERIMENTAL EVALUATION

We conduct a simple experimental evaluation of MVTL to answer some questions: Does MVTL enhance transaction concurrency and avoid aborts compared to alternatives? Does MVTL improve transaction throughput? On which workloads? Do the characteristics of the environment impact our conclusions? Does MVTL incur significant overheads in terms of state size?

To this end, we implement the distributed MVTL algorithm (§7) with a variant of the ϵ-clock algorithm (§5.3). In this variant, to execute a transaction T, a client obtains a timestamp t from its local clock and associates a timestamp interval $I = [t, t + \Delta]$ with T, where Δ is a small constant (we pick $\Delta = 5$ms in the experiments). When accessing a key k, the client tries to lock the timestamps in I for key k. If the client cannot lock the entire interval I, but manages to lock some subinterval, then the client replaces I with that subinterval to reduce the amount of locking on subsequent keys. We call this algorithm MVTIL. This is similar to the ϵ-clock algorithm but we do not assume that clients have synchronized clocks and we shrink I when clients fail to obtain some locks, as described above. We consider two variants of MVTIL: (i) *MVTIL-early*, which at commit time picks the smallest timestamp in I to commit, and (ii) *MVTIL-late*, which picks the largest. We compare MVTIL to 2PL and MVTO+.

8.1 Implementation details

Keys and values are small strings of eight characters. Clients are multi-threaded, each thread running a different transaction. When a client realizes that an ongoing transaction will abort (because it does not have a single timestamp locked across all accessed keys), it has the option of aborting or restarting the transaction, with an interval I adjusted based on the state it has already seen at the servers. Servers are multi-threaded, with hundreds of threads, each responsible to handle a client request. A server stores version and lock state in a hash table indexed by key; for each key, the hash table stores two skip lists, one for version state, one for lock state. The version state is a list of value-timestamp pairs ordered by timestamp. The lock state is a list of timestamp-timestamp pairs representing a locked time interval, ordered by the first timestamp. To coordinate access across threads, we use a concurrent hash table (from the Intel TBB library [14]), with a latch per entry in the hash table. Latches are held while a thread changes the lock and version lists of a key. We use Apache Thrift [2] for communication between clients and servers,

A timestamp service periodically broadcasts a message with a time T in the past, equal to the service's current time minus a constant K; we use $K = 15s$ in the local test bed, and $K = 60s$ in the cloud test bed (§8.2). This message has two effects. First, it causes servers to purge old versions of keys, namely versions that meet two criteria: their timestamp is smaller than T and they are not the most recent version of a key. If clients have ongoing transactions that later try to access a purged version, those transactions are aborted. However, because T is an old timestamp, there will be few such transactions, if any. The second effect of broadcasting T is that clients advance their local clocks to T if they are behind—this ensures that clients with slow clocks do not start new transactions that need purged versions and subsequently get aborted.

Our implementations of MVTO+ and 2PL use the same framework, but run a different client protocol and keep a different server state: 2PL stores a single readers-writer lock per key, while MVTO+ stores a single skip list per key containing versions and associated locks. The implementations of all schemes are available at https://github.com/LPD-EPFL/MVTIL.

8.2 Test beds

We use two test beds for the experiments: a local test bed with dedicated servers, and a public cloud test bed with virtual machine instances. The local test bed represents an enterprise setting with higher-performance machines and network, while the cloud test bed represents a low-cost shared environment with a less predictable network.

On the local test bed, we use three machines: (a) a server with four 2.7 GHz Intel Xeon 12-core E7-4830v3 processors and 512 GB of RAM; (b) a server with two 2.8 GHz Intel Xeon 10-core E5-2680 v2 processors and 256 GB of RAM; and (c) a server with four 2.1 GHz AMD Opteron 6172 12-core Processors and 128 GB of RAM. Machines are connected by a 1 Gbps network.

The public cloud test bed consists of several hundred Amazon EC2 *t2.micro* instances with 1 vCPU each.

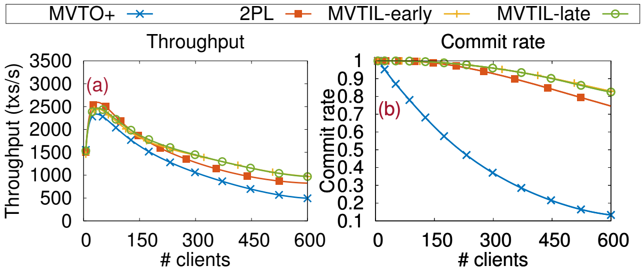


Figure 1: Effect of concurrency level on performance, local test bed.

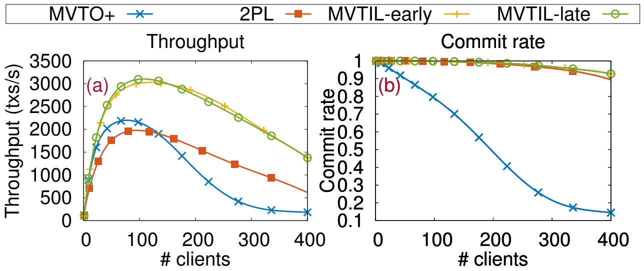


Figure 2: Effect of concurrency level on performance, cloud test bed.

8.3 Experiments

In an experiment, clients submit transactions repeatedly in a closed-loop. We measure the aggregate throughput of committed transactions and the commit rate, which is the fraction of transactions that commit. Before measuring, we run a warm-up stage of 40s to ensure all clients have started; we then measure the system for 20s. We repeat each experiment five times and report the average.

In each experiment, we fix the following parameters:

- algorithm (MVTIL, MVTO+, 2PL);
- number of clients, which determine the level of concurrency;
- size of transactions in number of operations;
- fraction of write operations in a transaction;
- size of the key space; and
- number of storage servers.

On the local test bed, which has three machines, we always run three servers on all machines, and we run client threads on a subset of the cores in those machines. For the cloud test bed, we run eight servers on eight different VM instances unless otherwise indicated, and we run each client on its own VM instance.

8.4 Results

We now present results regarding concurrency, fraction of write operations, transaction size, number of servers, and state size.

8.4.1 Level of concurrency. We study the effect of the level of concurrency on performance, under a workload where a majority of operations are reads—a common situation in practice. We vary the number of clients, while keeping the other parameters constant. We use transactions with 20 operations, 25% of which are writes. For the local test bed, we use 10K keys. For the larger cloud test bed, we use 50K keys.

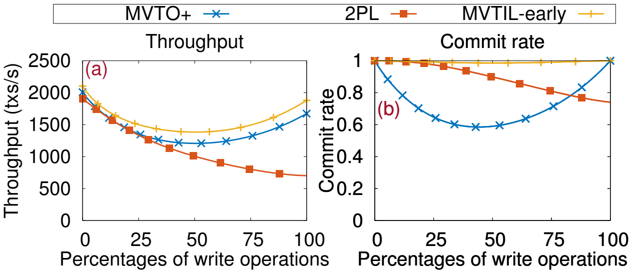


Figure 3: Effect of fraction of writes on performance.

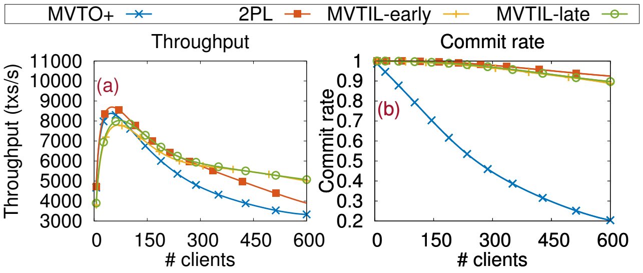


Figure 4: Effect of small transaction size on performance.

Figures 1 and 2 show throughput and commit rates for the local and cloud test beds, respectively. We see that MVTIL outperforms MVTO+ and 2PL in both test beds. Moreover, when concurrency increases, the commit rate of MVTO+ drops due to conflicts, but this does not happen for MVTIL because it can commit at many serialization points. The inefficiency—due to aborts in MVTO+ and waiting for locks in 2PL—is the reason for the difference in throughout. This is more pronounced in the cloud test bed, where resources are scarce: there, MVTIL has roughly 2x better throughput than the alternatives. The difference is smaller on the local test bed.

The commit rate for 2PL is not 100% because transactions abort after a timeout. This prevents deadlocks and starving transactions from limiting throughput. We set the timeout value to maximize throughput.

8.4.2 Write percentage. We next consider how the fraction of writes affect performance. We thus vary the fraction of writes and keep other parameters constant. We use the local test bed with 90 clients; transactions have 20 operations and 10K keys.

Figure 3 shows the results. For read-only transactions, the choice of protocol has little impact. Additionally, for write fractions close to 1, the workload consists mostly of blind writes, which allows multiversion protocols to commit nearly all transactions, as writes in such protocols do not conflict with each other. With a more balanced write fraction, MVTIL outperforms MVTO+ and 2PL. With 2PL, the more writes, the more time transactions wait for locks. MVTO+ has a high abort rate when the percentages of reads and writes are similar; this is where the chance of conflicts is highest in multiversion protocols. The issue impacts MVTIL less due to its ability to explore many serialization points to commit.

8.4.3 Transaction size. In previous experiments, we use transactions with 20 operations; we now consider smaller transactions with 8 operations. We vary the number of clients (level of concurrency) and observe the performance. We use the local test bed with a 50% fraction of writes and 10K keys.

Figure 4 shows the results. Under low concurrency, MVTIL behaves similar to MVTO+ and 2PL, but 2PL is ≈5% faster. This setting

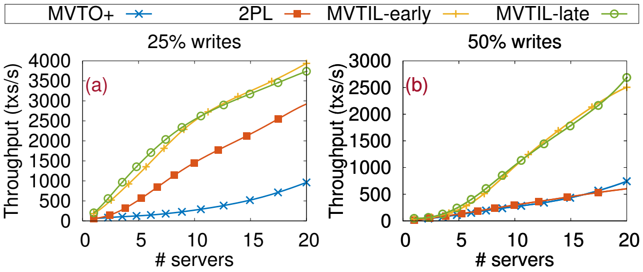


Figure 5: Effect of number of servers on performance.

with little concurrency, short transactions, and a local test bed with lots of resources is the only setting where MVTIL is worse than an alternative. However, as we increase concurrency, MVTIL again outperforms the others. This advantage is larger in the cloud test bed (not shown).

8.4.4 Number of servers. We now consider how the number of servers affect performance. Using the cloud test bed, we keep the number of clients constant to 400 and vary the number of server instances from 1 to 20. We use transactions with 20 operations with 25% or 50% writes, and 100K keys.

Figure 5 shows the results. The throughput of all protocols increases with the number of servers, but the scalability is better for MVTIL. MVTIL has a higher commit rate compared to MVTO+, and waits less for locks compared to 2PL; this is particularly visible with 50% writes.

8.4.5 State size. We now examine the size of the state kept by each algorithm, and the effectiveness of the garbage collection mechanisms. Most of the state of a multiversion protocol is the data versions and associated locks. We measure how the number of versions and locks evolve with time for MVTIL and MVTO+ without garbage collection, as well as MVTIL with garbage collection (MVTIL-GC) that activates every 15s to purge versions and locks. We use 50 clients running transactions with 20 operations, a fraction of 50% writes, and 8K keys, running on the local test bed.

Figure 6 shows the results. Without garbage collection, the state increases linearly with time. However, with garbage collection, the state size remains bounded in both the number of versions and locks. On average, there are ≈4 versions and ≈20 locks per key. Figure 7 shows how performance varies as time passes. Without garbage collection, throughput decreases after ≈5 minutes for MVTIL and MVTO+, because a larger state makes it slower to search for and access versions. Garbage collection removes this performance degradation. Moreover, comparing the performance with and without garbage collection at the beginning of the experiment, we see that the overhead of garbage collection is small.

8.5 Summary

We see that (i) with moderate contention, MVTIL outperforms alternatives, (ii) with no contention, MVTIL is at least as good as alternatives, and (iii) MVTIL's advantages are bigger in the cloud test bed that has limited processing power and unpredictable network latencies. MVTIL nevertheless represents just one of many MVTL-based algorithms. We believe that other MVTL algorithms will present different benefits on other workloads and environments.

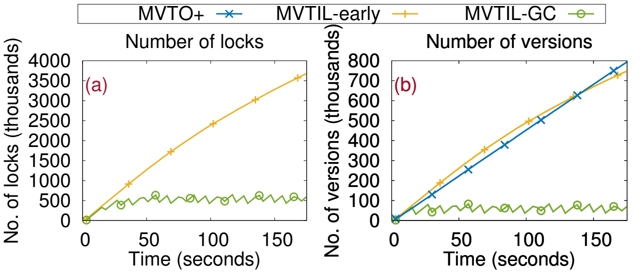


Figure 6: Number of locks and versions as time passes with garbage collection on and off.

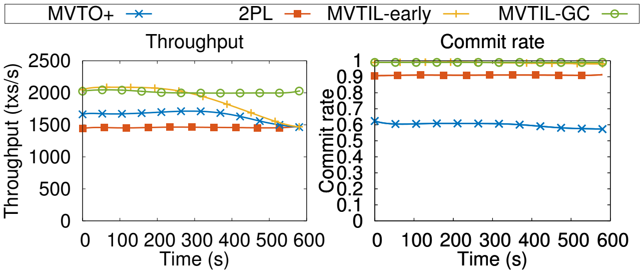


Figure 7: Performance as time passes with garbage collection on and off.

9 RELATED WORK

The main novelty of this work is the idea of locking individual timestamps, leading to a genre of multiversion algorithms called MVTL. No other work has proposed this idea, but because MVTL is a broad class, several existing algorithms become special cases of MVTL, leading to similarities in mechanism.

Multiversion concurrency control is an old idea [6] that has seen a resurgence in software transactional memory (STM) systems, several of which provide serializability [3, 7, 11, 15, 16, 20, 22–24]. Prior work in this space falls into three categories: (1) multiversion for read-only transactions, (2) conflict graph schemes, and (3) multiversion timestamp ordering algorithms. The first category [11, 22–24] are systems that use multiversion to benefit solely read-only transactions; update transactions rely on optimistic methods that, upon commit, validate the read-set and abort if any object has changed. While read-only transactions are important, these methods abort under simple concurrent update schedules, such as the following (where full multiversion schemes do not abort):

$$\begin{array}{llll} T_1: & R(X) & & W(Y) \\ T_2: & & W(X) & \end{array}$$

The second category (e.g., [3, 16, 20]) are multiversion systems that ensure serializability by detecting cycles in the *conflict graph*—a data structure that represents the conflicts across transactions—similarly to the MVSGT algorithm [26]. These algorithms have two drawbacks: they are complex and they incur significant computation overhead, as reported in some of these papers.

The third category are systems that extend multiversion timestamp ordering. Specifically, Kumar et al. [17] explain how to provide opacity, which is stronger than serializability. The algorithm suffers from the same drawbacks of multiversion timestamp ordering that we address in §5. It should be possible to extend MVTL to provide opacity using the ideas of Kumar et al., but this is future work.

Lomet et al. [19] introduce the multiversion timestamp range algorithm (MVTR). With MVTR, each transaction is assigned a range of timestamps, and this range shrinks as the transaction executes; at the end, MVTR commits if the range is non-empty. MVTR differs from MVTL because MVTR locks entire objects instead of timestamps. As a result, MVTR does not enjoy the full benefits of multiversion concurrency control, such as allowing two concurrent transactions to write the same object. Also, with MVTR one transaction manipulates the inner state of another transaction (e.g., by changing the range that another transaction uses), which requires careful synchronization of transactions using a scheduler or locks. Elastic transactions [12], aimed at search data structures, use timestamp ranges to determine if a transaction can commit based on its start time and when the accessed objects were written.

Snapshot isolation [4] is both an isolation property and a protocol. The protocol uses multiversioning and timestamps, similarly to multiversion timestamp ordering, but it does not provide serializability. Other protocols that use multiversioning and timestamps provide even weaker notions than snapshot isolation [25].

Optimistic concurrency control (OCC) [18] is another technique that can use multiversioning. With OCC, a transaction does not acquire locks when executing; to commit, the system checks that the versions that the transaction read are the latest. TicToc [28] optimizes OCC to serialize transactions based on the data they access. TicToc computes potential serialization points before the validation and commit phases. Thus, a transaction for which the read and write sets have been inspected might later abort. In contrast, MVTL ensures that once a serialization point has been found, the transaction commits. Bohm [10] is a multiversion protocol that pre-orders transactions before execution; in that sense, Bohm is more pessimistic than MVTL, which determines transaction ordering dynamically during execution. In addition, Bohm requires that the transaction be known ahead of time, and that its write-set be static.

Many practical systems with distributed transactions provide only snapshot isolation [9, 21] and abort on concurrent writes to the same object. Spanner [8] provides strict serializability using two-phase locking for read-write transactions, which limits parallelism.

10 CONCLUSION

This paper introduces a new genre of multiversion concurrency control algorithms called multiversion timestamp locking (MVTL). MVTL offers a new way to look at multiversion algorithms, based on locking individual timestamps. With this perspective, we find simple algorithms that improve the state of the art in different ways: by committing successfully more workloads than existing multiversion protocols, by avoiding the problems of serial aborts and ghost aborts, and by offering prioritized transactions. We can also view existing algorithms, such as MVTO and pessimistic concurrency control, as special cases of MVTL. We show how to realize MVTL in both centralized and distributed systems. Finally, we show experimental evidence of the benefits of MVTL in practice.

We believe that the algorithms proposed here are only a starting point for other possibilities opened up by MVTL. The design of other MVTL algorithms is a promising direction for future research.

Acknowledgements. We thank Dahlia Malkhi for valuable input early in the project. This work was supported in part by a VMware Fellowship.

REFERENCES

[1] M. K. Aguilera, T. David, R. Guerrraoui, and J. Wang. Locking timestamps versus locking objects. Technical report. https://infoscience.epfl.ch/record/229425, 2018.

[2] Apache thrift. https://thrift.apache.org.

[3] U. Aydonat and T. S. Abdelrahman. Serializability of transactions in software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data*, pages 1–10, May 1995.

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

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[7] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, Dec. 2006.

[8] J. 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 *Symposium on Operating Systems Design and Implementation*, pages 251–264, Oct. 2012.

[9] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE Symposium on Reliable Distributed Systems*, pages 173–184, Oct. 2013.

[10] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

[11] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[12] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. *Journal of Parallel and Distributed Computing*, 100(C):103–127, Feb. 2017.

[13] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised First Edition.* Morgan Kaufmann, 2012.

[14] Intel TBB. https://www.threadingbuildingblocks.org.

[15] I. Keidar and D. Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications.* Springer, 2015.

[16] I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. *ACM Transactions on Computer Systems*, 57(1):261–285, July 2015.

[17] P. Kumar, S. Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *International Conference on Distributed Computing and Networking*, pages 212–226, 2014.

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

[19] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. In *International Conference on Data Engineering*, pages 714–725, Apr. 2012.

[20] J. Napper and L. Alvisi. Lock-free serializable transactions. Technical Report TR-05-04, University of Texas at Austin, Feb. 2005.

[21] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Symposium on Operating Systems Design and Implementation*, pages 251–264, Oct. 2010.

[22] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective multi-versioning STM. In *International Symposium on Distributed Computing*, pages 125–140, Sept. 2011.

[23] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *ACM Symposium on Principles of Distributed Computing*, pages 16–25, July 2010.

[24] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*, pages 284–298, Sept. 2006.

[25] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles*, pages 385–400, Oct. 2011.

[26] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann, 2001.

[27] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, Mar. 2017.

[28] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. TicToc: Time traveling optimistic concurrency control. In *International Conference on Management of Data*, pages 1629–1642, June 2016.