Verifying vMVCC, a high-performance transaction library using multi-version concurrency control

Yun-Sheng Chang, Ralf Jung,[†] Upamanyu Sharma,
Joseph Tassarotti,[▽] M. Frans Kaashoek, and Nickolai Zeldovich

MIT CSAIL [†] *ETH Zurich* [▽] *New York University*

Abstract

Multi-version concurrency control (MVCC) is a widely used, sophisticated approach for handling concurrent transactions. vMVCC is the first MVCC-based transaction library that comes with a machine-checked proof of correctness, providing clients with a guarantee that it will correctly handle all transactions despite a complicated design and implementation that might otherwise be error-prone. vMVCC is implemented in Go, stores data in memory, and uses several optimizations, such as `RDTSC`-based timestamps, to achieve high performance (25–96% the throughput of Silo, a state-of-the-art in-memory database, for YCSB and TPC-C workloads). Formally specifying and verifying vMVCC required adopting advanced proof techniques, such as logical atomicity and prophecy variables, owing to the fact that MVCC transactions can linearize at timestamp generation prior to transaction execution.

1 Introduction

Applications routinely rely on databases not just for storing data durably on disk, but also for ensuring that transactions execute atomically despite concurrency and crashes. This simplifies application development, because the application developer no longer has to worry about concurrency bugs or partial state left over after a crash. Indeed, this pattern is so ubiquitous that it is common for cloud providers to offer databases as a black-box service to application developers. In this model, application correctness and performance crucially hinges on the database system correctly handling all possible corner cases and doing so efficiently.

Achieving both correctness and high performance in a database system for many concurrent transactions is challenging. In particular, when transactions read and write an overlapping set of data items, the database system must ensure the transactions appear to execute in a serial order. A widely used technique for improving performance in this setting is *multi-version concurrency control*, or MVCC [8, 30, 35, 36], in which the database stores not just the latest version of a data item, but also past versions. Storing past versions allows the database system to execute writes that add a new version, while also being able to use the older versions to execute reads from transactions that appear to execute earlier in the serial order.

Multi-version concurrency control requires a sophisticated implementation of its data structures, in order to efficiently track multiple versions of each tuple, implement garbage collection (GC), etc. The implementation must also employ low-level optimizations to get high performance. For instance, using a mutex on a shared counter to get a unique ID for each transaction is too costly, and highly scalable implementations must use contention-free approaches such as relying on the CPU timestamp counter. The end result, therefore, is a complex implementation that can have bugs leading to incorrect or non-serializable executions. These bugs can be costly: they can cause data to be lost or corrupted; they can lead to many applications being affected; and tracking down bugs in the database system can be difficult for application developers.

This paper presents vMVCC, a high-performance MVCC-based transaction library with a formal specification and a machine-checked proof of correctness. vMVCC addresses the core technical challenges faced by the transaction layer in a database, and can be used to build transactional applications. Verifying vMVCC requires addressing several challenges. First, we must formalize a specification that captures the guarantees provided by MVCC transactions in a concise manner. Second, we must develop proof techniques to show that MVCC achieves a serializable execution order in the presence of concurrency. Finally, we must be able to formally reason about high-performance implementations that use low-level programming techniques such as sharded data structures, accessing the CPU timestamp counter with an `RDTSC`-like instruction, etc.

The key technical challenge addressed in vMVCC lies in dealing with the fact that MVCC's linearization point happens before the transaction body runs—the linearization point is when the timestamp is obtained in `Begin()`. This makes it challenging to verify MVCC-based transactions because, at the linearization point, the transaction has not executed yet, and the proof does not know what data the transaction is going to write or whether it is going to commit or abort. However, it is important for the specification and proof to update the abstract state of the system at the linearization point, because subsequent transactions must observe these changes. In contrast, under two-phase locking, a transaction linearizes at the point when it commits, where it is well known what state the transaction modified and that it is about to commit.

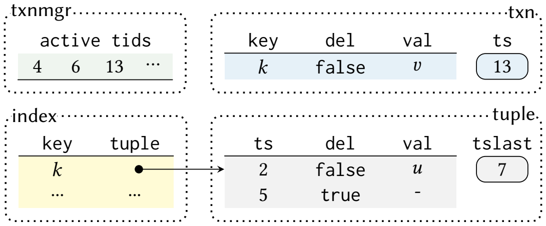


Figure 1: Overview of the main vMVCC data structures. Implementation details are not shown on the figure (e.g., the index and the active transaction IDs are partitioned into multiple shards for scalability).

vMVCC addresses this challenge by adopting *prophecy variables* [1, 18]. Our use of prophecy variables allows the proof to speculatively predict what state the transaction is going to modify and whether it will commit. This translates into the proof considering every possible prediction, allowing vMVCC to update the abstract state accordingly at the linearization point. As the transaction is about to commit, the proof can check whether the prediction was correct or not, and either stop considering further an incorrect prediction, or continue with a correct prediction. vMVCC is not the first to develop or use prophecy variables—many earlier frameworks developed support for them and proved that they are a sound proof technique—but it is the first to prove the correctness of MVCC-based transactions.

We implemented vMVCC in Go, and verified it using the Goose and Perennial frameworks. vMVCC implements sophisticated optimizations such as the use of RDTSC to generate strictly increasing timestamps, on-the-fly GC of past versions, and efficient data structures for storing multiple versions. vMVCC provides a transactional key-value store interface, similar to Silo [35]. For the YCSB benchmark with 32 worker threads, vMVCC achieves an aggregated throughput of 18.6M–52M transactions per second, which is 38–96% of that achieved by the unverified Silo database. For TPC-C, vMVCC achieves a throughput of 10.7K–33K transactions per second per warehouse, which is 25–43% of Silo's throughput.

The key technical contribution of vMVCC lies in demonstrating how to formally reason about transactions whose linearization point precedes the execution of their transaction body, using prophecy variables. This verification technique would be applicable to any system that uses MVCC [8, 10–12, 14, 19, 27, 30, 32, 35–37]. The second contribution is vMVCC itself, the first verified MVCC transaction library. The vMVCC artifact is interesting in its own right, providing a high-assurance and high-performance implementation, and can be used as a Go package independent of verification. vMVCC includes several other technical contributions, including a verified algorithm for computing strictly increasing transaction IDs using RDTSC, and a precise specification of a transaction library interface using logical atomicity [16].

One of the limitations of vMVCC is that it does not implement durability. In-memory databases are widely used in practice, but we do plan to extend vMVCC to store data durably on disk so that it persists across crashes, and to formally verify it using techniques from Perennial [3]. Another limitation of vMVCC is that it provides a simple key-value data model, as opposed to SQL's relational data, and does not support range scans.

2 Design and interface of vMVCC

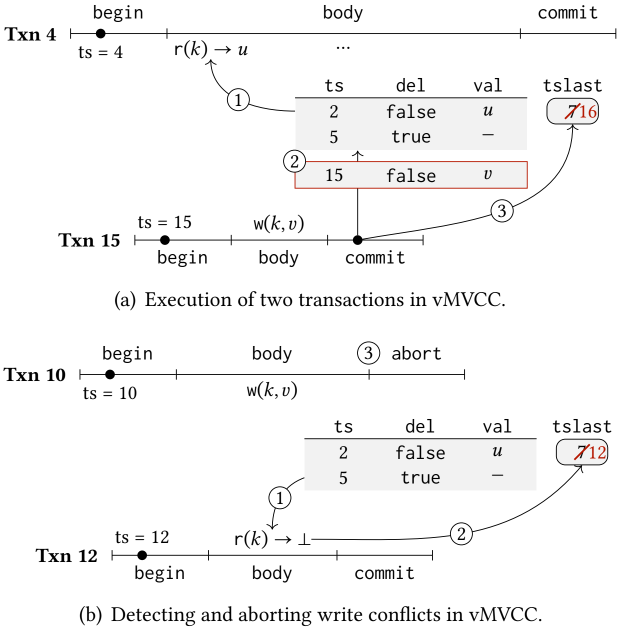
vMVCC is a transaction library, and applications interact with it through a standard interface for transactions, as follows (in Go syntax):

```
func (db  *DB)  Begin() *Txn
func (txn *Txn) Write(key K, value V)
func (txn *Txn) Delete(key K)
func (txn *Txn) Read(key K) (V, bool)
func (txn *Txn) Commit() bool
func (txn *Txn) Abort()
```

vMVCC uses an MVCC design closest to the original protocol as proposed by Reed [30] (also known as multi-version timestamp ordering [36]). The design is based around assigning a strictly increasing timestamp in Begin() to every transaction, and storing multiple versions for each key, corresponding to a range of timestamps for which that version is valid. When an application modifies a key, using Write(k,v) or Delete(k), the vMVCC transaction keeps track of the modification in a per-transaction write buffer. When an application invokes Read(k), the transaction first checks its local write buffer for pending writes to k; if there are no pending writes, it then searches from the global state the version of key k whose timestamp immediately precedes the transaction's timestamp. On successfully calling Commit(), the transaction creates a new version for each key in the write buffer with the transaction's timestamp as well. On calling Abort(), or a failed Commit(), the transaction drops its write buffer.

Read-only transactions always succeed in vMVCC because vMVCC retains all past versions required by active transactions (i.e., those that have begun but not yet committed or aborted). A transaction involving updates, however, might fail to commit if another transaction with a higher timestamp has read or updated the modified key in the meantime. The reason this requires aborting the first transaction is that, to achieve linearizability, the second transaction should have seen the update made by the first one, but it did not.

Data structures. Figure 1 shows the data structures that vMVCC uses to implement its design. The crux of multi-versioning lies in the data structure *tuple*, consisting of a list of versions, a tslast field to detect conflicts, and a mutex (not shown) used for synchronizing access to this data structure. Each version corresponds to a range of timestamps for which it is valid, represented by the ts field, which marks the start of the validity region. The version is valid until the next version's ts field, or, if this is the last version in the tuple, then it is the latest version. Each version also contains the value (val) and whether this key is deleted or not (del). The tslast field of each tuple represents the highest timestamp of any transaction that has read or written this tuple. It is



(a) Execution of two transactions in vMVCC.

(b) Detecting and aborting write conflicts in vMVCC.

Figure 2: Two example executions of concurrent transactions in vMVCC.

used to detect conflicts if a `Write` or a `Delete` with an earlier timestamp tries to commit later on. On top of tuples, vMVCC maintains its *index*, a hash map from keys to tuple pointers. Keys not present in the index are assumed to be not present (deleted) at every timestamp.

Every active transaction in vMVCC is represented using a *transaction object*, which consists of a unique timestamp `ts`, as well as a local write buffer keeping track of the modifications made by this transaction so far. When the transaction commits, it tries to acquire the mutexes of the tuples to be modified, and if successful, atomically applies the modification in its local writer buffer. Transaction IDs are generated by the *transaction manager*. For the purposes of GC, it also keeps track of the IDs of active transactions.

Execution examples. Figure 2(a) illustrates an example of two concurrent transactions accessing the same key k. The tuple in the example corresponds to key k. ① Txn 4 reads the value u from the tuple, as the timestamp of the corresponding version (i.e., 2) immediately precedes that of Txn 4. ② Txn 15 writes v to k by appending a new version tagged with its timestamp at commit time, and ③ increases `tslast` to 15 + 1, preventing transactions with timestamp below 16 from modifying this tuple.

This example also shows the concurrency advantages of MVCC over conventional concurrency control approaches such as two-phase locking (2PL) and optimistic concurrency control (OCC). With 2PL, Txn 15 cannot commit until Txn 4 commits, at which point the lock on k is released. With OCC, Txn 4 would have to abort as the value of k changes during the execution of Txn 4.

Figure 2(b) shows an example of how vMVCC detects and aborts conflicting writes. ① Txn 12 reads the second version of the tuple, and ② increases the `tslast` field to its timestamp. ③ Txn 10 attempts to commit and update the tuple, but fails because the timestamp of Txn 10 is less than `tslast` of the tuple (i.e., 12). Thus, Txn 10 aborts.

Garbage collection. To reclaim space occupied by unusable versions, vMVCC employs a garbage collector that runs in the background to remove those versions. The garbage collector must ensure that the versions it removes cannot be accessed by any transactions, including those that have not even begun. Concretely, the garbage collector first determines a lower bound on the transaction IDs of all active and future transactions. This lower bound can be computed by finding the minimal transaction ID among the active ones; if there are no active transactions, the current timestamp is used. Because timestamps are strictly increasing (as described below), the garbage collector can safely remove versions whose lifetime ends before that lower bound.

Generating timestamps with CPU timestamp counter. A key requirement for vMVCC is that every transaction is assigned a strictly increasing timestamp. However, assigning these timestamps by modifying a shared in-memory counter leads to contention on that counter. Instead, vMVCC uses the CPU timestamp counter (e.g., `RDTSC` on x86 machines) to generate timestamps in a scalable way. Modern hardware ensures that timestamps are monotonically increasing and consistent across cores and sockets [2].

One complication is that two threads running on different cores may obtain the same timestamp. vMVCC addresses this problem by using *transaction sites* to make transaction IDs unique. Each site has its own ID, which is a short integer value (e.g., from 0 to 63). When the transaction manager wants to assign a timestamp, it replaces the low bits of the timestamp counter with the site ID value. To ensure that the transaction manager does not use the same site for two transactions at the same time, vMVCC maintains an array of mutexes, one per site, and the transaction manager holds the site's mutex while computing the timestamp. The transaction manager can pick any site ID, such as the one associated with the local core. vMVCC takes a more flexible approach by assigning each thread a site ID in a round-robin manner. Having per-site mutexes ensures that the transaction manager does not contend when assigning timestamps on different sites.

Naïvely replacing the low bits of the timestamp counter with the site ID leads to subtle correctness issues. For example, Txn A may choose the highest possible site ID (all ones), quickly execute, and commit. Txn B, runs after Txn A but chooses the lowest possible site ID (all zeroes). The processor ensures that the `RDTSC` value seen by Txn B is higher than that seen by Txn A, but once the low bits are replaced with all-ones and all-zeroes, it may be that Txn B's transaction ID is lower than that of Txn A. One possible fix would be to represent the transaction ID as a tuple of the complete

64-bit RDTSC value and the site ID. However, since transaction IDs are used throughout vMVCC, this leads to a noticeable performance overhead.

Instead, vMVCC modifies the timestamp algorithm to ensure that timestamps are strictly increasing. To obtain a timestamp, the transaction manager first obtains t, the current RDTSC value, and then computes the next highest value $t' \geq t$ such that t' has the desired site ID in the low bits. The transaction manager then spins in a loop calling RDTSC until it returns a timestamp $t'' > t'$. The transaction manager then uses t' as the transaction's ID. The reason this loop-based design achieves strictly increasing transaction IDs is that the transaction manager is holding the site's mutex while the CPU timestamp counter passed through t'. This means no other thread could have generated the same transaction ID. In practice, of course, the loop runs for a few cycles at most, since the RDTSC value will quickly exceed the loop threshold.

Whole-transaction execution. For developer convenience, vMVCC provides an interface that wraps up the details of beginning, committing, and aborting a transaction, in db.Run, a higher-order function whose implementation is as follows:

```
func (db *DB) Run(body func(txn *Txn) bool) bool {
    t := db.Begin()
    commit := body(t)
    if commit {
        return t.Commit()
    } else {
        t.Abort()
        return false
    }
}
```

The developer provides the body of the transaction, which can use Read, Write, and Delete to access the system state. The transaction body returns a boolean to indicate whether it wants to commit or abort.

3 Using and specifying vMVCC

vMVCC is a transaction library that facilitates building and verifying applications by providing an atomic transaction abstraction. We begin with constructing on top of vMVCC an example application that atomically transfers some amount from one account to another, along the lines of what a bank application might do (§3.1). We then describe the formal specification of vMVCC and how to build arbitrary applications on top of it (§3.2).

3.1 Example: AtomicXfer

Figure 3 shows the implementation of AtomicXfer (ignore the inline proof for now). This code is implementing a simple bank, transferring amt from the src account to dst. If not enough funds are available in src, the transaction aborts. vMVCC ensures that the logical effect of the transaction body, xfer, appears to apply atomically. This frees the developer from worrying about other concurrent transactions that may affect the balance in src or dst, or about the versioning going on inside of vMVCC. vMVCC also ensures the transactions execute in a linearizable order, so that once AtomicXfer returns, any subsequent transactions will observe the effects of this AtomicXfer.

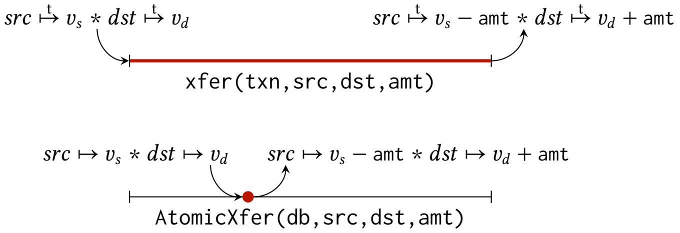
```
// { $src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d$ }
func xfer(txn *Txn, src, dst, amt uint64) bool {
    // { $src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d$ }
    sbal, _ := txn.Read(src)
    // { $src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d \wedge$ sbal $= v_s$ }
    if sbal < amt {
        // { $src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d \wedge$ sbal $<$ amt $\wedge \cdots$ }
        return false
    }
    // { $src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d \wedge$ sbal $= v_s \wedge$ sbal $\geq$ amt }
    txn.Write(src, sbal - amt)
    // { $src \overset{t}{\mapsto} v_s -$ amt $* dst \overset{t}{\mapsto} v_d \wedge \cdots$ }
    dbal, _ := txn.Read(dst)
    // { $src \overset{t}{\mapsto} v_s -$ amt $* dst \overset{t}{\mapsto} v_d \wedge$ dbal $= v_d \wedge \cdots$ }
    txn.Write(dst, dbal + amt)
    // { $src \overset{t}{\mapsto} v_s -$ amt $* dst \overset{t}{\mapsto} v_d +$ amt $\wedge \cdots$ }
    return true
}
// If returning false, then { $\top$ }
// Else { $src \overset{t}{\mapsto} v_s -$ amt $* dst \overset{t}{\mapsto} v_d +$ amt }

// $\langle src \mapsto v_s * dst \mapsto v_d \rangle$
func AtomicXfer(db *DB, src, dst, amt uint64) bool {
    body := func(t *Txn) bool {
        return xfer(t, src, dst, amt)
    }
    return db.Run(body)
}
// If returning false, then $\langle src \mapsto v_s * dst \mapsto v_d \rangle$
// Else $\langle src \mapsto v_s -$ amt $* dst \mapsto v_d +$ amt $\rangle$
```

Figure 3: Implementation and proof of AtomicXfer using vMVCC library.

Sequential reasoning in xfer. vMVCC formalizes the fact that the developer need not consider other concurrent transactions by allowing the developer to use sequential reasoning for the body of the transaction. To achieve this, vMVCC uses Iris [17], a modern concurrent separation logic (CSL) [29], to specify its interface. In Iris/CSL, threads can own logical *resources*, and resource ownership can be exclusive, meaning that if one thread owns a resource, no other thread can own the same resource. For example, the resource $k \mapsto v$ says that the value of k is v, and also says that the current thread *owns* k—that is, no other thread can own $k \mapsto v$ in the meantime (and thus no other thread can read or write k).

In our example, the proof of the transaction body, xfer, assumes ownership of $src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d$; the $*$ operator ("separating conjunction") says the thread owns both resources and they are disjoint. Having ownership of these resources allows the proof to assume that it is the only one accessing *src* and *dst*. This, in turn, allows the developer to prove xfer as if it was running in isolation, with no other concurrent transactions. The overall specification for xfer is that, starting with $\{src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d\}$, if xfer runs and terminates, then it either returns false to abort the transaction, or it returns true to commit, and the resources are



$src \overset{t}{\mapsto} v_s * dst \overset{t}{\mapsto} v_d$
$src \overset{t}{\mapsto} v_s - \mathtt{amt} * dst \overset{t}{\mapsto} v_d + \mathtt{amt}$
xfer(txn,src,dst,amt)
$src \mapsto v_s * dst \mapsto v_d$
$src \mapsto v_s - \mathtt{amt} * dst \mapsto v_d + \mathtt{amt}$
AtomicXfer(db,src,dst,amt)

Figure 4: Figurative specifications of xfer and AtomicXfer. We highlight the duration of owning the resources with red.

now $\{src \overset{t}{\mapsto} v_s - \mathtt{amt} * dst \overset{t}{\mapsto} v_d + \mathtt{amt}\}$. To prove this, the developer considers each line of code, and how that code affects the resources owned by the thread, as shown in the proof state comments between lines of code.

Concurrent specification for AtomicXfer. Specifying a function with exclusive resource ownership (like we did with xfer) simplifies the reasoning for that function, but at the cost of limiting its implementations to sequential ones—only the thread owning the required resources would be allowed to execute the function.

To specify the behavior of AtomicXfer in Iris/CSL without requiring ownership of src and dst for the entire duration of AtomicXfer, vMVCC uses the notion of *logical atomicity* [16]. Figure 4 shows the flow of resources in a logically atomic specification of AtomicXfer as compared to that of sequential xfer. The xfer specification says that the thread owns the resources throughout the entire execution of xfer, whereas in AtomicXfer, the specification says that there will be some point in time at which AtomicXfer appears to run atomically. One notable difference here is the kinds of resources appearing in the two specifications. Intuitively, the $k \overset{t}{\mapsto} v$ used by xfer says that "this transaction believes the value of k is v", whereas the $k \mapsto v$ used by AtomicXfer reflects "the actual value of k is v". We will explain the meaning of these resources in more depth in §3.2.

The resulting logically-atomic specification for AtomicXfer captures that any number of threads are allowed to concurrently invoke AtomicXfer, possibly with overlapping src and dst values. For each thread's invocation of AtomicXfer, the specification says that the transfer will execute correctly and atomically. The application can, in turn, prove that this maintains some application-level invariant, such as the sum of the balances of all accounts remains fixed.

Proving AtomicXfer. Proving AtomicXfer involves two parts. First, the developer proves that xfer meets its specification, as described above. Second, the developer uses the vMVCC library to obtain a proof that AtomicXfer's specification is the logically-atomic equivalent of xfer's sequential specification, as shown in Figure 3. The next subsection describes how vMVCC formally specifies db.Run in the general case to enable this second step.

$$\left\{ \mathop{\Large *}_{(k,v)\in m} k \overset{t}{\mapsto} v * P(m) \right\}$$
$$\mathtt{body(txn)}$$
$$\left\{ \text{RET } r.\ \text{if } r \text{ then } \mathop{\Large *}_{(k,v)\in m'} k \overset{t}{\mapsto} v * Q(m,m') \text{ else } \top \right\}$$
$$\Downarrow$$
$$\left\langle m.\ \mathop{\Large *}_{(k,v)\in m} k \mapsto v * P(m) \right\rangle$$
$$\mathtt{db.Run(body)}$$
$$\left\langle \text{RET } r.\ \text{if } r \text{ then } \mathop{\Large *}_{(k,v)\in m'} k \mapsto v * Q(m,m') \text{ else } \mathop{\Large *}_{(k,v)\in m} k \mapsto v \right\rangle$$

Figure 5: Specification of db.Run. The angle brackets indicate a logically atomic specification [16]. The vertical arrow indicates that, as a precondition for invoking db.Run, the developer must prove the standard Hoare-logic specification shown above the arrow for body. Not shown is the part of the specification that describes the representation predicates. We color the resources established for commit with green, and for abort with red.

3.2 Specifying the transaction interface

Transactions give users an illusion that they are "isolated" from each other. To capture this intuition, we define the resource $k \overset{t}{\mapsto} v$ (which already showed up in the above example) as the *transaction-local view* of the system state. We can then specify operations that manipulate the transaction-local view in terms of $k \overset{t}{\mapsto} v$:

$$\left\{ k \overset{t}{\mapsto} v \right\} \mathtt{txn.Read(k)} \left\{ \text{RET } v.\ k \overset{t}{\mapsto} v \right\}$$
$$\left\{ k \overset{t}{\mapsto} v \right\} \mathtt{txn.Write(k,u)} \left\{ k \overset{t}{\mapsto} u \right\}$$
$$\left\{ k \overset{t}{\mapsto} v \right\} \mathtt{txn.Delete(k)} \left\{ k \overset{t}{\mapsto} \bot \right\}$$

These specifications use standard Hoare-logic syntax, where $\{P\}$ op $\{Q\}$ means that, if op runs starting with the resources specified in precondition P, it will return with the resources as specified in the postcondition Q.

Next, we define the resource $k \mapsto v$ as the *logical view* of the system state, representing the linearizable state. The fact that only a single value of each key is exposed to users might seem counter-intuitive in the case of MVCC, given that the system physically stores multiple values for each key. However, from the application's point of view, it suffices to view the abstract state of the system as having a single value for each key at any given point in time, and updating that value at the transaction's linearization point. (We discuss this in more detail in §4.2.)

The specification of db.Run shown in Figure 5 connects these two kinds of resources. This is also the top-level theorem of vMVCC as a transaction library. The specification requires the developer to prove a sequential specification for body with a precondition that takes the transaction-local view of some set of key-value pairs, m, along with some

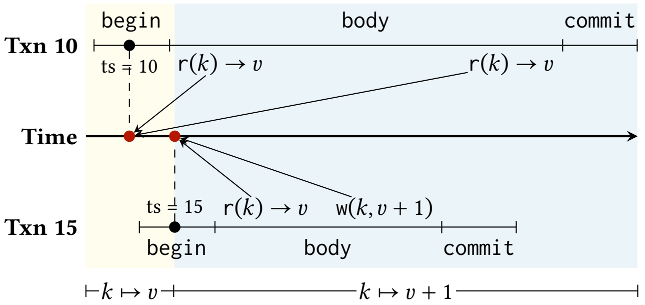


Figure 6: An example of two concurrently running vMVCC transactions. Both transactions appear to execute their reads and writes at their linearization points (marked as red). The reason linearization points appear at timestamp generation is that if Txn A linearizes (i.e., runs across its linearization point) before Txn B, then all reads and writes of A should appear to happen before that of B. This is precisely what timestamps are intended to do.

constraints on those values, represented by the predicate $P(m)$. The postcondition of body says that, if it chooses to commit, then it should return the transaction-local view of m' with some constraints $Q(m, m')$ on how these key-value pairs relate to the starting state.

Given such a specification for body, the specification of db.Run says that db.Run(body) will be the logically atomic equivalent: at some instant during its execution, it will swap the logical view of m satisfying $P(m)$ for that of m' satisfying $Q(m, m')$. Further, if this transaction aborts (either at its own will or because of conflicts with another transaction), then db.Run(body) keeps the logical view of m intact.

As an example, we can instantiate P and Q for AtomicXfer from §3.1 as follows:

$$P(m) \triangleq \text{dom}(m) = \{src, dst\}$$
$$Q(m, m') \triangleq m'[src] = m[src] - amt \wedge m'[dst] = m[dst] + amt$$

The use of P and Q as arbitrary predicates allows the db.Run specification to capture the behavior of body and transfer it to the logically atomic specification of db.Run(body). One technicality here is that P and Q are both *pure* predicates, meaning they cannot encode ownership of other resources, but merely restrict the values of m and m'.

The specification of db.Run can be regarded as a program-logic formalization of strict serializability [15] in the database literature. Serializability comes from the part of the specification that says transactions appear to observe and modify the system state one at a time (at their linearization point), with strictness owing to the fact that they do so during the course of their respective execution (and hence the serial order respects the transaction precedence order).

4 Proving vMVCC

This section describes the important aspects of our proof for vMVCC. We start with a key verification challenge and how we solve it with prophecy variables (§4.1). We describe how

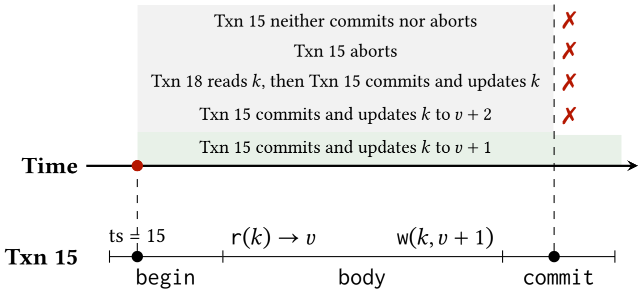

Txn 15 neither commits nor aborts ✗
Txn 15 aborts ✗
Txn 18 reads k, then Txn 15 commits and updates k ✗
Txn 15 commits and updates k to $v + 2$ ✗
Txn 15 commits and updates k to $v + 1$
Time
ts = 15
$\mathsf{r}(k) \to v$
$\mathsf{w}(k, v+1)$
Txn 15
begin
body
commit

Figure 7: Transaction futures, showing several example futures speculated through prophecy variables and their interaction with prophecy resolution.

we abstract a tuple from its physical representation containing multiple versions to its logical view with a single value, which potentially reflects some update that happens only in the future (§4.2). We present a key invariant about the prophecy variable used in vMVCC, and how the invariant helps maintain other system-wide invariants under correct and incorrect predictions (§4.3). We discuss how we define the transaction-local view of the system state, and its connection to the logical view (§4.4). We finally conclude this section with the challenges and the approach regarding proving strict monotonicity of transaction IDs (§4.5).

4.1 Speculation using prophecy variables

We introduce the verification challenge with an example shown in Figure 6. Observe that in the example, the value of k to be read by Txn 10 is determined up front by $k \mapsto v$ at its linearization point, despite the fact that by the second read of Txn 10, Txn 15 has already committed and updated the physical state of k. Similarly, the write of Txn 15 updates the logical state to $k \mapsto v + 1$ before it physically executes. This kind of "speculative" behavior of MVCC turns out to be tricky to reason about in a Hoare-logic reasoning style where the proof considers each line of code in turn and reasons about how that code updates the abstract and physical states.

The challenge arises from the fact that MVCC transactions linearize when their timestamp is generated. In the proof, the logical state must be updated at the transaction's linearization point, which happens before the transaction body runs. The changes to the logical state depend both on the transaction itself (i.e., what data the transaction decides to write), as well as conflicts with other transactions (i.e., whether another transaction reads or writes the same keys as this transaction in a way that will force this transaction to abort, as discussed in §2). This poses the question: how do we know, at the transaction's linearization point, what values will a transaction write, and whether a transaction will encounter a conflict and thus be forced to abort? To tackle this issue, we use prophecy variables.

Intuitively, prophecy variables allow the proof to speculate about future execution. In the case of vMVCC, the prophecy variable is a list of transaction actions, which describes what actions each transaction will perform, and in what order.

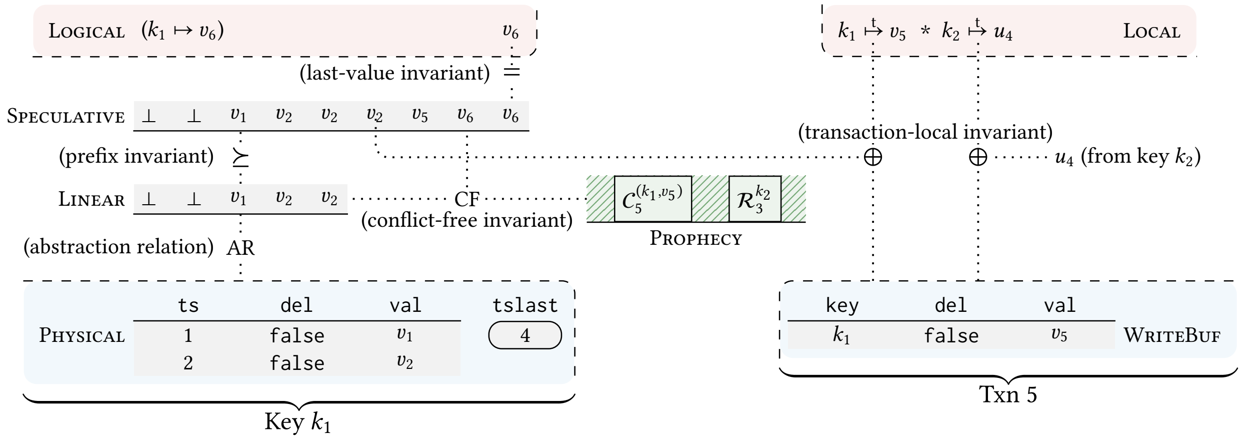

Logical $(k_1 \mapsto v_6)$ v_6
(last-value invariant) $\doteq$
Speculative $\bot$ $\bot$ v_1 v_2 v_2 v_2 v_5 v_6 v_6
(prefix invariant) $\succeq$
Linear $\bot$ $\bot$ v_1 v_2 v_2
CF
(conflict-free invariant)
$C_5^{(k_1,v_5)}$
$\mathcal{R}_3^{k_2}$
Prophecy
(abstraction relation) AR
Physical
ts del val tslast
1 false v_1 4
2 false v_2
Key k_1
$k_1 \overset{t}{\mapsto} v_5 * k_2 \overset{t}{\mapsto} u_4$ Local
(transaction-local invariant)
$\oplus$ $\oplus$ u_4 (from key k_2)
key del val
k_1 false v_5 WriteBuf
Txn 5

Figure 8: Overview of vMVCC's key physical states (in blue regions), logical states (in red regions), intermediate states, and the system-wide invariants relating them (the dotted lines). We introduce the tuple abstraction relation, the prefix invariant and the last-value invariant in §4.2, the conflict-free invariant in §4.3, and the transaction-local invariant in §4.4. Here u_4 is the value of the speculative view of k_2 at timestamp 5.

We refer to the list as the *future-action list*. There are two kinds of actions in vMVCC's proof: C_t^m ("Txn t commits and applies updates m to the system state") and $\mathcal{R}_t^k$ ("Txn t reads key k"). Transaction aborts are represented by a commit with an empty write-set.

At transaction begin time, vMVCC's proof uses the prophecy variable to speculatively predict the execution of the transaction, which allows the proof to update the logical state as if it knew what the transaction is going to do. The main challenge of using the prophecy variable, however, is that some of the predictions could be *incorrect*—it predicts something that does not match what happens later. As an example, Figure 7 shows five concrete predictions for Txn 15 that increases the value of k by 1. Only the bottom prediction turns out to be correct when the transaction actually commits. The incorrect predictions eventually diverge from the actual changes made by the transaction, and will make the logical state inconsistent with the physical state.

To deal with the divergence, the proof performs *prophecy resolution* at the point where the transaction actually commits and updates the physical state. Prophecy resolution allows the proof to stop considering cases corresponding to predictions that did not match reality, and continue only with the cases that did. We will elaborate more on correct/incorrect predictions and prophecy resolution with a concrete example in §4.3.

This description may make it sound like there are a large number of cases to consider in the proof, greatly increasing the proof burden. In practice, the predictions are symbolic, rather than concrete timestamps, keys, and values; for instance, the prophecy variable speculates the updates made by a transaction as a symbolic partial map. Furthermore, the proof can group together many speculative executions (e.g., those in which the transaction of interest is speculated to commit without encountering a conflict), and consider the entire family of executions just once.

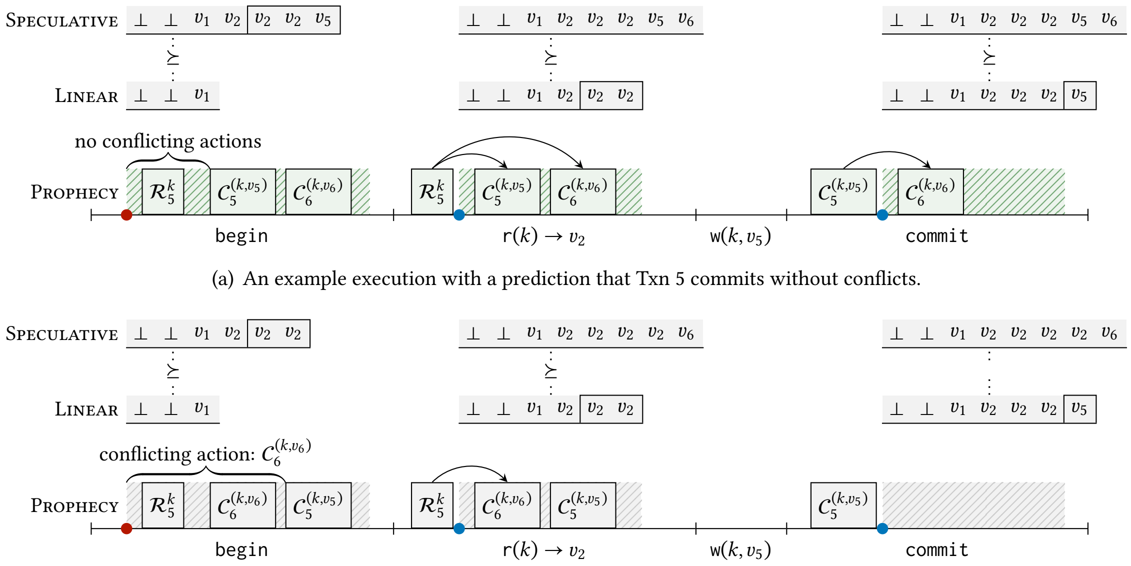
4.2 Incorporating speculation in abstract state

vMVCC exposes a single linearizable copy of the system state, thereby freeing the users from explicitly reasoning about the timestamps. Thus, the logical view (shown in the "logical" row of Figure 8) of vMVCC is a single value for every key, and the proof must connect this logical view to the physical state (shown in the "physical" row of Figure 8), consisting of the Go struct representing each tuple.

This connection is challenging for several reasons, including the fact that the Go data structure contains multiple versions, and the fact that the value in the logical view may not even be present in the Go data structure, if it is made by a write speculated by the prophecy variable for an active transaction. Moreover, reasoning about the physical layout of the tuple in all intermediate proofs is cumbersome.

To address these challenges, we introduce two intermediate layers modeled with *monotonic lists* (i.e., lists that only grow). The first is the *linear view* of the tuple, shown in the "linear" row of Figure 8. The linear view is a contiguous list of values, indexed by timestamps. The linear view gives us an elegant way to specify operations on tuples: reading a tuple with a given timestamp t just returns its value at index t. If the transaction needs to extend `tslast`, doing so extends the linear view up to the new `tslast` timestamp, filling in new entries with the last value in the list. Writing a tuple with a given timestamp t extends the tuple up to index t, and appends the new value to the end.

To capture the speculative behavior of MVCC as described in §4.1, we add the "speculative" layer, as shown in Figure 8 as well. The *speculative view* is yet another contiguous timestamp-indexed list, much like the linear view, but includes the writes from transactions that have linearized but have not yet finished executing and updating the physical state. The proof looks up and extends the speculative view at the linearization point (the ability for such extension is guaranteed by strict monotonicity of vMVCC's timestamps),



(a) An example execution with a prediction that Txn 5 commits without conflicts.

(b) An example execution with a prediction that Txn 5 commits despite conflicts.

Figure 9: Two example executions illustrating how the proof handles correct and incorrect predictions in Figure 9(a) and Figure 9(b), respectively. We indicate the linearization and prophecy resolution points with red and blue dots, respectively. For concreteness, we use Txn 5 in the examples, but in the actual proof, the timestamp is just a symbolic value t that represents all possible timestamps. Note that all these states are sealed in some global invariant to enable sharing—each thread (or, transaction) can access them only at its atomic steps, including the linearization and prophecy resolution points. This means that the states could have changed by another thread between two atomic steps.

based on the prophecy variable. The linear view is updated when physically reading and writing a tuple.

We use these intermediate views to relate vMVCC's physical state to its top-level logical view, as shown in Figure 8, for each key in the system state. The *tuple abstraction relation* describes how the physical tuple layout is connected to its abstract linear view. The *prefix invariant* requires that the linear view must be a prefix of the speculative view, capturing the intuition that the speculative view runs ahead of the linear one. Finally, the *last-value invariant* says the last element of the speculative view is equal to the top-level logical value of that key. vMVCC's proof heavily relies on the invariants maintained between these layers.

Modeling these intermediate views as monotonic lists allows the proof to seal their "authoritative" ownership in a global invariant for sharing among transactions, but at the same time enables the proof to retain knowledge about existing prefixes of the lists. As we will see in §4.4, this is crucial to bridge the gap between reading the logical state at the linearization point, and reading the physical state later on when the transaction actually executes.

Abstraction relation under GC. In the presence of GC, the tuple abstraction relation (shown as AR in Figure 8) cannot hold on all timestamps, as that would require mutating the existing part of the linear view when removing unusable versions from the physical state. As mentioned in §2, the key idea of GC safety is to identify versions that will not be accessed by any transactions, including those that have not even begun. We formalize this line of reasoning with a monotonic timestamp t_{safe}, which serves as a lower bound on the transaction IDs of all active and future transactions.

To start a round of GC, the garbage collector first computes a new t_{safe}, and then relaxes the abstraction relation of the target tuples so that it no longer places any constraints on versions whose lifetime ends before t_{safe}. Doing so allows the garbage collector to delete those versions without violating the abstraction relation. We further weaken the specification for reading and writing a tuple at timestamp t by requiring a proof of $t \geq t_{\text{safe}}$ in their precondition, ensuring that the deleted versions are never observed.

4.3 Maintaining invariants under speculation

One challenge in vMVCC's proof stems from the fact that prophecy variables can speculatively predict that a transaction will commit in the future, while at the same time predicting earlier transactions that conflict with it. This brings up two challenges related to the system-wide invariant maintained by vMVCC's proof.

The first challenge is that the invariant ensures that transactions cannot commit in the presence of conflicts, which would be at odds with the (ultimately incorrect) speculative prediction described above. This makes it impossible for the proof developer to update the logical state based on the incorrect predictions but still maintain the invariant, which must always hold. To get around this issue, vMVCC's proof treats such inconsistent predictions as transaction aborts, which makes it easy to maintain the invariant and thereby carry through the prediction to the prophecy resolution point.

The second challenge stems from the fact that an inconsistent prediction, such as the example above, involves multiple transactions, and therefore relies on prophecy resolution in multiple threads. The prediction about each individual transaction and thread could be correct in its own right, but it is the combination of them that leads to a contradiction. How should the proof be structured to establish the contradiction despite only doing one prophecy resolution at a time? vMVCC's proof addresses this challenge by maintaining a sufficiently strong invariant that carries along facts from each prediction to derive contradictions against later predictions as needed, as we describe below.

To illustrate this point, we first sketch out the proof for a correct prediction, where a transaction commits and there are no conflicts that would have forced it to abort, and then show how vMVCC's proof handles incorrect predictions.

Predicted commit without conflicts. In Figure 9(a), Txn 5 is speculated to commit without encountering any conflict. The reason is that at its linearization point, the commit action of Txn 5 in the future-action list, $\mathcal{C}_5^{(k,v_5)}$ (there might be multiple of them in the list, but the proof cares only about the first one), is *conflict-free* against all the actions prior to it. We define $\mathcal{C}_t^m$ to be conflict-free against an action a if (1) $a = \mathcal{R}_{t'}^k$, where $t' \le t \lor k \notin \mathrm{dom}(m)$, or (2) $a = \mathcal{C}_{t'}^{m'}$, where $t' < t \lor \mathrm{dom}(m') \cap \mathrm{dom}(m) = \emptyset$. Knowing that Txn 5 will commit without conflicts, the proof safely extends the speculative view up to timestamp 5 using the old value v_2, and appends the new value v_5 to it (which updates the logical view to v_5 as well) without violating the *conflict-free invariant*, as described below.

Intuitively, the conflict-free invariant requires that a transaction reflects its update to the speculative view only if the first commit action of the transaction is conflict-free against all the actions prior to it in the future-action list. As we will see below, this invariant is crucial to prove invariance of the prefix property between the linear and speculative views.

On reading key k, the proof resolves the head of the future-action list to $\mathcal{R}_5^k$. Then, it uses the conflict-free invariant to deduce that transactions which contain updates to the speculative view, but not to the linear view, must have timestamps greater than or equal to the timestamp of this read. This implies that the speculative view can differ from the linear view only after a timestamp $t > 5$, allowing the proof to re-establish the prefix invariant after extending the linear view. A similar reasoning goes for commit, except the proof additionally uses the promise that Txn 5 will commit to know the value at timestamp 5 of the speculative view is v_5.

Predicted commit despite conflicts. In Figure 9(b), Txn 5 is speculated to commit despite the presence of conflicts because its first commit action, $\mathcal{C}_5^{(k,v_5)}$, conflicts with an earlier action $\mathcal{C}_6^{(k,v_6)}$. The proof, as in the previous case, extends the speculative view up to timestamp 5 using the old value v_2; however, it does not append the new value v_5 as doing so would violate the conflict-free invariant, and proceeds as if the transaction will abort, which makes the invariant true.

For read, the proof of the prefix property is similar to the previous case. For commit, however, the proof cannot re-establish the prefix property after extending the linear view, because it indeed did not apply the new value v_5 at the linearization point. Fortunately, at this point the proof knows two facts that contradict each other: (1) reaching the prophecy resolution point for commit, the execution must have passed the conflict detection as illustrated in Figure 2(b), implying the length of the linear view $l \le 5 + 1$ (the +1 part is due to our lists being zero-indexed); (2) some conflicting action (in this case $\mathcal{C}_6^{(k,v_6)}$), which extends the linear view to at least timestamp $t > 5$, must have happened *before* Txn 5 commits, implying $l > 5 + 1$. The proof closes this case with the derived contradiction.

4.4 Abstract state of a transaction

As mentioned in §4.1 (and illustrated in Figure 6), the value of key k to be read by a transaction is determined up front by $k \mapsto v$ at the transaction's linearization point. Reading from the physical state, however, happens only at some later point in time, and the value is based on $k \overset{\mathsf{t}}{\mapsto} v'$, as specified in §3.2. This means the proof has to somehow connect $k \mapsto v$, $k \overset{\mathsf{t}}{\mapsto} v'$, and v'', the result of physically reading the tuple of k. This section describes how the system-wide invariants shown in Figure 8 establish that connection.

Let us first consider the case where the transaction has not written key k. Our first step then is to show $v = v'$. Recall that at the linearization point of Txn t that reads or writes k, we extend the speculative view of k up to t using its last value. Doing so, along with the last-value invariant, allows us to deduce that the value of the speculative view at index t is v (and will remain so since the speculative view is monotonic). The proof then follows from the definition of the *transaction-local invariant*, which says that if Txn t has not written k, then v', the transaction-local value, is equal to the value of the speculative view at index t.

Our next step is to show $v' = v''$. Again recall that physically reading the tuple of k at timestamp t means extending the linear view of k up to t (if the value at t is still absent), and looking up its value at index t. The proof of $v' = v''$ then follows immediately from the prefix invariant that requires the linear view to remain a prefix of the speculative one.

Now consider the case where the transaction has last written k with value u. As specified in §3.2, the logical effect of the write is $k \overset{\mathsf{t}}{\mapsto} u$. We thus define the remaining case for the transaction-local invariant: if the transaction has written k, then the transaction-local value is equal to the value in its local write buffer.

The contents of the write buffer are also what the speculated updates in a commit action (i.e., m in $\mathcal{C}_t^m$) resolve to. This allows us to obtain the equality between the speculated updates with the actual updates at the prophecy resolution point, which is crucial when re-establishing the prefix invariant as discussed in §4.3.

4.5 Strict monotonicity of transaction ID

Another challenge in vMVCC's proof is establishing strict monotonicity of transaction IDs, for vMVCC's RDTSC-based algorithm described in §2. The challenge lies not only in proving that the algorithm generates strictly increasing transaction IDs, but also in being able to logically execute the transaction (i.e., update the logical state) at the linearization point corresponding to that transaction ID. The reason this is challenging is that the linearization point for some transaction ID t' might not correspond to any line of code that was executed for this transaction—the algorithm simply spins in a loop waiting for RDTSC to advance past t', and linearization occurs when *any* transaction observes that t' has passed.

To formally reason about this algorithm, vMVCC's proof maintains a logical table of slots, one per timestamp. The slot contains the logical set of changes that a transaction with that timestamp wants to perform, represented as a ghost function. The actual state changes performed by this ghost function are determined by prophecy variables, as described above. The proof uses the slots to invoke the ghost functions for each timestamp in order, as the RDTSC clock advances; the proof maintains a "latest-slot" timestamp corresponding to the last table slot that has been invoked. The invariant associated with this proof states that this latest timestamp is always below (or equal to) the real RDTSC clock. Furthermore each future slot is protected by the site's mutex that corresponds to this timestamp.

When the transaction manager first computes t', it registers the t' slot in the table, putting in a ghost function that will perform its transaction's changes. Since $t' > t$, the proof has not yet invoked the ghost function for this slot, and the current thread also holds the site's mutex needed to fill this slot (which proves that no concurrent thread could fill the same slot). As the transaction manager runs the loop waiting for RDTSC to move past t', it updates the latest-slot with each iteration, executing all of the ghost functions in the slots that have been advanced over. The proof takes advantage of *later credits* [31] in Iris that enable verification of this "unsolicited helping" pattern.

The invariant for the slot table says that, for every slot with a timestamp below the latest-slot, its ghost function callback has been invoked. As a result, when the transaction manager's loop exits, it knows that the latest-slot is at least as high as t', and therefore its ghost callback must have been invoked (either by this same thread or by some other thread running the same loop).

5 Implementation and proof details

We implemented vMVCC in Go, and verified its implementation using the Perennial framework [3] (based on Iris [21–23] and Coq [33]), using Goose [4] to lift vMVCC's Go code into Perennial. To enable vMVCC's proofs of MVCC transaction linearizability, we incorporated prophecy variable support from Iris [18] into Perennial.

Component	Lines of code (Go) / proof (Coq)
Tuple	260 / 1947
Transaction	419 / 4489
Index	85 / 496
Timestamp	24 / 311
Misc	39 / 361
Ghost state	- / 947
Global invariants	- / 2566
Total	827 / 11117

Figure 10: Lines of code and proof for each component of vMVCC.

Figure 10 summarizes the implementation and proof effort, not including changes to Perennial that were necessary for the verification. The lines of proof include the specifications for each function in vMVCC's implementation. The proof effort for vMVCC required about 13× as many lines of proof as lines of code, which is in the same ballpark as other verified systems that handle concurrency [3, 6, 13].

The implementation contains several low-level optimizations that improve performance. We used RDTSC to generate transaction IDs. We also padded data structures to avoid false cache-line sharing that limits multi-core scalability, and sharded the index and the set of active transaction IDs to reduce contention from concurrent accesses.

5.1 Bugs found during verification

When we were first designing and implementing vMVCC, we were careful to structure the code in a way that makes it clear why the code is correct, what the invariants are, how they are maintained, and what guarantees each interface or function provides. Nevertheless, during the actual verification, we ran into several bugs in corner cases that we missed or did not correctly handle in the implementation, highlighting the importance of formal reasoning. In this subsection, we give several examples of such bugs.

One interesting bug we found when verifying vMVCC is related to garbage collection. The buggy code is:

```
func (site *TxnSite) getSafeTS() uint64 {
    site.mutex.Lock()
    var tidmin uint64 = MAX_U64 /* buggy */
    // var tidmin uint64 = site.getCurrentTS()
    for _, tid := range site.tidsActive {
        if tid < tidmin {
            tidmin = tid
        }
    }
    site.mutex.Unlock()
    return tidmin
}
```

When the garbage collector starts a new round of GC, it first calls getSafeTS on each site to collect the per-site minimal transaction ID, and then computes a globally minimal one

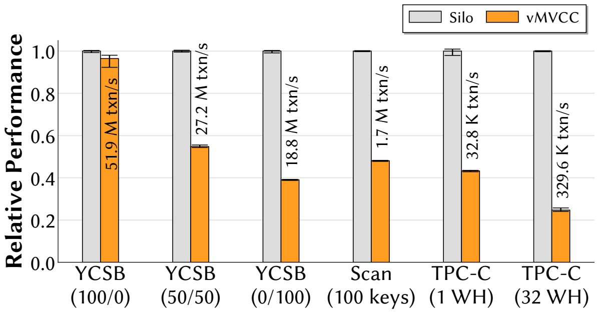


Figure 11: Comparison of Silo and vMVCC. For YCSB, each transaction reads or writes a key sampled from a uniform distribution with a certain R/W ratio. For TPC-C, the number of warehouses is same as the number of worker threads.

among them. If every transaction site is empty (i.e., if every site returned MAX_U64), the garbage collector generates a timestamp using an arbitrary site. (Recall that vMVCC always places a site ID in the low bits of the timestamp, and the choice of site ID does not matter, as it is purely there to ensure uniqueness.) The bug arises when a transaction enters the system right after getSafeTS returns, and then the garbage collector computes a timestamp larger than the ID of that transaction. Our fix to this bug is to generate a timestamp within each site, as shown in the commented-out code. Doing so ensures that future transaction IDs generated by this site will be larger than the one getSafeTS returns.

Another subtle bug we discovered is missing the wait loop when generating transaction IDs, violating the strict monotonicity of our timestamp generation scheme. The fix was the looping RDTSC algorithm described in §2. Finally, since our protocol is centered around timestamps, we also discovered several off-by-one errors in the implementation when conducting the verification of vMVCC (e.g., where greater-than comparisons should have been greater-than-or-equal-to comparisons).

6 Evaluation

We experimentally answer the following questions:

- Is vMVCC competitive with state-of-the-art unverified systems? (§6.2)
- Does the use of MVCC in vMVCC help with long-running read-only transactions? (§6.3)
- Are the low-level optimizations in vMVCC important for performance? (§6.4)
- Does vMVCC scale under high-contention workloads? (§6.5)

6.1 Experimental setup

All experiments were done on an AWS EC2 c5.9xlarge instance with 36 vCPUs (18 physical CPUs, each shared by 2 hardware threads via hyper-threading) and 72 GB of main memory, running Linux 5.15.0 and Go 1.20.3.

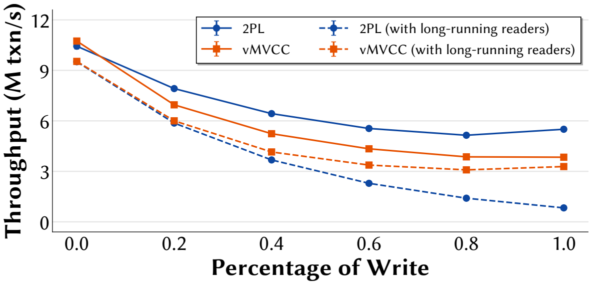


Figure 12: Comparison of 2PL and vMVCC under YCSB (4 keys accessed per transaction, $\theta = 0.85$, 24 threads), with and without 8 long-running reader threads that repeatedly read 1% of the entire key space.

We used the YCSB benchmark [7] to understand the performance characteristics of vMVCC under various workloads. Unless otherwise specified, we execute each YCSB put or get in a separate transaction. The data set contains 1M key-value pairs with each key being an 8-byte integer and value an 100-byte string. The access pattern follows the uniform distribution, or the Zipfian distribution, with a parameter θ controlling the skewness of the distribution. We vary the read-write ratio and the number of keys accessed in each transaction.

We also used the TPC-C benchmark, which involves more sophisticated transactions. TPC-C models the operation of a wholesale supplier, a common online-transaction processing (OLTP) workload. It contains 9 tables and 5 kinds of transactions, each with various workload characteristics. In particular, most transactions can be processed in a single warehouse, so it is natural and efficient to map each warehouse to one thread. Our current implementation of vMVCC requires the key to be an 8-byte integer, and every tuple needs a key. Because of these limitations we made two modification to TPC-C. First, we do not support "get customers by their last name" appearing in the OrderStatus and NewOrder transactions; they are replaced with just "get customers by customer ID". Second, the History table does not have a primary key, so we randomly generate one for it.

We employ a background GC thread for vMVCC in every experiment. We repeat each experiment 10 times, each for 30 seconds. We report the mean, minimum, and maximum (the last two as error bars) among the 10 runs.

6.2 Comparison with Silo

To evaluate whether vMVCC achieves competitive performance with state-of-the-art systems, we compare vMVCC to Silo [35], a high-performance transactional database system. Because vMVCC does not store data durably, we compare with MemSilo, a variant of Silo that does not persist its data. Silo is an OCC/MVCC based system, using OCC to provide serializability, and using MVCC to access a consistent snapshot of old versions. Unlike vMVCC, Silo does not generate a new version for every write, but only once per *snapshot epoch* (on the order of 1 second), which reduces memory management costs. Silo's OCC/MVCC design has the ad-

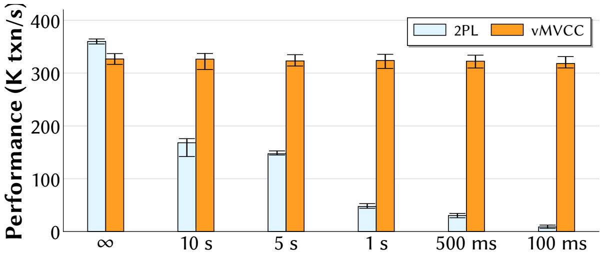


Figure 13: Comparison of 2PL and vMVCC under TPC-C (32 warehouses), with a thread periodically invoking the read-only transaction `StockScan`.

vantage of lower memory usage and allocation overhead over vMVCC's pure MVCC design. On the other hand, Silo only ensures its snapshot transactions (those that access past versions) always read a consistent snapshot, without imposing ordering constraints on them with respect to normal linearizable transactions, whereas in vMVCC, a "snapshot transaction" is simply a linearizable transaction that does not perform writes.

Figure 11 shows the results of the comparison for several configurations of YCSB and TPC-C, normalizing to the throughput achieved by Silo. Similarly to Silo, each worker thread in vMVCC generates the workload parameters and then immediately processes the transaction. We used a YCSB profile where each transaction accesses a single key sampled from a uniform distribution. vMVCC achieves 96.6% the throughput of Silo for a read-only workload in YCSB, and 38.8% for a write-only workload. For TPC-C, vMVCC achieves 43% the throughput of Silo for 1 warehouse and 25.7% for 32 warehouses. We hypothesize that the performance difference between Silo and vMVCC is largely due to (1) vMVCC's higher memory allocation overhead for storing past versions, and (2) its inefficient way of executing range scans—lacking a tree-like index structure, vMVCC relies on the continuity invariant of TPC-C [34], and expands a range query into multiple point queries. To test these hypotheses, we conducted the following two experiments.

First, we ran the same write-only YCSB workload, except that we fixed the write value to some statically allocated string, and modified vMVCC to perform in-place update on its tuples, without changing any other parts of the code (hence the resulting system is not even correct, but it is merely for us to understand more about vMVCC's performance characteristics). Applying these changes increases the relative performance from 38.8% to 87.3%.

Second, we ran an additional range scan workload where each transaction first randomly picks a starting key, and then reads the next 100 keys. Silo executes each transaction with a single scan, whereas vMVCC issues 100 reads. Figure 11 shows the results in the "scan" column. vMVCC achieves 48.1% of Silo's throughput; the difference mostly attributes to more cache misses in vMVCC.

Based on these experiments, we conclude that the gap between vMVCC's performance and that of Silo is indeed largely due to memory allocation and vMVCC's lack of support for range scans. For benchmarks that do not stress these two aspects, vMVCC achieves performance competitive with Silo.

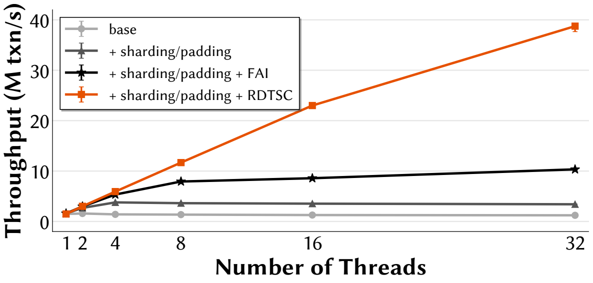


Figure 14: Throughput of vMVCC with different optimizations enabled. The benchmark is YCSB (1 key read per transaction, $\theta = 0.2$).

6.3 Robustness to long-running readers

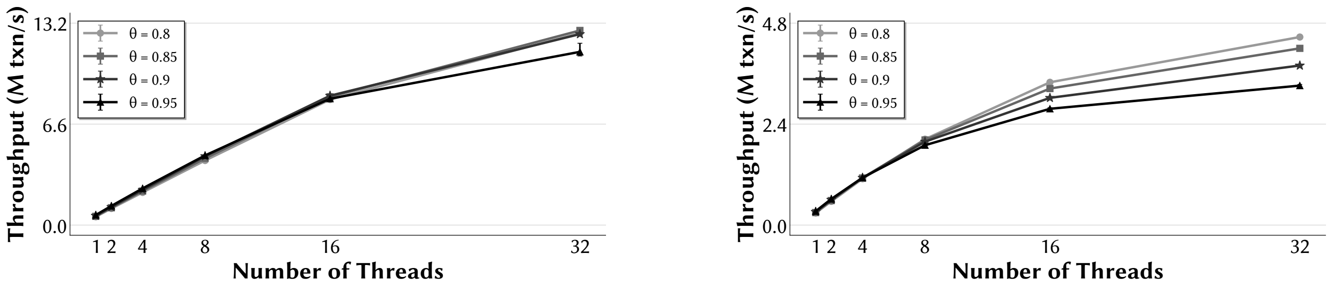
One main advantage of MVCC over traditional concurrency control protocols is that its performance should remain stable even in the presence of long-running readers. To confirm that vMVCC's design indeed achieves these performance benefits, we implemented a variant of vMVCC that uses two-phase locking for concurrency control instead of MVCC, and compared the performance of vMVCC with this 2PL variant.

YCSB. We first compare vMVCC and 2PL under the YCSB workload, using a YCSB profile where each transaction reads or writes 4 keys. We fixed the number of threads to 24, θ to 0.85, and varied the read-write ratio from 0% to 100%. We then ran one experiment without long-running readers, and another one where the workload includes 8 transactions repeatedly reading 10K keys (1% of the entire key space).

Figure 12 shows the results. In the absence of long-running readers, 2PL performs better than vMVCC for all read-write ratios except for the read-only workload (comparing the solid lines). The difference stems from MVCC's overhead of (1) generating timestamps and (2) keeping past versions and the associated memory allocation costs.

In the presence of long-running readers (comparing the solid and dashed lines of each system), vMVCC's throughput drops slightly between the range of 11.5%–22.2%, whereas 2PL's throughput drops significantly as the write ratio increases (e.g., 72.6% and 84.9% for write ratio 80% and 100%, respectively). As a result, the performance of 2PL with long-running readers is worse than that of vMVCC for workloads with 40% or more writes; for instance, under write ratios 80% and 100%, vMVCC performs 2.2× and 4× better than 2PL, respectively. The reason is that, in 2PL, the long-running readers hold read locks on keys for a long duration, preventing other transactions from writing to those keys.

TPC-C. We also compare vMVCC and 2PL under the TPC-C workload. Similarly to prior work [36], we add a read-only transaction `StockScan` that counts the number of each item in



(a) Read scalability of vMVCC under high-contention workloads.

(b) Write scalability of vMVCC under high-contention workloads.

Figure 15: Scalability analysis under high contention. The benchmark is YCSB (4 keys accessed per transaction).

all warehouses. We parametrize the workload by the interval of invoking StockScan. Figure 13 shows the results.

When no StockScan is invoked (represented by the ∞ interval on the x-axis), 2PL performs better than vMVCC by around 14%. However, when there are StockScan transactions running, NewOrder transactions that update the stock table will conflict with StockScan, and block under 2PL concurrency control. As the interval between StockScan transactions decreases, 2PL's performance drops significantly, whereas vMVCC throughput remains more-or-less the same, since StockScan accesses old versions of tuples and does not impact other transactions. For StockScan intervals 500 and 100 ms, the throughput of vMVCC is 11× and 54.4× that of 2PL. In terms of latency, vMVCC maintains its 99.9% latency around 3.4 ms across all StockScan intervals, whereas the 99.9% latency of 2PL increases from 3.2 ms in the absence of StockScan transactions, to a few tens and occasionally hundreds of ms when StockScan is invoked every 100 ms.

6.4 Low-level optimizations

vMVCC implements (and verifies) two low-level optimizations to achieve high performance on many cores: (1) padding and sharding data structures and mutexes, to avoid cache-line contention, and (2) using RDTSC to generate transaction IDs without shared-memory contention. To understand whether they are important for performance, we enable each optimization in turn and measure the resulting performance. To stress the implementation, we chose a lightweight YCSB profile where each transaction accesses a single key. We chose a low-contention setting ($\theta = 0.2$) so that transactions largely access different portions of the key space; we will evaluate scalability under high contention in the next subsection.

To evaluate the benefit of the RDTSC-based transaction ID generation, we compare with two alternatives. The first is a lock-based design where the transaction manager acquires a mutex to get (and increment) the next transaction ID counter. The second is a lock-free implementation that uses the fetch-and-increment (FAI) instruction to atomically obtain the next transaction ID.

Figure 14 shows the results. The optimizations have little effect on a single core, but significantly improve vMVCC's performance on 32 cores. Partitioning and padding index and transaction sites improves vMVCC's performance by 2.8× at 32 cores. Using FAI increases throughput by a further 3× over the lock-based design at 32 cores. Finally, RDTSC-based transaction IDs achieve yet another 3.7× improvement in throughput compared to FAI at 32 cores. In summary, the results show that all of these optimizations are important for scaling vMVCC's performance with many cores.

Enabling all the optimizations, vMVCC's throughput scales by 15.6× using 16 threads. The result suggests that vMVCC eliminates almost all contention on its internal data structures (when the keys themselves do not contend). The throughput scales further by 1.66× when doubling the number of threads to 32, implying that vMVCC can benefit from hyper-threading even though not as much as from having more physical cores.

6.5 Scalability under contention

The previous section showed that vMVCC scales nearly linearly for a low-contention workload, with its low-level optimizations. In this section, we evaluate vMVCC's scalability under high-contention workloads, using a YCSB profile where each transaction issues 4 reads/writes, with the skewness parameter θ ranging from 0.8 to 0.95.

Figure 15 shows the results. For reads (Figure 15(a)), before reaching the hyper-threading threshold (i.e., 18 cores), the throughput scales almost linearly with respect to the number of threads, except for extremely contended workloads (e.g., $\theta = 0.95$): the aggregated throughput of 16 threads is 14.9× that of a single thread for $\theta = 0.8$, and 12.6× for $\theta = 0.95$. Scalability drops after reaching the hyper-threading threshold because of interference, especially for higher skewness: using 32 threads achieves 22.8× better performance for $\theta = 0.8$, and 16.9× for $\theta = 0.95$.

For writes (Figure 15(b)), asides from hyper-threading interference, having more contention also causes more conflicts between transactions, and hence higher abort rates. For instance, the abort rate at 32 threads for $\theta = 0.8$ is 4.8%, whereas for $\theta = 0.95$ is 27.6%. The result is that vMVCC's throughput with 32 threads is 11.7× that of a single thread for $\theta = 0.8$ and 9.9× for $\theta = 0.95$.

The results show that vMVCC's performance scales with the number of cores even for workloads of high contention.

7 Related work

vMVCC is the first formally verified MVCC-based system, but builds on prior work on formal verification and specification of transactions, as we now discuss.

Verified systems. The closest related work to vMVCC is GoTxn [6], a verified transaction library that uses 2PL for concurrency control. GoTxn stores data durably to disk and uses the verified GoJournal [5] journaling system to provide crash atomicity. vMVCC uses a more sophisticated concurrency control plan (MVCC), which allows it to achieve high performance for long-running read-only transactions, while GoTxn uses standard 2PL which does not perform well with long-running readers. vMVCC also implements and verifies sophisticated optimizations, such as strictly increasing `RDTSC`-based timestamps, which are not present in GoTxn.

Malecha et al. [28] verified a simple relational database, focusing on SQL queries, the relational data model, and the use of B+-trees on disk. These issues are complementary to the focus of vMVCC, which targets handling concurrent transactions using sophisticated concurrency control protocols and low-level optimizations.

Prophecy variables. Abadi and Lamport [1] first proposed prophecy variables as a proof technique to establish refinement mappings between state machines. Jung et al. [18] later integrated it in a Hoare-style program logic. Prior work using prophecy variables is mostly focused on verification of protocols and small examples of data structures and algorithms, such as RDCSS, the Herlihy-Wing queue [18], and the atomic snapshot algorithm [24].

In this paper, we apply prophecy variables in a sophisticated transaction library. We use prophecy variables to make more elaborate predictions about the behavior of transactions, including what data they read and write, and we demonstrate that prophecy variables are useful for reasoning about transactions.

Framework for specifying and verifying transactions. Lesani et al. [25] develop a framework for verifying software transactional memory systems and apply it to the NOrec transactional memory algorithm [9]. NOrec uses a form of OCC, in which transactions check whether they have been invalidated by conflicting writes during commit time. As with 2PL, NOrec transaction's linearization point occurs during commit, and hence does not appear to require prophecy variables in its proof.

vMVCC uses logically atomic triples to specify transactions, instead of classical serializability and linearizability definitions [15] that are based on trace equivalence (e.g., as used by GoTxn). This makes it easier to verify clients of a transaction library by proving Hoare triples about code running inside of the transaction library. Prior work has similarly found it useful to introduce alternate specifications for transactions and serializability in the context of formal verification. The Push/Pull model [20] provides a set of primitive operations which can be used to describe a variety of transactions. Any system that can be decomposed into these operations is guaranteed to be serializable. C4 [26] is a framework that supports verifying transactional objects, that is, concurrent data structures that allow chaining multiple operations together in an atomic transaction. The framework supports composing transactional objects as components of a higher-level transactional object.

8 Conclusion

This paper presented vMVCC, the first MVCC-based transaction library with a machine-checked proof of correctness. A key challenge in verifying vMVCC lies in reasoning about the linearization of transactions under MVCC, where the linearization point occurs before the transaction body actually runs. vMVCC addressed this challenge by using prophecy variables to speculate whether upcoming transactions are going to commit, and what values they are going to write, thereby allowing vMVCC to state and prove a simple yet general specification for its top-level transaction interface using logical atomicity. vMVCC incorporates further low-level optimizations, such as using `RDTSC` to generate strictly increasing transaction IDs, with corresponding proofs of correctness, to achieve high performance. An evaluation demonstrated that, for a range of YCSB and TPC-C workloads, vMVCC's throughput is 25–96% of the throughput of Silo, a state-of-the-art unverified system; that vMVCC benefits from MVCC to achieve good performance for long-running read-only transactions compared to two-phase locking; and that vMVCC's low-level optimizations are important for achieving high performance. At the same time, vMVCC's proof effort—13× as many lines of proof as lines of code—is on par with other verified concurrent systems.

Acknowledgments

We are grateful to Anish Athalye, Sanjit Bhat, Alexandra Henzinger, Jon Howell, Derek Leung, the anonymous reviewers, and our shepherd, Adriana Szekeres, for their valuable feedback that improved this paper. We thank Tej Chajed for discussions on transactions and the Perennial framework. This work was supported by a grant from Amazon AWS through the Science Hub program, and by NSF awards CCF-2123864 and CCF-2318722. The code and proof of vMVCC is available at:

https://pdos.csail.mit.edu/projects/vmvcc.html

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd Annual IEEE Symposium on Logic in Computer Science*, pages 165–175, Edinburgh, Scotland, July 1988.

[2] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Sept. 2014.

[3] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 243–258, Huntsville, Ontario, Canada, Oct. 2019.

[4] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent Go code in Coq with Goose. In *Proceedings of the 6th International Workshop on Coq for Programming Languages (CoqPL)*, New Orleans, LA, Jan. 2020.

[5] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 423–439, Virtual conference, July 2021.

[6] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 447–463, Carlsbad, CA, July 2022.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, Indianapolis, IN, June 2010.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 67–78, Bangalore, India, Jan. 2010.

[10] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, NY, June 2013.

[11] etcd Authors. etcd API, Apr. 2023. https://etcd.io/docs/v3.6/learning/api/#revisions.

[12] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Kohala Coast, HI, Aug.–Sept. 2015.

[13] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, Nov. 2016.

[14] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon Redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, May–June 2015.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

[16] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–282, Austin, TX, Jan. 2011.

[17] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[18] R. Jung, R. Lepigre, G. Parthasarathy, M. Rapoport, A. Timany, D. Dreyer, and B. Jacobs. The future is ours: prophecy variables in separation logic. In *Proceedings of the 47th ACM Symposium on Principles of Programming Languages (POPL)*, pages 45:1–45:32, New Orleans, LA, Jan. 2020.

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

[20] E. Koskinen and M. Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 186–195, Portland, OR, June 2015.

[21] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *Proceedings of the 26th European Symposium on Programming (ESOP)*, pages 696–723, Uppsala, Sweden, Apr. 2017.

[22] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–217, Paris, France, Jan. 2017.

[23] R. Krebbers, J. Jourdan, R. Jung, J. Tassarotti, J. Kaiser, A. Timany, A. Charguéraud, and D. Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 77:1–30, St. Louis, MO, Sept. 2018.

[24] L. Lamport and S. Merz. Prophecy made simple. *ACM Transactions on Programming Languages and Systems*, 44(2):6:1–6:27, Apr. 2022.

[25] M. Lesani, V. Luchangco, and M. Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR)*, page 516–530, Newcastle upon Tyne, UK, Sept. 2012.

[26] M. Lesani, L. Xia, A. Kaseorg, C. J. Bell, A. Chlipala, B. C. Pierce, and S. Zdancewic. C4: verified transactional objects. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA):1–31, 2022.

[27] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, Chicago, IL, May 2017.

[28] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, Madrid, Spain, Jan. 2011.

[29] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007.

[30] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, Sept. 1978. http://hdl.handle.net/1721.1/16279.

[31] S. Spies, L. Gäher, J. Tassarotti, R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Later credits: Resourceful reasoning for the later modality. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Ljubljana, Slovenia, Sept. 2022.

[32] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, Portland, OR, June 2020.

[33] The Coq Development Team. *The Coq Proof Assistant, version 8.15*, Jan. 2022. URL https://doi.org/10.5281/zenodo.5846982.

[34] Transaction Processing Performance Council (TPC). TPC benchmark C standard specification, revision 5.11, Feb. 2010. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

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

[36] 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.

[37] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach, D. Rosenthal, X. Dong, W. Wilson, B. Collins, D. Scherer, A. Grieser, Y. Liu, A. Moore, B. Muppana, X. Su, and V. Yadav. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, pages 2653–2666, Virtual conference, June 2021.