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

Yun-Sheng Chang [ϕ]Ralf Jung Upamanyu Sharma
[†]Joseph Tassarotti Frans Kaashoek Nickolai Zeldovich

MIT CSAIL [ϕ]ETH Zurich [†]NYU

Transactions simplify application development, but...

Achieving high performance requires sophisticated concurrency techniques

- Multi-version concurrency control (MVCC), contention-free data structures, etc.

Transactions simplify application development, but...

Achieving high performance requires sophisticated concurrency techniques

- Multi-version concurrency control (MVCC), contention-free data structures, etc.

Hard to implement a correct and high-performance transaction layer

- Zheng et al. [OSDI ’14], Elle [VLDB ’20], TxCheck [OSDI ’23], etc.

Transactions simplify application development, but...

Achieving high performance requires sophisticated concurrency techniques

- Multi-version concurrency control (MVCC), contention-free data structures, etc.

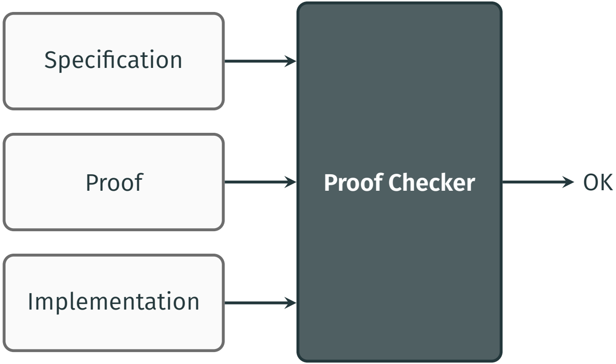
Hard to implement a correct and high-performance transaction layer

- Zheng et al. [OSDI ’14], Elle [VLDB ’20], TxCheck [OSDI ’23], etc.

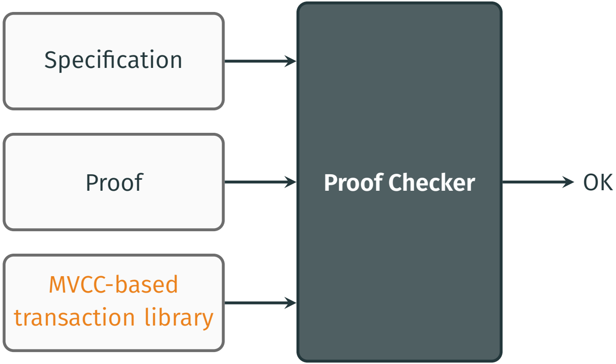
Transaction bugs can lead to severe consequences

- Corrupted databases, data losses, security issues, etc.

Approach: Formal verification

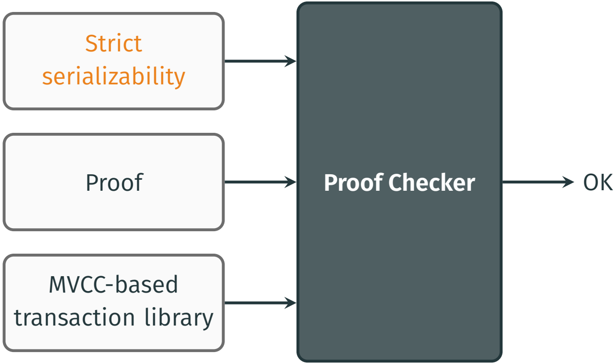


Approach: Formal verification



1. improving concurrency with multiple versions
2. ordering transactions with timestamps

Approach: Formal verification



1. improving concurrency with multiple versions
2. ordering transactions with timestamps

Benefit of formal verification

Proof establishing strictly serializable execution of transactions

Benefit of formal verification

Proof establishing strictly serializable execution of transactions

$\Longrightarrow$ a wide range of bugs are eliminated

- Race conditions
- Out-of-bound accesses
- Off-by-one errors
- Incorrect garbage collection (GC) of versions
- Violation of timestamp monotonicity
- ...

Challenges

1. Requiring a specification for strictly serializable transactions

Challenges

1. Requiring a specification for strictly serializable transactions
2. Proving MVCC transactions execute in some total order despite reordering

Challenges

1. Requiring a specification for strictly serializable transactions
2. Proving MVCC transactions execute in some total order despite reordering
3. Reasoning about garbage collection (GC) and RDTSC-based timestamps

Contributions of this work

vMVCC is the first MVCC-based transaction library with a machine-checked proof of correctness

Contributions of this work

vMVCC is the first MVCC-based transaction library with a machine-checked proof of correctness

- High-performance Go implementation including GC and `RDTSC`-based timestamps

Contributions of this work

vMVCC is the first MVCC-based transaction library with a machine-checked proof of correctness

- High-performance Go implementation including GC and `RDTSC`-based timestamps
- Succinct and application-friendly specification

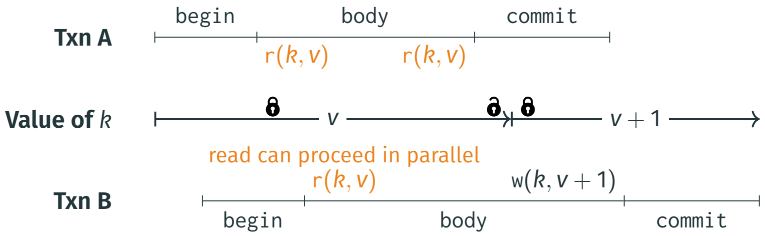
Contributions of this work

vMVCC is the first MVCC-based transaction library with a machine-checked proof of correctness

- High-performance Go implementation including GC and `RDTSC`-based timestamps
- Succinct and application-friendly specification
- Proof adopting prophecy variables [LICS ’88] for MVCC transaction linearization

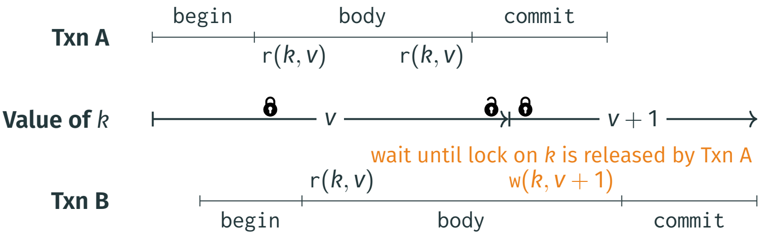
Transactions using two-phase locking (2PL)

Acquiring a lock before reading/writing a key



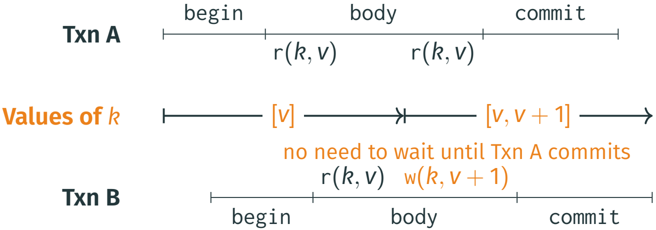
Transactions using two-phase locking (2PL)

Acquiring a lock before reading/writing a key



Benefit of MVCC: More concurrency

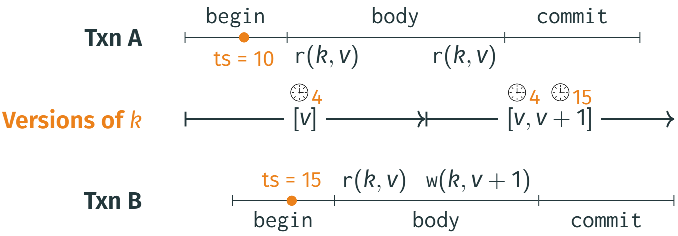
Keeping past values to improve concurrency



Benefit of MVCC: More concurrency

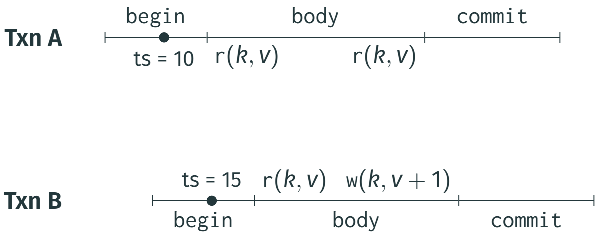
Keeping past values to improve concurrency

Ordering transactions with timestamps



Specifying strictly serializable transactions

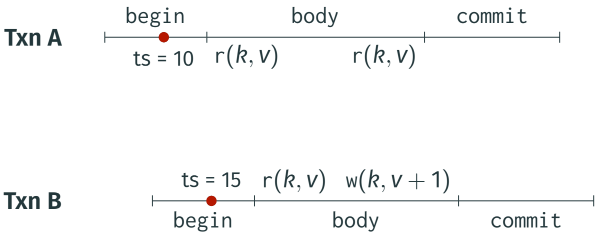
Each transaction appears to execute its reads and writes at its linearization point



Specifying strictly serializable transactions

Each transaction appears to execute its reads and writes at its linearization point

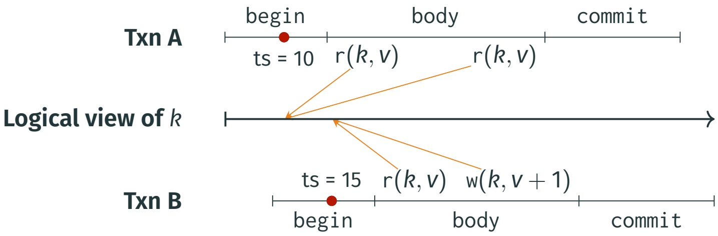
- MVCC transactions linearize exactly when timestamp is generated



Specifying strictly serializable transactions

Each transaction appears to execute its reads and writes at its linearization point

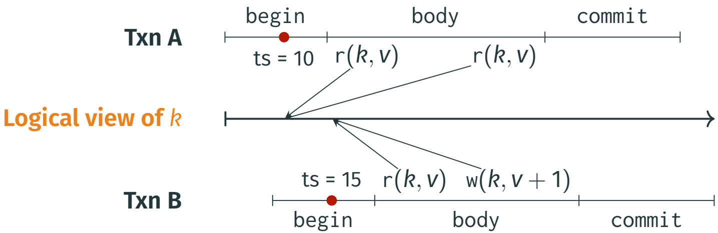
- MVCC transactions linearize exactly when timestamp is generated



Specifying strictly serializable transactions

Each transaction appears to execute its reads and writes at its linearization point

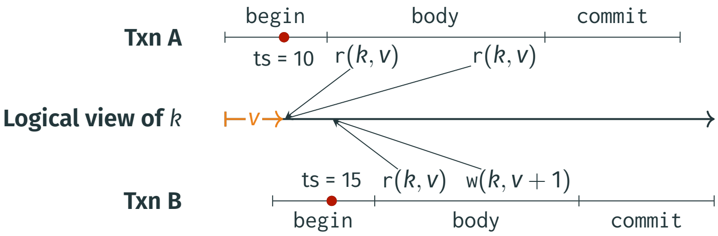
- MVCC transactions linearize exactly when timestamp is generated
- Logical view of the system: the current value for each key



Specifying strictly serializable transactions

Each transaction appears to execute its reads and writes at its linearization point

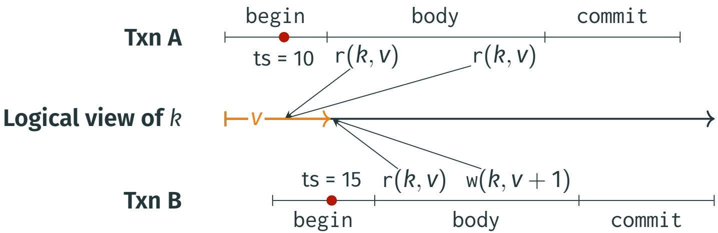
- MVCC transactions linearize exactly when timestamp is generated
- Logical view of the system: the current value for each key



Specifying strictly serializable transactions

Each transaction appears to execute its reads and writes at its linearization point

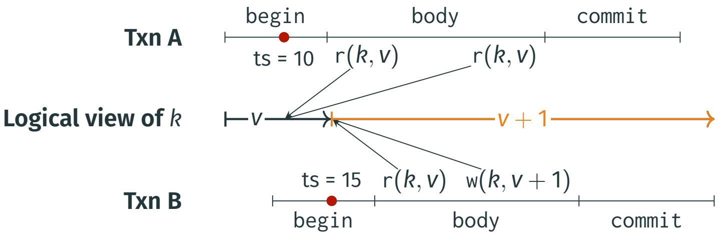
- MVCC transactions linearize exactly when timestamp is generated
- Logical view of the system: the current value for each key



Specifying strictly serializable transactions

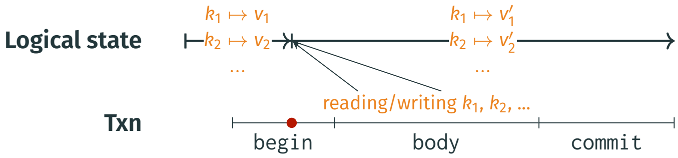
Each transaction appears to execute its reads and writes at its linearization point

- MVCC transactions linearize exactly when timestamp is generated
- Logical view of the system: the current value for each key



Succinct specification catches a wide range of implementation bugs

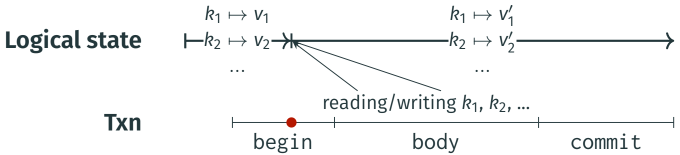
Reading and writing the logical state around the linearization point



Succinct specification catches a wide range of implementation bugs

Reading and writing the logical state around the linearization point

Versioning and timestamps are not mentioned in the specification

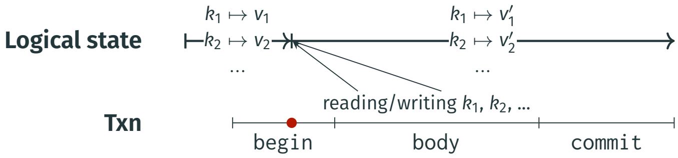


Succinct specification catches a wide range of implementation bugs

Reading and writing the logical state around the linearization point

Versioning and timestamps are not mentioned in the specification

$\implies$ proof ensures correct handling of implementation details



Application-friendly specification reduces proof effort

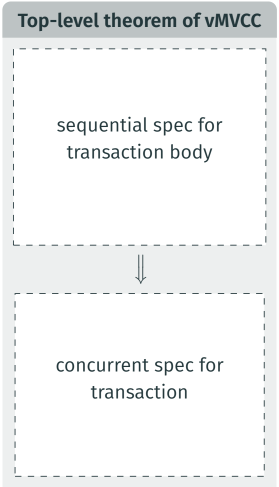
Top-level theorem of vMVCC

sequential spec for transaction body

⇓

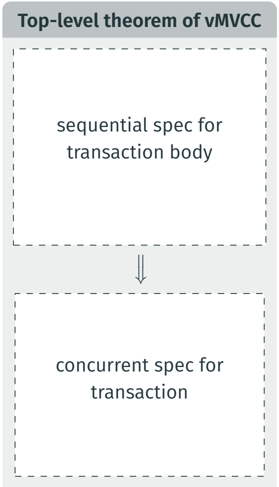
concurrent spec for transaction

Application-friendly specification reduces proof effort



1. Application developer proves the transaction body in an isolated world

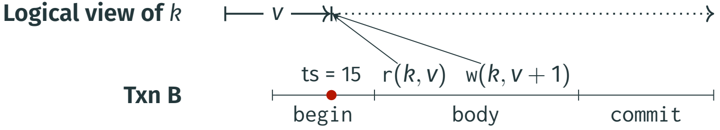
Application-friendly specification reduces proof effort



1. Application developer proves the transaction body in an isolated world
2. vMVCC's top-level theorem ensures safety to run the transaction concurrently

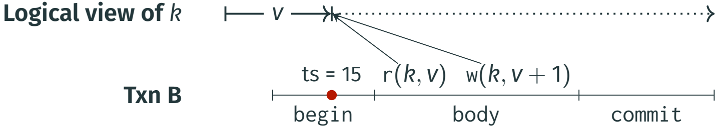
Verification challenge: Transactions linearize before their body runs

Update the logical state requires knowing transaction execution in the future



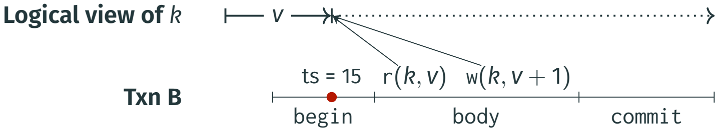
Addressing the challenge with prophecy variables [LICS '88]

1. Speculate whether a transaction commits/aborts and its updates



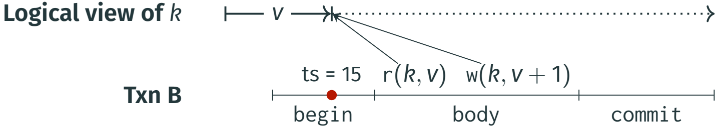
Addressing the challenge with prophecy variables [LICS '88]

1. Speculate whether a transaction commits/aborts and its updates
2. Update the logical state accordingly



Addressing the challenge with prophecy variables [LICS ’88]

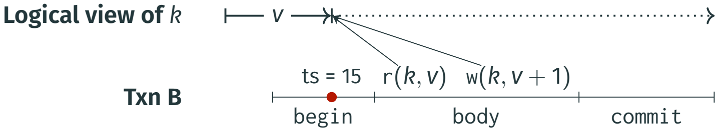
1. Speculate whether a transaction commits/aborts and its updates
2. Update the logical state accordingly
3. Reconcile speculation with reality on commit/abort



Addressing the challenge with prophecy variables [LICS '88]

- ✓ txn B commits and updates k to $v+1$
- ✗ txn B commits and updates k to $v+2$
- ✗ txn B aborts
- ⋮

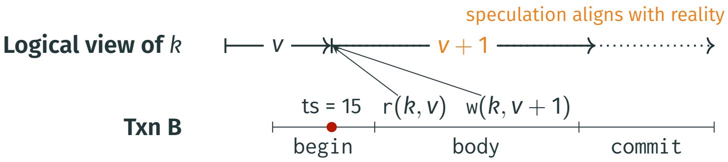
} speculating all possible executions



Addressing the challenge with prophecy variables [LICS ’88]

- ✓ txn B commits and updates k to $v + 1$
- ✗ txn B commits and updates k to $v + 2$
- ✗ txn B aborts
- ⋮

} speculating all possible executions



Addressing the challenge with prophecy variables [LICS '88]

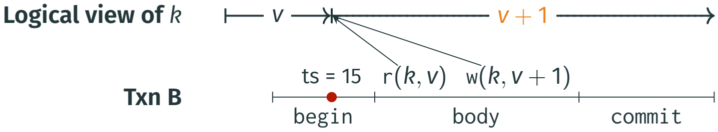
✓ txn B commits and updates k to $v+1$

✗ txn B commits and updates k to $v+2$

✗ txn B aborts

⋮

} speculating all possible executions



Addressing the challenge with prophecy variables [LICS '88]

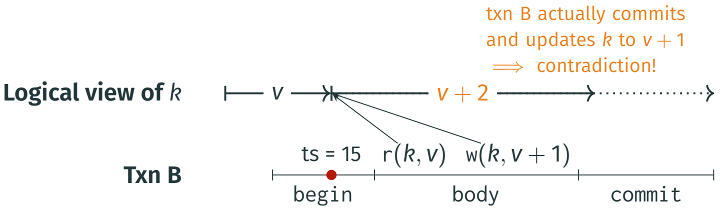
✓ txn B commits and updates k to $v + 1$

✗ txn B commits and updates k to $v + 2$

✗ txn B aborts

⋮

} speculating all possible executions



vMVCC: Implementation and proof efforts

Implementation feature and optimization

- Concurrent GC of unusable versions
- Sharding and padding shared data structures
- Timestamp generation with `RDTSC`

Component	Lines of code
Program	827 (Go)

vMVCC: Implementation and proof efforts

Implementation feature and optimization

- Concurrent GC of unusable versions
- Sharding and padding shared data structures
- Timestamp generation with `RDTSC`

Component	Lines of code
Program	827 (Go)

Proof framework

- Translating Go code with Goose [CoqPL ’20]
- Proof in Perennial [SOSP ’19], Iris [JFP ’18], Coq

vMVCC: Implementation and proof efforts

Implementation feature and optimization

- Concurrent GC of unusable versions
- Sharding and padding shared data structures
- Timestamp generation with RDTSC

Proof framework

- Translating Go code with Goose [CoqPL '20]
- Proof in Perennial [SOSP '19], Iris [JFP '18], Coq

Component	Lines of code
Program	827 (Go)
Spec (4 ops)	42 (Coq)

vMVCC: Implementation and proof efforts

Implementation feature and optimization

- Concurrent GC of unusable versions
- Sharding and padding shared data structures
- Timestamp generation with RDTSC

Proof framework

- Translating Go code with Goose [CoqPL '20]
- Proof in Perennial [SOSP '19], Iris [JFP '18], Coq

Component	Lines of code
Program	827 (Go)
Spec (4 ops)	42 (Coq)
Proof	∼11K (Coq)

vMVCC: 13×

Prior work: 11–20×
GoTxn, CSPEC, CertiKOS, etc.

What good is the proof?

Clarify design by writing spec for whole system and internal components

What good is the proof?

Clarify design by writing spec for whole system and internal components

Encourage writing cleaner code to reduce proof efforts

What good is the proof?

Clarify design by writing spec for whole system and internal components

Encourage writing cleaner code to reduce proof efforts

Caught subtle bugs

- Premature GC of still valid versions
- Violation of strict monotonicity of timestamps
- Off-by-one errors

Evaluation: Is the performance of vMVCC competitive to unverified systems?

Database benchmarks

- YCSB: reading or writing (given a certain R/W ratio) a key sampled uniformly
- TPC-C: modelling the operations of a warehouse wholesale supplier

Experimental setup

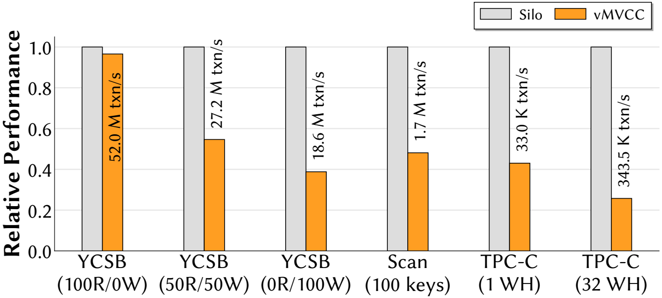
- AWS EC2 instance with 36 vCPUs and 72 GB of main memory

Silo [SOSP ’13]: a state-of-the-art research system

- Single-node in-memory transactional key-value store

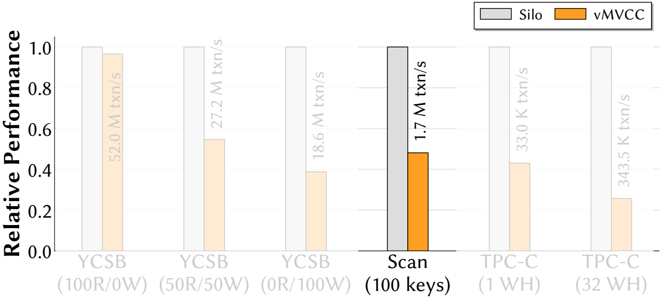
vMVCC is competitive with Silo, the state-of-the-art unverified system

25%–96% of Silo's throughput for YCSB and TPC-C workloads



vMVCC is competitive with Silo, the state-of-the-art unverified system

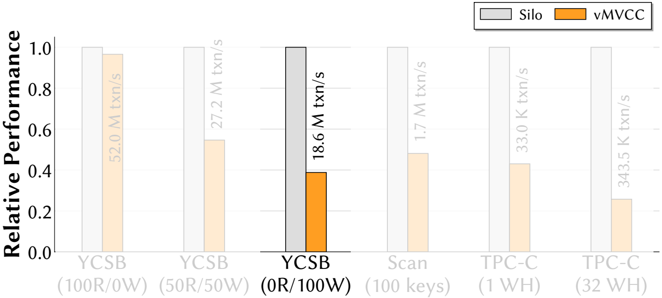
vMVCC lacks a tree-based index



vMVCC is competitive with Silo, the state-of-the-art unverified system

vMVCC lacks a tree-based index

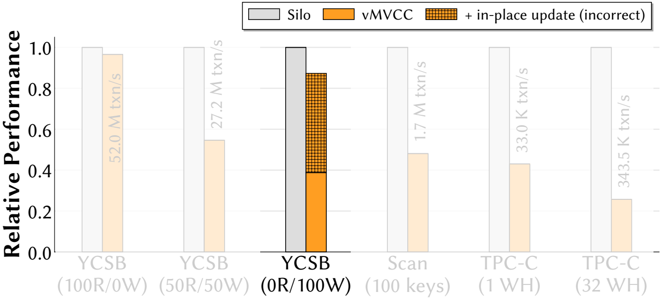
Silo has lower versioning overhead but weaker consistency guarantee



vMVCC is competitive with Silo, the state-of-the-art unverified system

vMVCC lacks a tree-based index

Silo has lower versioning overhead but weaker consistency guarantee



Related work

Reasoning about transactions

- Push/pull model [PLDI ’15]
- C4 [OOPSLA ’22]

Related work

Reasoning about transactions

- Push/pull model [PLDI ’15]
- C4 [OOPSLA ’22]

Prophecy variables

- RDCSS, Herlihy-Wing Queue [POPL ’20]
- Atomic snapshot [TOPLAS ’22]

Related work

Reasoning about transactions

- Push/pull model [PLDI ’15]
- C4 [OOPSLA ’22]

Prophecy variables

- RDCSS, Herlihy-Wing Queue [POPL ’20]
- Atomic snapshot [TOPLAS ’22]

Verified transaction library

- GoTxn [OSDI ’22]

Conclusion

vMVCC is the first MVCC-based transaction library with a machine-checked proof of correctness

- Sophisticated implementation to achieve high performance
- Succinct and application-friendly specification
- Formal proof adopting prophecy variables for MVCC transactions

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