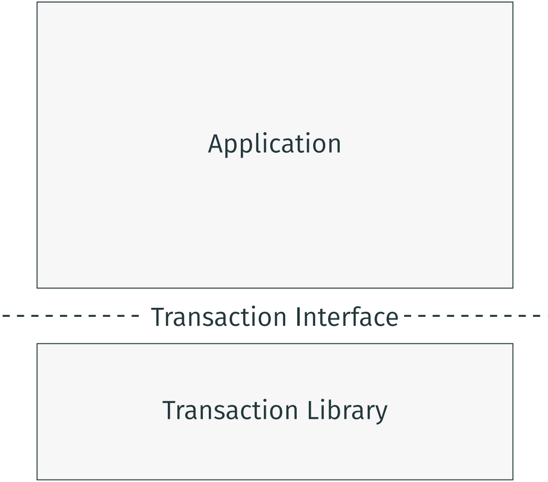
Verifying vMVCC, a high-performance database using multi-version concurrency control

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

MIT CSAIL [ϕ]ETH Zürich [†]NYU

Transactional programming model



Transactional programming model

```
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
}
```

---------- Transaction Interface ----------

Transaction Library

Transactional programming model

```go
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
}
```

---------- Transaction Interface ----------

Transaction Library

Transactional programming model

```go
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
    txn.Write(src, sbal - amt)



}
```

---------- Transaction Interface ----------

Transaction Library

Transactional programming model

```go
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
    txn.Write(src, sbal - amt)
    dbal, _ := txn.Read(dst)
    txn.Write(dst, dbal + amt)

}
```

---------- Transaction Interface ----------

Transaction Library

Transactional programming model

```
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
    txn.Write(src, sbal - amt)
    dbal, _ := txn.Read(dst)
    txn.Write(dst, dbal + amt)
    return true
}
```

Transaction Interface

Transaction Library

Transactional programming model

```go
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
    txn.Write(src, sbal - amt)
    dbal, _ := txn.Read(dst)
    txn.Write(dst, dbal + amt)
    return true
}
```

```go
txn := Begin()
c := xfer(txn, src, dst, amt)
if c {
    txn.Commit()
} else {
    txn.Abort()
}
```

---------- Transaction Interface ----------

Transaction Library

Transactional programming model

```
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
    txn.Write(src, sbal - amt)
    dbal, _ := txn.Read(dst)
    txn.Write(dst, dbal + amt)
    return true
}
```

```
txn := Begin()
c := xfer(txn, src, dst, amt)
if c {
    txn.Commit()
} else {
    txn.Abort()
}
```

---------- Transaction Interface ----------

vMVCC

Transactional programming model

```go
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
    txn.Write(src, sbal - amt)
    dbal, _ := txn.Read(dst)
    txn.Write(dst, dbal + amt)
    return true
}
```

```go
txn := Begin()
c := xfer(txn, src, dst, amt)
if c {
    txn.Commit()
} else {
    txn.Abort()
}
```

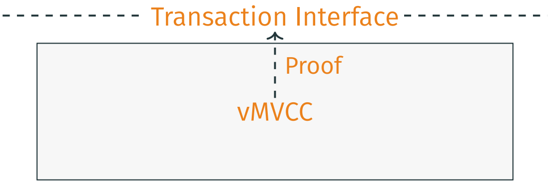
---------- Transaction Interface ----------

vMVCC

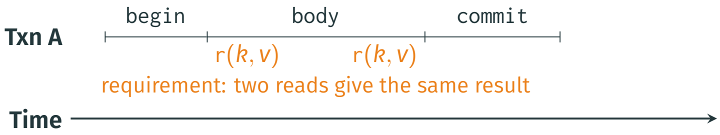
Transactional programming model

```
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    if sbal < amt {
        return false
    }
    txn.Write(src, sbal - amt)
    dbal, _ := txn.Read(dst)
    txn.Write(dst, dbal + amt)
    return true
}
```

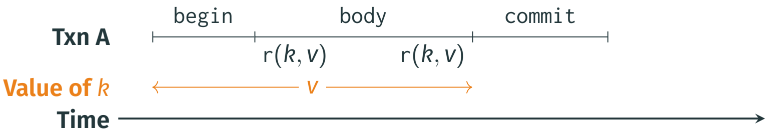
```
txn := Begin()
c := xfer(txn, src, dst, amt)
if c {
    txn.Commit()
} else {
    txn.Abort()
}
```



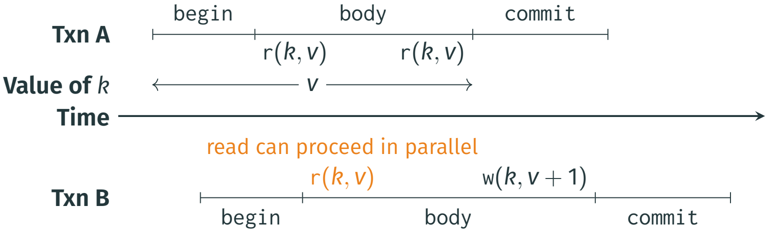
Transactions using locks



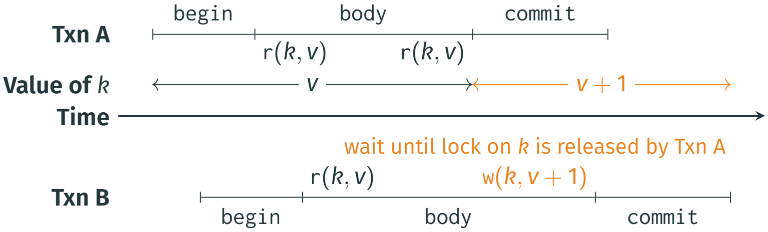
Transactions using locks



Transactions using locks

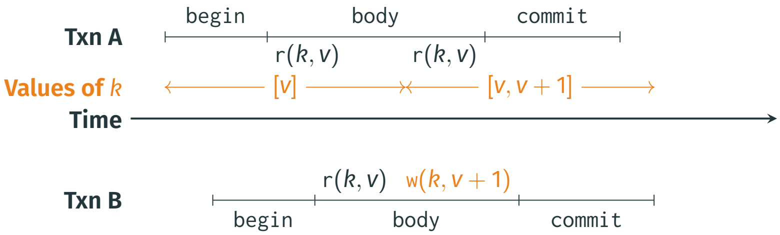


Transactions using locks



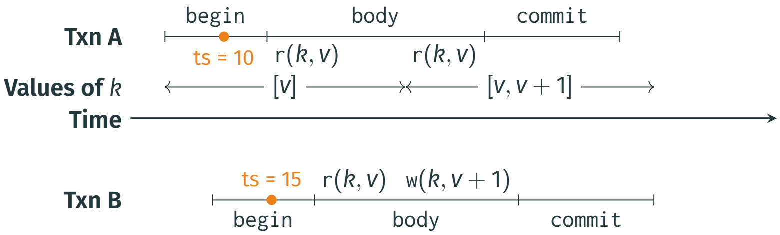
Transactions using multi-version concurrency control (MVCC)

- Keeping past values to improve concurrency



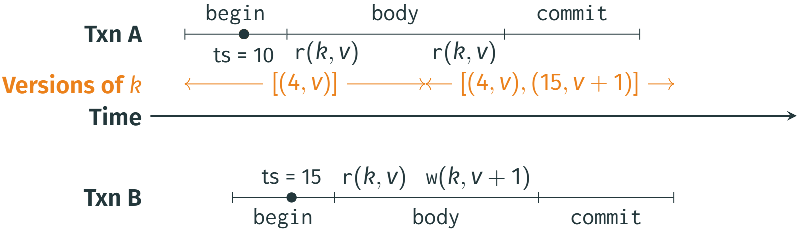
Transactions using multi-version concurrency control (MVCC)

- Keeping past values to improve concurrency
- Ordering transactions with timestamps



Transactions using multi-version concurrency control (MVCC)

- Keeping past values to improve concurrency
- Ordering transactions with timestamps



Contribution: Verifying an MVCC-based transaction library implementation

- A practical and high-performance implementation written in Go
  - E.g., concurrent GC of unusable versions and RDTSC-based timestamps

Contribution: Verifying an MVCC-based transaction library implementation

- A practical and high-performance implementation written in Go
  - E.g., concurrent GC of unusable versions and `RDTSC`-based timestamps
- Requiring sophisticated reasoning techniques
  - E.g., logical atomicity and prophecy variables

Outline

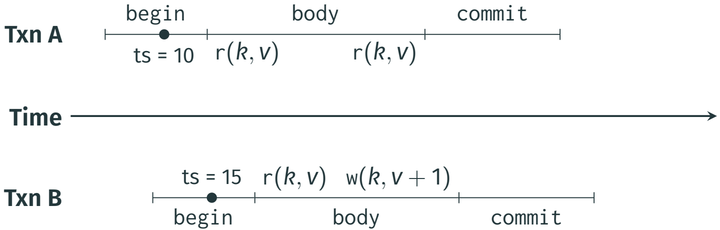
- Specifying and verifying MVCC transactions
- Low-level optimization: RDTSC-based timestamps
- Evaluation
- Conclusion

Outline

- Specifying and verifying MVCC transactions
- Low-level optimization: `RDTSC`-based timestamps
- Evaluation
- Conclusion

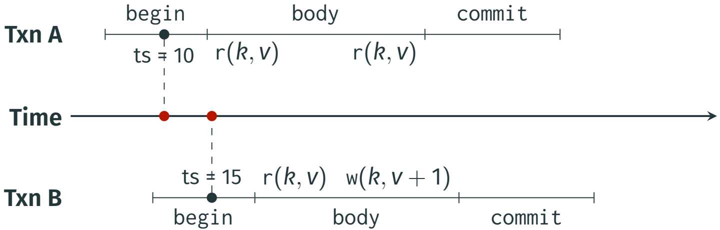
Transactions, intuitively

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



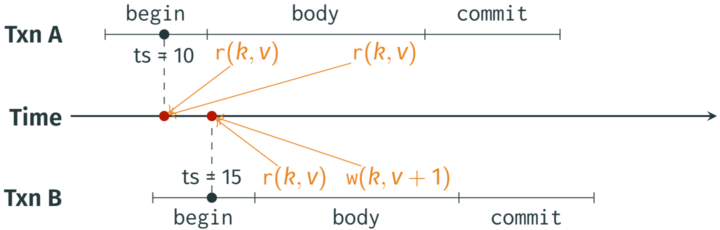
Transactions, intuitively

- Each transaction appears to execute its reads and writes at its linearization point
  - MVCC transactions linearize exactly when timestamp is generated

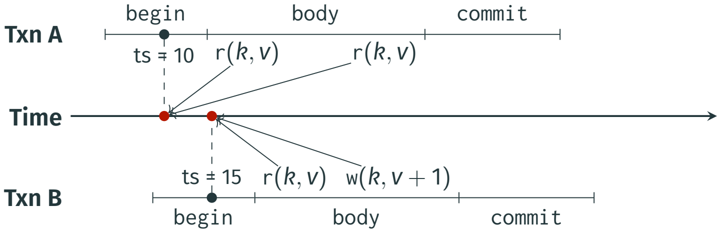


Transactions, intuitively

- Each transaction appears to execute its reads and writes at its linearization point
  - MVCC transactions linearize exactly when timestamp is generated

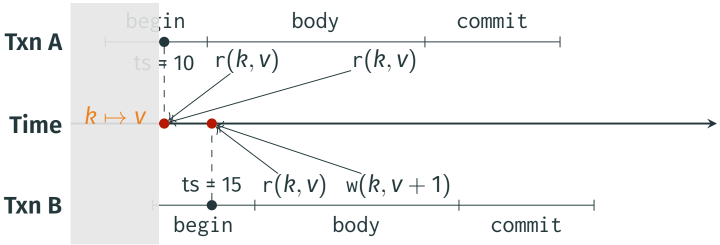


Logical (ghost) state of the database

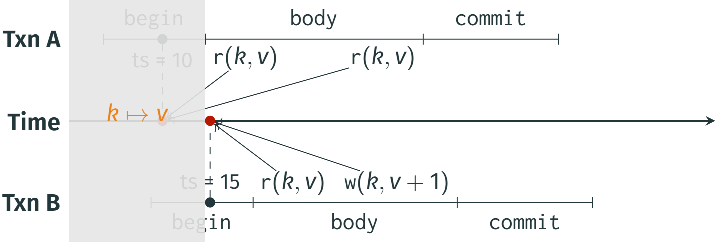


Logical (ghost) state of the database

- The current value for each key: $k \mapsto v$

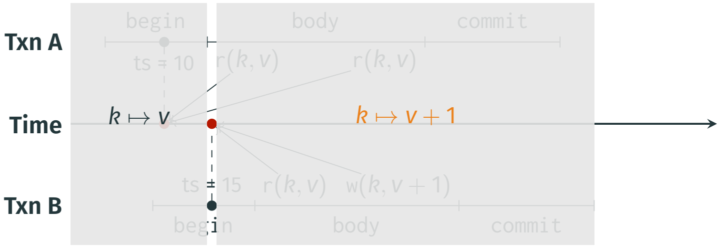


- The current value for each key: $k \mapsto v$



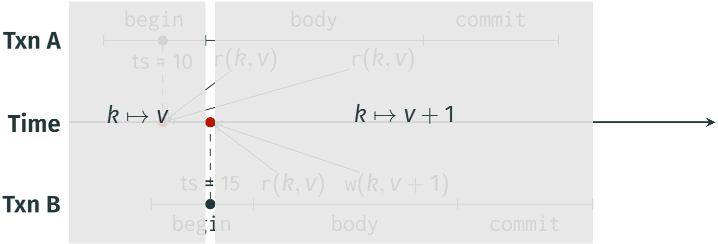
Logical (ghost) state of the database

- The <u>current value</u> for each key: $k \mapsto v$

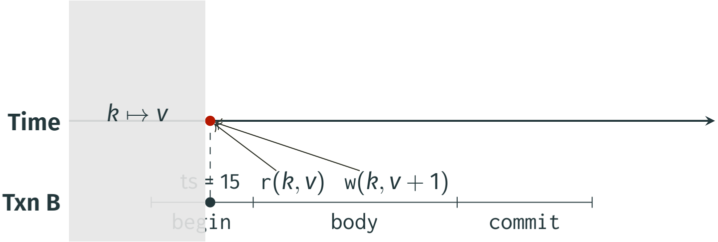


Logical (ghost) state of the database

- The <u>current value</u> for each key: $k \mapsto v$
  - Mismatch in MVCC: multi-version physical layout vs. single-value logical view

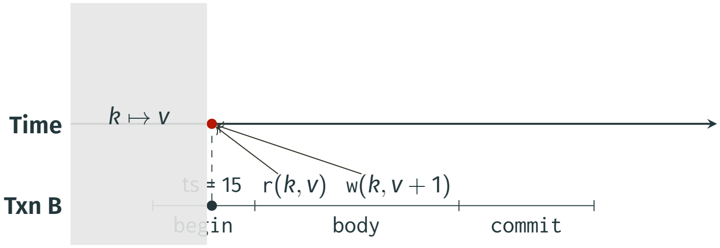


Verification challenge: Transaction linearizes before its body runs



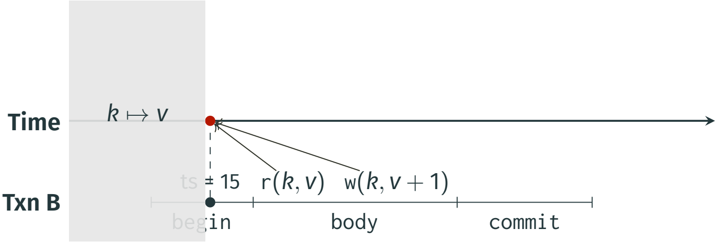
Verification challenge: Transaction linearizes before its body runs

- To update the logical state, we need to know:
  - <u>Will</u> this transaction commit or abort?



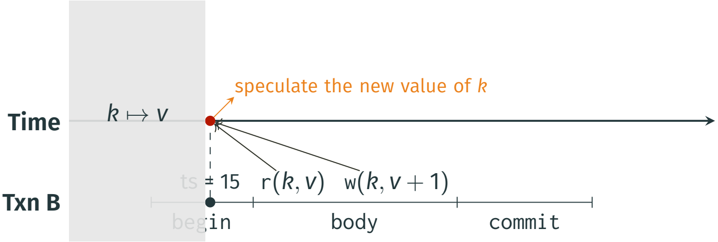
Verification challenge: Transaction linearizes before its body runs

- To update the logical state, we need to know:
  - Will this transaction commit or abort?
  - What will this transaction write?

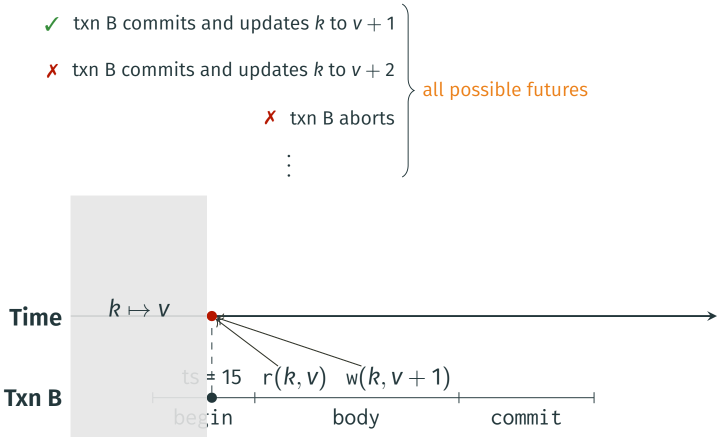


Verification challenge: Transaction linearizes before its body runs

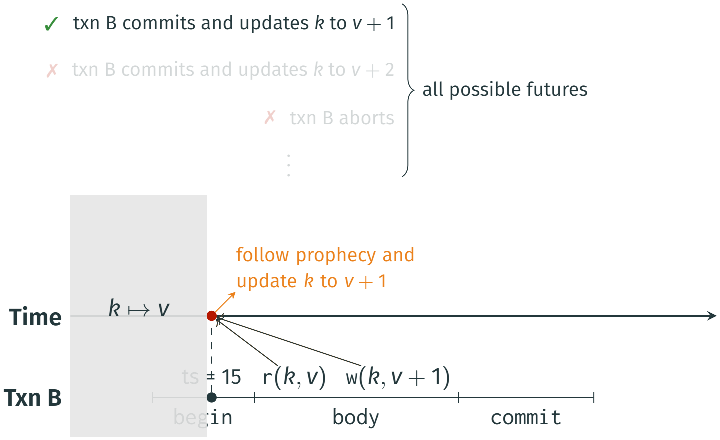
- To update the logical state, we need to know:
  - Will this transaction commit or abort?
  - What will this transaction write?
- **Solution**: Prophecy variables



Peeking into the future with prophecy variable



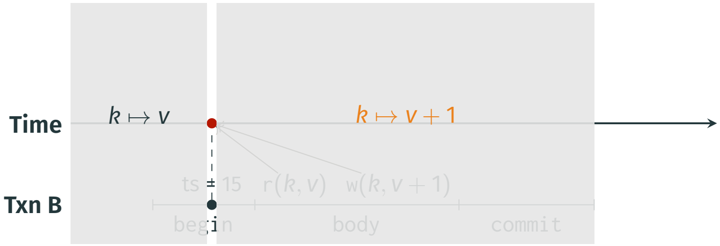
Peeking into the future with prophecy variable



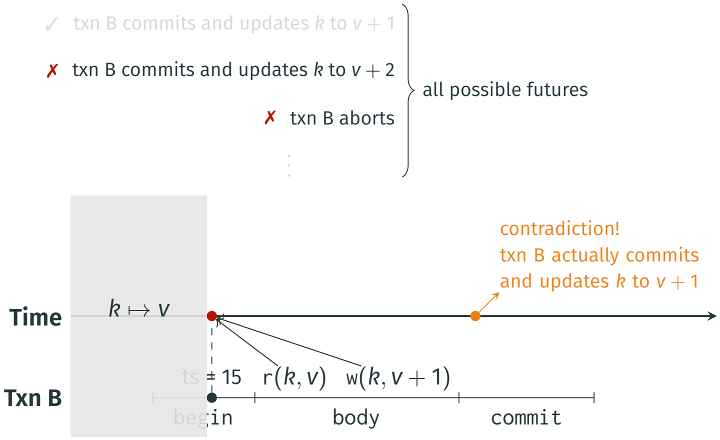
Peeking into the future with prophecy variable

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

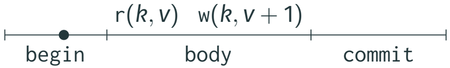
} all possible futures



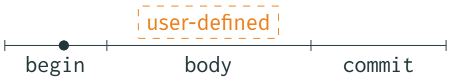
Peeking into the future with prophecy variable



Specifying general transactions with logical atomicity



Specifying general transactions with logical atomicity



Specifying general transactions with logical atomicity

Specifying general transactions with logical atomicity

```
func xfer(txn *Txn, src, dst, amt uint64) bool {
   sbal, _ := txn.Read(src)
   if sbal < amt {
      return false
   }
   txn.Write(src, sbal - amt)
   dbal, _ := txn.Read(dst)
   txn.Write(dst, dbal + amt)
   return true
}
```

Specifying general transactions with logical atomicity

∀ body.

sequential spec for
body(txn)

Specifying general transactions with logical atomicity

∀ body.

sequential spec for
body(txn)

⇓

concurrent spec for
db.Run(body)

Specifying general transactions with logical atomicity

$$\forall\, \text{body}.\ \left\{ \qquad\qquad \right\}$$

`body(txn)`

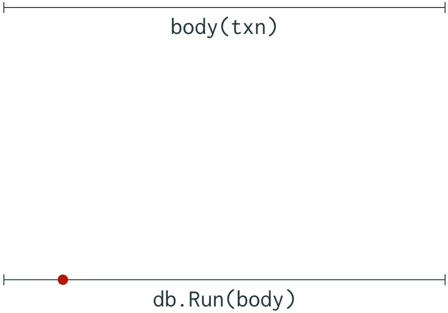
$$\left\{ \qquad\qquad \right\}$$

$$\Downarrow$$

$$\left\langle \qquad\qquad \right\rangle$$

`db.Run(body)`

$$\left\langle \qquad\qquad \right\rangle$$



Specifying general transactions with logical atomicity

$$\forall\, \text{body}.\ \left\{ \mathop{\ast}_{(k,v)\in m} k \overset{t}{\mapsto} v \right\}$$

`body(txn)`

$$\left\{ \qquad \right\}$$

$$\Downarrow$$

$$\left\langle \qquad \right\rangle$$

`db.Run(body)`

$$\left\langle \qquad \right\rangle$$

Specifying general transactions with logical atomicity

$$\forall\, \text{body}.\ \left\{ \mathop{\ast}_{(k,v)\in m} k \overset{t}{\mapsto} v \right\}$$

`body(txn)`

$$\left\{ \qquad \right\}$$

$$\Downarrow$$

$$\left\langle \qquad \right\rangle$$

`db.Run(body)`

$$\left\langle \qquad \right\rangle$$

take $\overset{t}{\mapsto}$: transaction-local view of database

$k_1 \overset{t}{\mapsto} v_1 * \cdots * k_n \overset{t}{\mapsto} v_n$

`body(txn)`

`db.Run(body)`

Specifying general transactions with logical atomicity

$$\forall \text{ body}.\ \left\{ \mathop{\Huge\ast}_{(k,v)\in m} k \overset{t}{\mapsto} v * P(m) \right\}$$

`body(txn)`

$$\Downarrow$$

`db.Run(body)`

take $P(m)$ describes what's in the starting state

$k_1 \overset{t}{\mapsto} v_1 * \cdots * k_n \overset{t}{\mapsto} v_n$

`body(txn)`

`db.Run(body)`

Specifying general transactions with logical atomicity

$$\forall\, \text{body}.\ \left\{ \mathop{\huge\ast}_{(k,v)\in m} k \overset{t}{\mapsto} v * P(m) \right\}$$

$$\texttt{body(txn)}$$

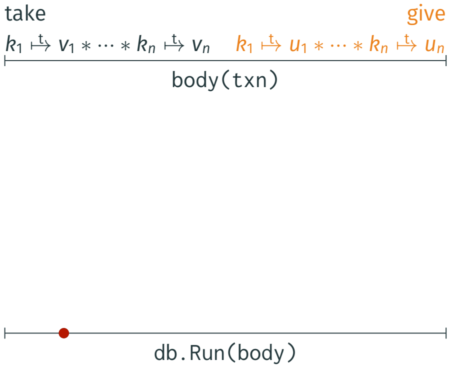
$$\left\{ \mathop{\huge\ast}_{(k,v)\in m'} k \overset{t}{\mapsto} v \right\}$$

$$\Downarrow$$

$$\left\langle \qquad \right\rangle$$

$$\texttt{db.Run(body)}$$

$$\left\langle \qquad \right\rangle$$



Specifying general transactions with logical atomicity

$$\forall\, \text{body}.\ \left\{ \mathop{\Large *}_{(k,v)\in m} k \overset{t}{\mapsto} v * P(m) \right\}$$

body(txn)

$$\left\{ \mathop{\Large *}_{(k,v)\in m'} k \overset{t}{\mapsto} v * Q(m, m') \right\}$$

$$\Downarrow$$

$$\Big\langle \qquad \Big\rangle$$

db.Run(body)

$$\Big\langle \qquad \Big\rangle$$

take $\quad Q(m, m')$ describes the changes $\quad$ give

$k_1 \overset{t}{\mapsto} v_1 * \cdots * k_n \overset{t}{\mapsto} v_n \qquad k_1 \overset{t}{\mapsto} u_1 * \cdots * k_n \overset{t}{\mapsto} u_n$

body(txn)

db.Run(body)

Specifying general transactions with logical atomicity

$$\forall\, \text{body}.\ \left\{ \mathop{\Huge *}_{(k,v)\in m} k \stackrel{t}{\mapsto} v * P(m) \right\}$$

$$\texttt{body(txn)}$$

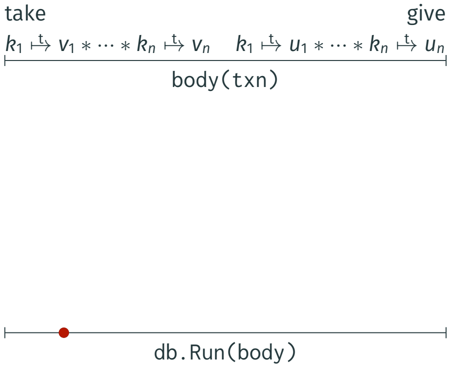
$$\left\{ \mathop{\Huge *}_{(k,v)\in m'} k \stackrel{t}{\mapsto} v * Q(m, m') \right\}$$

$$\Downarrow$$

$$\left\langle m.\ \mathop{\Huge *}_{(k,v)\in m} k \mapsto v * P(m) \right\rangle$$

$$\texttt{db.Run(body)}$$

$$\left\langle \mathop{\Huge *}_{(k,v)\in m'} k \mapsto v * Q(m, m') \right\rangle$$



Specifying general transactions with logical atomicity

$$\forall\, \mathrm{body}.\ \left\{ \mathop{\Large\ast}_{(k,v)\in m} k \overset{t}{\mapsto} v \ast P(m) \right\}$$

body(txn)

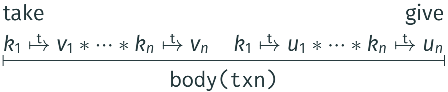
$$\left\{ \mathop{\Large\ast}_{(k,v)\in m'} k \overset{t}{\mapsto} v \ast Q(m, m') \right\}$$

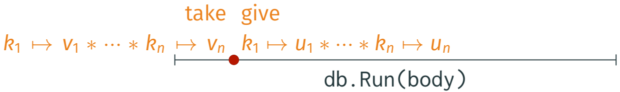
$$\Downarrow$$

$$\left\langle m.\ \mathop{\Large\ast}_{(k,v)\in m} k \mapsto v \ast P(m) \right\rangle$$

db.Run(body)

$$\left\langle \mathop{\Large\ast}_{(k,v)\in m'} k \mapsto v \ast Q(m, m') \right\rangle$$





Specifying general transactions with logical atomicity

$$\forall\, \text{body}.\ \left\{ \mathop{\Large\ast}_{(k,v)\in m} k \overset{t}{\mapsto} v \ast P(m) \right\}$$

`body(txn)`

$$\left\{ \mathop{\Large\ast}_{(k,v)\in m'} k \overset{t}{\mapsto} v \ast Q(m, m') \right\}$$

$$\Downarrow$$

$$\left\langle m.\ \mathop{\Large\ast}_{(k,v)\in m} k \mapsto v \ast P(m) \right\rangle$$

`db.Run(body)`

$$\left\langle \mathop{\Large\ast}_{(k,v)\in m'} k \mapsto v \ast Q(m, m') \right\rangle$$

take give

$k_1 \overset{t}{\mapsto} v_1 \ast \cdots \ast k_n \overset{t}{\mapsto} v_n \quad k_1 \overset{t}{\mapsto} u_1 \ast \cdots \ast k_n \overset{t}{\mapsto} u_n$

`body(txn)`

transactions are allowed to run concurrently!

`db.Run(body)`

vMVCC: Abstractions and invariants

LOGICAL $(k \mapsto v_4)$ v_4

PHYSICAL

ts	del	val	tslast
1	false	v_1	4
3	false	v_3	

vMVCC: Abstractions and invariants

LOGICAL $(k \mapsto v_4)$ v_4

LINEAR $\bot$ $\bot$ v_1 v_1 v_3

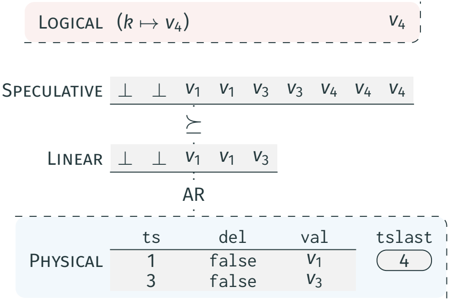
AR

PHYSICAL

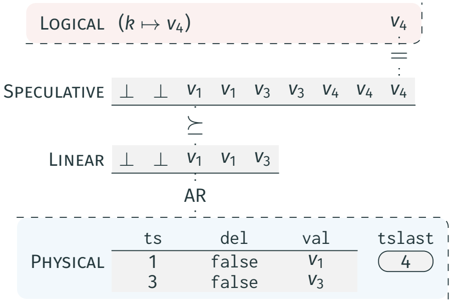
ts	del	val
1	false	v_1
3	false	v_3

tslast: 4

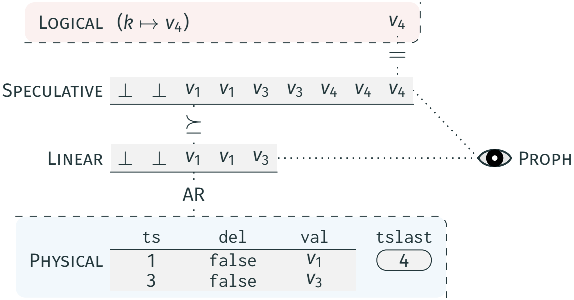
vMVCC: Abstractions and invariants



vMVCC: Abstractions and invariants



vMVCC: Abstractions and invariants



Outline

- Specifying and verifying transactions
- Low-level optimization: `RDTSC`-based timestamps
- Evaluation
- Conclusion

Generating strictly increasing timestamps

Timestamp schemes

- a global lock on a shared counter

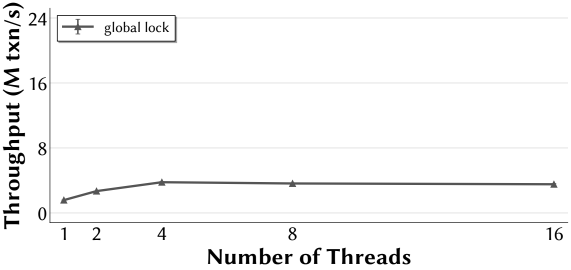


Figure 1: Scalability analysis of different timestamp schemes under the YCSB workload (1 key accessed per transaction, $\theta = 0.2$).

Generating strictly increasing timestamps

Timestamp schemes

- a global lock on a shared counter
- `FAI` on a shared counter

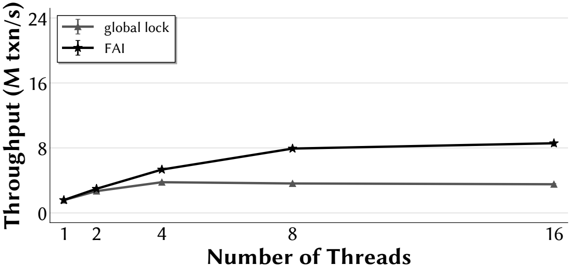


Figure 1: Scalability analysis of different timestamp schemes under the YCSB workload (1 key accessed per transaction, $\theta = 0.2$).

Generating strictly increasing timestamps

Timestamp schemes

- a global lock on a shared counter
- `FAI` on a shared counter
- `RDTSC` on CPU hardware counters

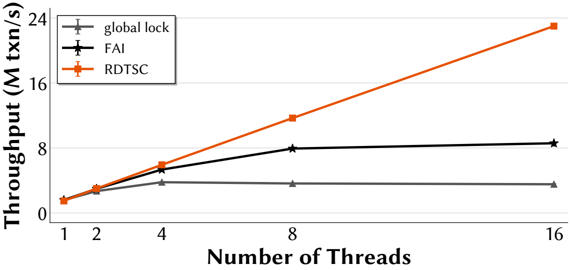


Figure 1: Scalability analysis of different timestamp schemes under the YCSB workload (1 key accessed per transaction, $\theta = 0.2$).

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()



    return tid
}
```

RDTSC()

↓

tid = 101011110

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()



    return tid
}
```

RDTSC()

↓

tid = 101011110

Transaction site

- Each site is assigned a short unique ID (e.g., 5 bits)

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()
    tid = RoundUp(tid, sid)


    return tid
}
```

RDTSC()

↓

tid = 101100101

(on site 5)

Transaction site

- Each site is assigned a short unique ID (e.g., 5 bits)

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()
    tid = RoundUp(tid, sid)


    return tid
}
```

RDTSC()

↓

tid = 101100101

(on site 5)

Transaction site

- Each site is assigned a short unique ID (e.g., 5 bits)
- Acquiring a per-site mutex before calling `GenTID`

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()
    tid = RoundUp(tid, sid)
    for RDTSC() <= tid {
    }
    return tid
}
```

RDTSC()

↓

tid = 101<u>00101</u>

(on site 5)

Transaction site

- Each site is assigned a short unique ID (e.g., 5 bits)
- Acquiring a per-site mutex before calling `GenTID`

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()
    tid = RoundUp(tid, sid)
    for RDTSC() <= tid {
    }
    return tid
}
```

RDTSC()

↓

tid = 101<u>00101</u>

(on site 5)

Transaction site

- Each site is assigned a short unique ID (e.g., 5 bits)
- Acquiring a per-site mutex before calling `GenTID`

Proof challenge: helping

- Linearizes at the next `RDTSC` returning a larger value (might be called by a different thread)

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()
    tid = RoundUp(tid, sid)
    for RDTSC() <= tid {
    }
    return tid
}
```

RDTSC()

↓

tid = 101<u>00101</u>

(on site 5)

Transaction site

- Each site is assigned a short unique ID (e.g., 5 bits)
- Acquiring a per-site mutex before calling `GenTID`

Proof challenge: unsolicited helping

- Linearizes at the next `RDTSC` returning a larger value (might be called by a different thread)
- No explicit communication between threads

Generating strictly increasing timestamps with RDTSC

```
func GenTID(sid uint64) uint64 {
    var tid uint64
    tid = RDTSC()
    tid = RoundUp(tid, sid)
    for RDTSC() <= tid {
    }
    return tid
}
```

RDTSC()

↓

tid = 101<u>00101</u>

(on site 5)

Transaction site

- Each site is assigned a short unique ID (e.g., 5 bits)
- Acquiring a per-site mutex before calling `GenTID`

Proof challenge: unsolicited helping

- Linearizes at the next `RDTSC` returning a larger value (might be called by a different thread)
- No explicit communication between threads
- Later credits!

vMVCC: Implementation and proof efforts

Implementation feature and optimization

- Concurrent garbage collection of unusable versions
- Lock sharding and padding
- Timestamp generation with `RDTSC`

Component	Lines of code
Program	∼800 (Go)

vMVCC: Implementation and proof efforts

Implementation feature and optimization

- Concurrent garbage collection of unusable versions
- Lock sharding and padding
- Timestamp generation with `RDTSC`

Proof framework

- Translating Go code with Goose and proving in Perennial/Iris/Coq

Component	Lines of code
Program	$\sim$800 (Go)
Proof	$\sim$11K (Coq)

Outline

- Specifying and verifying transactions
- Low-level optimization: `RDTSC`-based timestamps
- Evaluation
- Conclusion

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

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

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

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

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

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

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

- Single-node in-memory transactional key-value store
- Creating one version every one second $\Longrightarrow$ less memory and better performance

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

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

- Single-node in-memory transactional key-value store
- Creating one version every one second $\implies$ less memory and better performance
- Read-only transactions not linearizable $\implies$ weaker consistency level

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

Observation

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

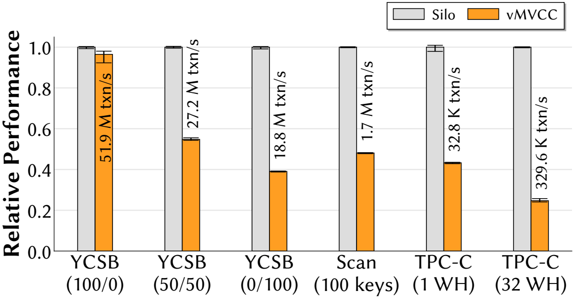


Figure 2: Comparison of Silo and vMVCC. For YCSB, each transaction reads or writes a key sampled from a uniform distribution with a certain R/W ratio. For TPC-C, the number of warehouses is same as the number of worker threads.

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

Observation

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

Performance difference

- Lack of a tree-based index

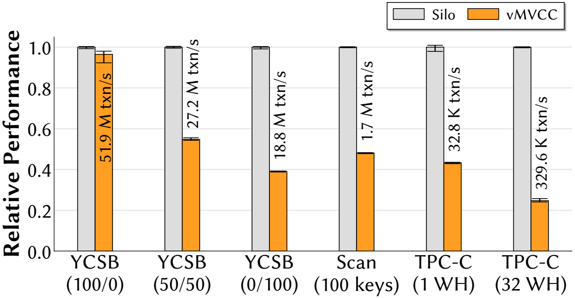


Figure 2: Comparison of Silo and vMVCC. For YCSB, each transaction reads or writes a key sampled from a uniform distribution with a certain R/W ratio. For TPC-C, the number of warehouses is same as the number of worker threads.

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

Observation

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

Performance difference

- Lack of a tree-based index
- Higher memory management overhead

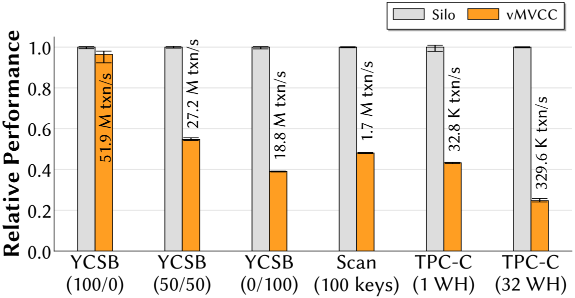


Figure 2: Comparison of Silo and vMVCC. For YCSB, each transaction reads or writes a key sampled from a uniform distribution with a certain R/W ratio. For TPC-C, the number of warehouses is same as the number of worker threads.

Conclusion

Contribution

- A logically-atomic specification for transactions
- A proof approach using prophecy variable for MVCC transaction linearization
- A verified high-performance transaction library using MVCC and low-level optimizations such as `RDTSC`-based timestamps

Conclusion

Contribution

- A logically-atomic specification for transactions
- A proof approach using prophecy variable for MVCC transaction linearization
- A verified high-performance transaction library using MVCC and low-level optimizations such as `RDTSC`-based timestamps

Thank you Iris!

- Specification: logical atomicity and resource algebras
- Proof: invariants, prophecy variables, and later credits