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

txn := Begin()
c := xfer(txn, src, dst, amt)
if c {
    txn.Commit()
} else {
    txn.Abort()
}
func xfer(txn *Txn, src, dst, amt uint64) bool {
    sbal, _ := txn.Read(src)
    
}

--- Transaction Interface ---

--- Transaction Library ---
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 ---
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 ---
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
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
}
**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
}
```

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

Txn A

begin

body

commit

\( r(k,v) \) \( r(k,v) \)

requirement: two reads give the same result

Time
Transactions using locks

**Requirement:** two reads give the same result

**Txn A**
- `begin`
- `r(k, v)`
- `body`
- `r(\overline{k}, \overline{v})`
- `commit`

**Txn B**
- `begin`
- `r(k, v)`
- `body`
- `w(k, \overline{v} + 1)`
- `commit`

Read can proceed in parallel
Wait until lock on `k` is released by Txn A

**Value of k**

**Time**

2 versions of `k`
Transactions using locks

**Value of $k$**

<table>
<thead>
<tr>
<th>Txn A</th>
<th>begin</th>
<th>body</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r(k, v)$</td>
<td>$r(k, v)$</td>
<td></td>
</tr>
</tbody>
</table>

**Time**

<table>
<thead>
<tr>
<th>Txn B</th>
<th>begin</th>
<th>body</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r(k, v)$</td>
<td>$w(k, v + 1)$</td>
<td></td>
</tr>
</tbody>
</table>

**Requirement:** Two reads give the same result.

Read can proceed in parallel when the lock on $k$ is released by Txn A.
Transactions using locks

**Transaction A**
- **Begin**
- **Body**: \( r(k, v) \) \( \rightarrow \) \( r(k, v) \)
- **Commit**

**Transaction B**
- **Begin**
- **Body**: \( r(k, v) \)
- **Commit**

**Value of** \( k \)
- **Time**: \( v \) \( \rightarrow \) \( v + 1 \)

**Wait until lock on** \( k \) **is released by Transaction A**

**Versions of** \( k \)
- \( v \) \( + \) \( 1 \)

**Diagram**

- **Txn A**
  - Begin
  - Body: \( r(k, v) \) \( \rightarrow \) \( r(k, v) \)
  - Commit

- **Txn B**
  - Begin
  - Body: \( r(k, v) \)
  - Commit

- **Value of** \( k \)
  - \( v \) \( \rightarrow \) \( v + 1 \)

- **Wait until lock on** \( k \) **is released by** Txn A

**Version**
- \( k \) \( + \) \( 1 \)
Transactions using multi-version concurrency control (MVCC)

- Keeping past values to improve concurrency

\[ \begin{align*}
\text{Txn A} & \quad \text{begin} \quad \text{body} \quad \text{commit} \\
& \quad r(k, v) \quad r(k, v) \\
\text{Values of } k & \quad \leftarrow [v] \times [v, v + 1] \rightarrow \\
\text{Time} & \\
\text{Txn B} & \quad \text{begin} \quad \text{body} \quad \text{commit} \\
& \quad r(k, v) \quad w(k, v + 1)
\end{align*} \]
Transactions using multi-version concurrency control (MVCC)

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

\[
\begin{align*}
\text{Begin} & \quad \text{body} \quad \text{commit} \\
\text{Txn A} & \\
\text{values of } k & \\
\text{time} & \\
\text{Txn B} & \\
\end{align*}
\]
Transactions using multi-version concurrency control (MVCC)

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

```
Time

<table>
<thead>
<tr>
<th>Txn A</th>
<th>begin</th>
<th>body</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ts = 10 r(k, v) r(k, v)</td>
<td></td>
</tr>
</tbody>
</table>

Versions of k

<table>
<thead>
<tr>
<th>k</th>
<th>[(4, v)]</th>
<th>[4, v), (15, v + 1)]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Txn B</th>
<th>begin</th>
<th>body</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ts = 15 r(k, v) w(k, v + 1)</td>
<td></td>
</tr>
</tbody>
</table>
```
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

```
begin
body
commit

Txn A

<table>
<thead>
<tr>
<th></th>
<th>begin</th>
<th>body</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts = 10</td>
<td>r(k, v)</td>
<td>r(k, v)</td>
<td></td>
</tr>
</tbody>
</table>

Versions of k

<table>
<thead>
<tr>
<th></th>
<th>r(k, v)</th>
<th>w(k, v + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts = 15</td>
<td>(4, v)</td>
<td>(4, v), (15, v + 1)</td>
</tr>
</tbody>
</table>

Time
```

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

**Timelines:**
- **Txn A**
  - Begin: $ts = 10$
  - Body: $r(k,v)$
  - Commit: $r(k,v)$

- **Txn B**
  - Begin: $ts = 15$
  - Body: $r(k,v)$
  - Write: $w(k,v+1)$

**Versions of $k$:**
- $v$, $v+1$
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

- The current value for each key: $k_7!_v$

Mismatch in MVCC: multi-version physical layout vs. single-value logical view

Time

- **Txn A**
  - begin $ts = 10$
  - body $r(k,v)$ $r(k,v)$

- **Txn B**
  - begin $ts = 15$
  - body $r(k,v)$ $w(k,v + 1)$

Versions of $k_7!_v$ $k_7!_v$ $k_7!_v$ $k_7!_v + 1$
Logical (ghost) state of the database

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

```
begin
body
commit

Txn A

ts = 10
r(k, v)

Time

k \mapsto v

Ts = 15
r(k, v)
w(k, v + 1)

Txn B

begin
body
commit
```
Logical (ghost) state of the database

- The current value for each key: $k \mapsto v$
Logical (ghost) state of the database

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

![Diagram showing transactions and time stamps](image)
• The current value 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

To update the logical state, we need to know:

- Will this transaction commit or abort?
- What will this transaction write?

Solution: Prophecy variables

\[
\begin{align*}
&\text{Time} \quad k \mapsto v \\
&\text{Txn B} \quad \begin{array}{c}
\text{begin} \\
\text{commit}
\end{array} \\
&\begin{array}{c}
\text{ts} = 15 \\
\text{r}(k, v) \\
\text{w}(k, v + 1)
\end{array}
\end{align*}
\]
Verification challenge: Transaction linearizes before its body runs

• To update the logical state, we need to know:
  • Will this transaction commit or abort?

Solution: Prophecy variables

Time

\[
\text{Txn A} \begin{array}{c}
\text{begin} \quad r(k,v) \\
\text{commit} \\
\text{ts} = 10 \\
\end{array}
\]

\[
\text{Txn B} \begin{array}{c}
\text{begin} \quad r(k,v) \\
\text{body} \quad w(k, v + 1) \\
\text{commit} \\
\end{array}
\]
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?

\[\begin{align*}
  \text{begin} & \quad \text{body} & \quad \text{commit} \\
  \text{Txn A} & \quad r(k, v) & \quad w(k, v + 1) \\
  \text{ts} = 10 & & \\
  \text{Txn B} & \quad r(k, v) & \quad w(k, v + 1) \\
  \text{ts} = 15 & & \\
\end{align*}\]
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

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

(all possible futures)

Versions of $k$ follow prophecy and update $k$ to $v + 1$.

Contradiction!

txn B actually commits and updates $k$ to $v + 1$.
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

Versions of $k$ follow prophecy and update $k$ to $v + 1$
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

$\begin{align*}
\text{Time} & \\
\text{txn B} & \\
\text{begin} & \\
\text{body} & \\
\text{commit} & \\
\end{align*}$
Peeking into the future with prophecy variable

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

$\times$ txn B commits and updates $k$ to $v + 2$

$\times$ txn B aborts

$\{ \text{all possible futures} \}$

contradiction!

txn B actually commits and updates $k$ to $v + 1$
Specifying general transactions with logical atomicity

```
begin
body
commit

r(k, v)

w(k, v + 1)

db.Run(body)

body(txn)

\{ (k, v) \to m \mapsto k \mapsto v_1 \mapsto P(m) \}

\{ (k, v) \to m' \mapsto k \mapsto v_2 \mapsto Q(m, m_0) \}

\langle m \mapsto (k, v) \mapsto m \mapsto k \mapsto v_1 \mapsto P(m) \rangle

\langle m' \mapsto (k, v) \mapsto m_0 \mapsto k \mapsto v_2 \mapsto Q(m, m_0) \rangle

take k_1 \mapsto v_1 \mapsto k_n \mapsto v_n

give k_1 \mapsto u_1 \mapsto k_n \mapsto u_n

\text{transaction-local view of database}

t_7! P(m) \text{ describes what's in the starting state}

Q(m, m_0) \text{ describes the changes}
```

transactions are allowed to run concurrently!
Specifying general transactions with logical atomicity

begin
body
commit

\( r(k, v) \)
\( w(k, v+1) \)

\[
\begin{align*}
\text{db.Run(body)} \\
\text{body(txn)} = \\
\text{\{ } \langle m. (k, v) \rangle \text{ \} P(m) } \\
\text{db.Run(body)} = \\
\text{body(txn)} = \\
\text{\{ } \langle m'. (k, v) \rangle \text{ \} Q(m, m_0) }
\end{align*}
\]

\( \langle m_1. (k_1, v_1) \rangle \)
\( \langle m_2. (k_2, v_2) \rangle \)

\( \text{transactions are allowed to run concurrently!} \)

\[
\begin{align*}
xfer(txn*, Txn, src, dst, amt: uint64) = \text{bool} \\
\text{sbal, _} := \text{txn.Read(src)} \\
\text{if sbal < amt} \{ \text{return false} \} \\
\text{txn.Write(src, sbal - amt)} \\
\text{dbal, _} := \text{txn.Read(dst)} \\
\text{txn.Write(dst, dbal + amt)} \\
\text{return true}
\end{align*}
\]
Specifying general transactions with logical atomicity

begin
body
commit
r((k, v))
w((k, v + 1))
user-defined
db.Run(body)

body(txn)

\[
\begin{cases}
(k, v) & \text{if } m = k^t \text{ and } v = P(m) \\
(k, v) & \text{if } m' = k^t \text{ and } v = Q(m, m_0)
\end{cases}
\]

\[\langle m . (k, v) \rangle\]
db.Run(body)

\[\langle m . (k, v) \rangle\]
take k_1 t v_1

\[\langle m . (k, v) \rangle\]
give k_1 t u_1

: transaction-local view of database
t_7!
P(m)
describes what's in the starting state
Q(m, m_0)
describes the changes

transactions are allowed to run concurrently!

\[\text{func xfer(txn \ast Txn, src, dst, amt \text{uint64}) \text{bool}}\]

\[
\begin{aligned}
sbal, _ & = \text{txn.Read(src)} \\
\text{if } sbal & < \text{amt} \\
& \text{return false} \\
\text{txn.Write(src, } sbal \text{ - amt)} \\
dbal, _ & = \text{txn.Read(dst)} \\
\text{txn.Write(dst, } dbal + \text{amt}) \\
\text{return true}
\end{aligned}
\]
Specifying general transactions with logical atomicity

```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
}
```

```
db.Run(body)
```

```
⟨m.⟨(k, v)⟩⟩
```

```
⟨m′.⟨(k, v)⟩⟩
```

```
take k1 t7!v1
```

```
give k1 t7!u1
```

transactions are allowed to run concurrently!

```
body(txn)
```

```
body(txn)
```

```
t7!P(m)
```

```
t7!Q(m, m0)
```

```
t7!P(m)
```

describes what's in the starting state

describes the changes
Specifying general transactions with logical atomicity

∀ body.

sequential spec for body(txn)

body(txn)

db.Run(body)

take k1 t7! v1

give k1 t7! u1

transactions are allowed to run concurrently!

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)

↓

concurrent spec for db.Run(body)

body(txn)

db.Run(body)
Specifying general transactions with logical atomicity

∀ body. 

\[
\begin{align*}
&\text{body(txn)} \\
&\{ \\
&\quad \{ \\
&\quad \quad \text{db.Run(body)} \\
&\quad \}
\}
\end{align*}
\]

body(txn)

db.Run(body)

\[\langle m.\ (k, v) \rangle \]

\[\langle \rangle \]

\[
\begin{align*}
&\text{begin (k, v)} \\
&\quad \text{r((k, v))} \\
&\quad \text{w((k, v)+1)} \\
&\end{align*}
\]
Specifying general transactions with logical atomicity

∀ body. \{ \begin{align*} (k, v) &\in m \\ k &\mapsto v \\ \text{body(txn)} &\end{align*} \} 

\text{db.Run(body)}

db.Run(body)

\text{take} \\
\begin{align*} k_1 &\mapsto v_1 \\ \cdots & \\ k_n &\mapsto v_n \\ \text{body(txn)} &\end{align*}
Specifying general transactions with logical atomicity

∀ body. \{ \begin{array}{c}
\{ \begin{array}{c}
\times \quad k \mapsto v \\
(k,v) \in m
\end{array} \\
\text{body(txn)}
\end{array} \} \\
\text{db.Run(body)}
\}

\longrightarrow

\{ \\
\{ \begin{array}{c}
take \mapsto \text{transaction-local view of database} \\
k_1 \mapsto v_1 \cdots k_n \mapsto v_n \\
\text{body(txn)}
\end{array} \\
\text{db.Run(body)}
\}
Specifying general transactions with logical atomicity

∀ body. \( \{ \star k \mapsto v \cdot \lambda P(m) \} \)

\( \text{body(txn)} \)

\( \{ \} \)

\( \downarrow \)

\( \langle \rangle \)

\( \text{db.Run(body)} \)

\( \langle \rangle \)

\text{take } P(m) \text{ describes what's in the starting state}

\( k_1 \mapsto v_1 \ast \ldots \ast k_n \mapsto v_n \)

\( \text{body(txn)} \)

\( \text{db.Run(body)} \)
Specifying general transactions with logical atomicity

∀ body. \{ \begin{align*}
&\{ k \mapsto v \, \mathbb{P}(m) \} \\
&\{ k \mapsto v \} \\
&\text{body(txn)}
\end{align*} \}

\downarrow

db.Run(body)

take
\begin{align*}
k_1 &\mapsto v_1 \ast \cdots \ast k_n \mapsto v_n \\
&\text{body(txn)}
\end{align*}
give
\begin{align*}
k_1 &\mapsto u_1 \ast \cdots \ast k_n \mapsto u_n \\
&\text{body(txn)}
\end{align*}

transactions are allowed to run concurrently!
Specifying general transactions with logical atomicity

∀ body. \{ 
\forall (k,v) \in m \exists k \mapsto v \ast P(m) \\
\forall (k,v) \in m' \exists k \mapsto v \ast Q(m,m') \\
\} 

body(txn)

\[
\begin{aligned}
&\left\langle (k_1, v_1) \mapsto \ldots \mapsto (k_n, v_n) \right| \\
&\left\langle (k_1, u_1) \mapsto \ldots \mapsto (k_n, u_n) \right|
\end{aligned}
\]

take \quad Q(m,m') \text{ describes the changes} 

give 

\[
\begin{aligned}
db.Run(body)
\end{aligned}
\]
Specifying general transactions with logical atomicity

∀ body. \left\{ \begin{array}{c}
\ast \quad k \mapsto v \cdot P(m) \\
(k,v) \in m
\end{array} \right. \\
body(txn)

\downarrow

\left\{ \begin{array}{c}
\ast \quad k \mapsto v \cdot Q(m, m') \\
(k,v) \in m'
\end{array} \right.

\langle m. \ast \quad k \mapsto v \cdot P(m) \rangle \\
(k,v) \in m

db.Run(body)

\langle \ast \quad k \mapsto v \cdot Q(m, m') \rangle \\
(k,v) \in m'

db.Run(body)

take \quad \begin{array}{c}
k_1 \mapsto v_1 \ast \cdots \ast k_n \mapsto v_n \\
k_1 \mapsto u_1 \ast \cdots \ast k_n \mapsto u_n
\end{array}

give

body(txn)

\text{take} \quad k_1 \mapsto v_1 \ast \cdots \ast k_n \mapsto v_n \\
\text{give} \quad k_1 \mapsto u_1 \ast \cdots \ast k_n \mapsto u_n

\text{transactions are allowed to run concurrently!}
Specifying general transactions with logical atomicity

∀ body. \[
\begin{align*}
&\star k \mapsto v \cdot P(m) \\
&\star k \mapsto v \cdot Q(m, m') \\
&\langle m. \star k \mapsto v \cdot P(m) \rangle \\
&\langle \star k \mapsto v \cdot Q(m, m') \rangle
\end{align*}
\]

\[
\begin{array}{c}
take \\
\text{take} \quad \text{give}
\end{array}
\]

\[
\begin{align*}
k_1 &\mapsto v_1 \ast \cdots \ast k_n \mapsto v_n \quad k_1 \mapsto u_1 \ast \cdots \ast k_n \mapsto u_n \\
\text{body(txn)} &\mapsto \text{body(txn)}
\end{align*}
\]

\[
\begin{array}{c}
take \\
give
\end{array}
\]

\[
\begin{align*}
k_1 &\mapsto v_1 \ast \cdots \ast k_n \mapsto v_n \quad k_1 \mapsto u_1 \ast \cdots \ast k_n \mapsto u_n \\
\text{db.Run(body)} &\mapsto \text{db.Run(body)}
\end{align*}
\]
Specifying general transactions with logical atomicity

∀ body. \left\{ \begin{array}{c}
\forall (k,v) \in m \quad k \mapsto v \cdot P(m) \\
\forall (k,v) \in m' \quad k \mapsto v \cdot Q(m', m')
\end{array} \right\}

body(txn)

\left\{ \begin{array}{c}
\forall (k,v) \in m' \quad k \mapsto v \cdot Q(m', m')
\end{array} \right\}

db.Run(body)

\left\{ \begin{array}{c}
\forall (k,v) \in m \quad k \mapsto v \cdot P(m)
\end{array} \right\}

take \quad k_1 \mapsto v_1 \cdot \cdots \cdot k_n \mapsto v_n \quad k_1 \mapsto u_1 \cdot \cdots \cdot k_n \mapsto u_n

give

body(txn)

transactions are allowed to run concurrently!
vMVCC: Abstractions and invariants

**Logical** $(r \mapsto v_4)$

<table>
<thead>
<tr>
<th>Physical</th>
<th>ts</th>
<th>del</th>
<th>val</th>
<th>tslast</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>false</td>
<td>$v_1$</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>false</td>
<td>$v_3$</td>
<td></td>
</tr>
</tbody>
</table>
vMVCC: Abstractions and invariants

\[
\text{LOGICAL } (r \mapsto \nu_4)
\]

\[
\text{LINEAR } \quad \text{AR}
\]

\[
\begin{array}{ccccc}
\text{ts} & \text{del} & \text{val} & \text{tslast} \\
1 & \text{false} & \nu_1 & & \\
3 & \text{false} & \nu_3 & & 4
\end{array}
\]

\[
\nu_1, \nu_2, \nu_3, \nu_4
\]
vMVCC: Abstractions and invariants

Logical: \( (k \mapsto v_4) \)

Speculative: 

Linear: 

Physical:

<table>
<thead>
<tr>
<th>ts</th>
<th>del</th>
<th>val</th>
<th>tslast</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
<td>( v_1 )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>( v_3 )</td>
<td>( 4 )</td>
</tr>
</tbody>
</table>
vMVCC: Abstractions and invariants

**Logical** \((r \mapsto v_4)\)

**Speculative**

**Linear**

**Physical**

<table>
<thead>
<tr>
<th>ts</th>
<th>del</th>
<th>val</th>
<th>ts</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
<td>(v_1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>(v_3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
vMVCC: Abstractions and invariants

Logical: \((k \mapsto v_4)\)

Speculative: \(\bot \bot v_1 v_1 v_3 v_3 v_4 v_4 v_4\)

Linear: \(\bot \bot v_1 v_1 v_3\)

AR

Physical:

<table>
<thead>
<tr>
<th>ts</th>
<th>del</th>
<th>val</th>
<th>tslast</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>false</td>
<td>(v_1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>(v_3)</td>
<td>4</td>
</tr>
</tbody>
</table>

PROPH
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

![Graph showing scalability analysis of different timestamp schemes](image)

**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

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

    return tid
}
```

RDTSC()

\[\downarrow\]

\[tid = 101011110\]
Generating strictly increasing timestamps with RDTSC

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

**Transaction site**

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

```
RDTSC()

↓

tid = 101011110
```
Generating strictly increasing timestamps with RDTSC

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

Transaction site

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

RDTSC()

\[
tid = 101100101
\]

(On site 5)
Generating strictly increasing timestamps with RDTSC

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

### Transaction site

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

```
RDTSC()

↓

tid = 101100101
(on site 5)
```
Generating strictly increasing timestamps with RDTSC

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

**Transaction site**

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

RDTSC()

\[
\text{tid} = 101100101
\]

(on site 5)
Generating strictly increasing timestamps with RDTSC

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

**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)

\[
\text{RDTSC}() \downarrow
\]

\[
tid = 101100101
\]

(on site 5)
Generating strictly increasing timestamps with RDTSC

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

**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

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

**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!

`RDTSC()`

\[
tid = 101100101
\]

(on site 5)
**vMVCC: Implementation and proof efforts**

**Implementation feature and optimization**

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

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>~800 (Go)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>~800 (Go)</td>
</tr>
<tr>
<td>Proof</td>
<td>~11K (Coq)</td>
</tr>
</tbody>
</table>
• Specifying and verifying transactions
• Low-level optimization: RDTSC-based timestamps
• Evaluation
• Conclusion
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 \( \implies \) 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  \(\rightarrow\) less memory and better performance
- Read-only transactions not linearizable  \(\rightarrow\) 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.
**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.
**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