No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System

Dominik Durner, Thomas Neumann
April 10, 2019

Technische Universität München
Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc.

Therefore, unexpected behavior and also unnecessary aborts are introduced.

Spurious aborts due to implementation artifacts that are hard to understand.
Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc.

Therefore, unexpected behavior and also unnecessary aborts are introduced.

Spurious aborts due to implementation artifacts that are hard to understand.

For example, 2PL cannot accept:

\[ t_1 \quad t_2 \]

\[ r(x) \quad w(x) \quad r(y) \quad c \quad r(x) \quad w(z) \quad c \]
Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc.

Therefore, unexpected behavior and also unnecessary aborts are introduced.

Spurious aborts due to implementation artifacts that are hard to understand.

For example, 2PL cannot accept:

\[
\begin{align*}
  t_1 & : \ r(x) \quad w(x) \quad r(y) \quad c \\
  t_2 & : \ r(x) \quad w(z) \quad c
\end{align*}
\]
Motivation

- Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc
- Therefore, unexpected behavior and also unnecessary aborts are introduced
- Spurious aborts due to implementation artifacts that are hard to understand
- For example, 2PL cannot accept:

```
\begin{align*}
\text{t}_1 & \quad r(x) \quad w(x) \quad r(y) \quad c \\
\text{t}_2 & \quad r(x) \quad w(z) \quad c
\end{align*}
```

- Only Serialization Graph Testing (SGT) accepts all valid schedules
- SGT seems to be too expensive and not scalable
Conflict graphs allow to accept all conflict serializable schedules.
Conflict graphs allow to accept all conflict serializable schedules.
Recoverability is independent of serializability.

\[ S_2 \subseteq P \cap RC \]
Conflict graphs allow to accept all **conflict serializable** schedules

Recoverability is independent of serializability

DBMS users expect to see committed changes

Note that $S_2 \subseteq COCSR \cap RC$
Motivation: Desired Schedules

- Conflict graphs allow to accept all conflict serializable schedules
- Recoverability is independent of serializability
- DBMS users expect to see committed changes

Note that $S2PL \not\subseteq COCSR \cap RC$
Our approach leverages the conflict graph and
1. accepts all useful $COCSR \cap RC$ schedules
2. meets users’ expectations
3. has low overhead for maintaining the graph
4. scales to many-core systems
Theorem: $s \in CSR \iff CG(s)$ is acyclic

- Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic
- Remove all outgoing edges of a node at its deletion
Theorem: $s \in CSR \iff CG(s)$ is acyclic

Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic

Remove all outgoing edges of a node at its deletion

Example: $s = r_0[x] w_0[x]$
Theorem: \( s \in CSR \iff CG(s) \) is acyclic

- Update \( CG(s) \) at operation arrival and allow if \( CG(s) \) is acyclic
- Remove all outgoing edges of a node at its deletion

Example: \( s = r_0[x] w_0[x] r_1[x] \)
Theorem: $s \in CSR \iff CG(s)$ is acyclic

Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic

Remove all outgoing edges of a node at its deletion

Example: $s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x]$
Theorem: $s \in CSR \iff CG(s)$ is acyclic

Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic

Remove all outgoing edges of a node at its deletion

Example: $s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x] w_2[y] c_2 c_0 c_1$

$\Rightarrow s \in CSR$
SGT Lacked Practical Relevance

- SGT has the best theoretical properties of accepting all valid schedules
- However, previous work fails to implement SGT efficiently in practice
SGT Lacked Practical Relevance

- SGT has the best theoretical properties of accepting all valid schedules.
- However, previous work fails to implement SGT efficiently in practice.

We developed the first practical and scalable algorithm that leverages the theoretical superior concept of graph-based serialization testing.
Pitfall: Deletion of a committed node $t_c$
Pitfall: Deletion of a committed node $t_c$

Example: $s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x] w_2[y] c_2$

$\Rightarrow s \not\in \text{CSR}$, but not detectable if $t_2$ was deleted.

Deletion of committed node is only allowed if all incoming edges are removed.
Pitfall: Deletion of a committed node $t_c$

Example: $s = r_0[x] \; w_0[x] \; r_1[x] \; r_2[x] \; w_2[x] \; w_2[y] \; c_2$
Prerequisites for Node Deletions

Pitfall: Deletion of a committed node $t_c$

Example: $s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x] w_2[y] c_2 r_0[y] c_0 c_1$
Prerequisites for Node Deletions

Pitfall: Deletion of a committed node $t_c$

Example: $s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x] w_2[y] c_2 r_0[y] c_0 c_1$

$\Rightarrow s \not\in CSR$, but not detectable if $t_2$ was deleted
Prerequisites for Node Deletions

Pitfall: Deletion of a committed node $t_c$

Example: $s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x] w_2[y] c_2 r_0[y] c_0 c_1$

$\Rightarrow s \notin CSR$, but not detectable if $t_2$ was deleted

Deletion of committed node is only allowed if all incoming edges are removed
Every transaction commit needs to wait until it is not dependent on in-flight results

Example: $s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x] w_2[y] c_2 c_0 c_1$
Every transaction commit needs to wait until it is not dependent on in-flight results.

Example: \( s = r_0[x] w_0[x] r_1[x] r_2[x] w_2[x] w_2[y] c_2 c_0 c_1 a_0 a_1 \)
Every transaction commit needs to **wait until it is not dependent on in-flight results**

Example: \( s = r_0[x] \, w_0[x] \, r_1[x] \, r_2[x] \, w_2[x] \, w_2[y] \, c_2 \, c_0 \, c_1 \, a_0 \, a_1 \)

No incoming write-read, write-write edge from an uncommitted node allowed
Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order
No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x] w_1[x] c_1$
Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: \( s = r_0[x] \lor_1 w_1[x] \prec d_1 \)

\[ \begin{array}{c}
\text{t}_1 \\
\downarrow \\
\text{t}_0
\end{array} \]
Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x] \swarrow w_1[x] \searrow d_1 r_2[y] c_2$

Diagram:

- $t_0 \rightarrow t_1 \rightarrow t_2$
Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: \( s = r_0[x].w_1[x] \rightleftharpoons d_1 r_2[y].c_2 w_0[y] \)
No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x] \, w_1[x] \, d_1 \, r_2[y] \, c_2 \, w_0[y] \, c_0$
Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: \( s = r_0[x] \rightarrow w_1[x] \rightarrow d \rightarrow r_2[y] \rightarrow c_2 \rightarrow w_0[y] \rightarrow c_0 \rightarrow c_1 \)

\( t_1 \)
Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x] w_1[x] \cap d_1 r_2[y] c_2 w_0[y] c_0 c_1$

$s_{orig} = r_0[x] w_1[x] c_1 r_2[y] c_2 w_0[y] c_0$

with $s' = t_2 t_0 t_1$, but $s_{orig} \notin COCSR$
Preserving the Commit Order

**No (uncommitted) incoming edge at commit time to preserve the commit order**

Example: $s = r_0[x] \ w_1[x] \ d_1 \ r_2[y] \ c_2 \ w_0[y] \ c_0 \ c_1$

$s_{orig} = r_0[x] \ w_1[x] \ c_1 \ r_2[y] \ c_2 \ w_0[y] \ c_0$

with $s' = t_2 \ t_0 \ t_1$, but $s_{orig} \notin COCSR$

All useful $COCSR \cap RC$ schedules accepted due to commit delays

Committed nodes are deleted directly including all outgoing edges
Scaling of our SGT-based Approach

- No incoming edges to commit simplifies cycle check
- **Conflict graph** is accessed *concurrently* by multiple threads
- No other transaction is allowed to modify a node during its final check
No incoming edges to commit simplifies cycle check

Conflict graph is accessed concurrently by multiple threads

No other transaction is allowed to modify a node during its final check

Transaction local shared/exclusive locks help to scale the graph
Example of our SGT-based Approach

$\begin{align*}
\text{sharedLocks: } & \{\} \\
\text{exclusiveLock: } & \text{false}
\end{align*}$
Example of our SGT-based Approach

- \( t_0 \)
  - \( r(x) \)
  - \( w(x) \)
  - \( c_{start} \)
  - \( c \)

- \( t_1 \)
  - \( r(x) \)
  - \( c_{start} \)
  - \( c \)

- wait for \( t_0 \)

Lock States:

- \( t_0 \)
  - \( \text{sharedLocks: } \{ t_1 \} \)
  - \( \text{exclusiveLock: } \text{false} \)

- \( t_1 \)
  - \( \text{sharedLocks: } \{ \} \)
  - \( \text{exclusiveLock: } \text{false} \)
Example of our SGT-based Approach

\[ r(x), w(x), c_{\text{start}}, c \]

wait for \( t_0 \)

\[ r(x), c_{\text{start}}, c \]

\[ \text{sharedLocks: \{\} } \]
\[ \text{exclusiveLock: false} \]

\[ \text{sharedLocks: \{\} } \]
\[ \text{exclusiveLock: true} \]
Example of our SGT-based Approach

```
sharedLocks: {}
exclusiveLock: false
sharedLocks: {}
exclusiveLock: true
sharedLocks: {}
exclusiveLock: false
sharedLocks: {}
exclusiveLock: false
sharedLocks: {}
exclusiveLock: false
sharedLocks: {}
exclusiveLock: true
```

![Diagram showing the example of our SGT-based Approach](image-url)
Example of our SGT-based Approach

\[ r(x) \quad w(x) \quad c_{\text{start}} \quad c \]

\[ t_0 \]

\[ \text{wait for } t_0 \]

\[ t_1 \]

\[
\begin{align*}
\text{sharedLocks: } & \{ \} \\
\text{exclusiveLock: } & \text{true}
\end{align*}
\]
Experimental Evaluation

Setup:
- 4-socket Intel Xeon server (60 cores) with 1TB DRAM
- Every transaction is scheduled on one worker thread
- Aborts require undos and restarts of the aborted transactions

Algorithms:
- Our SGT-based approach
- TicToc
- 2PL with row based atomic read-write locks and deadlock prevention
SmallBank Medium Contention (1000 Customers)
SmallBank Medium Contention (1000 Customers)

![Graphs showing the relationship between OLTP threads and TX/s, Abort Rate, 2PL, TicToc, and SGT for SmallBank Medium Contention.](image-url)

The graphs display the performance metrics under different contention scenarios.

- **TX/s**: The throughput measured in transactions per second (TX/s) increases as the number of OLTP threads increases.
- **Abort Rate**: The abort rate also increases with the number of OLTP threads, indicating higher contention levels.

The graphs illustrate how different concurrency control mechanisms (2PL, TicToc, SGT) influence performance metrics under varying loads.

---

The data suggests that SGT offers better performance in terms of TX/s and Abort Rate compared to 2PL and TicToc, especially at higher contention levels.
Our SGT has competitive throughput while reducing aborts significantly!
Our SGT has **competitive throughput** while reducing aborts significantly!
Summary: Our graph-based concurrency control algorithm accepts all useful \( COCSR \cap RC \) schedules.
Summary: Our graph-based concurrency control algorithm accepts all useful \( COCSR \cap RC \) schedules and reduces aborted schedules and meets users' expectations. It has low protocol overhead and scales to many-core systems.
Summary: Our graph-based concurrency control algorithm accepts all useful COCSR ∩ RC schedules.

- Reduces aborted schedules and meets users' expectations.
- Has low protocol overhead and scales to many-core systems.

| all schedules | RC |
|---------------|----|
| CSR           |    |
| OCSR          |    |
| COCSR         |    |

![Graph showingAbort Rate vs OLTP threads](image1)

![Graph showingTX/s vs OLTP threads](image2)