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

Dominik Durner, Thomas Neumann
April 10, 2019

Technische Universität München
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
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:

\[
\begin{array}{l}
  t_1 \quad r(x) \quad w(x) \quad r(y) \quad c \\
  t_2 \quad r(x) \quad w(z) \quad c
\end{array}
\]
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:

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

Only Serialization Graph Testing (SGT) accepts all valid schedules.

SGT seems to be too expensive and not scalable.
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{3} & \quad r(x) \quad w(x) \quad r(y) \quad c \\
\text{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.
Motivation: Desired Schedules

- Conflict graphs allow to accept all conflict serializable schedules

![Diagram](attachment:image.png)

- CSR
- all schedules

Note that $S_2 \subseteq \text{CSR} \cap \text{RC}$
Motivation: Desired Schedules

- Conflict graphs allow to accept all conflict serializable schedules
- Recoverability is independent of serializability

Note that $S_2 \subseteq \text{CSR} \cap \text{RC}$
Motivation: Desired Schedules

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

![Diagram](image-url)
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 $S_{2PL} \subset COCSR \cap RC$
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 $t_c$
Prerequisites for Node Deletions

**Pitfall: Deletion of a committed node** $t_c$

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

Pitfall: Deletion of a committed node $t_c$

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

**Pitfall**: Deletion of a committed node $t_c$

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

![Diagram of nodes $t_0$ and $t_1$ with an edge from $t_0$ to $t_1$]
Prerequisites for Node Deletions

Pitfall: Deletion of a committed node $t_c$

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

$\Rightarrow s \notin CSR$, 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
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: $s = r_0[x] \ w_1[x] \ c_1$

\begin{tikzpicture}
  \node[shape=circle,draw=black] (1) at (0,0) {$t_0$};
  \node[shape=circle,draw=black] (2) at (1,1) {$t_1$};
  \draw[->,thick] (1) -- (2);
\end{tikzpicture}
Preserving the Commit Order

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

Example: $s = r_0[x] \, w_1[x] \rightleftharpoons d_1$

Diagram:

```
+---+---+
| t1 | t0 |
+---+---+
```
Preserving the Commit Order

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

Example: \( s = r_0[x] \ x w_1[x] \ x d_1 \ x r_2[y] \ x c_2 \)

\[
\begin{array}{c}
\text{t}_0 \\
\text{t}_1 \\
\text{t}_2
\end{array}
\]
Preserving the Commit Order

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

Example: \( s = r_0[x] \quad w_1[x] \quad d_1 \quad r_2[y] \quad c_2 \quad w_0[y] \)
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: \[ s = r_0[x] w_1[x] \otimes d_1 r_2[y] c_2 w_0[y] c_0 c_1 \]

\[ t_1 \]
Preserving the Commit Order

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

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

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

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

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

Example: \( s = r_0[x] \, w_1[x] \bigotimes d_1 \, r_2[y] \, c_2 \, w_0[y] \, c_0 \, c_1 \)

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

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

All useful \( \text{COCSR} \cap \text{RC} \) 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
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

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

\[
\begin{align*}
  t_0 & \quad r(x) \quad w(x) \quad c_{start} \quad c \\
  t_1 & \quad \underline{\text{wait for } t_0} \quad r(x) \quad c_{start} \quad c \\
\end{align*}
\]
Example of our SGT-based Approach

```
t0 r(x) w(x) cstart c

t1 wait for t0

r(x) cstart c

sharedLocks: {}
exclusiveLock: false
```

```
t0

sharedLocks: {t1}
exclusiveLock: false

sharedLocks: {}
exclusiveLock: false

sharedLocks: {}
exclusiveLock: true
```
Example of our SGT-based Approach

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

\[
\begin{align*}
\text{sharedLocks: } & \{\} & \text{exclusiveLock: false} \\
\text{sharedLocks: } & \{\} & \text{exclusiveLock: true} \\
\text{sharedLocks: } & \{\} & \text{exclusiveLock: false} \\
\text{sharedLocks: } & \{t_0\} & \text{exclusiveLock: false}
\end{align*}
\]
Example of our SGT-based Approach

$t_0 \quad r(x) \quad w(x) \quad c_{\text{start}} \quad c$

$t_1 \quad \overline{\text{wait for } t_0} \quad \overline{r(x)} \quad c_{\text{start}} \quad c$

$t_1$

sharedLocks: {}
exclusiveLock: true
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)

![Graph showing TX/s vs. OLTP threads](image1)

![Graph showing Abort Rate vs. OLTP threads](image2)
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.

The diagram shows a table comparing concurrency control algorithms: SGT, TicToc, 2PL, with respective abort rates and TX/s performance metrics. The table indicates that our algorithm reduces aborted schedules and meets users' expectations, has low protocol overhead, and scales to many-core systems.
Summary: Our graph-based concurrency control algorithm accepts all useful $\text{COCSR} \cap \text{RC}$ schedules and reduces aborted schedules and meets users' expectations.
Summary: Our graph-based concurrency control algorithm

accepts all useful \( COCSR \cap RC \) schedules

reduces aborted schedules and meets users' expectations

has low protocol overhead and scales to many-core systems