Serializability of Transactions in Software Transactional Memory

Utku Aydonat

uaydonat@eecg.toronto.edu

Department of Electrical and Computer Engineering
University of Toronto

Workshop on Transactional Computing (TRANSACT) 2008
Outline

• Motivation
  – STM systems perform poorly for some applications.

• 2-Phase Locking
  – Why can it be an overly conservative technique?

• Conflict Serializability
  – What is the advantage of conflict-serializability?

• Ensuring Conflict-Serializability
  – How can we implement conflict-serializability efficiently?

• Multi-Versioning

• Experimental Evaluation

• Conclusions
Motivation

- Short transactions with low data sharing
  - Successful STM implementations (blocking STMs, time-based STMs, etc.)
  - Aborts are insignificant for performance.
    - For RBT, the abort rate is 0.23% at 32 threads, avg. tx. time is 11 µsec.
- Throughput:

![Graph showing throughput vs. threads]
Motivation

- Long transactions with high data sharing
  - Popular STM implementations suffer in performance.
  - Abort rates impact the performance.
    - For LL, the abort rate is 79% at 32 threads, avg. tx. time is 197 µsec.
    - Throughput:
Consistency in STMs

- Are STMs too eager to abort transactions?
- **Linearizability**: Correctness criteria for STMs.
  - Transactions appear to execute atomically at some point between their start time and commit time.
Consistency in STMs

- Are STMs too eager to abort transactions?
- **Linearizability**: Correctness criteria for STMs.
  - Transactions appear to execute atomically at some point between their start time and commit time.

Equivalent sequential execution!
Consistency in STMs

- Are STMs too eager to abort transactions?
- Linearizability: Correctness criteria for STMs.
  - Transactions appear to execute atomically at some point between their start time and commit time.

Equivalent sequential execution!
Consistency in STMs

- Are STMs too eager to abort transactions?
- Linearizability: Correctness criteria for STMs.
  - Transactions appear to execute atomically at some point between their start time and commit time.
Consistency in STMs

- Are STMs too eager to abort transactions?
- Linearizability: Correctness criteria for STMs.
  - Transactions appear to execute atomically at some point between their start time and commit time.

Equivalent sequential execution!
Two Phase Locking

- STMs emulate two phase locking (2PL) to ensure linearizability.
  - Conflicting accesses are not allowed from the access time till the transaction commits.
  - More conservative than linearizability.
  - Easy to implement: fast transactions but frequent aborts.
Two Phase Locking

- STMs emulate two phase locking (2PL) to ensure linearizability.
  - Conflicting accesses are not allowed from the access time till the transaction commits.
  - More conservative than linearizability.
  - Easy to implement: fast transactions but frequent aborts.

Delay or Abort!
Two Phase Locking

- STMs emulate two phase locking (2PL) to ensure linearizability.
  - Conflicting accesses are not allowed from the access time till the transaction commits.
  - More conservative than linearizability.
  - Easy to implement: fast transactions but frequent aborts.

```
ObjA
ObjB
ObjC
ObjD
```

```
TX1 R
W
R
W
TX2
```

equivalent seq. exec.  TX1  TX2
Conflict-Serializability (CS)

- Allows conflicting accesses as long as we can find a valid ordering for transactions; an ordering that matches the order of conflicting accesses.
- More relax than 2PL & linearizability.

![Diagram showing conflicting accesses and equivalent sequence execution]

```
ObjA

| ObjB | TX1       |
|      | W         |
|      | R         |

| ObjC | TX2       |
|      | W         |
|      |            |

| ObjD | TX3       |
|      | W         |
|      | R         |

| ObjE | TX3       |
|      | W         |
```
Conflict Serializability (CS)

• Conclusion: We can reduce abort rates in STMs by implementing CS rather than 2PL.

• We can build a precedence/ordering/serializability graph.
  – Too slow

• We use Serializability Order Numbers (SONs)
Serializability Order Numbers (SONs)

- Each transaction gets a SON number when it commits.
- The order of SON numbers matches the order of conflicting accesses for all transactions.
  - Rule1: If TX1 accesses an object committed by TX2
    \[\text{SON}(\text{TX1}) > \text{SON}(\text{TX2})\]
  - Rule2: If TX2 commits an object already read by TX1
    \[\text{SON}(\text{TX2}) > \text{SON}(\text{TX1})\]
- Keep an SON range for each active transaction \([lb, ub]\)
  - Assign SON number based on \(lb\) \& \(ub\), such that \(lb < \text{SON} < ub\)
  - If \(up < lb\) at any point, transaction cannot be serialized.
Serializability Order Numbers (SONs)

\[ [0:\infty] \quad \text{Start TX1;} \quad [0:\infty] \quad \text{Start TX2;} \quad [0:\infty] \quad \text{Start TX3;} \]

Equivalent sequential execution!
Serializability Order Numbers (SONs)

Equivalent sequential execution!
Serializability Order Numbers (SONs)

Equivalent sequential execution!
Serializability Order Numbers (SONs)

[0:∞] Start TX1;
[0:∞] Read(A);

[0:∞] Start TX2;
[0:∞] Read(B);

[0:∞] Start TX3;

[0:∞] Write(B);

SON=3 Commit;

Equivalent sequential execution!
Serializability Order Numbers (SONs)

\[
\begin{align*}
&[0:\infty] \text{ Start TX1; } \\
&[0:3] \text{ Read(A); } \\
&\text{.} \\
&\text{.} \\
&\text{.} \\
&\text{.} \\
&[0:3] \text{ SON=3 Commit; }
\end{align*}
\]

\[
\begin{align*}
&[0:\infty] \text{ Start TX2; } \\
&[0:\infty] \text{ Read(B); } \\
&\text{.} \\
&\text{.} \\
&[0:\infty] \text{ Write(B); } \\
&[0:\infty] \text{ Start TX3; }
\end{align*}
\]

Equivalent sequential execution!
Serializability Order Numbers (SONs)

[0:∞] Start TX1;
[0:∞] Read(A);
.  
.  
.  
[0:3] Write(A);
Commit;

[0:∞] Start TX2;
[0:∞] Read(B);
.  
.  
[0:3] Write(B);
SON=3 Commit;

[0:∞] Start TX3;

Equivalent sequential execution!
Serializability Order Numbers (SONs)

Equivalent sequential execution!
Serializability Order Numbers (SONs)

\[
\begin{align*}
0: \infty & \quad \text{Start TX1;} \\
0: \infty & \quad \text{Read(A);} \\
0: \infty & \quad . \\
0: \infty & \quad . \\
0: \infty & \quad . \\
0: 2 & \quad \cdots \quad \text{TX1} \\
\end{align*}
\]

\[
\begin{align*}
0: \infty & \quad \text{Start TX2;} \\
0: \infty & \quad \text{Read(B);} \\
0: 3 & \quad \text{Write(A);} \\
0: 3 & \quad \text{Write(B);} \\
0: \infty & \quad \text{Commit;} \\
\end{align*}
\]

\[
\begin{align*}
0: \infty & \quad \text{Start TX3;} \\
0: \infty & \quad . \\
0: \infty & \quad . \\
0: 3 & \quad \text{Commit;} \\
\end{align*}
\]

Equivalent sequential execution!
Serializability Order Numbers (SONs)

Equivalent sequential execution!
**Serializability Order Numbers (SONs)**

```
[0:∞] Start TX1;
[0:∞] Read(A);
. 
. 
. 
[3:2] Read(B);

[0:∞] Start TX2;
[0:∞] Read(B);
. 
. 
[0:3] Write(A);

[0:∞] Start TX3;
[0:∞] Write(B);
. 
[0:3] Commit;

[0:3] SON=3 Commit;

Equivalent sequential execution!
```
Serializability Order Numbers (SONs)

[0:∞] Start TX1;
[0:∞] Read(A);
   .
   .
[0:2]  Read(B);
  Abort;

[0:∞] Start TX2;
[0:∞] Read(B);
   .
   .
[0:3]  Write(A);

[0:∞] Start TX3;
[0:∞] Write(B);
[0:∞] Write(B);

SON=2  Commit;

Equivalent sequential execution!
Multi-Versioning

• If reading the latest version of data cannot be serialized, perhaps reading an older version can be.
  – Keep older versions for each object.
  – Update SON ranges based on the version read.

• Avoids having to reject operations that arrive too late to be serializable.
Toronto STM (TSTM)

- Implements CS with multi-versioning.
- C++, Object-based, indirection with wrappers, obstruction-free, visible readers.
- High overheads
  - Accessing SON ranges.
  - Iterating reader transactions.
  - Pays off when aborts are important.
- Adaptive TSTM
  - Switches consistency model based on abort rates: CS vs. 2PL
Experimental Evaluation

• **Platform:** Sun-Fire 15K, 72 UltraSPARCIII 1.2 GHz proc.
  – Use only 32 processors.
  – Single thread per processor, no nested transactions.

• **Benchmarks:**
  – LinkedList (LL), RB Tree (RBT), Graph (GR).
  – Value ranges: 16K(LL), 64K(RBT), 4K(GR).

• **Evaluated algorithms:**
  – Fine-grain locking (FG), Coarse-grain locking (CG).
  – TSTM, TSTM-adaptive.
  – RSTM, TL2.
Linked List

![throughput vs. threads graph]

- CG
- FG
- TL2
- RSTM
- TSTM
- TSTM-adaptive
Linked List
RBT

throughput vs. threads

- CG
- FG
- TL2
- RSTM
- TSTM
- TSTM-adaptive
Related Work

• Early Release & Open Nesting & Collection Classes

• Snapshot Isolation

• Z-linearizability

• Databases
Conclusions

- 2PL is too restrictive for applications with long transactions and high data sharing.
- CS gives better performance despite its high overheads
  - 2.5 times better throughput for LL.
  - 4 times better throughput for GR.
- Poor performance for applications with short transactions
- Dynamically adapting the model yields the best performance
- We need to look into more models
  - Is adapting between several different models viable?