Approximately Opaque Multi-version Permissive Transactional Memory

Basem Assiri
Costas Busch
Louisiana State University
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- Transactional Memory and Consistency
- Approximately Opaque TM for Read-only Transactions
- Results
- Garbage Collector
- Conclusion
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- Transactional Memory and Consistency
- Approximately Opaque TM for Read-only Transactions
- Results
- Garbage Collector
- Conclusion
The Challenges of Distributed Computing and Performance

Multi-core processors improve the performance through concurrent computing.

However, synchronizing memory accesses makes writing concurrent applications much harder than sequential ones.
Transactional Memory

Transactional memory is an important way to cope with such challenge!

Transactional memory allows concurrent accesses by using the concept of transaction

Transaction is a piece of code or a finite sequence of instructions that access local and shared memory. Transactions are read-only or update. Transactions commit or abort
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- Transactional Memory and Consistency
- Approximately Opaque TM for Read-only transaction
- Results
- Garbage Collector
- Conclusion
Objectives and Motivation

• **Improve TM performance** (increase throughput).
  The precision level in consistency.

• **In large scale network systems**
• **long read-only transactions**
• **In real life, some types of systems do not require precise computations.**

Non-sensitive data and frequent changes.
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- **Transactional Memory and Consistency**
- Approximately Opaque TM for Read-only Transactions
- Results
- Garbage Collector
- Conclusion
We can easily prove the correctness of sequential execution.

So, we prove the correctness of any concurrent execution if we can match it with a sequential one.
TM Consistency

- Event: “x.write()” = invocation + response
- History (H): a sequence of events.
TM Consistency (Order)

Total Order: \( T_1 \rightarrow T_2 \)

Partial Order: \( T_1 \rightarrow T_2 \) or \( T_2 \rightarrow T_1 \)?

Memory:
- \( x=0 \)
- \( x=5 \)

- \( T_1 \): \text{x.write(5)}
- \( T_2 \): \text{x.read(0)} \text{x.read(5)}
Complete History (H\'): all transactions are either committed or aborted.

Sequential History (S): is complete, and is a total order.

with respect to committed transactions
and reads from aborted transactions

Memory
X=5

T1
x.write(5)

T2
x.read(5)

Total Order: T1 \(\rightarrow\) T2
TM Consistency (Legality)

Memory

\( X=0 \quad X=5 \)

\( x.\text{write}(5) \)

legal

\( x.\text{read}(5) \)

\( X=5 \)

Abort

Memory

\( x.\text{write}(5) \)

illegal

\( x.\text{read}(0) \)

\( X=0 \quad X=5 \)
TM Consistency (K-legality)

Memory

$X = 0$  $X = 5$

T1

x.write(5)

T2

legal

x.read(5)

$X = 0$  $X = 5$

T1

x.write(5)

T2

K=2

K-legal

x.read(0)
TM Consistency (K-legality)

Legal S = T₀ → T₁ → T₂

K-Legal S = T₀ → T₁ → T₂

Memory

X = 0  X = 5

x.write(5)

K = 2

x.read(0)

Legal

K-legal

T₁

x.write(5)

T₂

x.read(5)
• **Opacity:**
  Concurrent H -> H’ -> *legal* S where S respects the *real time order* of transactions in H.

• **K-opacity:**
  Concurrent H -> H’ -> *K-legal* S where S respects the *real time order* of transactions in H.
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- Transactional Memory and Consistency
- Approximately Opaque TM for Read-only Transactions
- Results
- Garbage Collector
- Conclusion
Single-version

- Two concurrent read-only and update transactions sharing two objects \((x\) and \(y\))
Single-version

- Two concurrent read-only and update transactions sharing two objects (x and y)

Memory

| X=4 | Y=4 |

T1: x.read(0) → y.read(4)

T2: y.write(4) → x.write(4)

Abort
Multi-version

• Two concurrent read-only and update transactions sharing two objects (x and y)

Memory

T1: x.read(0)  y.read(0)

T2: y.write(4)  x.write(4)

S = T1 → T2
Approximately Opaque TM for Read-only Transactions

We relax the opacity to K-opacity

(i) It reduces the number of aborts since there is smaller chance for read-only transactions to abort update transactions.

(ii) It reduces space requirements, since a new version is saved once every K object updates, which reduces the total number of saved object versions by a factor of K.
Approximately Opaque TM for Read-only Transactions
Approximately Opaque TM for Read-only Transactions

T0:
- \text{write}(0)

\text{Objects (x)}
- \text{commitsCounter} = 0
- ts = 0
- x = 0

T1:
- \text{read}(0)
- \text{......}
Approximately Opaque TM for Read-only Transactions

T0: x.write(0)

Objects (x)

T1: x.read(0) ..........

T2: x.write(2)

commitsCounter=0

X=0

ts=0
Approximately Opaque TM for Read-only Transactions

Objects ($x$)

$commitsCounter > 1$
$ts = 0$
$X = 0$

$commitsCounter < K$

$T_0$
$x.write(0)$

$T_1$
$x.read(o)$

$T_2$
$x.write(2)$

$\vdots$

$T_{3+}$
$x.read(o)$
Approximately Opaque TM for Read-only Transactions

Objects (x)

commitsCounter = K

T0 x.write(0)

T1 x.read(0) ........

T2 x.write(2)

T3+ x.read(0)

Tv x.write(v)

commitsCounter = K
Approximately Opaque TM for Read/write Object

Objects (x)

commitsCounter = 0

0
v

X = 0
X = v

S = T₀ → T₁ → T₂ → ... → T₃⁺ → Tᵥ

K-opaque
Another Issue

Objects (x)

commitsCounter > 1

commitsCounter < K

T0

x.write(0)

T1

......

T2

x.write(2)

X=0

ts=0
Another Issue

Objects (x)

commitsCounter > 1

T0 x.write(0)

T1 x.read(0)

T2 x.write(2)

T3 x.read(0)

Tn x.read(0)

commitsCounter < K

ts=0

X=0

X=0
Solution

Objects (x)

commitsCounter > 1

VersionsList

ts = 0

X = 0

lastCommit

ts = 2

X = 2

commitsCounter < K

T0: x.write(0)

T1:  

T2: x.write(2)

X = 0

X = 2

X = 0

……
Solution

commitsCounter < K

T0
x.write(0)

Objects (x)

commitsCounter > 1

VersionsList

<table>
<thead>
<tr>
<th>ts=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=0</td>
</tr>
</tbody>
</table>

lastCommit

<table>
<thead>
<tr>
<th>ts=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=2</td>
</tr>
</tbody>
</table>

T1
x.read(0)

T2
x.write(2)

T3
x.read(2)

Tn
x.read(2)
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- Transactional Memory and Consistency
- Approximately Opaque TM for Read-only Transactions
- Results
- Garbage Collector
- Conclusion
Results

Micro-benchmarks : (tinySTM-1.0.5)
(a) Linked-list
(b) Read-black Tree
(c) Bank

- We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 5500 milliseconds using 10 threads.
Results
Results
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- Transactional Memory and Consistency
- Approximately Opaque TM for Read-only Transactions
- Results
- Garbage Collector
- Conclusion
Garbage Collector

minT

LiveT

9
11
12
15
16

T9
T11
T12
T15
T16

X.vl

4 2380
7 9000
10 90
15 68
23 510

Y.vl

5 6810
7 80
8 28
14 9000
12 6810
Approximately Opaque TM for Queue Object

\( \text{min}T \)
Outline

- Challenges of Distributed Computing and Performance
- Objectives and Motivation
- Transactional Memory and Consistency
- Approximately Opaque TM for Read-only Transactions
- Results
- Garbage Collector
- Conclusion
- Improve the performance and the space complexity by relaxing the consistency

- We extend this work and apply the K-opacity concept on update transactions and different data structures.

- For future direction we can use different kind of relaxation such time-based or value-based relaxation.
Conclusion

Snapshot Isolation allows older reads, but it still ensures that each transaction sees a consistent snapshot.

However, Snapshot Isolation does not preserve serializability.

While in K-opacity transactions may see a K-consistent snapshot but it preserves a relaxed (Approximate) serializability.
Thank you!

Questions??