Software Transactional Memory for Dynamic-sized Data Structures

Maurice Herlihy, Victor Luchango, Mark Moir, William N. Scherer III

Presented by: Irina Botan
Outline

• Introduction
• Dynamic Software Transactional Memory (DSTM)
• DSTM Implementation
  – Transactions and Transactional Objects
  – Contention Management
• DSTM performance
• DSTM2
• DSTM2 performance
Parallel Computing

Moore’s law => chip multiprocessing (CMP) => Increased Parallelism
Complex problems can be divided into smaller ones which can be then executed in parallel
Concurrent Access to Shared Data

T1

```c
int x[N];
i := N-1;
...
if (i < N){
    x[i] := i;
}
```

T2

```c
i := i+1;
...
```
Locking

- Coarse-grained or fine-grained locking
- Locking conventions
- Vulnerability to thread failures and delays
- Poor support for code composition and reuse

=> too difficult to develop, debug, understand and maintain programs

Coarse-grained locking

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock()</td>
<td>lock()</td>
</tr>
<tr>
<td>i := N-1;</td>
<td>i := i + 1;</td>
</tr>
<tr>
<td>...</td>
<td>unlock()</td>
</tr>
<tr>
<td>if ( i&lt;N )</td>
<td>unlock()</td>
</tr>
<tr>
<td>x[i]:=i;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Software Transactional Memory (STM)

• Low-level API for synchronizing access to shared data without using locks
  – Alleviates the difficulty of programming
  – Maintains performance

• Transactional Model
  – Transaction = atomic sequence of steps executed by a single thread (process); protects access to shared (transactional) objects

• Only for static data structures
  – Transactional objects and transactions defined apriori
From STM to D(ynamic)STM

• Transactions and transactional objects can be created dynamically
  
  if (object1.value == 1)
  
  object2.value = 2;

  else

  object3.value = 2;

• Well suited for dynamic-sized data structures, like linked lists and trees
Outline

• Introduction

• Dynamic Software Transactional Memory (DSTM)

• DSTM Implementation
  – Transactions and Transactional Objects
  – Contention Management

• DSTM performance

• DSTM2

• DSTM2 performance
Transactional Object

- Container for a shared object

- Creation

  ```java
  List newNode = new List(v);
  TMObject newTMNode = new TMObject(newNode);
  ```

- Access

  ```java
  List current = (List)newTMNode.open(WRITE);
  current.value = 1;
  ```
Transaction

• Short-lived single-threaded computation that either commits (the changes take effect) or aborts (the changes are discarded)

• Linearizability = transactions appear as if they were executed one-at-a-time
public boolean insert (int v){

    List newList = new List(v);
    TMObject newNode = new TMObject(newList);
    TMThread thread = (TMThread)thread.currentThread();
    while(true){
        thread.beginTransaction();
        boolean result = true;
        try{
            List prevList = (List)this.first.open(WRITE);
            List currList = (List)prevList.next.open(WRITE);
            while (currList.value < v){…}
        } catch (Denied d){}
        if  (thread.commitTransaction())
            return result;
    }
    return result;
}
Synchronization Conflict

• Two transactions attempting to access the same object and at least one of them wants to write it
Check Synchronization Conflicts

• If a conflict occurs, open() throws a Denied exception
  – The transaction knows it will not commit successfully and will retry execution

```java
public boolean insert (int v){
    ...
    while(true){
        thread.beginTransaction();
        try{
            List prevList = (List)this.first.open(READ);
            ...
        } catch (Denied d){}
        if (thread.commitTransaction())
            return result;
    }
}
```
Conflicting Reduction = Early Release

- `Release` an object opened in READ mode before commit
- Useful for shared pointer-based data structures (e.g., lists, trees)
- Programmer’s job to ensure correctness (linearizability)

```
A
open(i, READ)
... ...
release(i)
open(i, WRITE)
...
B
open(i, READ)
...
open(j, READ)
... ...
commit
```
Progress Guarantee

• Wait-freedom
  – Every thread makes progress

• Lock-freedom
  – At least one thread makes progress

• Obstruction-freedom
  – Any thread that runs by itself for long enough makes progress
Obstruction-Freedom

• A transaction can abort any other transaction

• + Simpler and more efficient (in absence synchronization conflicts) than lock-freedom

• - Livelocks possible
Livelock

- Two competing processes constantly change state with respect to one another, none making progress

- E.g., two people meeting in a narrow corridor, each trying to be polite by moving aside to let the other pass*

Outline

• Introduction
• Dynamic Software Transactional Memory (DSTM)
  • DSTM Implementation
    – Transactions and Transactional Objects
    – Contention Management
  • DSTM performance
• DSTM2
  • DSTM2 performance
Transactional Object Implementation

“transaction” points to the transaction that most recently opened the object in WRITE mode

<table>
<thead>
<tr>
<th>Transaction State</th>
<th>Old Object</th>
<th>New Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Committed</td>
<td>Meaningless</td>
<td>Current object</td>
</tr>
<tr>
<td>Aborted</td>
<td>Current Object</td>
<td>Meaningless</td>
</tr>
<tr>
<td>Active</td>
<td>Current Object</td>
<td>Tentative new current object</td>
</tr>
</tbody>
</table>
Transactional Object Access

- Avoid generating inconsistencies
- How to atomically access all three fields?

![Transactional Object Diagram]

Transaction
new object
old object

Transactional Object

State
Data
Data
Atomically Access the Transactional Object’s Fields

• Introduce another level of indirection
  – CAS (Compare and Swap) to swing the Start object from one locator object to the other
Open Transactional Object in WRITE Mode (Previous Transaction Committed)
Open Transactional Object in WRITE Mode (Previous Transaction Aborted)
Open Transactional Object in WRITE Mode (Previous Transaction Active)
Open Transactional Object in READ Mode

1. Identify the last committed version of the transactional object (exactly as for WRITE)
2. Add the pair (O,V) to a thread-local read-only table
Transaction Validation

- Ensure that the user never sees an inconsistent state
- After open() determined the version
  1. For each pair \((O, V)\) verify that \(V\) is still the most recently committed version of the object \(O\)
  2. Check that status field of transaction still ACTIVE

<table>
<thead>
<tr>
<th>Object</th>
<th>Version</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>V1</td>
<td>2</td>
</tr>
<tr>
<td>O2</td>
<td>V2</td>
<td>1</td>
</tr>
<tr>
<td>O3</td>
<td>V</td>
<td>1</td>
</tr>
</tbody>
</table>
Transaction Commit

1. Validate entries in the read-only table
2. Change the status field of the transaction from ACTIVE to COMMITTED

<table>
<thead>
<tr>
<th>Object</th>
<th>Version</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1</td>
<td>V1</td>
<td>2</td>
</tr>
<tr>
<td>O2</td>
<td>V2</td>
<td>1</td>
</tr>
<tr>
<td>O3</td>
<td>V</td>
<td>1</td>
</tr>
</tbody>
</table>
Contestation Management

• Ensures progress

• Each thread has a Contention Manager
  – Consults it to decide whether to force another conflicting thread to abort

• Correctness requirement for contention managers
  – Any active transaction can eventually abort another transaction (“obstruction-freedom”)
  – Should avoid livelock
Contestation Manager Policies Examples

• Aggressive
  – Always and immediately grants permission to abort any conflicting transaction

• Polite
  – In case of a conflict, sleep for a time interval $t$
    * Idea: wait for the other transaction to finish
  – Retry and increase waiting time with each attempt
  – After a fixed number of tries, immediately abort the other transaction
Costs

- $W$ – number of objects opened in WRITE mode
- $R$ – number of objects opened in READ mode

- *In the absence of conflicts*
  - $(W+1)$ CAS operations (for each open() call and one commit)
- *Synchronization conflicts*
  - More CAS operations to abort other transactions
- *Costs of copying objects* (uses simple load and store operations)
- *Validating a transaction*
  - Requires $O(R)$ work

- Total overhead due to DSTM implementation
  - $O((R+W)R)$ + clone each object opened for writing once
Outline

• Introduction
• Dynamic Software Transactional Memory (DSTM)
• DSTM Implementation
  – Transactions and Transactional Objects
  – Contention Management
• DSTM performance
• DSTM2
• DSTM2 performance
Experimental Setup

- Integer Set and Red-black tree

- Measure: how many operations completed in 20 seconds, varying the number of threads

- Goal: compare performance of different implementation approaches
Experimental Results

![Graph showing experimental results for different locking strategies. The x-axis represents the number of threads (72-processor machine), and the y-axis represents operations per millisecond. Different locking methods are compared, including Simple Locking, IntSetSimple/Aggressive, IntSetSimple/Polite, IntSetRelease/Aggressive, IntSetRelease/Polite, RBTTree/Aggressive, and RBTTree/Polite.]
Outline

• Introduction
• Dynamic Software Transactional Memory (DSTM)
• DSTM Implementation
  – Transactions and Transactional Objects
  – Contention Management
• DSTM performance
• DSTM2
• DSTM2 performance
Lessons Learned from DSTM

```java
TMObject<Node> tmNode = new TMObject<>(new Node());
Node rNode = tmNode.open(READ);
Node wNode = tmNode.open(WRITE);
```

• The programmer must not modify the object referenced by rNode
• If wNode is opened before rNode, changes to wNode are visible through rNode, but not if they are opened in the opposite order
• rNode and wNode references must not linger (programmer’s job)
• The Node class must provide a clone() method
• Programmers must be aware of the container based implementation:

```java
Class Node{
    Int value;
    TMObject<Node> next; //not Node
}
```
DSTM2

• Software transactional memory library (a collection of Java packages that supports transactional API)

• Safe, convenient and flexible API for application programmers

• Allows users to plug-in their own synchronization and recovery mechanisms (*transactional factories*)
Atomic Classes Comparison

**DSTM**

```java
TMObject<Node> newNode = new TMObject<>(new Node());
Node rNode = newNode.open(READ);
Node wNode = newNode.open(WRITE);

Class Node{
    Int value;
    TMObject<Node> next; //not Node
}
```

**DSTM2**

```java
@atomic public interface INode{
    int getValue();
    void setValue(int value);
    INode getNext();
    void setNext(INode value);
    ...
}

Factory<INode> factory = dstm2.Thread.makeFactory(INode.class);

INode newNode = factory.create();
```
Atomic Interface

- @atomic
  - Objects satisfying this interface should be safe to share
- Defines one or more properties (pairs of get and set) of certain type
- Property type: either scalar or @atomic interface
- May define other specialized methods

```java
@atomic public interface INode{
    int getValue();
    void setValue(int value);
    INode getNext();
    void setNext(INode value);
    ...
}
```
Transactional Factory

- Atomic interface is passed to a transactional factory constructor

- Use specific methods to create class implementing the interface

- The factory then creates instances of the class

```java
Factory<INode> factory = dstm2.Thread.makeFactory(INode.class);
INode newNode = factory.create();
```
Atomic Interface and Transactional Factory

• Semantics of get and set is clear

• Each factory is free to provide its own implementation for the methods declared

```java
@atomic public interface INode {
    int getValue();
    void setValue(int value);
    INode getNext();
    void setNext(INode value);
    ...
}
```

```java
Factory<INode> factory = dstm2.Thread.makeFactory (INode.class);

INode newNode = factory.create();
```
Obstruction-Free Factory

- Start
- transaction
  - new object
  - old object
  - oldLocator
- transaction
  - new object
  - old object
  - newLocator
- B(commit)
- Data
- Data
- copy
- Data
- A
- Data
Obstruction-Free Factory Variants

• Invisible reads
  – At commit time, a transaction must validate itself
    • Checks that the versions read are still current

• Visible reads
  – Each object maintains a list of reader transactions descriptors
  – A transaction intending to modify the object must first abort them
Shadow Factory

- committed
- transaction
- field1
- shadow1
- field2
- shadow2
- field3
- shadow3

- aborted
- transaction
- field1
- shadow1
- field2
- shadow2
- field3
- shadow3

- backup
- restore
Transactions

**DSTM**

```java
public boolean insert (int v){
    ...

    TMThread thread = (TMThread)thread.currentThread();

    while(true){
        thread.beginTransaction();
        try{
            ...
        } catch (Denied d){}
        if  (thread.commitTransaction())
            return result;
    }
}
```

**DSTM2**

```java
result = Thread.doIt (new Callable<Boolean>(){
    public boolean call(){
        return intSet.insert(v);
    }
}

public static <T> T doIt(Callable<T> xaction){
    while (!Thread.stop){
        beginTransaction();
        try{
            result=xaction.call();
        }catch (AbortedException d){}
        if  (commitTransaction()){
            return result;
        }
    }
}
```
Outline

• Introduction
• Dynamic Software Transactional Memory (DSTM)
• DSTM Implementation
  – Transactions and Transactional Objects
  – Contention Management
• DSTM performance
• DSTM2
• DSTM2 performance
DSTM2 Experimental Setup

• Linked-list and Skip List

• Configurations: obstruction-free factory (visible reads), obstruction-free factory with invisible reads, shadow factory

• 0%, 50%, 100% updates out of all operations

• Measure: transactions/second in a 20 second period

• Goal: show how DSTM2 can be used experimentally to evaluate the relative performance of different factories
DSTM2 Performance

• Linked List
  – The shadow factory 3-5 times higher throughput than the obstruction-free factories
    • Slightly higher when the percentage of updates decreases
  – Obstruction-free factories roughly the same results

• Skip List
  – Shadow factory better for high percentage of updates
Conclusions

• STM – API for low-level synchronized access to shared data without using locks

• DSTM – dynamic STM
  – Dynamic creation of transactions and transactional objects
  – Detect and reduce synchronization conflicts
  – Contention Manager (obstruction-freedom)

• DSTM2
  – Flexible API for application programmers