

# Transactional Memory

Michał Kapalka

EPFL, LPD

STiDC'08, 1.XII 2008

# How to Deal with Multi-Threading?

- Locks?
- Wait-free (or lock-free/obstruction-free), atomic objects?
- Transactional memory (this lecture)

# A Counter (not thread-safe)

```
public class Counter {  
    private int c = 0;  
  
    public void inc() {  
        c := c + 1;  
    }  
    public int get() {  
        return c;  
    }  
}
```

```
Counter cnt = new Counter();  
  
cnt.inc();  
k := cnt.get();
```

# A Counter with Locks

```

public class Counter {
    ...
    synchronized
    public void inc() {
        c := c + 1;
    }
    synchronized
    public int get() {
        return c;
    }
}

```

```

Counter cnt = new Counter();

cnt.inc();
k := cnt.get();

synchronized(cnt) {
    cnt.inc();
    k := cnt.get();
}

```

# Difficult Issues

How to **atomically**:

- 1 Fetch & increment 100 counters?
- 2 Fetch & increment a subset  $S$  of 100 counters?
- 3 Fetch & increment a counter and put the value in a synchronized hash table?
- 4 ...

# Ideal Transactional Memory (1)

```
public class Counter {
    ...
    @Atomic
    public void inc() {
        c := c + 1;
    }
    @Atomic
    public int get() {
        return c;
    }
}

Counter cnt = new Counter();

cnt.inc();
k := cnt.get();
```

# Ideal Transactional Memory (2)

**@Atomic**

```
public class Counter {
    ...
    public void inc() {
        c := c + 1;
    }

    public int get() {
        return c;
    }
}
```

```
Counter cnt = new Counter();
k := incAndGet();
```

...

```
@Atomic {
    cnt.inc();
    return cnt.get();
}
```

# Multiple Counters

```
@Atomic {  
    for(Counter cnt : counters) {  
        k[i++] := cnt.get();  
        cnt.inc();  
    }  
}
```



# Many Objects

```
@Atomic {  
    k := cnt.get();  
    cnt.inc();  
    table.put(k);  
}
```

# Implementing Transactional Memory

- In hardware (e.g., [Herlihy and Moss 93])
- In software (library, compiler, VM, etc.). Examples: DSTM ([Herlihy et al. 03]), TL2 ([Dice et al. 06])
- Hardware-software hybrids

# Basic Idea

**Atomicity** (formally defined as **opacity**) = transactions do not observe any concurrency:

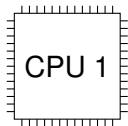
- Committed transactions: changes applied instantaneously
- Aborted transactions: changes never visible to others

Possible implementation of transaction atomicity:

- Many transactions can read the same object
- Writing requires exclusive ownership
- Conflicts  $\Rightarrow$  abort some transactions

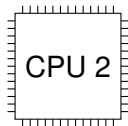
# Hardware Transactional Memory

State: non-transactional

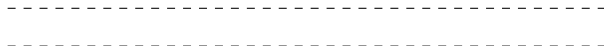


cache

State: non-transactional



cache

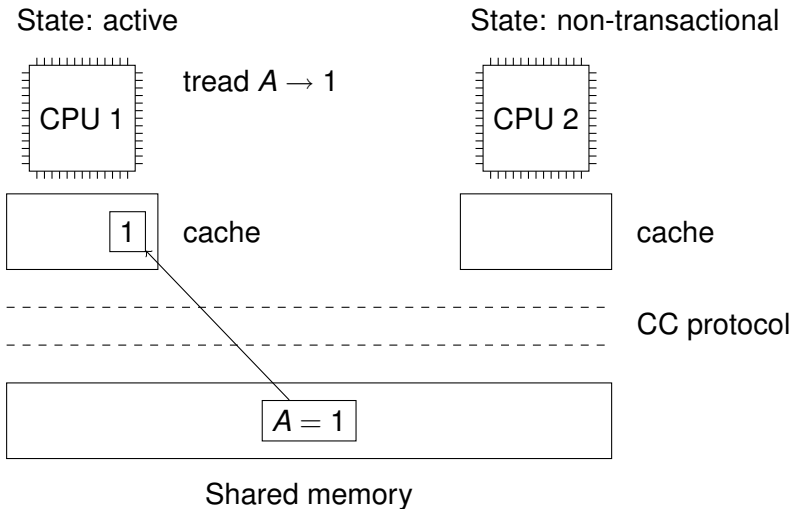


CC protocol

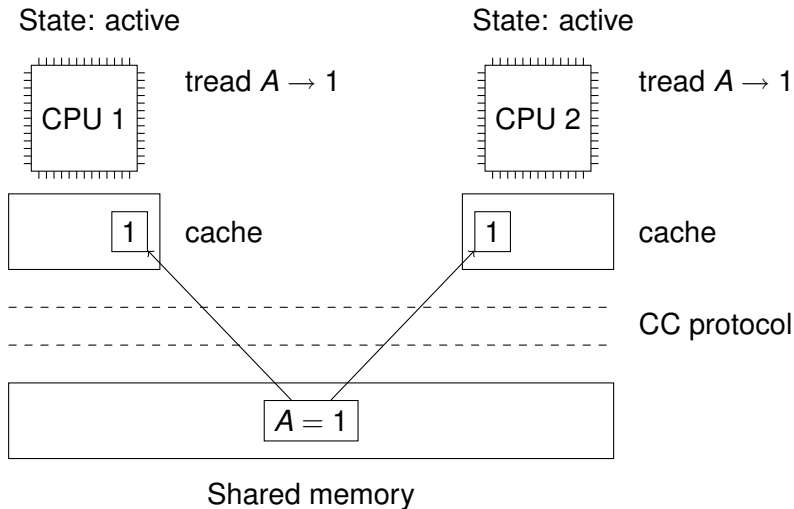


Shared memory

# Hardware Transactional Memory

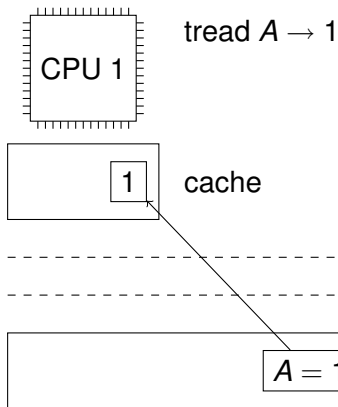


# Hardware Transactional Memory

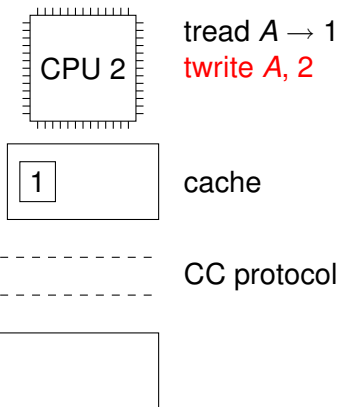


# Hardware Transactional Memory

State: active

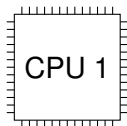


State: aborted

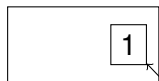


# Hardware Transactional Memory

State: active

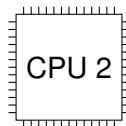


tread A  $\rightarrow$  1



cache

State: non-transactional



tread A  $\rightarrow$  1

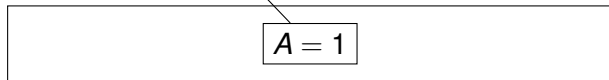
*twrite A, 2*

*commit  $\rightarrow$  false*



cache

CC protocol

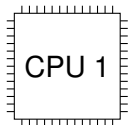


Shared memory

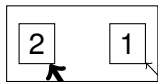


# Hardware Transactional Memory

State: active

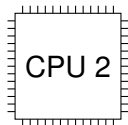


tread A  $\rightarrow$  1  
 twrite B, 3

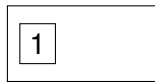


cache

State: non-transactional

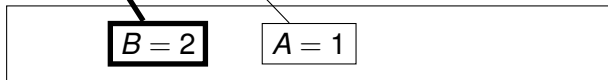


tread A  $\rightarrow$  1  
 twrite A, 2  
 commit  $\rightarrow$  *false*



cache

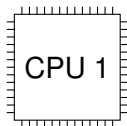
CC protocol



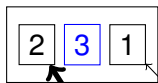
Shared memory

# Hardware Transactional Memory

State: active

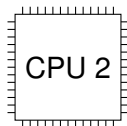


tread A  $\rightarrow$  1  
 twrite B, 3

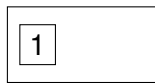


cache

State: non-transactional

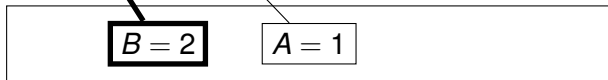


tread A  $\rightarrow$  1  
 twrite A, 2  
 commit  $\rightarrow$  *false*



cache

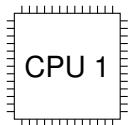
CC protocol



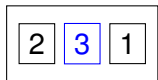
Shared memory

# Hardware Transactional Memory

State: committed

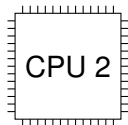


tread A  $\rightarrow$  1  
 twrite B, 3  
 commit  $\rightarrow$  *true*



cache

State: non-transactional

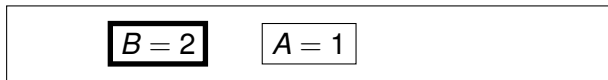


tread A  $\rightarrow$  1  
 twrite A, 2  
 commit  $\rightarrow$  *false*



cache

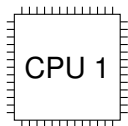
CC protocol



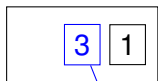
Shared memory

# Hardware Transactional Memory

State: committed

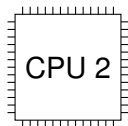


tread A  $\rightarrow$  1  
 twrite B, 3  
 commit  $\rightarrow$  *true*



cache

State: non-transactional

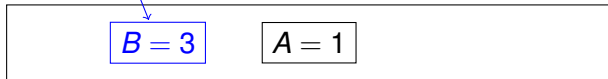


tread A  $\rightarrow$  1  
 twrite A, 2  
 commit  $\rightarrow$  *false*



cache

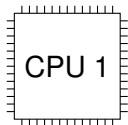
CC protocol



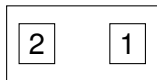
Shared memory

# Hardware Transactional Memory

State: committed

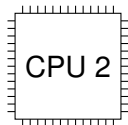


tread A  $\rightarrow$  1  
 twrite B, 3  
 abort



cache

State: non-transactional

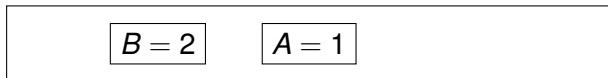


tread A  $\rightarrow$  1  
 twrite A, 2  
 commit  $\rightarrow$  *false*



cache

CC protocol



Shared memory

# From Hardware TM to Software TM

Example: DSTM

# Thread-Safe Counter using DSTM

```
TMObject cnt = new TMObject(new Counter());
```

```
beginTransaction();
```

```
Counter tmp_cnt = cnt.openWrite();
```

```
tmp_cnt.inc();
```

```
boolean committed = commitTransaction();
```

```
beginTransaction();
```

```
tmp_cnt = cnt.openRead();
```

```
k := tmp_cnt.get();
```

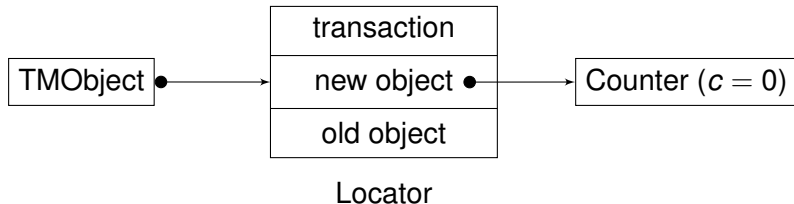
```
committed = commitTransaction();
```

# Thread-Safe Counter using DSTM

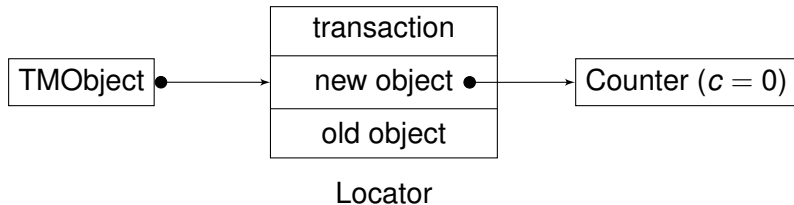
```
TMObject cnt = new TMObject(new Counter());  
  
beginTransaction();  
Counter tmp_cnt = cnt.openWrite();  
tmp_cnt.inc();  
k := tmp_cnt.get();  
boolean committed = commitTransaction();
```



# DSTM – The Idea

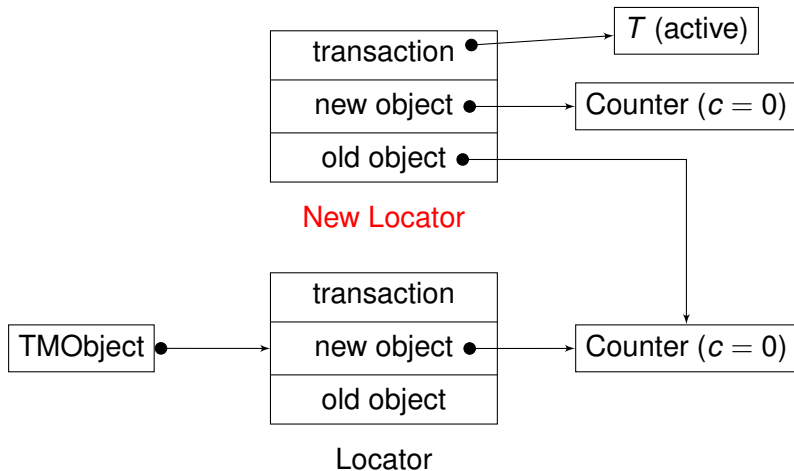


# DSTM – The Idea



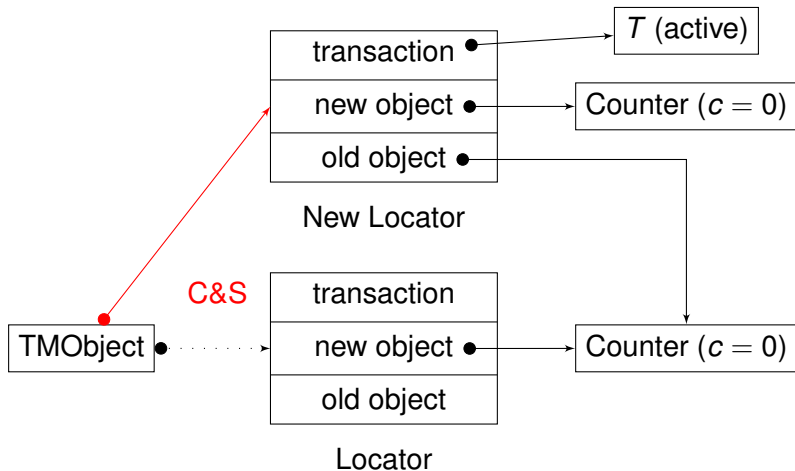
Transaction  $T$  wants to increment the counter

# DSTM – The Idea



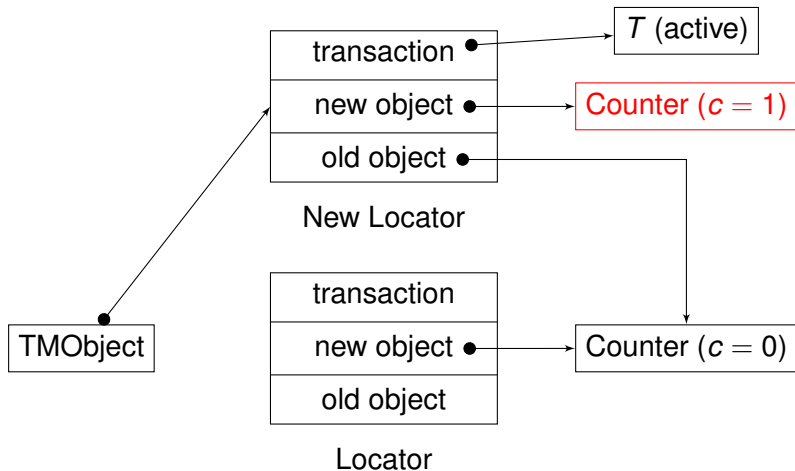
Step 1: Create new locator, clone the counter

# DSTM – The Idea



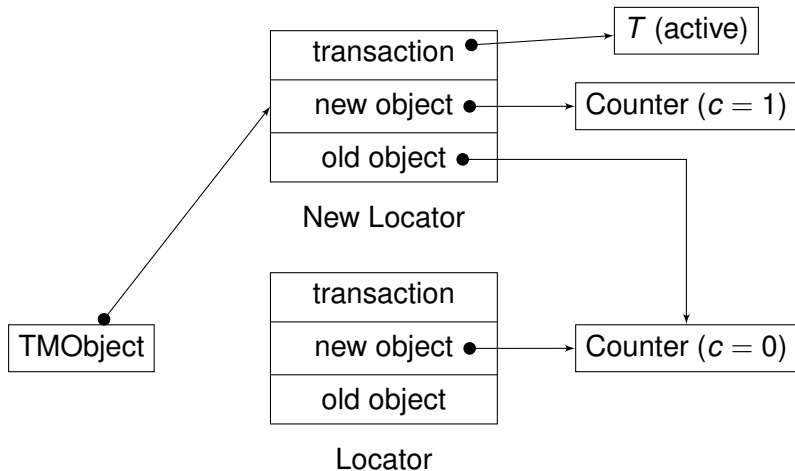
Step 2: Compare&Swap TMOBJECT

# DSTM – The Idea



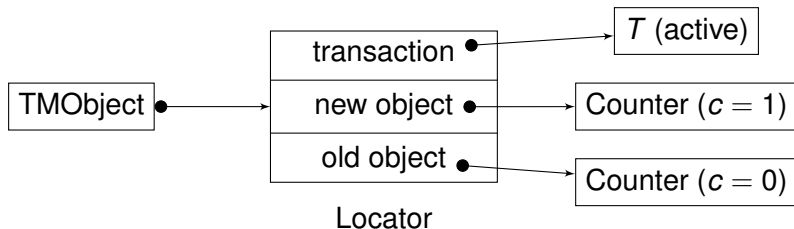
Step 3: Invoke `inc()` on the new counter

# DSTM – The Idea



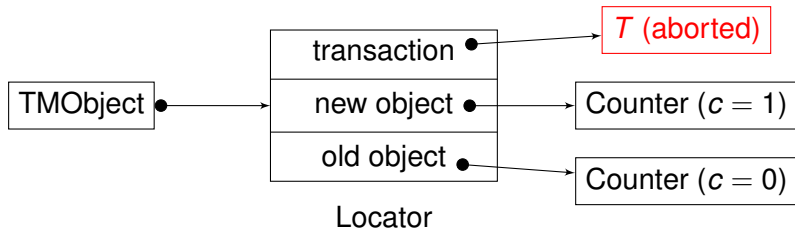
Old objects can be eventually garbage collected

# Resolving Conflicts



Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

# Resolving Conflicts

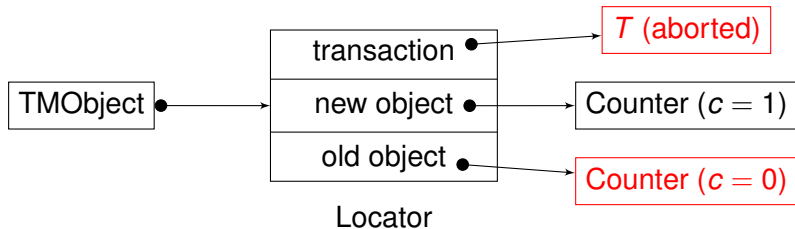


Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

Variant 1: **abort** transaction  $T$  (Compare&Swap on state of  $T$ )



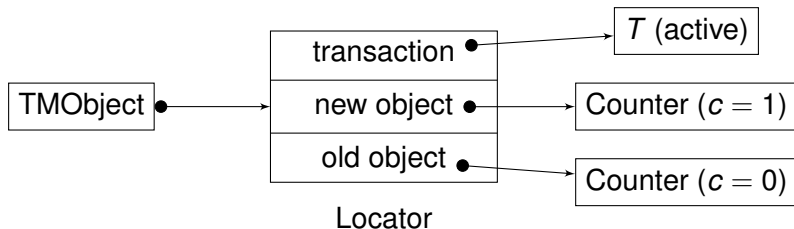
# Resolving Conflicts



Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

Variant 1: **abort** transaction  $T$  (Compare&Swap on state of  $T$ )  
 $\Rightarrow$  invoke `get()` on **old** counter

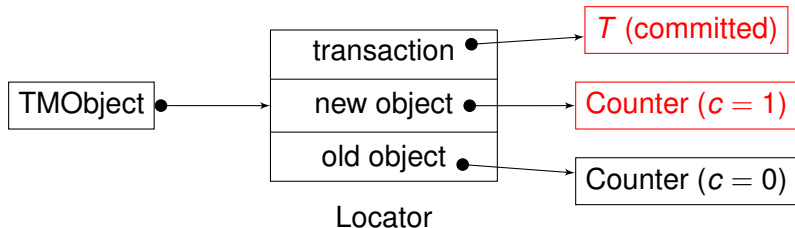
# Resolving Conflicts



Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

Variant 2: wait until  $T$  commits or aborts, then:

# Resolving Conflicts

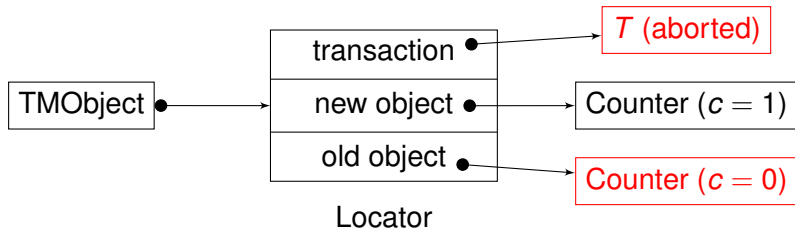


Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

Variant 2: wait until  $T$  commits or aborts, then:

$T$  **committed**  $\Rightarrow$  invoke `get()` on **new** counter

# Resolving Conflicts

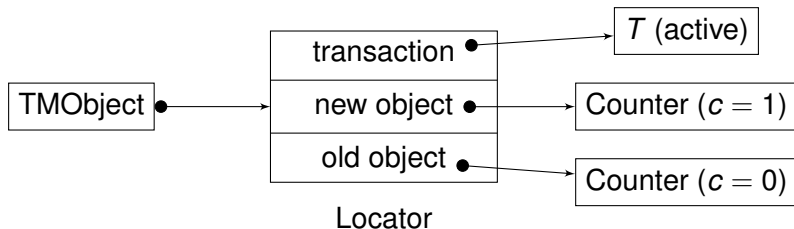


Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

Variant 2: wait until  $T$  commits or aborts, then:

$T$  **aborted**  $\Rightarrow$  invoke `get()` on **old** counter

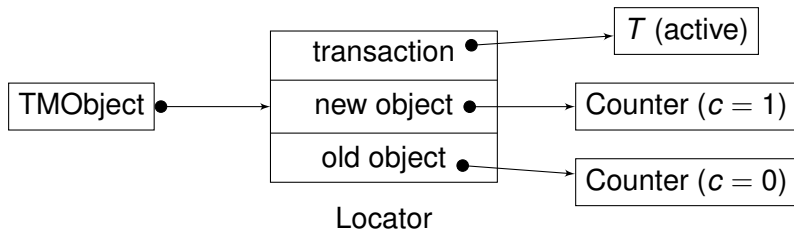
# Resolving Conflicts



Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

Variant 3: abort transaction  $T'$

# Resolving Conflicts



Now another transaction  $T'$  wants to read the counter  $\Rightarrow$  three possibilities:

Which variant to choose?  $\Rightarrow$  **contention manager** module decides

# Reading Objects

Transaction  $T$  wants to read an object:

- 1 If the object written by an active transaction  $\Rightarrow$  resolve conflict
- 2 Then, two techniques possible:
  - **Visible** reads:  $T$  adds itself to a **shared** list of readers (pointed by the locator)  $\Rightarrow$  readers have to write to shared memory (cache!)
  - **Invisible** reads (DSTM):  $T$  reads the object and remembers **locally** the value  $\Rightarrow$  writers do not know about readers  $\Rightarrow$  need **validation**
- 3 Validation: make sure no object previously read has changed (for  $n$  objects read so far,  $O(n)$  complexity in DSTM!)

# The Last Steps

- $T$  wants to **commit**:
  - 1 Validate (again!)
  - 2 Change state to “committed” (using C&S)
- $T$  wants to **abort**: change state to “aborted”

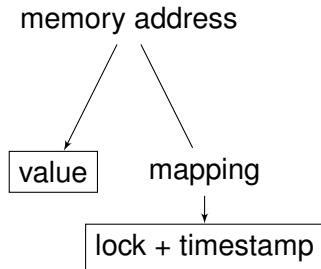


# From Hardware TM to Software TM

Example: TL2

# TL2 – The Idea

- Use **locks**  $\Rightarrow$  no copies, no indirection
- Invisible reads
- Make validation cheaper: **timestamps**
- Lock and write to objects only on commit time



# The Algorithm (1)

Uses a global version (strong) counter  $V$

Locks are 1-bit values (1 = locked)

Local variables:  $rver$ ,  $rset$ ,  $wver$ ,  $wset$

**upon** *beginTransaction*

$rver \leftarrow V.read()$

**upon** *write(addr, val)*

$wset \leftarrow wset \cup \{(addr, val)\}$

(Note that this is just a rough approximation of the TL2 algorithm. For detailed description see [Dice et al. 06])

# The Algorithm (2)

**upon** *read(addr)*

**if**  $addr \in wset$  **then return**  $wset[addr].val$

$(l_1, v_1) \leftarrow lockver[addr]$

$val \leftarrow$  read value from  $addr$

$(l_2, v_2) \leftarrow lockver[addr]$

**if**  $l_1 = 1$  **or**  $l_2 = 1$  **or**  $v_1 \neq v_2$  **or**  $v_2 > rver$  **then abort**

$rset \leftarrow rset \cup \{(addr, val)\}$

**return**  $val$

# The Algorithm (3)

**upon** *commitTransaction*

**foreach**  $(addr, val) \in wset$  **do**

    try to acquire lock in  $lockver[addr]$

**if** *failed to acquire* **then abort**

$wver \leftarrow V.inc()$

**if**  $wver \neq rver + 1$  **then**

**foreach**  $(addr, val) \in rset$  **do**

$(l, v) \leftarrow lockver[addr]$

**if**  $v > rver$  **or**  $l = 1$  **then abort**

**foreach**  $(addr, val) \in wset$  **do**

    store  $val$  at address  $addr$

$lockver[addr] \leftarrow (0, wver)$

# Further Reading



M. Herlihy, J. E. B. Moss.

Transactional memory: architectural support for lock-free data structures.

In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 289–300, 1993.



M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III.

Software transactional memory for dynamic-sized data structures.

In *Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*, pp. 92–101, 2003.



D. Dice, O. Shalev, and N. Shavit.

Transactional locking II.

In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.