Theory results in Transactional Memory

Panagiota Fatourou

FORTH ICS & University of Crete

HTDC 2013
La Plagne, France, March 2013
Topics to be discussed

• Basics of TM
• Safety
• Liveness
• Universal Constructions & their relation to TM
• Lower Bounds
• TM Algorithms
PART 1: Basics of TM
Software Transactional Memory (STM)

- An alternative parallel programming paradigm
- Relieves naive programmer from
  - using locks
  - developing non-blocking algorithms

**Key Idea:**
- STM system is developed by expert programmers
- Implementation details are hidden from the user
- Parallel programming for average user becomes easier
  - parallel code resembles to its sequential analog, so it is easily understandable, verifiable
STM

• Supports execution of transactions
  - blocks of sequential code, each of which contain accesses to pieces of data, called data items
  - In a concurrent environment, the data items may be accessed concurrently by several processes and, therefore, synchronization is needed
• STM guarantees that operations of some transaction
  - will be atomically executed, if transaction commits
  - will never become apparent, if transaction aborts
• To ensure consistency, STMs employ ownerships on data items
  - a transaction must acquire the ownership of some data item before accessing it
• Each STMs maintains additional information (metadata)
  - for each transaction, e.g. its status (pending or completed)
  - for each data item, e.g. the id of the transaction holding its ownership
Useful Definitions

• A **TM algorithm** provides, for each process, an implementation for the following functions:
  - T.ReadDI(x), where x is a handle for a data item
  - T.WriteDI(x,v), where x is a handle for a data item and v is a value
  - T.BeginTrans()
  - T.CommitTrans()
  - T.CreateDI(size), T.AbortTrans()

• Each time a transaction T calls one of these routines we say that it **invokes** an operation. Each operation returns a **response**.
Example: An ordered linked list

**Sequential Code**
```c
typedef struct node {
    int num;
    struct node *next;
} NODE;

NODE *L;
```

**Transactional Code**
```c
typedef struct node {
    int num;
    TmVar next;
} NODE;

TmVar L;
```
Example: An ordered linked list

**Sequential Code**

```c
Node *Insert (Node *L, int x) {
    NODE *p, *prevp = NULL, *newNode;
    1. p = L;
    2. while (p!=null && p->num < x) {
        3. prevp = p;
        4. p = p->next;
    }
    6. if (p!=null && p->num==x)
        7. return L;
    8. newNode = malloc(sizeof(Node));
    9. newNode->num = x;
   10. newNode->next = p;
   11. if (prevp != NULL)
        12. prevp->next = newNode;
    13. else return p;
```

**Transactional Code**

```c
<TmVar,boolean> Insert (TmVar L, int x) {
    NODE *p, *prevp = NULL, *newNode;
    TmVar pTmVar, prevpTmVar = NULL, newNodeTmVar;
    boolean abort;
    1. while (1) {
        2. BeginTrans ();
        3. pTmVar = L;
        4. (abort, p) = ReadDI (t, pTmVar);
        5. while  (! abort && p != null && p->num < x) {
            6. prevpTmVar = pTmVar;
            7. prevp = p;
            8. pTmVar = p->next;
            9. (abort, p) = ReadDI (t, pTmVar);
        }
        11. if (abort) { AbortTrans (); return <ABORT, L>; }
        12. if (p!=null && p->num==x)
            13. if (CommitTransaction()) return <COMMIT, L>;
            14. else return <ABORT, L>;
        15. ...
        16. } // while } // Insert
```
Example: An ordered linked list

**Sequential Code**

```c
boolean Insert (Node **L, int x) {
    NODE *p , *prevp, *newNode ;
    1.   p = L;
    2.   while ( P!=null && P->num < x)
    3.     prevp = p;
    4.     p = p->next;
    5.     if (p!=null && p->num==x)
    6.         return (FALSE);
    7.    newNode = malloc(sizeof(Node));
    8.    newNode->num = x;
    9.    newNode->next = p;
   10.   if (prevp != NULL)
   11.       prevp->next = newNode;
   12.   else return L = p;
   13.   return L;
```

**Transactional Code**

```c
void Insert (TmVar L, int x) {
    ...
    17.   newNode = (Node *)malloc (sizeof(Node));
    18.   newNode->num = num;
    19.   newNode->next = pTmVar;
   20.   newNodeTmVar = CreateDI (sizeof(NODE));
    21.   abort = WriteDI (newNodeTmVar, newNode);
    22.   if (abort) return <ABORT,L>;
    23.   If (prevp != NULL) {
    24.         newprev.num = prevp->num;
    25.         newprev.next = newNodeTmVar;
    26.         abort = WriteDI (prevpTmVar, &newprev);
    27.   }
    28.   else abort = WriteDI (L, newNodeTmVar);
    29.   if (abort) return <ABORT,L>;
    30.   if (CommitTransaction) return<COMMIT, L>;
}  // Insert
```

Panagiota Fatourou  faturu@ics.forth.gr  HTDC’13 - La Plagne
PART 2: Safety

- strict serializability
- serializability
- opacity
- snapshot isolation
- other consistency conditions found in TM
the operations supported by base objects are called primitives

processes may experience crash failures

Model

\[ \text{Processes: } p_1, p_2, \ldots, p_n \]

\[ \text{Shared Objects} \]

\[ \text{Register } R \]
- supports \text{read}(R) and \text{write}(R,v) \\

\[ \text{LL/SC object } R \]
- supports \text{LL}(R) and \text{SC}(R,v) \\

\[ \text{CAS object } R \]
- supports \text{read}(R) and \text{CAS}(R, v_{old}, v_{new}) \\

Panagiota Fatourou
faturu@ics.forth.gr

HTDC'13 - La Plagne
Model

• A configuration is a vector consisting of the state of each process and the value of each base object.

• A step of a process consists of a primitive on a base object, the response to this primitive, and possibly some local computation by the process.

• An execution is a sequence of steps taken by the processes.
Useful Definitions

- A **history** $H$ is a sequence of operation invocations and responses performed by transactions.


- $a,b,d$: data items, initially 0
- $T_1$ by $p_1$: $W(a,1)$ $W(b,1)$
- $T_2$ by $p_2$: $W(a,2)$ $W(d,2)$
- $T_3$ by $p_3$: $R(b)$
- $T_4$ by $p_4$: $R(d)$
Useful Definitions

- **H|T**: longest subsequence of H consisting only of invocations and responses of T
  - **H | T**: \( T_1.\text{BeginTrans}(), T_1.\text{ok}, T_1.\text{W(a,1)}, T_1.\text{ack}, T_1.\text{W(b,1)}, T_1.\text{ack}, T_1.\text{CommitTrans}(), T_1.\text{A}_1 \)
  - **H | T_4**: \( T_4.\text{BeginTrans}(), T_4.\text{ok}, T_4.\text{R(d)}, T_4.\text{<2>}, T_4.\text{CommitTrans}, T_4.\text{C}_4 \)

- **T** is live in H if H|T is not empty and does not end with T.A or T.C

- **H|p**: longest subsequence of H consisting only of invocations and responses of transactions executed by process p

- **H** is well-formed if \( \forall T \) in H:
  - **H|T** is a sequence of invocations and responses starting with BeginTrans(), ok
  - every invocation of ReadDI is followed either by a value or by \( A_T \)
  - each invocation of WriteDI is followed either by an ack or \( A_T \)
  - each invocation of CommitTrans is followed by \( C_T \) or \( A_T \)
  - no invocation follows after \( C_T \) or \( A_T \)

- For each execution a of a TM algorithm, we denote by \( H_a \) the sequence of invocations and responses performed by the transactions executed in a.
Partial Orders induced by histories

- A history $H$ induces an irreflexive partial order $\prec_H$ on transactions:
  - $T_1 \prec_H T_2$ if $T_1.A$ or $T_1.C$ appears in $H$ before $T_2.\text{BeginTrans}()$
- Informally, $\prec_H$ captures the real time precedence ordering of transactions in $H$.
- Transactions unrelated by $\prec_H$ are said to be concurrent.
- If $H$ is sequential, $\prec_H$ is a total order on transactions.

**Example**

$T_1 \prec_H T_4$, $T_2 \prec_H T_4$, $T_3 \prec_H T_4$

$T_1$, $T_2$ and $T_3$ are concurrent

---

Panagiota Fatourou  
faturu@ics.forth.gr  
HTDC’13 - La Plagne
Useful Definitions

• A history is **sequential** if no two transactions are concurrent in it.


• Histories $H$ and $H'$ are **equivalent** if they contain the same transactions and for every transaction $T$ in $H$, 
  $H|T = H'|T$.

  - $H$ and $S$ are equivalent.
Useful Definitions

• $\text{comp}(H)$: set of all histories which can be derived from $H$ by:
  - appending to $H$ an $A_T$ response for each transaction $T$ which has invoked an operation other than CommitTrans and has not received a response for it
  - appending either a $C_T$ or an $A_T$ response for every transaction $T$ that has invoked CommitTrans without having received a response for it
  - appending an invocation of an AbortTrans() and the response $A_T$ for each other transaction $T$.

• A sequential history $S$ is legal if its restriction to committed and live transactions (i.e., $\text{comm}(S)$) respects the sequential specification of the data items accessed in $\text{comm}(S)$.
We say that an execution $a$ is **strictly serializable** if it is possible to do all of the following:

- For each committed transaction $T$, to insert a serialization point $*_{T}$ somewhere between $T$'s first invocation and $T$'s last response in $a$.

- To choose a subset $A$ of the live transactions in $a$ and, for each transaction $T \in A$, insert a serialization point $*_{T}$ somewhere after $T$'s first invocation.

- These serialization points should be inserted, so that, in the sequential execution $\sigma$ constructed by serially executing each transaction $T$ at the point that its serialization point has been inserted:
  - if $T \not\in A$, the same operations, as in $a$, are invoked by $T$ and the response of each such operation is the same as that in $a$, and
  - if $T \in A$, a prefix of the operations invoked by $T$ in $\sigma$ are the same as all operations invoked by $T$ in $a$ and the response of each such operation is the same as that in $a$. 
Strict Serializability

• A history $H$ is **final-state strictly serializable** if there exists a history $H' \in \text{comp}(H)$ s.t.:
  1. $\text{comm}(H')$ is equivalent to some legal sequential history $S$,
  2. $\prec_{\text{comm}(H')} \subseteq \prec_{\text{comm}(S)}$

• A history $H$ is **strictly serializable** if each of its finite prefixes is final-state strictly serializable.

• The above definition of strict serializability is not equivalent to that provided in the previous slide.
Strict Serializability - Example

T₁ by p₁: W(a,1) W(b,1)
T₂ by p₂: W(a,2) W(d,2)
T₃ by p₃: R(b) -> 0
T₄ by p₄: R(d) -> 2


What if T1 also commits?


Serializability
Papadimitriou, J. ACM, 1979

• We say that an execution $a$ is **serializable** if it is possible to do all of the following:
  
  ✓ For each committed transaction $T$, to insert a serialization point $*_{T}$ somewhere between $T$'s first invocation and $T$'s last response in $a$.
  
  ✓ To choose a subset $A$ of the live transactions in $a$ and, for each transaction $T \in A$, insert a serialization point $*_{T}$ somewhere after $T$'s first invocation.
  
  ✓ These serialization points should be inserted, so that, in the sequential execution $\sigma$ constructed by serially executing each transaction $T$ at the point that its serialization point has been inserted:
    
    ✓ if $T \notin A$ the same operations, as in $a$, are invoked by $T$ and the response of each such operation is the same as that in $a$.
    
    ✓ if $T \in A$, a prefix of the operations invoked by $T$ in $\sigma$ are the same as all operations invoked by $T$ in $a$ and the response of each such operation is the same as that in $a$. 
Serializability

• A history $H$ is **final-state serializable** if there exists a history $H' \in \text{comp}(H)$ s.t.:
  1. $\text{comm}(H')$ is equivalent to some legal sequential history $S$,
  2. $\text{comm}(H') \subseteq \text{comm}(S)$ for each process $p$, $\text{comm}(H') \upharpoonright p = S \upharpoonright p$

• A history $H$ is **serializable** if each of its finite prefixes is final-state serializable.

• The above definition of serializability is not equivalent to that provided in the previous slide.
Serializability - Example

The above execution is not strictly serializable.

However, it is serializable.

**Strict serializability is a stronger condition than serializability:**

Any strictly serializable execution is serializable.
The opposite is not TRUE!
• In addition to ensuring conditions on the behavior of committed (or ready to commit) transactions, opacity places conditions on the behavior of non-committed transactions as well:
  - non-committed transactions should not behave inappropriately, i.e., their execution should not result in:
    • entering infinite loops, or
    • abnormal termination (e.g. segmentation fault)
• A history $H$ is final-state opaque if there exists a history $H' \in \text{comp}(H)$ s.t.:
  - $H'$ is equivalent to some legal sequential history $S$,
  - $<H' \subseteq <S$
  - for every transaction $T \in H'$, the longest subsequence of $S$ made of (1) all transactions preceding $T$ in $S$, and (2) every prefix of $S|T$ s.t. $T$ is live in $S|T$, is a legal history.
• A history $H$ is opaque if each of its finite prefixes is final-state opaque.
Opacity - Example

This execution is not opaque.

This execution is opaque.
Snapshot Isolation (SI)
Lu et al., IEEE Trans. on Knowledge & Data Eng. (DB),
Riegel et al., Transact 2006 (TM)

• **Snapshot isolation** requires that transactions should be executed as if every readDI operation reads from some snapshot of the memory that was taken when the transaction started.

• **Snapshot isolation is appealing for TM:**
  - it provides the potential to increase throughput for workloads with long transactions
Snapshot Isolation

- Let $H$ be a history and $T$ be a transaction in $H$.
- $T|\text{read}$: longest subsequence of $T$ consisting only of read invocations
- $T|\text{write}$: longest subsequence of $T$ consisting only of write invocations
- If $T$ is committing (or live):
  - $T_r = T|\text{read}$, $\text{CommitTrans}(T_r)$, $C_{Tr}$ if $T|\text{read} \neq \lambda$, and $\lambda$ otherwise
  - $T_w = T|\text{write}$, $\text{CommitTrans}(T_w)$, $C_{Tw}$ if $T|\text{write} \neq \lambda$, and $\lambda$ otherwise
- If $T$ is aborting:
  - $T_r = T|\text{read}$ if $T|\text{read}$ contains $A_{Tr}$ or if $T|\text{read} = \lambda$, and
  - $T_r = T|\text{read} \text{CommitTrans}(T_r) A_{Tr}$ otherwise
  - $T_w = \lambda$
Snapshot Isolation (roughly speaking)

- An execution $a$ satisfies **snapshot isolation**, if for every completed transaction $T$ in $a$ (and for some of the live transactions) it is possible to insert a read serialization point $*_{T,r}$ and a write serialization point $*_{T,w}$ s.t:
  - $*_{T,r}$ precedes $*_{T,w}$
  - both $*_{T,r}$ and $*_{T,w}$ are inserted within the execution interval of $T$,
  - if $\sigma_a$ is the sequence defined by these serialization points, in order, and $H_{\sigma_a}$ is the history we get by replacing each $*_{T,r}$ with $T_r$ and each $*_{T,w}$ with $T_w$ in $\sigma_a$, then $H_{\sigma_a}$ is legal.
Snapshot Isolation - Example

\[ T_1 \xrightarrow{R(w) \to 0} R(z) \to 0 \xrightarrow{W(x,1)} W(y,1) \]
\[ *r_{T1} \]
\[ T_2 \xrightarrow{R(x) \to 0} \xrightarrow{W(w,1)} \]
\[ *r_{T2} * \]
\[ T_3 \xrightarrow{R(y) \to 1} \xrightarrow{W(z,1)} \]
\[ *r_{T3} * \]
\[ W_{T1} \]
\[ s \]
\[ W_{T2} \]
\[ W_{T3} \]

\( x,y,w,z \): data items, initially 0

\( T_1 \) by \( p_1 \): \( R(w) \ R(z) \ W(x,1) \ W(y,1) \)
\( T_2 \) by \( p_2 \): \( R(x) \ W(w,1) \)
\( T_3 \) by \( p_3 \): \( R(y) \ W(z,1) \)

This execution is not strictly serializable.

However, it ensures snapshot isolation.
Other Consistency Conditions in TM

- **Causal Consistency** (Imbs et al., INRIA TR, 2008)
- Virtual World Consistency: Weaker version of opacity where for non-committed transactions only causal consistency is ensured (Imbs et al., INRIA TR, 2008)
- **TMS1 and TMS2**: variants of opacity (Doherty et. al, WTTM 2012)
- **Z-linearizability**: transactions are partitioned into two sets, long and short transactions, and different consistency conditions are ensured for each set. (Riegel et. al, Universite de Neuchatel TR, 2007)
- Many consistency conditions have previously appeared in the DB literature and probably make sense for TM computing as well: (Attiya et. al, Transact 2012)
  - Recoverability, Strictness, Rigorousness, Avoiding Cascading aborts
  - I am sure that there are more...
- etc.
Privatization

- strong atomicity/isolation
- weak atomicity/isolation
- single lock atomicity
- disjoint lock atomicity
- asymmetric flow ordering
- encounter time lock atomicity
- selective strict serializability
- internal consistency
- race-free atomicity
PART 2: Liveness
Conventional Progress Conditions

Blocking algorithms

- An arbitrary and unexpected delay by a process may prevent all other processes from making progress.
- e.g., a process that holds a lock
- Deadlock-freedom: no deadlock occurs
  - the system as a whole makes progress but progress to individual processes is not guaranteed
- Starvation-freedom: no starvation occurs
  - Every process eventually makes progress (e.g., it acquires the lock).

Blocking algorithms do not ensure progress if processes crash
Conventional Progress Conditions

Non-blocking algorithms

- An arbitrary and unexpected delay by any process does not prevent other processes from making progress.
- Obstruction-freedom
  - Every process makes progress if it executes solo for long enough (starting from any reachable configuration).
- Lock-freedom
  - Progress is ensured for some of the active processes
- Wait-freedom
  - Progress is ensured for each active process

Non-blocking algorithms ensure progress even if processes crash
TM Progress

✓ A TM progress property should additionally ensure that some transactions commit

• Local Progress*
  - Each process that keeps executing a transaction (e.g. restarts it in case it aborts) eventually commits it.
  - Similar to wait-freedom

• Global Progress*
  - Some process that keeps executing a transaction eventually commits it.
  - Similar to lock-freedom

• Solo Progress*
  - Each process which eventually runs solo while it keeps executing a transaction eventually commits it.


• Obstruction-freedom#
  - A transaction that does not encounter step contention during the course of its execution must commit.

TM Progress

- **Permissiveness**: a transaction must commit unless doing so violates correctness

- **Progressiveness**: a transaction must commit if it doesn't conflict with any other transaction

- **Strong Progressiveness**: progressiveness + if a number of transactions conflict on a single t-var, then at least one of them must be able to commit.
TM Progress

- **Read-only transactions never abort**
    - update transactions may abort and they require locks to execute some of the transactional instructions

- **No transaction ever aborts**
    - read-only transactions are wait-free
    - write-only transactions are blocking and they are executing one-after-the-other using a global lock
    - no conflict ever occurs, so no transaction ever aborts
    - parallelism is fine-grained; it is achieved at the level of transactional instructions instead of transactions themselves
Universal Constructions

- Provides a general mechanism to **automatically execute** pieces of **sequential code** in a concurrent environment.

- For each process $p$, an operation `Perform` is supported.

- Takes as **parameters**:
  - piece of sequential code
  - input arguments

- Applies code to the simulated state.

- Returns a response to $p$.

---

Panagioti Fatourou  
faturu@ics.forth.gr  
HTDC'13 - La Plagne
Universal constructions and TM algorithms are closely related

**Same Goal**
Simplify parallel programming by providing mechanisms to efficiently execute code in a concurrent environment.

**Main Differences**

1. **Support of “Abort”**
   - A TM algorithm informs the external environment when a transaction is aborted.
   - Universal constructions, thus far, did not have the facility to abort the execution of the piece of code that was passed as a parameter.

2. **Access to transaction’s code**
   - A TM system invokes actions for reading or writing a data item, initiating, committing or aborting a transaction.
     - In this talk we will call this model `tcode-unaware`
   - A universal construction requires that the transaction’s code is placed in a routine and a pointer to this routine is passed as an argument to `Perform`.
     - In many cases, e.g. in data structures, this is indeed the case
     - A lot of TM implementations (DSTM, TL II) assume that a pointer to a function containing the transaction’s code is passed as an argument to the implementation.
     - We will call this model `tcode-aware`

- Programming in the `tcode-unaware` model is probably easier in some cases
Results

• There does not exist a tcode-unaware TM implementation that ensures both local progress and opacity. (Bushkov et al, PODC 2012)

• There exist several universal constructions that ensure wait-freedom and can serve as wait-free tcode-aware TM systems
  - Afek et al., STOC 1995
  - Anderson & Moir, PODC 1995
  - Chuong et al., SPAA 2010
  - Fatourou et al., DISC 2009, SPAA 2011
  - Herlihy, PPoPP 1990, TOPLAS 1991

It is easy to ensure that no transaction ever aborts by allowing waiting or helping.
## Results

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Possible/Impossible/NP-hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opacity + progressiveness</td>
<td>Possible</td>
</tr>
<tr>
<td>Imbs et al., OPODIS 2008</td>
<td></td>
</tr>
<tr>
<td>Opacity + permissiveness + invisible reads</td>
<td>Impossible</td>
</tr>
<tr>
<td>Crain et al., IRISA TR, 2010</td>
<td></td>
</tr>
<tr>
<td>Opacity + permissiveness</td>
<td>NP-hard</td>
</tr>
<tr>
<td>Guerraoui et al., DISC 2008</td>
<td></td>
</tr>
<tr>
<td>Opacity + probabilistic permissiveness + lock freedom</td>
<td>Possible</td>
</tr>
<tr>
<td>Guerraoui et al., DISC 2008</td>
<td></td>
</tr>
<tr>
<td>Virtual world consistency + probabilistic permissiveness + invisible reads</td>
<td>Possible</td>
</tr>
<tr>
<td>Crain et al., IRISA TR, 2010</td>
<td></td>
</tr>
</tbody>
</table>
PART 3: Impossibility Results & Lower Bounds
Useful Definitions

• An execution $a$ is **legal** starting from a configuration $C$ if the sequence of steps performed by each process follows the algorithm for that process (starting from its state in $C$), and for each object, the responses to the operations performed on the object are in accordance with its specification (and the value stored in the object at $C$).

• An execution $a$ is **indistinguishable** from another execution $a'$ for some processes, if each of these processes take the same steps in $a$ and $a'$, and each of these steps has the same response in $a$ and $a'$.
Parallelizability

- We say that two transactions **conflict** in an execution \( a \), if they both invoke an operation on a common data item.

- We say that two executions **contend** on a base object \( o \) if they both contain a primitive on \( o \) and one of these primitives is non-trivial (i.e. it may update \( o \)).

- We say that a TM implementation is **strictly disjoint-access parallel** (DAP) if, in each execution \( a \), and for every two transactions \( T_1 \) and \( T_2 \) in \( a \), if \( a|T_1 \) and \( a|T_2 \) contend on some base object, then \( T_1 \) conflicts with \( T_2 \) in \( a \).
Parallelizability

**Conflict Graph of an execution interval**

- vertices represent transactions whose execution overlaps with the interval
- edges connect transactions that conflict
- A TM implementation $I$ is disjoint-access-parallel (dap) if, for every execution $a$ of $I$ which contains two transactions $T_1$ and $T_2$, $a|T_1$ and $a|T_2$ contend on some base object, only if there is a path between $T_1$ and $T_2$ the conflict graph of the minimal execution interval $a$ containing $T_1$ and $T_2$.

Panagiota Fatourou  
faturu@ics.forth.gr  
HTDC’13 - La Plagne
Impossibility Results

1. No TM implementation ensures obstruction-freedom, strict DAPism and strict serializability

2. No TM implementation ensures obstruction-freedom, strict DAPism and snapshot isolation.

3. No TM implementation (where each aborted transaction is restarted) ensures DAPism and wait-freedom.
Obstruction-freedom, Strict serializability and Strict DAPism: Impossible

\[ x, y, w, z: \text{data items, initially 0} \]

\[ T_1 \text{ by } p_1: R(w) \ R(z) \ W(x,1) \ W(y,1) \]

\[ T_2 \text{ by } p_2: R(x) \ W(w,1) \]

\[ T_3 \text{ by } p_3: R(y) \ W(z,1) \]

Critical step

- If each of \( T_2, T_3 \) runs solo starting from the configuration before \( s \), its read returns 0.
- If each of \( T_2, T_3 \) runs solo starting from the configuration after \( s \), the read of at least one of \( T_2 \) or \( T_3 \) returns 1; wlog, assume that it is \( T_3 \) that reads the value 1 for \( y \).
Obstruction-freedom, Strict serializability & Strict DAPism: Impossible

This execution violates strict serializability:
• Since $T_2$ commits, $T_1$ must abort.
• Since $T_3$ commits, $T_1$ must commit.
A contradiction!

- $a_2$ does not perform any non-trivial operation on $o$ or any other base object read in $a_3$.
  • $s$ is legal after $a_1a_2$
  • $a_3$ is legal after $a_1a_2s$

$T_1$ by $p_1$: $R(w) R(z) W(x,1) W(y,1)$
$T_2$ by $p_2$: $R(x) W(w,1)$
$T_3$ by $p_3$: $R(y) W(z,1)$
Obstruction-freedom, Snapshot Isolation & strict DAPism: Impossible

Main Ideas

• Proof by contradiction. Assume there is an obstruction-free TM alg A which ensures snapshot isolation and strict DAPism.

• We will construct two executions:
  \[ \delta_1 = a_1 a_2 s_1 s_2 \gamma_6, \text{ and} \]
  \[ \delta_2 = a_1 a_2 s_2 s_1 \gamma'_6, \text{ s.t.:} \]
  - \( \gamma_6 \) and \( \gamma'_6 \) are solo executions of \( T_6 \)
  - in \( \gamma_6 \), \( T_6 \) reads the value 2 for \( a \)
  - in \( \gamma'_6 \), \( T_6 \) reads the value 1 for \( a \)
  - \( \gamma_6 \) and \( \gamma'_6 \) are indistinguishable.

• Thus, \( T_6 \) should read the same value for \( a \) in both executions. A contradiction!
Definition of $a_1$, $a_2$, $s_1$ and $s_2$

1. $s_1$ applies a non-trivial operation on some base object $o_1$ s.t $T_3$ reads $o_1$ in $\gamma_3$

2. $s_2$ applies a non-trivial operation on some base object $o_2$ s.t $T_4$ reads $o_2$ in $\gamma_4$

$T_1$ by $p_1$: $W(a,1)$ $W(b,1)$ $W(c,1)$
$T_2$ by $p_2$: $W(a,2)$ $W(d,2)$ $W(e,2)$
$T_3$ by $p_3$: $R(b)$
$T_4$ by $p_4$: $R(d)$
$T_5$ by $p_5$: $R(e)$
$T'_5$ by $p_5$: $R(c)$
$T_6$ by $p_6$: $R(a)$

Panagiota Fatourou
faturu@ics.forth.gr
HTDC’13 - La Plagne
**Useful Claims**

Claim 1: \( T'_5 \) reads 1 for c in \( \beta'_5 \).

Claim 2: \( T_5 \) reads 2 for e in \( \gamma_5 \).
- Proved in a similar way as Claim 1

Claim 3: \( o_1 \neq o_2 \).
- \( \gamma_3 \) is legal after \( a_1 a_2 s_1 \)
- \( \gamma_3 \) reads \( o_1 \)
- \( \gamma_3 \) does not read any base object written by \( a_2 s_2 \)
What is our goal?

Goal
• To construct two executions:
  \[ \delta_1 = a_1 a_2 s_1 s_2 \gamma_6, \text{ and} \]
  \[ \delta_2 = a_1 a_2 s_2 s_1 \gamma_6, \text{ s.t.:} \]
  - \( \gamma_6 \) and \( \gamma'_6 \) are solo executions of \( T_6 \)
  - in \( \gamma_6 \), \( T_6 \) reads the value 2 for \( a \)
  - in \( \gamma'_6 \), \( T_6 \) reads the value 1 for \( a \)
  - \( \gamma_6 \) and \( \gamma'_6 \) are indistinguishable.

Claim 1: \( T'_5 \) reads 1 for \( c \) in \( \beta'_5 \).
Claim 2: \( T_5 \) reads 2 for \( e \) in \( \gamma_5 \).
Claim 3: \( o_1 \neq o_2 \).

Lemma 1: \( \gamma_6 \) and \( \gamma'_6 \) are indistinguishable.
\(T_6\) reads 2 for \(a\) in \(\gamma_6\)

Recall that (1) \(T_3\) reads 1 for \(b\) in \(\gamma_3\), and (2) \(T_3\) reads \(o_1\)

**Claim 4:** \(T_4\) reads 0 for \(d\) in \(\gamma_4\)
- \(\gamma_4':\) solo execution of \(T_4\) after \(a_1\) \(a_2\)
- \(\gamma_4\) and \(\gamma_4'\) are indistinguishable

Recall that \(\gamma_5\) is the solo execution of \(T_5\) after \(a_1\) \(a_2\) \(s_2\)

Recall that (1) \(T_5\) reads 2 for \(e\) in \(\gamma_5\) and (2) \(o_1 \neq o_2\)

**Claim 5:** \(T_5\) reads 2 for \(e\) in \(\gamma_5'\)
- \(\gamma_5\) and \(\gamma_5'\) are indistinguishable

If \(\gamma_5'\) does not read \(o_2 \rightarrow \gamma_5\) and \(\gamma_5'\) are indistinguishable.

Otherwise, \(\gamma_4\) does not write \(o_2\), since then \(\gamma_4\) and \(\gamma_5'\) conflict which violates strict DAPism.
\( T_6 \) reads 2 for \( a \) in \( \gamma_6 \)

- \( \gamma_6 \) and \( \gamma''_6 \) are indistinguishable

- \( T_6 \) reads 2 for \( a \) in \( \gamma_6 \)

\[C_0\ a_1\ a_2\ s_1\ s_2\ \cdots\ \cdots\ \cdots\ s_2\ y_3\ y_4\ y_5\ y'_5\ y_6\ y''_6\]

- \( T_3 \) runs solo until it commits
- \( T_4 \) runs solo until it commits
- \( T_5 \) runs solo until it commits
- \( T_6 \) runs solo until it commits

- \( T_1 \) by \( p_1 \): \( W(a,1) \) \( W(b,1) \) \( W(c,1) \)
- \( T_2 \) by \( p_2 \): \( W(a,2) \) \( W(d,2) \) \( W(e,2) \)
- \( T_3 \) by \( p_3 \): \( R(b) \)
- \( T_4 \) by \( p_4 \): \( R(d) \)
- \( T_5 \) by \( p_5 \): \( R(e) \)
- \( T_5 \) by \( p_5 \): \( R(c) \)
- \( T_6 \) by \( p_6 \): \( R(a) \)

Panagiota Fatourou

faturu@ics.forth.gr

HTDC'13 - La Plagne
**Claim 6:** $T_5$ reads 2 for $e$ in $\gamma'_5$

$\cdot \gamma_5$ and $\gamma'_5$ are indistinguishable

**Claim 7:** $\gamma'_6$ and $\gamma''_6$ are indistinguishable

$\blacktriangleleft T_6$ reads 1 for $a$ in $\gamma'_6$

$T_1$ by $p_1$: $W(a,1) W(b,1) W(c,1)$

$T_2$ by $p_2$: $W(a,2) W(d,2) W(e,2)$

$T_3$ by $p_3$: $R(b)$

$T_4$ by $p_4$: $R(d)$

$T_5$ by $p_5$: $R(e)$

$T_5$ by $p_5$: $R(c)$

$T_6$ by $p_6$: $R(a)$

Panagiota Fatourou  

faturu@ics.forth.gr  

HTDC'13 - La Plagne
Putting it all together

We have constructed two executions:
\[ \delta_1 = \alpha_1 \alpha_2 s_1 s_2 \gamma_6, \text{ and} \]
\[ \delta_2 = \alpha_1 \alpha_2 s_2 s_1 \gamma'6, \text{ s.t.:} \]
- \( \gamma_6 \) and \( \gamma'6 \) are solo executions of \( T_6 \)
- in \( \gamma_6 \), \( T_6 \) reads the value 2 for a
- in \( \gamma'6 \), \( T_6 \) reads the value 1 for a
- \( \gamma_6 \) and \( \gamma'6 \) are indistinguishable.

It follows that \( T_6 \) should read the same value for \( a \) in both executions. A contradiction!

\[
\begin{array}{cccccc}
C_0 & a_1 & a_2 & s_1 & s_2 & \gamma_6 \\
\hline
\end{array}
\]

\( T_6 \) runs solo until it commits

\[
\begin{array}{cccccc}
C_0 & a_1 & a_2 & s_2 & s_1 & \gamma'6 \\
\hline
\end{array}
\]

\( T_6 \) runs solo until it commits
DAPism & wait-freedom: Impossible

**Proof intuition:** there is an execution where
- process \( p \) performs a single instance of \( \text{Search}(0) \) that never terminates
- process \( q \) performs instances of \( \text{Append}(i) \), for \( i \geq 1 \)

Proof intuition: there is an execution where
- process \( p \) performs a single instance of \( \text{Search}(0) \) that never terminates
- process \( q \) performs instances of \( \text{Append}(i) \), for \( i \geq 1 \)

Panagiota Fatourou  
faturu@ics.forth.gr  
HTDC’13 - La Plagne
Proof
1. Construction of a valid infinite execution
2. We prove that for each $i \geq 3$, at least one of $\gamma_i$, $\gamma_{i+1}$,
   and $\gamma_{i+2}$ is nonempty
DAPism & wait-freedom: Impossible

Proof
3. As an example, we consider $i = 3$ and by contradiction, we assume that $\gamma_3$, $\gamma_4$, and $\gamma_5$ are empty

3.1. $p$ continues after $\beta_2$ (by accessing some base object $o$)

Panagiota Fatourou faturu@ics.forth.gr HTDC’13 - La Plagne
DAPism & wait-freedom: Impossible

3.1. \( p \) continues after \( \beta_2 \) (by accessing some base object \( o \) that is accessed by at least one of \( \text{Append}(5), \text{Append}(6), \ldots \))

3.2. the solo executions of \( \text{Append}(5) \) and \( \text{Search}(0) \) starting from \( d' \) access \( o \)

3.3. the solo executions of \( \text{Append}(5) \) and \( \text{Search}(0) \) starting from \( d' \) access no common base object

---

**Contradiction!**

Search(0) starting from \( d \) responds with false and Search(0) starting from \( d' \) responds with true

---

Panagiota Fatourou  
faturu@ics.forth.gr  
HTDC'13 - La Plagne
### Possible/Impossible Combinations for strict serializable Implementations

<table>
<thead>
<tr>
<th>Combination</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>strict dap + obstruction-freedom</strong></td>
<td>Impossible</td>
</tr>
<tr>
<td>Guerraoui &amp; Kapalka, SPAA 2008</td>
<td></td>
</tr>
<tr>
<td><strong>dap + nonblocking</strong></td>
<td>Possible</td>
</tr>
<tr>
<td>Fraser UCAM-CL-TR, 2003</td>
<td></td>
</tr>
<tr>
<td><strong>dap + wait-freedom</strong></td>
<td>Impossible</td>
</tr>
<tr>
<td>Ellen et al., PODC 2012</td>
<td></td>
</tr>
<tr>
<td><strong>dap + wait-freedom for operations with bounded data sets</strong></td>
<td>Possible</td>
</tr>
<tr>
<td>Ellen et al., PODC 2012</td>
<td></td>
</tr>
<tr>
<td><strong>dap + nonblocking + read-only transactions never abort</strong></td>
<td>Impossible</td>
</tr>
<tr>
<td>Attiya et al., SPAA 2009</td>
<td></td>
</tr>
<tr>
<td><strong>dap + read-only transactions never abort</strong></td>
<td>Possible</td>
</tr>
<tr>
<td>Avni and Shavit, SIROCCO 2008</td>
<td></td>
</tr>
</tbody>
</table>
Complexity

opacity + permissiveness:
• $\Omega(r)$ synchronization primitives required for a transaction that reads $r$ data items*

opacity + progressiveness:
• $O(1)$ synchronization primitives suffice*

opacity + progressiveness + dap:
• $\Omega(w)$ synchronization primitives required for a transaction that writes $w$ data items*

* Kuznetsov et al., OPODIS 2011
Complexity

- strict serializability + dap + wait-free read only transactions: reading \( r \) data items requires updating \( 2^r - 1 \) base objects
  Attiya et al., SPAA 2009

- strong atomicity + dap + invisible reads + progressiveness/obstruction freedom: the data set of a privatizing transaction must contain all privatized data items
  Attiya & Hillel, DISC 2010

- strong atomicity + dap + progressiveness/obstruction freedom: in the worst case, a transaction privatizing \( k \) data items must access at least \( k \) base objects
  Attiya & Hillel, DISC 2010
Part 5: TM Algorithms
Design of STM systems - Ownerships

- Recall that to ensure consistency, TM algorithms acquire *ownerships* on data items.

- How long an ownership can be hold?
  - non-preemptive ownerships (e.g. locks)
    - lead to blocking STM algorithms
  - preemptive ownerships
    - a transaction may either *forcefully abort* some other transaction, or
    - *help* it so that the ownerships it holds are released
    - lead to non-blocking STM algorithms

- When ownerships are acquired?
  - upon first access to datum (*eager acquisition*)
  - after accessing datum, e.g. at commit time (*lazy acquisition*)
Design of STM systems

- **Writing** data items
  - When a transaction aborts, its modifications must be discarded and never become apparent
  - acquire the ownership before writing and maintain it until transaction’s completion
    - with eager acquisition an **undo log** is maintained
    - with lazy acquisition a **redo log** is maintained

- **Reading** data items
  - it is possible some data item to be updated after reading it. Then, what was read may be **inconsistent** (in some executions)
    - the algorithm’s correctness is violated
    - zombie or doomed transactions
  - other problems may be introduced due to inconsistencies
    - e.g. infinite loops, dereferencing null pointers, dividing by zero
Design of STM systems - Example of an indefinite loop

- Transactions $T_1$ and $T_2$ are concurrently accessing a linked list
  - $T_1$ searches the node containing number 4
  - $T_2$ executes a MoveToFront procedure

- Execution:
  - $T_1$ begins and visits node with number 2
  - $T_1$ visits node with number 1
  - $T_2$ begins, executes MoveToFront(8) and commits
  - $T_1$ visits node with number 8
  - Infinite Loop!

![Diagram showing initial state and after $T_2$'s MoveToFront snapshot]
Design of STM systems - Reading Shared Data

- 1st Solution:
  - acquire the ownership of some data item before reading it
  - visible reads are used

- 2nd Solution: (when visible reads are not used)
  - introducing the notion of **version** for each data item
    - in each write, the version of the data item is also updated
    - maintained to data item's metadata
  - each transaction,
    - maintains the version for each data item it reads
    - checks if it has changed
      - a **validation mechanism** is used
STM Algorithms

Several STM algorithms already exist

- non-blocking:
  - SSTM (Shavit & Touitou, PODC 1995)
  - DSTM (Herlihy et. al, PODC 2003)
  - NBSTM (Kosmas, MSc Thesis, Un. of Ioannina, 2008)
  - ASTM (Marathe et. al, DISC 2005)
  - NZSTM (Tabba et. al, SPAA 2009)
  - this list is not exhaustive

- blocking:
  - TL (Dice et. al, TRANSACT 2006)
  - TLII (Dice et. al, DISC 2006)
  - RingSTM (Spear et. al, SPAA 2008)
  - TinySTM (Felber et. al, PPoPP 2008)
  - Norec (Dalessandro et. al, PPoPP 2010)
  - this list is not exhaustive
**DSTM - Dynamic STM**

- **Supports**
  - *dynamic transactions*
  - the weakest non-blocking liveness property *obstruction-freedom*

- The representation of some data item $x$ that describes an object *obj*:

  - $\text{status} \in \{\text{Active, Committed, Aborted}\}$
  - **Read list** is used to implement validation
  - Tm-variables are *CAS registers*

---

Panagiota Fatourou  

faturu@ics.forth.gr  

HTDC’13 - La Plagne
DSTM

Example: A sorted linked list in DSTM

Transactions | data items | Locators | Memory
---|---|---|---
Transaction 1: Committed | start | oldData | 5
Transaction 1: readList | newData | tran | |
Transaction 2: Committed | start | oldData | 10
Transaction 2: readList | newData | tran | |
Transaction 2: | | | 15

Panagiota Fatourou
faturu@ics.forth.gr
HTDC’13 - La Plagne
**DSTM - Ownership acquisition**

- **Eager ownership acquisition**
- It is important how the current data of some data item are determined

- **When status is Active?**

---

Panagiota Fatourou  
faturu@ics.forth.gr  
HTDC’13 - La Plagne
Some transaction forcefully aborts another by changing its status field to Aborted
  - Obstruction-freedom

Ownership release: A transaction changes its status to either Committed or Aborted
  - execution of a single command simultaneously updates or restores the data of all data items
Ownership Acquisition

Transaction 1
Committed

Transaction 2
Committed

Transaction 3
Active

Creation of new element

Panagiota Fatourou
faturu@ics.forth.gr
HTDC'13 - La Plagne
DSTM - Validation mechanism

- Invisible reads

- **Version** of data item: The memory addresses in which the corresponding data item is stored

- **Read List**. For each data item read, is maintained:
  - its data item
  - its version

- **Validation Mechanism**
  - during execution of ReadTmVar operation all the elements of read list are checked
  - if **at least one of them** has been changed, the transaction is aborted
Thank you!

Questions?