Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- **abstraction**: an object storing certain data and providing certain functionality may be used without reference to its internals
- **composition**: several objects can be combined to a new object without interference

Both, abstraction and composition are closely related, since the ability to compose hinges on the ability to abstract from details.
Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- **abstraction**: an object storing certain data and providing certain functionality may be used without reference to its internals
- **composition**: several objects can be combined to a new object without interference

Both, **abstraction** and **composition** are closely related, since the ability to compose hinges on the ability to abstract from details.

Consider an example:

- A linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`
- A set object may internally use the list object and expose a set of operations, including `PushLeft`

The `Insert` operation uses the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

Wrapping the linked list in a mutex does not help to make the `set` thread-safe.

- Wrap the two calls in `Insert` in a mutex

---

Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- **abstraction**: an object storing certain data and providing certain functionality may be used without reference to its internals
- **composition**: several objects can be combined to a new object without interference

Both, **abstraction** and **composition** are closely related, since the ability to compose hinges on the ability to abstract from details.

Consider an example:

- A linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`
- A set object may internally use the list object and expose a set of operations, including `PushLeft`

The `Insert` operation uses the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

Wrapping the linked list in a mutex does not help to make the `set` thread-safe.

- Wrap the two calls in `Insert` in a mutex
- But other list operations can still be called – use the `same` mutex
Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- **abstraction**: an object storing certain data and providing certain functionality may be used without reference to its internals.
- **composition**: several objects can be combined to a new object without interference.

Both, abstraction and composition, are closely related, since the ability to compose hinges on the ability to abstract from details.

Consider an example:

- a linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`.
- a set object may internally use the list object and expose a set of operations, including `PushLeft`.

The `Insert` operations uses the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

Wrapping the linked list in a mutex does not help to make the `set` thread-safe.

- `~` wrap the two calls in `Insert` in a mutex.
- but other list operations can still be called `~` use the same mutex.

Unlike sequential algorithms, thread-safe algorithms cannot always be composed to give new thread-safe algorithms.

---

Transactional Memory [2]

Idea: automatically convert **atomic** blocks into code that ensures atomic execution of the statements.

```cpp
atomic {
  // code
  if (cond) retry;
  atomic {
    // more code
  }
  // code
}
```

Execute code as **transaction**:

- execute the code of an atomic block.

---

Transactional Memory [2]

Idea: automatically convert **atomic** blocks into code that ensures atomic execution of the statements.

```cpp
atomic {
  // code
  if (cond) retry;
  atomic {
    // more code
  }
  // code
}
```

Execute code as **transaction**:

- execute the code of an atomic block.
- nested atomic blocks act like a single atomic block.
- check that it runs without **conflicts** due to accesses from another thread.
Transactional Memory [2]

Idea: automatically convert atomic blocks into code that ensures atomic execution of the statements.

```java
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
    // code
}
```

Execute code as transaction:
- execute the code of an atomic block
- nested atomic blocks act like a single atomic block
- check that it runs without conflicts due to accesses from another thread
- if another thread interferes through conflicting updates:

---

Transactional Memory [2]

Idea: automatically convert atomic blocks into code that ensures atomic execution of the statements.

```java
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
    // code
}
```

Execute code as transaction:
- execute the code of an atomic block
- nested atomic blocks act like a single atomic block
- check that it runs without conflicts due to accesses from another thread
- if another thread interferes through conflicting updates:
  - undo the computation done so far
  - re-start the transaction

---

Managing Conflicts

**Definition (Conflicts)**

A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
Managing Conflicts

Definition (Conflicts)
A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- **optimistic vs. pessimistic concurrency control:**
  - pessimistic: conflict occurrence, detection, resolution occur at once
  - resolution here is usually delaying one transaction

Managing Conflicts

Definition (Conflicts)
A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- **optimistic vs. pessimistic concurrency control:**
  - pessimistic: conflict occurrence, detection, resolution occur at once
  - resolution here is usually delaying one transaction
  - can be implemented using locks: deadlock problem
Managing Conflicts

**Definition (Conflicts)**
A conflict *occurs* when accessing the same piece of data, a conflict is *detected* when the TM system observes this, it is *resolved* when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- **optimistic vs. pessimistic concurrency control:**
  - pessimistic: conflict *occurrence*, *detection*, *resolution* occur at once
    - resolution here is usually *delaying* one transaction
    - can be implemented using *locks*: deadlock problem
  - optimistic: detection and resolution can happen after a conflict occurs
    - resolution here must be *aborting* one transaction
    - need to repeated aborted transaction: livelock problem

---

Managing Conflicts

**Definition (Conflicts)**
A conflict *occurs* when accessing the same piece of data, a conflict is *detected* when the TM system observes this, it is *resolved* when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- **optimistic vs. pessimistic concurrency control:**
  - pessimistic: conflict *occurrence*, *detection*, *resolution* occur at once
    - resolution here is usually *delaying* one transaction
    - can be implemented using *locks*: deadlock problem
  - optimistic: detection and resolution can happen after a conflict occurs
    - resolution here must be *aborting* one transaction
    - need to repeated aborted transaction: livelock problem
  - **eager vs. lazy version management**: how read and written data are managed during the transaction
Managing Conflicts

Definition (Conflicts)
A conflict **occurs** when accessing the same piece of data, a conflict is **detected** when the TM system observes this, it is **resolved** when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- **optimistic vs. pessimistic concurrency control**:
  - **pessimistic**: conflict occurrence, detection, resolution occur at once
  - resolution here is usually **delaying** one transaction
  - can be implemented using **locks**: deadlock problem
  - **optimistic**: detection and resolution can happen after a conflict occurs
  - resolution here must be **aborting** one transaction
  - need to repeated aborted transaction: livelock problem
- **eager vs. lazy version management**: how read and written data are managed during the transaction
  - **eager**: writes modify the memory and an undo-log is necessary if the transaction aborts
  - **lazy**: writes are stored in a redo-log and modifications are done on committing

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:
- **granularity** of conflict detection: may be a cache-line or an object, **false conflicts** possible
Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- **conflict detection**: 
  - **eager**: conflicts are detected when memory locations are first accessed
  - **validation**: check occasionally that there is no conflict yet, always validate when committing
Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, \textit{false conflicts} possible
- **conflict detection:**
  - **eager:** conflicts are detected when memory locations are first accessed
  - **validation:** check occasionally that there is no conflict yet, always validate when committing
  - **lazy:** conflicts are detected when committing a transaction

Reference of conflict (for non-eager conflict detection)
Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- **conflict detection:*
  - **eager:** conflicts are detected when memory locations are first accessed
  - **validation:** check occasionally that there is no conflict yet, always validate when committing
  - **lazy:** conflicts are detected when committing a transaction
- **reference of conflict (for non-eager conflict detection)**
  - **tentative** detect conflicts before transactions commit, e.g. aborting when transaction TA reads while TB may writes the same location
  - **committed** detect conflicts only against transactions that have committed

Design choices for TM that allow conflicts to happen:

- **granularity** of conflict detection: may be a cache-line or an object, *false conflicts* possible
- **conflict detection:*
  - **eager:** conflicts are detected when memory locations are first accessed
  - **validation:** check occasionally that there is no conflict yet, always validate when committing
  - **lazy:** conflicts are detected when committing a transaction

Concurrent control

- [Optimistic]
  - Version management
    - Lazy undo
  - Eager undo
  - Conflict detection
    - Eager
    - Validating
    - Lazy
      - Reference of conflict
        - Tentative
        - Committed
The goal is to use transactions to specify atomic executions. Transactions are rooted in databases where they have the ACID properties:

atomicity: a transaction completes or seems not to have run

- we call this failure atomicity to distinguish it from atomic executions

consistency: each transaction transforms a consistent state to another consistent state
Semantics of Transactions

The goal is to use transactions to specify *atomic executions.* Transactions are rooted in databases where they have the ACID properties:

- **atomicity**: a transaction completes or seems not to have run
  - we call this *failure atomicity* to distinguish it from *atomic executions*
- **consistency**: each transaction transforms a consistent state to another consistent state
  - a consistent state is one in which certain *invariants* hold
- **isolation**: transactions do not influence each other
- **durability**: the effects are permanent ✓

---

Semantics of Transactions

The goal is to use transactions to specify *atomic executions.* Transactions are rooted in databases where they have the ACID properties:

- **atomicity**: a transaction completes or seems not to have run
  - we call this *failure atomicity* to distinguish it from *atomic executions*
- **consistency**: each transaction transforms a consistent state to another consistent state
  - a consistent state is one in which certain *invariants* hold
  - invariants depend on the application (e.g. queue data structure)
- **isolation**: transactions do not influence each other
  - not so evident with respect to non-transactional memory
- **durability**: the effects are permanent ✓

Transactions themselves must be *serializable.*
Semantics of Transactions

The goal is to use transactions to specify *atomic executions*. Transactions are rooted in databases where they have the ACID properties:

- **atomicity**: a transaction completes or seems not to have run
  - we call this *failure atomicity* to distinguish it from *atomic executions*
- **consistency**: each transaction transforms a consistent state to another consistent state
  - a consistent state is one in which certain *invariants* hold
  - invariants depend on the application (e.g. queue data structure)
- **isolation**: transactions do not influence each other
  - not so evident with respect to non-transactional memory
- **durability**: the effects are permanent ✓

Transactions themselves must be *serializable*:

- the result of running current transactions must be identical to *one* execution of them in sequence

Consistency During Transactions

**Consistency during a transaction.**

ACID states how committed transactions behave but not what may happen until a transaction commits.

- a transaction that is run on an inconsistent state may generate an inconsistent state ⇨ zombie transaction

```java
atomic {
    int tmp1 = x;  // preserved invariant: x=y
    int tmp2 = y;
    assert(tmp1-tmp2==0);
}
```

Consistency During Transactions

**Consistency during a transaction.**

ACID states how committed transactions behave but not what may happen until a transaction commits.

- a transaction that is run on an inconsistent state may generate an inconsistent state ⇨ zombie transaction
- this is usually ok since it will be aborted eventually
- but transactions may cause havoc when run on inconsistent states
Consistency During Transactions

Consistency during a transaction.

ACID states how committed transactions behave but not what may happen until a transaction commits.

- a transaction that is run on an inconsistent state may generate an inconsistent state → zombie transaction
- this is usually ok since it will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

```c
atomic {
    int tmp1 = x;
    int tmp2 = y;
    assert(tmp1-tmp2==0);
}
```

- critical for C/C++ if, for instance, variables are pointers

Definition (opacity)

A TM system provides opacity if failing transactions are serializable w.r.t. committing transactions.

→ failing transactions still sees a consistent view of memory

Weak- and Strong Isolation

If guarantees are only given about memory accessed inside atomic, a TM implementation provides weak isolation.

Can we mix transactions with code accessing memory non-transactionally?

- no conflict detection for non-transactional accesses
- standard race problems as in unlocked shared accesses

```c
atomic {
    x = 42;
}
```
Weak- and Strong Isolation

If guarantees are only given about memory accessed inside \texttt{atomic}, a TM implementation provides \textit{weak isolation}.

Can we mix transactions with code accessing memory non-transactionally?

\begin{itemize}
  \item no conflict detection for non-transactional accesses
  \item standard \textit{race} problems as in unlocked shared accesses
  \begin{verbatim}
  // Thread 1
  atomic {
    x = 42;
    int tmp = x;
  }
  \end{verbatim}
  \item give programs with \textit{races the same semantics as if using a single global lock for all atomic blocks}
\end{itemize}

// Thread 2
atomic {
  x = 42;
  int tmp = x;
}

\item give programs with races the same semantics as if using a single global lock for all \texttt{atomic} blocks

\item \textbf{strong isolation}: retain order between accesses to TM and non-TM

\begin{equation}
\begin{aligned}
\text{atomic} (k = i+j;)
\end{aligned}
\end{equation}

Observation:

\textbf{Definition (SLA)}

The \textit{single-lock atomicity} is a model in which the program executes as if all transactions acquire a single, program-wide mutual exclusion lock.
Properties of Single-Lock Atomicity

Observation:
- SLA enforces order between TM and non-TM accesses ✓
  - this guarantees strong isolation between TM and non-TM accesses
- within one transactions, accesses may be re-ordered ✓

Properties of Single-Lock Atomicity

Observation:
- SLA enforces order between TM and non-TM accesses ✓
  - this guarantees strong isolation between TM and non-TM accesses
- within one transactions, accesses may be re-ordered ✓
- the content of non-TM memory conveys information which atomic block has executed, even if the TM regions do not access the same memory
### Properties of Single-Lock Atomicity

![Diagram showing atomic operations and dependencies between different variables]

**Observation:**
- SLA enforces order between TM and non-TM accesses.
  - This guarantees *strong isolation* between TM and non-TM accesses.
- Within one transactions, accesses may be re-ordered.
- The content of non-TM memory conveys information which atomic block has executed, even if the TM regions do not access the same memory.
  - SLA makes it possible to use atomic block for synchronization.

### Disadvantages of the SLA model

The SLA model is *simple* but often too strong:

- **SLA has a weaker progress guarantee than a transaction should have**
  ```c
  // Thread 1
  atomic {
      while (true) {};
  }
  int tmp = x; // x in TM
  ```
  ```c
  // Thread 2
  atomic {
      while (true) {};
  }
  int tmp = x; // x in TM
  ```

- **SLA correctness is too strong in practice**
  ```c
  // Thread 1
  atomic {
      int tmp = data;
  }
  ```
  ```c
  // Thread 2
  atomic {
      int tmp = x; // x in TM
  }
  ```
  ```c
  // Thread 1
  atomic {
      if (ready) {
          // use tmp
      }
      ready = 1;
  }
  ```
  ```c
  // Thread 2
  atomic {
      if (ready) {
          // use tmp
      }
  }
  ```
  ```c
  // Thread 1
  data = 1;
  ```
  ```c
  // Thread 2
  atomic {
  }
  ```

- **under the SLA model, atomic {} acts as barrier**
Disadvantages of the SLA model

The SLA model is simple but often too strong:

- SLA has a weaker progress guarantee than a transaction should have

```java
// Thread 1
atomic {
    while (true) {};
}
// Thread 2
atomic {
    int tmp = x; // x in TM
}
```

- SLA correctness is too strong in practice

```java
// Thread 1
atomic {
    data = 1;
    int tmp = data;
    atomic {
        if (ready) {
            // use tmp
        }
    }
}
// Thread 2
atomic {
    ready = 1;
}
```

- Under the SLA model, `atomic {}` acts as barrier
- Intuitively, the two transactions should be independent rather than synchronize

~~ need a weaker model for more flexible implementation of strong isolation

Transaction Semantics

---

Transactional Sequential Consistency

How about a more permissive view of transaction semantics?

- TM should not have the blocking behaviour of locks
- The programmer cannot rely on synchronization

Definition (TSC)

The **transactional sequential consistency** is a model in which the accesses within each transaction are sequentially consistent.

---

Disadvantages of the SLA model

The SLA model is simple but often too strong:

- SLA has a weaker progress guarantee than a transaction should have

```java
// Thread 1
atomic {
    while (true) {};
    int tmp = x; // x in TM
}
// Thread 2
atomic {
    while (true) {};
    int tmp = x; // x in TM
}
```

- SLA correctness is too strong in practice

```java
// Thread 1
atomic {
    data = 1;
    int tmp = data;
    atomic {
        if (ready) {
            // use tmp
        }
    }
    ready = 1;
}
// Thread 2
atomic {
    if (ready) {
        // use tmp
    }
}
```

- Under the SLA model, `atomic {}` acts as barrier
- Intuitively, the two transactions should be independent rather than synchronize

~ need a weaker model for more flexible implementation of strong isolation

Transaction Semantics

---

Transactional Sequential Consistency

How about a more permissive view of transaction semantics?

- TM should not have the blocking behaviour of locks
- The programmer cannot rely on synchronization

Definition (TSC)

The **transactional sequential consistency** is a model in which the accesses within each transaction are sequentially consistent.

```java
A
    \[\text{atomic \{ \text{k = i+j; \}}\]\nB
    \[\text{atomic \{ \text{k = i+j; \}}\]\n```

- TSC is weaker: gives **strong isolation**, but allows parallel execution
- TSC is stronger: accesses within a transaction may **not** be re-ordered

---

Concurrency: Transactions

Transaction Semantics
Quick Quiz

Associate one item on the left with one or two on the right.

- a transaction waits rather than creating a conflict
- in case of a conflict, a kind of log is needed
- no opacity: a zombie transaction sees an inconsistent state
- no guarantee if a program accesses variables via TM and non-TM
- a write in a transaction is immediately globally visible

- redo and undo
- conflict detection
- concurrency control
- isolation
- version management
- eager
- lazy
- optimistic, pessimistic
- strong, weak

Translation of atomic-Blocks

A TM system must track which shared memory locations are accessed:

- convert every read access $x$ from a shared variable to $\text{ReadTx}(\&x)$
- convert every write access $x\leftarrow e$ to a shared variable to $\text{WriteTx}(\&x, e)$

Translation of atomic-Blocks

A TM system must track which shared memory locations are accessed:

- convert every read access $x$ from a shared variable to $\text{ReadTx}(\&x)$
- convert every write access $x\leftarrow e$ to a shared variable to $\text{WriteTx}(\&x, e)$

Convert atomic blocks as follows:

$$\text{atomic } \{ \text{// code} \} \quad \Rightarrow \quad \{ \text{\begin{align*}&\text{do } \{ \\
&\text{StartTx();} \\
&\text{// code with ReadTx and WriteTx} \\
&\text{\} while (!\text{CommitTx();})}
\end{align*}} }\}$$
Translation of atomic-Blocks

A TM system must track which shared memory locations are accessed:
- convert every read access $x$ from a shared variable to $\text{ReadTx}(\langle x \rangle)$
- convert every write access $x=e$ to a shared variable to $\text{WriteTx}(\langle x, e \rangle)$

Convert atomic blocks as follows:

```c
atomic {
    // code
    do {
        StartTx();
        // code with ReadTx and WriteTx
    } while (!CommitTx());
}
```

- translation can be done using a pre-processor
  - determining a minimal set of memory accesses that need to be transactional requires a good static analysis
  - idea: translate all accesses to global variables and the heap as TM
  - more fine-grained control using manual translation
- an actual implementation might provide a `retry` keyword
  - when executing `retry`, the transaction aborts and re-starts
  - the transaction will again wind up at `retry` unless its `read set` changes
  - block until a variable in the read-set has changed
- similar to condition variables in monitors

Transactional Memory for the Queue

If a preprocessor is used, `PopRight` can be implemented as follows:

```c
int PopRight(DQueue* q) {
    QNode* oldRightNode;
    QNode* rightSentinel = q->right;
    atomic {
        oldRightNode = rightSentinel->left;
        if (oldRightNode == leftSentinel) retry;
        QNode* newRightNode = oldRightNode->left;
        newRightNode->right = rightSentinel;
        rightSentinel->left = newRightNode;
    }
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

- the transaction will abort if other threads call `PopRight`
- if the queue is empty, it may abort if `PushLeft` is executed

A Software TM Implementation

A software TM implementation allocates a `transaction descriptor` to store data specific to each atomic block, for instance:

- `undo-log` of writes if writes have to be undone if a commit fails
- `redo-log` of writes if writes are postponed until a commit
- `read- and write-set`: locations accessed so far
- `read- and write-version`: time stamp when value was accessed

Consider the TL2 STM (software transactional memory) algorithm [1]:
A Software TM Implementation

A software TM implementation allocates a transaction descriptor to store data specific to each atomic block, for instance:

- **undo-log** of writes if writes have to be undone if a commit fails
- **redo-log** of writes if writes are postponed until a commit
- **read-** and **write-set**: locations accessed so far
- **read-** and **write-version**: time stamp when value was accessed

Consider the TL2 STM (software transactional memory) algorithm [1]:

- provides **opacity**: zombie transactions do not see inconsistent state
- uses **lazy versioning**: writes are stored in a redo-log and done on commit

A Software TM Implementation

A software TM implementation allocates a transaction descriptor to store data specific to each atomic block, for instance:

- **undo-log** of writes if writes have to be undone if a commit fails
- **redo-log** of writes if writes are postponed until a commit
- **read-** and **write-set**: locations accessed so far
- **read-** and **write-version**: time stamp when value was accessed

Consider the TL2 STM (software transactional memory) algorithm [1]:

- provides **opacity**: zombie transactions do not see inconsistent state
- uses **lazy versioning**: writes are stored in a redo-log and done on commit
Principles of TL2

The idea: obtain a version \( tx.RV \) from the global clock when starting the transaction, the \( read-version \), and set the versions of all written cells to a new version on commit. A read from a field at offset of object \( obj \) is implemented as follows:

```c
transactional read

int ReadTx(TMDesc tx, object obj, int offset) {
    if (&(obj[offset]) in tx.redoLog) {
        return tx.redoLog[&obj[offset]];
    } else {
        atomic { v1 = obj.timestamp; locked = obj.sem<1; }
        result = obj[offset];
        v2 = obj.timestamp;
        if (locked || v1 != v2 || v1 > tx.RV) AbortTx(tx);
        tx.readSet = tx.readSet.add(obj);
        return result;
    }
}
```

WriteTx is simpler: add or update the location in the redo-log.

Committing a Transaction

A transaction can succeed if none of the read locations has changed:

```c
bool CommitTx(TMDesc tx) {
    foreach (e in tx.writeSet)
        if (!try_wait(e.obj.sem)) goto Fail;
    WV = FetchAndAdd(&globalClock);
    foreach (e in tx.readSet)
        if (e.obj.version > tx.RV) goto Fail;
    foreach (e in tx.redoLog)
        e.obj[e.offset] = e.value;
    foreach (e in tx.writeSet) {
        e.obj += WV; signal(e.obj.sem);
    }
    return true;
    Fail:
        // signal all acquired semaphores
        return false;
}
```

Properties of TL2

Opacity is guaranteed by aborting a read access with an inconsistent value:

- StartTx
- ReadTx
- WriteTx
- ReadTx
- CommitTx

- memory state seems to be consistent
- validate redo-log
- increment global clock

Other observations:
Properties of TL2

Opacity is guaranteed by aborting a read access with an inconsistent value:

`StartTx  ReadTx  WriteTx  ReadTx  CommitTx`

```
memory state seems to be consistent
validate read set
write redo-log
increment global clock
```

Other observations:
- read-only transactions just need to check that read versions are consistent (no need to increment the global clock)
- writing values still requires locks
  - deadlocks are still possible
  - since other transactions can be aborted, one can preempt transactions that are deadlocked

---

Properties of TL2

Opacity is guaranteed by aborting a read access with an inconsistent value:

`StartTx  ReadTx  WriteTx  ReadTx  CommitTx`

```
memory state seems to be consistent
validate read set
write redo-log
increment global clock
```

Other observations:
- read-only transactions just need to check that read versions are consistent (no need to increment the global clock)
- writing values still requires locks
  - deadlocks are still possible
  - since other transactions can be aborted, one can preempt transactions that are deadlocked
  - since lock accesses are generated, computing a lock order up-front might be possible
Properties of TL2

Opacity is guaranteed by aborting a read access with an inconsistent value:

StartTx  ReadTx  WriteTx  ReadTx  CommitTx

memory state seems to be consistent
validate read set
increment global clock
write redo-log

Other observations:
- read-only transactions just need to check that read versions are consistent (no need to increment the global clock)
- writing values still requires locks
  - deadlocks are still possible
  - since other transactions can be aborted, one can preempt transactions that are deadlocked
  - since lock accesses are generated, computing a lock order up-front might be possible
- at least two memory barriers are necessary in ReadTx

General Challenges when using TM

Executing atomic blocks by repeatedly trying to executing them non-atomically creates new problems:
- a transaction might unnecessarily be aborted