Script generated by TTT

Title: Petter: Programmiersprachen (05.11.2014)

Date: Wed Nov 05 14:16:18 CET 2014

Duration: 89:20 min

Pages: 94

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Transactions

Dr. Axel Simon and Dr. Michael Petter
Winter term 2014

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.

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.

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.

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

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

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.

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

Execute code as *transaction*:

Transactional Memory [2]

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

```
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.

```
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.

```
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.

```
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

Transactional Memory [2]

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

```
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
- provide a `retry` keyword similar to the `wait` of monitors

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*:

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

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

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

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

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:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible

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:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *conflict detection*:

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *conflict detection*:
   - *eager*: conflicts are detected when memory locations are first accessed

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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
3. reference of conflict (for non-*eager conflict* detection)

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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
3. 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

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible
2. *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

Choices for Optimistic Concurrency Control

Design choices for TM that allow conflicts to happen:

1. *granularity* of conflict detection: may be a cache-line or an object, *false conflicts* possible

Concurrency control
- pessimistic
- optimistic
  - version management
    - lazy — redo
    - eager — undo
  - Conflict detection
    - eager
    - validating
    - lazy
      - reference of conflict
        - tentative
        - committed

Semantics of Transactions

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

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

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*

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

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

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

Concurrency: Transactions Transaction Semantics 6 / 34

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 ✓

Concurrency: Transactions Transaction Semantics 6 / 34

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*:

Concurrency: Transactions Transaction Semantics 6 / 34

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

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
- serializability for transactions is insufficient to perform synchronization between threads

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

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

```
atomic {                              // preserved invariant: x==y
  int tmp1 = x;                       atomic {
  int tmp2 = y;                         x = 10;
  assert(tmp1-tmp2==0);                 y = 10;
}                                     }
```

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

```
atomic {                          // preserved invariant: x==y
  int tmp1 = x;                   atomic {
  int tmp2 = y;                     x = 10;
  assert(tmp1-tmp2==0);             y = 10;
}                                 }
```

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

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

```
atomic {                          // preserved invariant: x==y
  int tmp1 = x;                   atomic {
  int tmp2 = y;                     x = 10;
  assert(tmp1-tmp2==0);             y = 10;
}                                 }
```

- 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?

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

```
// Thread 1
atomic {                          // Thread 2
  x = 42;                         int tmp = x;
}
```

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

```
// Thread 1
atomic {                    // Thread 2
  x = 42;                   int tmp = x;
}
```

- ⇝ give programs with races the same semantics as if using a single global lock for all `atomic` blocks

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

```
// Thread 1
atomic {                    // Thread 2
  x = 42;                   int tmp = x;
}
```

- ⇝ give programs with races the same semantics as if using a single global lock for all `atomic` blocks
- *strong isolation*: retain order between accesses to TM and non-TM

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

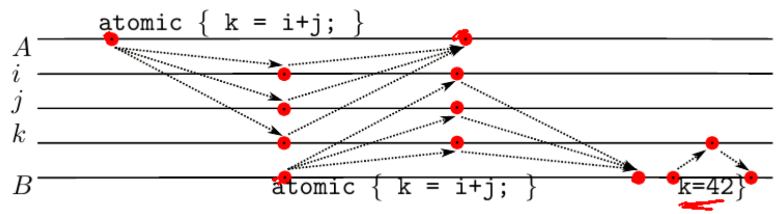
```
// Thread 1
atomic {                    // Thread 2
  x = 42;                   int tmp = x;
}
```

- ⇝ give programs with races the same semantics as if using a single global lock for all `atomic` blocks
- *strong isolation*: retain order between accesses to TM and non-TM

Definition (SLA)

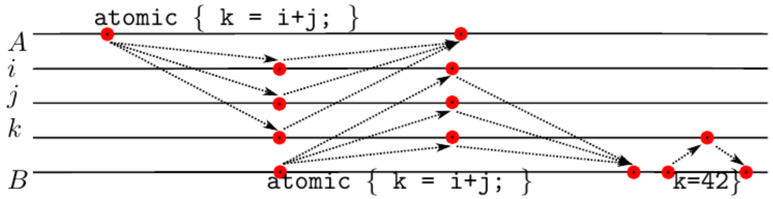
The *single-lock atomicity* is a model in which the program executes as if all transactions acquire a single, program-wide mutual exclusion lock.

Properties of Single-Lock Atomicity



Observation:

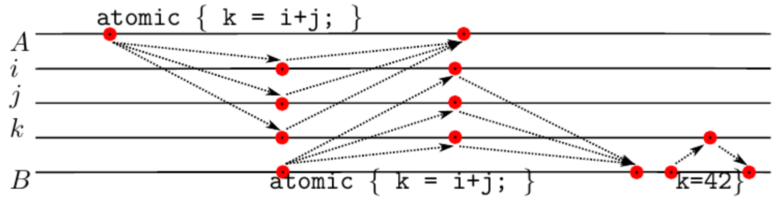
Properties of Single-Lock Atomicity



Observation:

- SLA enforces order between TM and non-TM accesses ✓

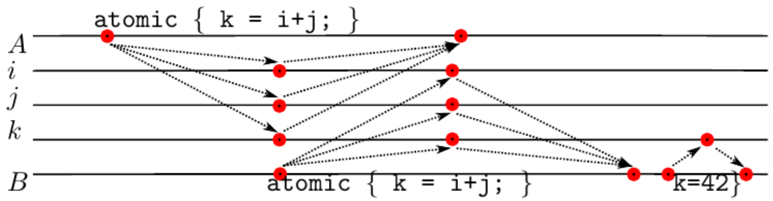
Properties of Single-Lock Atomicity



Observation:

- SLA enforces order between TM and non-TM accesses ✓
  - this guarantees *strong isolation* between TM and non-TM accesses

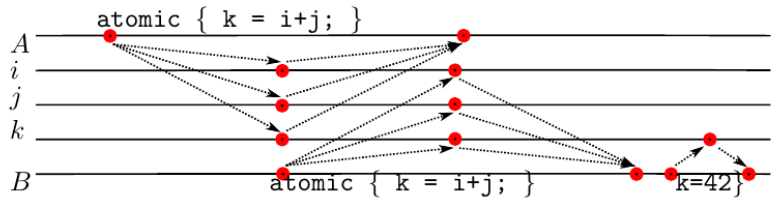
Properties of Single-Lock Atomicity



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 ✓

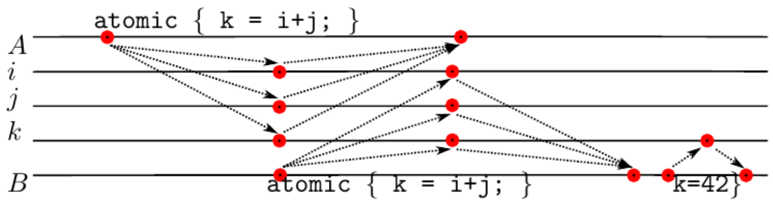
Properties of Single-Lock Atomicity



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



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:

1. SLA has a weaker *progress* guarantee than a transaction should have

```
// Thread 1
atomic {
  while (true) {};
}
```

```
// Thread 2
atomic {
  int tmp = x; // x in TM
}
```


Disadvantages of the SLA model

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

1. SLA has a weaker *progress* guarantee than a transaction should have

```
// Thread 1
atomic {
  while (true) {};
}
```

```
// Thread 2
atomic {
  int tmp = x; // x in TM
}
```

2. SLA correctness is too strong in practice

```
// Thread 1
data = 1;
atomic {
}
ready = 1;
```

```
// Thread 2
atomic {
  int tmp = data;
  // Thread 1 not in atomic
  if (ready) {
    // use tmp
  }
}
```


Disadvantages of the SLA model

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

1. SLA has a weaker *progress* guarantee than a transaction should have

```
// Thread 1
atomic {
  while (true) {};
}
```

```
// Thread 2
atomic {
  int tmp = x; // x in TM
}
```

2. SLA correctness is too strong in practice

```
// Thread 1
data = 1;
atomic {
}
ready = 1;
```

```
// Thread 2
atomic {
  int tmp = data;
  // Thread 1 not in atomic
  if (ready) {
    // use tmp
  }
}
```

- ▸ under the SLA model, `atomic {}` acts as barrier

Disadvantages of the SLA model

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

1. SLA has a weaker *progress* guarantee than a transaction should have

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

2. SLA correctness is too strong in practice

```
                               // Thread 2
                               atomic {
// Thread 1                      int tmp = data;
data = 1;                        // Thread 1 not in atomic
atomic {                         if (ready) {
}                                  // use tmp
ready = 1;                       }
                               }
```

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

Disadvantages of the SLA model

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

1. SLA has a weaker *progress* guarantee than a transaction should have

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

2. SLA correctness is too strong in practice

```
                               // Thread 2
                               atomic {
// Thread 1                      int tmp = data;
data = 1;                        // Thread 1 not in atomic
atomic {                         if (ready) {
}                                  // use tmp
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*

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.

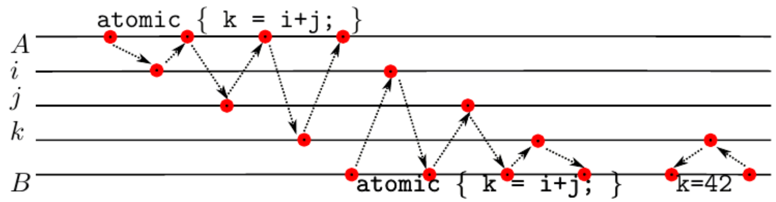
Transactional Sequential Consistency

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.



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

Quick Quiz

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

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

- redo and undo 2
- conflict detection 3
- concurrency control 1
- isolation 4
- version management 5

- eager, 5
  lazy 3
- optimistic,
  pessimistic 1
- strong,
  weak 4

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 `ReadTx(&x)`

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 `ReadTx(&x)`
- convert every write access `x=e` to a shared variable to `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 `ReadTx(&x)`
- convert every write access `x=e` to a shared variable to `WriteTx(&x,e)`

Convert `atomic` blocks as follows:

```
atomic {
  // code
}
```

$\Longrightarrow$

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

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 `ReadTx(&x)`
- convert every write access `x=e` to a shared variable to `WriteTx(&x,e)`

Convert `atomic` blocks as follows:

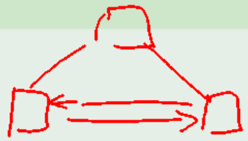
```
atomic {                    do {
  // code         ⟹          StartTx();
}                             // code with ReadTx and WriteTx
                            } while (!CommitTx());
```

- 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:

double-ended queue: removal



```
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`

Transactional Memory for the Queue

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

double-ended queue: removal

```
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

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
- *validating conflict detection*: accessing a modified address aborts

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
- *validating conflict detection*: accessing a modified address aborts

TL2 stores a *global version* counter and:

- a read version in each *object* (allocate a few bytes more in each call to `malloc`, or inherit from a *transaction object* in e.g. Java)
- a redo-log in the transaction descriptor
- a read- and a write-set in the transaction descriptor
- a read-version: the version when the transaction started

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:

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;
}
```


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:

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

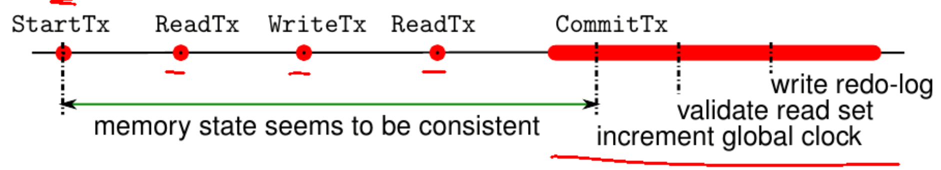
A transaction can succeed if none of the read locations has changed:

committing a transaction

```
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

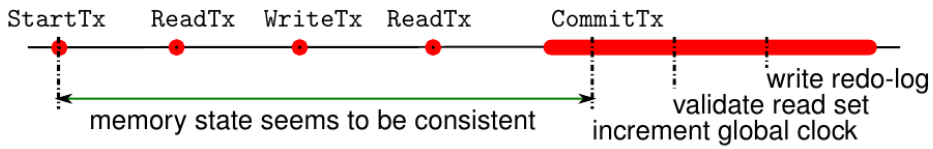
Opacity is guaranteed by aborting a read access with an inconsistent value:



Other observations:

Properties of TL2

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

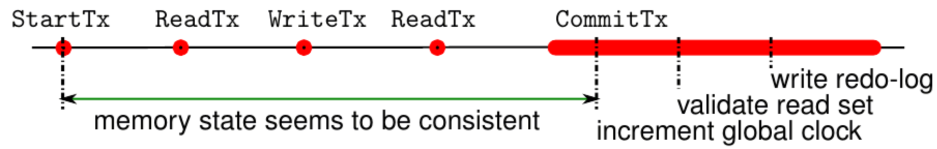


Other observations:

- read-only transactions just need to check that read versions are consistent (no need to increment the global clock)
- writing values still requires locks

Properties of TL2

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

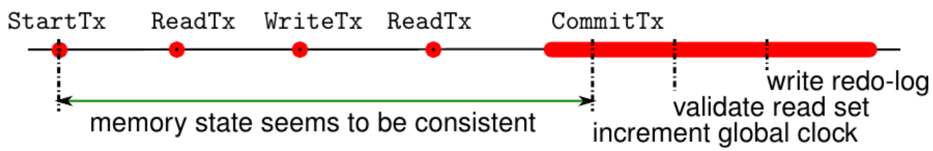


Other observations:

- 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

Properties of TL2

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

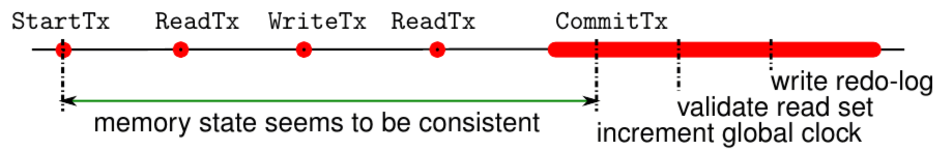


Other observations:

- 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:

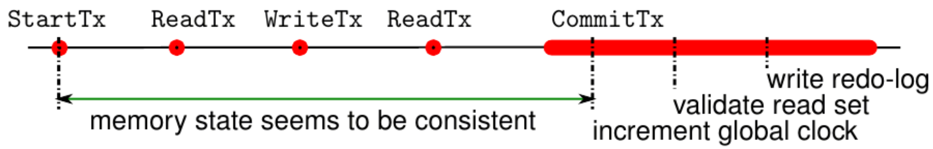


Other observations:

- 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:

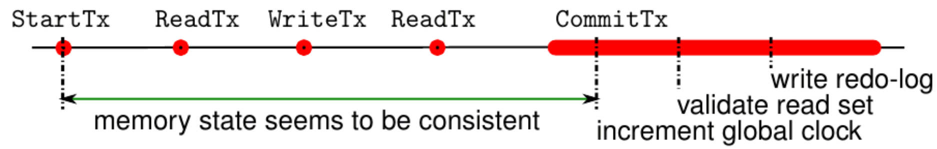


Other observations:

- 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`

Properties of TL2

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



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`
  - read version+lock, `lfence`, read value, `lfence`, read version

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