Script generated by TTT

Title: Simon: Programmiersprachen (23.11.2012)

Date: Fri Nov 23 11:05:06 CET 2012

Duration: 94:06 min

Pages: 113

Deadlocks with Monitors

Definition (Deadlock)

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Consider this Java class:

```
class Foo {
  public Foo other = null;
  public synchronized void bar() {
    ... if (*) other.bar(); ...
  }
}
```

and two instances:

```
Foo a = new Foo();
Foo b = new Foo();
a.other = b; b.other = a;
// in parallel:
a.bar() || b.bar();
```

Sequence leading to a deadlock:

- threads A and B execute `a.bar()` and `b.bar()`
- `a.bar()` acquires the monitor of `a`
- `b.bar()` acquires the monitor of `b`
- A happens to execute `other.bar()`
- A blocks on the monitor of b
- B happens to execute `other.bar()`
- $\rightsquigarrow$ both *block* indefinitely

How can this situation be avoided?

Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:

1. *mutual exclusion*: processes require exclusive access
2. *wait for*: a process holds resources while waiting for more
3. *no preemption*: resources cannot be taken away form processes
4. *circular wait*: waiting processes form a cycle

The occurrence of deadlocks can be:

Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:

1. *mutual exclusion*: processes require exclusive access
2. *wait for*: a process holds resources while waiting for more
3. *no preemption*: resources cannot be taken away form processes
4. *circular wait*: waiting processes form a cycle

The occurrence of deadlocks can be:

1. *ignored*: for the lack of better approaches, can be reasonable if deadlocks are rare
2. *detection*: check within OS for a cycle, requires ability to *preempt*

Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:

1. *mutual exclusion*: processes require exclusive access
2. *wait for*: a process holds resources while waiting for more
3. *no preemption*: resources cannot be taken away form processes
4. *circular wait*: waiting processes form a cycle

The occurrence of deadlocks can be:

1. *ignored*: for the lack of better approaches, can be reasonable if deadlocks are rare
2. *detection*: check within OS for a cycle, requires ability to *preempt*
3. *prevention*: design programs to be deadlock-free

Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:

1. *mutual exclusion*: processes require exclusive access
2. *wait for*: a process holds resources while waiting for more
3. *no preemption*: resources cannot be taken away form processes
4. *circular wait*: waiting processes form a cycle

The occurrence of deadlocks can be:

1. *ignored*: for the lack of better approaches, can be reasonable if deadlocks are rare
2. *detection*: check within OS for a cycle, requires ability to *preempt*
3. *prevention*: design programs to be deadlock-free
4. *avoidance*: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [1]:

1. *mutual exclusion*: processes require exclusive access
2. *wait for*: a process holds resources while waiting for more
3. *no preemption*: resources cannot be taken away form processes
4. *circular wait*: waiting processes form a cycle

The occurrence of deadlocks can be:

1. *ignored*: for the lack of better approaches, can be reasonable if deadlocks are rare
2. *detection*: check within OS for a cycle, requires ability to *preempt*
3. *prevention*: design programs to be deadlock-free
4. *avoidance*: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

$\rightsquigarrow$ *prevention* is the only safe approach on standard operating systems

- can be achieve using *lock-free* algorithms
- but what about algorithms that require locking?

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, that is, the set of locks that may be in the "acquired" state at program point p.

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, that is, the set of locks that may be in the "acquired" state at program point p.

We require the transitive closure σ^+ of a relation σ:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

$$\begin{aligned} \sigma^0 &= \sigma \\ \sigma^{i+1} &= \sigma^i \cup \{\langle x_1, x_3 \rangle \mid \exists x_2 \in X \,.\, \langle x_1, x_2 \rangle \in \sigma^i \wedge \langle x_2, x_3 \rangle \in \sigma^i\} \end{aligned}$$

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, that is, the set of locks that may be in the "acquired" state at program point p.

We require the transitive closure σ^+ of a relation σ:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

$$\begin{aligned} \sigma^0 &= \sigma \\ \sigma^{i+1} &= \sigma^i \cup \{\langle x_1, x_3 \rangle \mid \exists x_2 \in X \,.\, \langle x_1, x_2 \rangle \in \sigma^i \wedge \langle x_2, x_3 \rangle \in \sigma^i\} \end{aligned}$$

Each time a lock is acquired, we track the lock set at p:

Definition (lock order)

Define $\lhd \subseteq L \times L$ such that $l \lhd l'$ iff $l \in \lambda(p)$ and the statement at p is of the form `wait(l')` or `monitor_enter(l')`. Define the strict lock order $\prec = \lhd^+$.

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, that is, the set of locks that may be in the "acquired" state at program point p.

We require the transitive closure σ^+ of a relation σ:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

$$\begin{aligned} \sigma^0 &= \sigma \\ \sigma^{i+1} &= \sigma^i \cup \{\langle x_1, x_3 \rangle \mid \exists x_2 \in X \,.\, \langle x_1, x_2 \rangle \in \sigma^i \wedge \langle x_2, x_3 \rangle \in \sigma^i\} \end{aligned}$$

Each time a lock is acquired, we track the lock set at p:

Definition (lock order)

Define $\lhd \subseteq L \times L$ such that $l \lhd l'$ iff $l \in \lambda(p)$ and the statement at p is of the form `wait(l')` or `monitor_enter(l')`. Define the strict lock order $\prec = \lhd^+$.

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.

Suppose a program blocks on semaphores (mutexes) at L_S and on monitors at L_M such that $L = L_S \cup L_M$.

Theorem (freedom of deadlock for monitors)

If $\nexists a \in L_S \,.\, a \prec a$ and $\nexists a \in L_M, b \in L \,.\, a \neq b \wedge a \prec b \wedge b \prec a$ then the program is free of deadlocks.

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.

Suppose a program blocks on semaphores (mutexes) at L_S and on monitors at L_M such that $L = L_S \cup L_M$.

Theorem (freedom of deadlock for monitors)

If $\nexists a \in L_S \,.\, a \prec a$ and $\nexists a \in L_M, b \in L \,.\, a \neq b \wedge a \prec b \wedge b \prec a$ then the program is free of deadlocks.

Note: the set L contains *instances* of a lock.

- the set of lock instances can vary at runtime
- if we statically want to ensure that deadlocks cannot occur:
  - summarize every monitor that may have several instances into one
  - a summary lock $\bar{a} \in L_M$ represents several concrete ones
  - thus, if $\bar{a} \prec \bar{a}$ then this might not be a self-cycle
  - $\rightsquigarrow$ require that $\bar{a} \not\prec \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^n = L_M \setminus L_M^s$
- sort locks into ascending order according to lock sets
- modify code that locks are only acquired in strictly ascending order

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^n = L_M \setminus L_M^s$
- sort locks into ascending order according to lock sets
- modify code that locks are only acquired in strictly ascending order

⚠ Ordering might be hard or impossible to find:

- determining which locks may be acquired at each program point is undecidable ⇝ approximate lock set
- an array of locks: lock in increasing array index sequence
- if $l\lambda(P)$ exists where $l' \prec l$ should be locked: release l, acquire l', then acquire l again ⇝ inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^n = L_M \setminus L_M^s$
- sort locks into ascending order according to lock sets
- modify code that locks are only acquired in strictly ascending order

⚠ Ordering might be hard or impossible to find:

- determining which locks may be acquired at each program point is undecidable ⇝ approximate lock set
- an array of locks: lock in increasing array index sequence
- if $l\lambda(P)$ exists where $l' \prec l$ should be locked: release l, acquire l', then acquire l again ⇝ inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

an example for the latter is the `Foo` class: two instances of the same class call each other

Refining the Queue: Concurrent Access

Add a second lock `s->t` to allow concurrent removal:

double-ended queue: removal

```
int PopRight(DQueue* q, int val) {
  QNode* oldRightNode;
  wait(q->t); // wait to enter the critical section
  QNode* rightSentinel = q->right;
  oldRightNode = rightSentinel->left;
  if (oldRightNode==leftSentinel) { signal(q->t); return -1; }
  QNode* newRightNode = oldRightNode->left;
  if (newRightNode==leftSentinel) wait(q->s);
  newRightNode->right = rightSentinel;
  rightSentingel->left = newRightNode;
  if (newRightNode==leftSentinel) signal(q->s);
  signal(q->t); // signal that we're done
  int val = oldRightNode->val;
  free(oldRightNode);
  return val;
}
```


Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

double-ended queue: removal

```
void PopRight() {
  ...
  wait(q->t);
  ...
  if (*) { signal(q->t); return; }
  ...
  if (c) wait(q->s);
  ...
  if (c) signal(q->s);
  signal(q->t);
}
```

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

double-ended queue: removal

```
void PopRight() {
  ...
  wait(q->t);
  ...
  if (*) { signal(q->t); return; }
  ...
  if (c) wait(q->s);
  ...
  if (c) signal(q->s);
  signal(q->t);
}
```

- in `PushLeft`, the lock set for s is empty
- here, the lock set of s is $\{t\}$
- $t \lhd s$ and transitive closure is $t \prec s$
- $\rightsquigarrow$ the program cannot deadlock

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

double-ended queue: removal

```
void PopRight() {
  ...
  wait(q->t);
  ...
  if (*) { signal(q->t); return; }
  ...
  if (c) wait(q->s);
  ...
  if (c) signal(q->s);
  signal(q->t);
}
```

- nested `atomic` blocks still describe one atomic execution
- $\rightsquigarrow$ locks convey additional information over `atomic`
- locks cannot easily be recovered from `atomic` declarations

Outlook

Writing `atomic` annotations around sequences of statements is a convenient way of programming.

Outlook

Writing `atomic` annotations around sequences of statements is a convenient way of programming.

Idea: Replace `atomic` sections with locks:

- a single lock could be use to protect all `atomic` blocks
- more concurrency is possible by using several locks
  - compare the `PushLeft`, `PopRight` example
- some statements might modify variables that are never read by other threads $\rightsquigarrow$ no lock required
- statements in one `atomic` block might access variables in a different oder to another `atomic` block $\rightsquigarrow$ deadlock prevention when creating locks
- creating too many lock can decrease the performance, especially when required to release locks in $\lambda(l)$ when acquiring l

References

- E. G. Coffman, M. Elphick, and A. Shoshani.
  System deadlocks.
  ACM Comput. Surv., 3(2):67–78, June 1971.
- Tim Harris, James Larus, and Ravi Rajwar.
  Transactional memory, 2nd edition.
  Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.

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
- 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

Concurrency: Transactions Motivation 2 / 26

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
- check that it runs without *conflicts* due to accesses from another thread

Concurrency: Transactions Motivation 3 / 26

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
- 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
- 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

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
- 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*:
  - *pressimistic*: 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*:
  - *pressimistic*: 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*:
  - *pressimistic*: 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*:
  - *pressimistic*: 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*:
  - *pressimistic*: 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

Concurrency: Transactions | Transaction Semantics | 4 / 26

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*:
  - *pressimistic*: 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

Concurrency: Transactions | Transaction Semantics | 4 / 26

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*:
  - *pressimistic*: 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

Concurrency: Transactions | Transaction Semantics | 4 / 26

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*:
   - *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
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

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

Semantics of Transactions

atomic blocks

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
- not so evident with respect to non-transactional memory

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

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 doomed

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

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

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


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

- $\rightsquigarrow$ 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                    // Thread 2
atomic {                       int tmp = x;
  x = 42;
}
```

- $\rightsquigarrow$ 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                    // Thread 2
atomic {                       int tmp = x;
  x = 42;
}
```

- $\rightsquigarrow$ 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 singel, program-wide mutual exclusion lock.

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

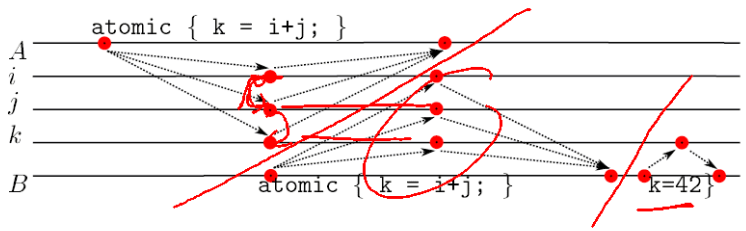
- $\rightsquigarrow$ give programs with races the same semantics as if using a single global lock for all `atomic` blocks
- *strong isolation*: retain order between accesses to TM and non-TM

Definition (SLA)

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

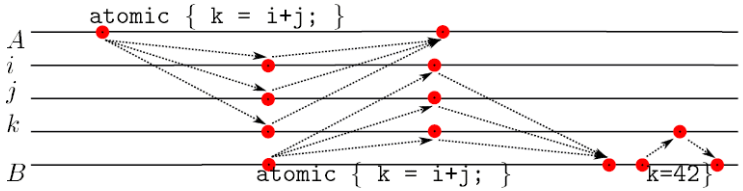
$\rightsquigarrow$ like *sequential consistency*, SLA is a statement about program equivalence

Properties of Single-Lock Atomicity



Observation:

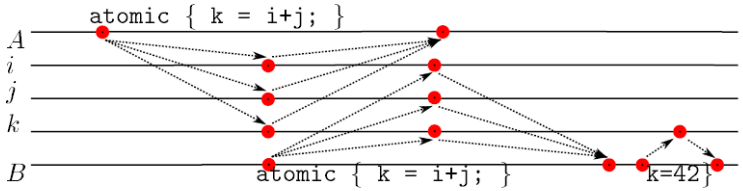
Properties of Single-Lock Atomicity



Observation:

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

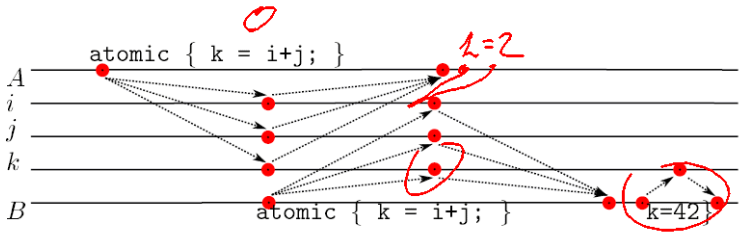
Properties of Single-Lock Atomicity



Observation:

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

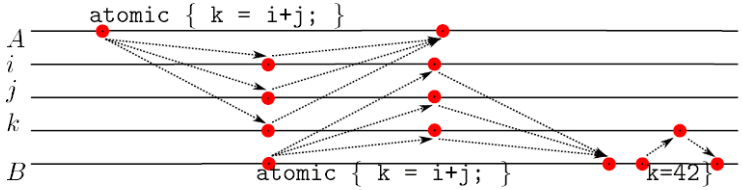
Properties of Single-Lock Atomicity



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                    // Thread 2
atomic {                       atomic {
  while (true) {};               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                    // Thread 2
atomic {                       atomic {
  while (true) {};               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                    // Thread 2
atomic {                       atomic {
  while (true) {};               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
- intuitively, the two transactions should be independent rather than synchronize

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.

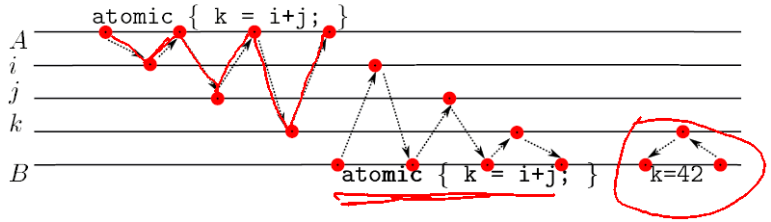
Transactional Sequential Consistency

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 ⚠

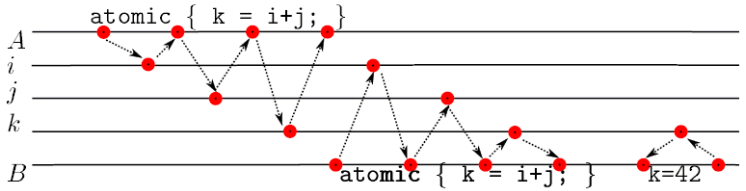
Transactional Sequential Consistency

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 ⚠

⇝ actual implementations use TSC with some *race free* re-orderings

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. a zombie transaction sees an inconsistent state
4. no guarantee if a transaction accesses non TM
5. 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 `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)`

Convert `atomic` blocks as follows:

```
atomic {                    ⟹    do {
  // code                           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, int val) {
  QNode* oldRightNode;
  atomic {
    QNode* rightSentinel = q->right;
    oldRightNode = rightSentinel->left;
    if (oldRightNode==leftSentinel) retry;
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentingel->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, int val) {
  QNode* oldRightNode;
  atomic {
    QNode* rightSentinel = q->right;
    oldRightNode = rightSentinel->left;
    if (oldRightNode==leftSentinel) retry;
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentingel->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 (lines 8,9)

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 TM2 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 TM2 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
- *eager conflict detection*: a transaction aborts as soon as it conflicts

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

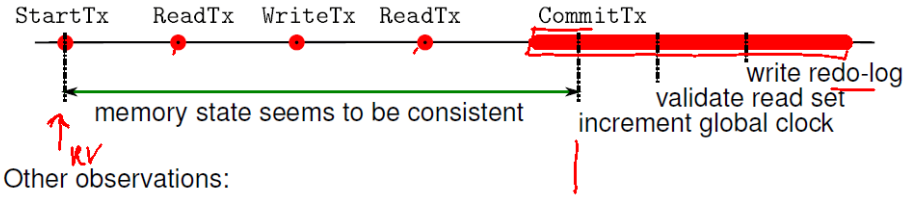
A transaction can succeed if none of the read locations has changed:

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

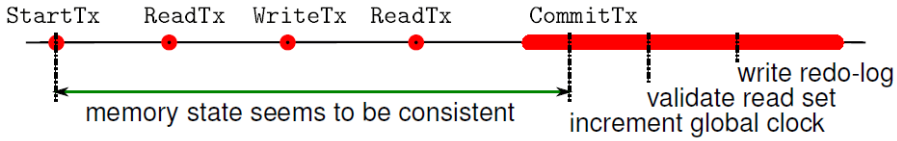
Opacity is guaranteed by aborting a read access with an inconsistent value:



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)

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid
  - if a cache-line in the read set is invalidated, a transaction becomes invalid

⇝ due to limited size, a STM backup must be provided

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid
  - if a cache-line in the read set is invalidated, a transaction becomes invalid

⇝ due to limited size, a STM backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid
  - if a cache-line in the read set is invalidated, a transaction becomes invalid

⇝ due to limited size, a STM backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - track an extra bit with each cache-line that is set if the transaction became invalid, return this bit after each access
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid
  - if a cache-line in the read set is invalidated, a transaction becomes invalid

⇝ due to limited size, a STM backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - track an extra bit with each cache-line that is set if the transaction became invalid, return this bit after each access
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked
   - provide a target to jump to when transaction aborts

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - if a cache-line in the write set is evicted, a transaction becomes invalid
  - if a cache-line in the read set is invalidated, a transaction becomes invalid

⇝ due to limited size, a STM backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional HTM: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - track an extra bit with each cache-line that is set if the transaction became invalid, return this bit after each access
2. Implicit Transactional HTM: only the beginning and end of a transaction are marked
   - provide a target to jump to when transaction aborts
   - track a read- and write-set per core and check these against invalidations
   - performing any instruction affecting the CPU aborts a transaction (examples: OS calls, interrupts, IO)

Concurrency: Transactions | Implementation of Software TM | 20 / 26

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): *implicit transactional*

1. Hardware Lock Elision

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): *implicit transactional*

1. Hardware Lock Elision
   - provides a way to execute a critical section without the atomic updates necessary to acquire and release the lock

Concurrency: Transactions | Implementation of Software TM | 21 / 26

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): *implicit transactional*

1. Hardware Lock Elision
   - provides a way to execute a critical section without the atomic updates necessary to acquire and release the lock
   - requires annotations

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): *implicit transactional*

1. Hardware Lock Elision
   - provides a way to execute a critical section without the atomic updates necessary to acquire and release the lock
   - requires annotations
     - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
     - instruction that increments the semaphore must be prefixed with `XRELEASE`
     - these prefixes are ignored on older platforms
   - after `XACQUIRE instr`, all accesses are stored in read-/write-sets
   - the value v that `instr` is updating to v' is only read, not written
   - any accessed cache line is tracked in the read-/write-sets
   - if any other processor invalidates any of these cache lines, the transaction aborts
   - when `XRELEASE instr'` is seen, tracking of read-/write-sets stops
   - if `XRELEASE instr'` writes a value different to v, the transaction aborts ($\leadsto$ nested locks)
   - aborting a transaction requires:
     - a shadow copy of the processor state at `XACQUIRE`

Example for HTM: Intel

Intel's Haswell microarchitecture (March - June 2013): *implicit transactional*

1. Hardware Lock Elision
   - provides a way to execute a critical section without the atomic updates necessary to acquire and release the lock
   - requires annotations
     - instruction setting the semaphore to 0 must be prefixed with `XACQUIRE`
     - instruction that increments the semaphore must be prefixed with `XRELEASE`
     - these prefixes are ignored on older platforms
   - after `XACQUIRE instr`, all accesses are stored in read-/write-sets
   - the value v that `instr` is updating to v' is only read, not written
   - any accessed cache line is tracked in the read-/write-sets
   - if any other processor invalidates any of these cache lines, the transaction aborts
   - when `XRELEASE instr'` is seen, tracking of read-/write-sets stops
   - if `XRELEASE instr'` writes a value different to v, the transaction aborts ($\leadsto$ nested locks)
   - aborting a transaction requires:
     - a shadow copy of the processor state at `XACQUIRE`
     - an invalidation of all cache lines in the read-/write sets
     - a re-execution of the code with normal lock semantics

Restricted Transactional Memory

1. Hardware Lock Elision
   - ...
2. Restricted Transactional Memory
   - provides new instructions `XBEGIN`, `XEND`, `XABORT`, and `XTEST`
   - `XBEGIN` takes an instruction address where execution continues if the transaction aborts
   - `XABORT` aborts the current transaction with an error code
   - `XTEST` checks if the processor is executing transactionally
   - internal operations similar to lock elision
   - aborts on every use of OS calls, IO, accesses to non-MESI addresses, etc.
   - programmer must provide alternative code path
   - in contrast to lock elision, there is no progress guarantee
   - $\leadsto$ semantics needs to be re-implemented with locks or STM

Integrating Non-TM Resources

Allowing access to other resources than memory inside an `atomic` block poses problems:

- storage management, condition variables, `volatile` variables, input/output
- semantics should be as if `atomic` implements SLA or TSC semantics

Usual choice is one of the following:

- *Prohibit It.* Certain constructs do not make sense. Use compiler to reject these programs.
- *Execute It.* Library routines may be executable as transactions.
- *Irrevocably Execute It.* Universal way to deal with operations that cannot be undone: ensure that this transaction is able to terminate before starting *It* by making all other transactions conflict.
- *Integrate It.* Re-write code to be transactional: error logging, writing data to a file, ...

Integrating Non-TM Resources

Allowing access to other resources than memory inside an `atomic` block poses problems:

- storage management, condition variables, `volatile` variables, input/output
- semantics should be as if `atomic` implements SLA or TSC semantics

Usual choice is one of the following:

- *Prohibit It.* Certain constructs do not make sense. Use compiler to reject these programs.
- *Execute It.* Library routines may be executable as transactions.
- *Irrevocably Execute It.* Universal way to deal with operations that cannot be undone: ensure that this transaction is able to terminate before starting *It* by making all other transactions conflict.
- *Integrate It.* Re-write code to be transactional: error logging, writing data to a file, ...

$\rightsquigarrow$ currently best to use TM only for memory; check if TM supports irrevocable transactions

Concurrency: Transactions | Implementation of Software TM | 23 / 26

Transactional Memory: Summary

Transactional memory aims to provide `atomic` blocks for general code:

- frees the user from deciding how to lock data structures
- compositional way of communicating concurrently
- can be implemented using software (locks, atomic updates) or hardware

Transactional Memory: Summary

Transactional memory aims to provide `atomic` blocks for general code:

- frees the user from deciding how to lock data structures
- compositional way of communicating concurrently
- can be implemented using software (locks, atomic updates) or hardware

The devil lies in the details:

- semantics of *explicit HTM* and *STM* transactions quite subtle when mixing with non-TM (*weak* vs. *strong isolation*)
- *single-lock atomicity* and *transactional sequential consistency* semantics
- STM not the right tool to synchronize threads
- STM providing *opacity* require *eager conflict detection*

Concurrency: Transactions | Implementation of Software TM | 24 / 26

Outlook

Several other principles exist for concurrent programming:

1. non-blocking message passing (the actor model)
   - a program consists of actors that send messages
   - each actor has a queue of incoming messages
   - messages can be processed and new messages can be sent
   - special filtering of incoming messages
   - *example:* Erlang, many add-ons to existing languages
2. blocking message passing (CSP, π-calculus, join-calculus)
   - a process sends a message over a channel and blocks until the recipient accepts it
   - channels can be send over channels (π-calculus)
   - *examples:* Occam, Occam-π, Go
3. (immediate) priority ceiling
   - declare *processes* with priority and *resources* that each process may acquire
   - each resource has the maximum (ceiling) priority of all processes that may acquire it
   - a process' priority at run-time increases to the maximum of the priorities of held resources
   - the process with the maximum (run-time) priority executes

References

D. Dice, O. Shalev, and N. Shavit.
Transactional Locking II.
In *Distributed Coputing*, LNCS, pages 194–208. Springer, Sept. 2006.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.