### 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 \( \rightsquigarrow \) zombie transaction
- this is usually ok since it will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

```c
atomic {
    int tmp1 = x;          // preserved invariant: x==y
    atomic {
        int tmp2 = y;
        assert(tmp1==0);  // not !tmp1
        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.

\( \rightsquigarrow \) failing transactions still sees a consistent view of memory

#### Concurrency: Transactions | Transaction Semantics

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**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 \( \rightsquigarrow \) zombie transaction
- this is usually ok since it will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

```c
atomic {
    int tmp1 = x;          // preserved invariant: x==y
    atomic {
        int tmp2 = y;          // preserved invariant: x==y
        assert(tmp1==0);  // not !tmp1
        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 \( \rightsquigarrow \) zombie transaction
- this is usually ok since it will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

```c
atomic {
    int tmp1 = x;  // preserved invariant: x==y
    int tmp2 = y;
    assert(tmp1-tmp2==0);
}
```
- critical for C/C++ if, for instance, variables are pointers

Definition (opacity)
A TM system provides \textit{opacity} if failing transactions are serializable w.r.t. committing transactions.

\( \rightsquigarrow \) failing transactions still sees a consistent view of memory

Weak- and Strong Isolation

If guarantees are only given about memory accessed inside \texttt{atomic}, a TM implementation provides \textit{weak isolation}.
Can we mix transactions with code accessing memory non-transactionally?

- 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

```c
atomic { // Thread 1
    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

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

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

**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.

- like *sequential consistency*, SLA is a statement about program equivalence
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
Disadvantages of the SLA model

The SLA model is simple but often too strong:

- SLA has a weaker progress guarantee than a transaction should have
  ```
  // Thread 1
  atomic {
  while (true) {}
  }
  // Thread 2
  atomic {
  int tmp = x; // x in TM
  }
  ```
- SLA correctness is too strong in practice
  ```
  // Thread 1
  data = 1;
  atomic {
  if (ready) {
  // use tmp
  }
  }
  // Thread 2
  atomic {
  int tmp = data;
  }
  ```
  ▶ 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:

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

A \( k = i + j \)

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

**Translation of atomic-Blocks**
A TM system must track which shared memory locations are accessed:
- convert every read access \( x \) from a shared variable to \( \text{ReadTx}(kx) \)

**Translation of atomic-Blocks**
A TM system must track which shared memory locations are accessed:
- convert every read access \( x \) from a shared variable to \( \text{ReadTx}(kx) \)
- convert every write access \( x_w \) to a shared variable to \( \text{WriteTx}(kx,w) \)
Translation of atomic-Blocks
A TM system must track which shared memory locations are accessed:
- convert every read access \( x \) from a shared variable to \( \text{ReadTx}(\&x) \)
- convert every write access \( x=e \) to a shared variable to \( \text{WriteTx}(\&x,e) \)

Convert atomic blocks as follows:

```c
atomic {
    // code
    do {
        \text{StartTx}();
        // code with \text{ReadTx} and \text{WriteTx}
    } while (!\text{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 \text{retry} keyword
  - when executing \text{retry}, the transaction aborts and re-starts
  - the transaction will again wind up at \text{retry} unless its \text{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, \text{PopRight} can be implemented as follows:

```c
int \text{PopRight}(\text{DQueue}^* q) {
    \text{QNode}^* oldRightNode;
    \text{atomic} {
        \text{QNode}^* rightSentinel = q->right;
        oldRightNode = rightSentinel->left;
        if (oldRightNode==leftSentinel) retry;
        \text{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 \text{PopRight}
- if the queue is empty, it may abort if \text{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**: timestamp 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
- **eager conflict detection**: a transaction aborts as soon as it conflicts

---

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**: timestamp 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
- **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 \( \text{obj} \) is implemented as follows:

```java
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;
}
```

**Concurrency: Transactions**

**Implementation of Software TN**

15/33

Committed a Transaction

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

```java
bool CommitTx(TMDesc tx) {
    foreach (e in tx.writeSet)
        if (try_wait(e.obj,sem)) goto Fail;
    numVar = FetchAndAdd(e.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 = numVar; signal(e.obj,sem);
    }
    return true;

    Fail:
    // signal all acquired semaphores
    return false;
}
```

**Concurrency: Transactions**

**Implementation of Software TN**

Concurrent Transactions

Implementation of Software TN

16/33

Properties of TL2

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

```
StartTx  ReadTx  WriteX  ReadX  CommitTx
```

- Memory state seems to be consistent
- Increment global clock
- Validate read set
- Write redo-log

Other observations:

- For some \( \mathbf{c} \), \( \mathbf{y} \), \( \mathbf{z} \) we have:
  - \( y, z \) is consistent
  - \( x, \mathbf{c} \) and \( \mathbf{y}, \mathbf{z} \) are inconsistent
Properties of TL2

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

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

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

Other observations:
- reading values 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:

StartTx  ReadTx  WriteTx  ReadTx  CommitTx

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

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

Concurrence: Transactions  Implementation of Software TN  17/32

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

Concurrence: Transactions  Implementation of Software TN  19/32

Properties of TL2

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

StartTx  ReadTx  WriteTx  ReadTx  CommitTx

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

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

Concurrence: Transactions  Implementation of Software TN  17/32

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
  - the granularity of what is locked might be too large

Concurrence: Transactions  Implementation of Software TN  19/32
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
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:
    ```java
    // Thread 1
    atomic {
    ...
    // x is shared
    x = 42;
    }
    int r = ReadTx(x, 0);
    ```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object if the old object has not changed
  - idea of the original STM proposal

---

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
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:
    ```java
    // Thread 1
    atomic {
    ...
    // x is shared
    x = 42;
    }
    int r = ReadTx(x, 0);
    ```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object if the old object has not changed
  - idea of the original STM proposal

- TM system should figure out which memory locations must be logged
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
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:
    ```c
    // Thread 1
    atomic {
      atomic {
        ...
        // x is shared
        x = 42;
      }
    }
    int r = ReadTx(&x, 0);
    }
    ```
- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object
    if the old object has not changed
  - ~~idea of the original STM proposal~~
- TM system should figure out which memory locations must be logged
- danger of live-locks: transaction B might abort A which might abort B ...

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.** I/O operations may only happen in some runs (e.g. file writes usually go to a buffer). Abort if I/O happens.
- **Irrevocably Execute It.** Universal way to deal with operations that cannot be undone: enforce that this transaction terminates (possibly before starting) 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.** I/O operations may only happen in some runs (e.g. file writes usually go to a buffer). Abort if I/O happens.
- **Irrevocably Execute It.** Universal way to deal with operations that cannot be undone: enforce that this transaction terminates (possibly before starting) by making all other transactions conflict.
- **Integrate It.** Re-write code to be transactional: error logging, writing data to a file, ...

~~currently best to use TM only for memory; check if TM supports irrevocable transactions~~
## 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:
  - additional hardware makes it cheap to perform conflict detection

---

<table>
<thead>
<tr>
<th>Concurrency: Transactions</th>
<th>Hardware Transactional Memory</th>
</tr>
</thead>
</table>

---

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:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the 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:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

≈ limited by fixed hardware resources, a software 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:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

≈ limited by fixed hardware resources, a software 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`

≈ mixing transactional and non-transactional accesses is problematic

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:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts
- limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:
- **Explicit Transactional HTM:** each access is marked as transactional
  - similar to `StartTx, ReadTx, WriteTx, and CommitTx`
  - requires separate transaction instructions
  - a transaction has to be translated differently
  - mixing transactional and non-transactional accesses is problematic
- **Implicit Transactional HTM:** only the beginning and end of a transaction are marked
  - same instructions can be used, hardware interprets them as transactional

---

Example for HTM

AMD Advanced Synchronization Facilities (ASF):
- defines a logical *speculative region*

LOCK MOV instructions provide *explicit* data transfer between normal memory and speculative region

---

Example for HTM

AMD Advanced Synchronization Facilities (ASF):
- defines a logical *speculative region*
- LOCK MOV instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations
Example for HTM

AMD Advanced Synchronization Facilities (ASF):
- defines a logical **speculative region**
- **LOCK** MOV instructions provide explicit data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (since Sep 2013):
- **implicit transactional**, can use normal instructions within transactions
- tracks read/write set using a single transaction bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)

Example for HTM

AMD Advanced Synchronization Facilities (ASF):
- defines a logical **speculative region**
- **LOCK** MOV instructions provide explicit data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's Haswell microarchitecture (since Sep 2013):
- **implicit transactional**, can use normal instructions within transactions
- tracks read/write set using a single transaction bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
Example for HTM

AMD Advanced Synchronization Facilities (ASF):
- defines a logical *speculative region*
- **LOCK MOV** instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel’s Haswell microarchitecture (since Sep 2013):
- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources
- may abort at any time due to lack of resources

---

Example for HTM

AMD Advanced Synchronization Facilities (ASF):
- defines a logical *speculative region*
- **LOCK MOV** instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel’s Haswell microarchitecture (since Sep 2013):
- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources
- may abort at any time due to lack of resources

Intel provides two software interfaces to TM:
- Restricted Transactional Memory
- Hardware Lock Elision

---

Restricted Transactional Memory (Intel)

Provides new instructions **XBEGIN, XEND, XABORT**, and **XTEST**:
- **XBEGIN** takes an instruction address where execution continues if the transaction aborts
Example for HTM
AMD Advanced Synchronization Facilities (ASF):
- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel’s Haswell microarchitecture (since Sep 2013):
- *implicit transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources
- aborting in an inner transaction means aborting all of them

Intel provides two software interfaces to TM:
- Restricted Transactional Memory
- Hardware Lock Elision

Restricted Transactional Memory (Intel)
Provides new instructions `XBEGIN`, `XEND`, `XABORT`, and `XTEST`:
- `XBEGIN` takes an instruction address where execution continues if the transaction aborts
- `XEND` commits the transaction started by the last `XBEGIN`

```c
int data[100]; // shared
void update(int idx, int value) {
    if (_xbegin() == -1) {
        data[idx] += value;
        _xend();
    } else {
        // transaction failed
    }
}
```

**~ user must provide fall-back code**
**Considerations for the Fall-Back Path**

Consider executing the following code in parallel with itself:

```c
int data[100]; // shared
void update(int idx, int value) {
  if (_xbegin() == -1) {
    data[idx] += value;
    _xend();
  } else {
    data[idx] += value;
  }
}
```

Problem:
- if the fall-back code is executed, it might be interrupted by the transaction
- the write in the fall-back path thereby overwrites the value of the transaction

---

**Implementing RTM using the Cache**

Transactional operation:
- augment each cache line with an extra bit $T$

---

**Protecting the Fall-Back Path**

Use a lock to prevent the transaction from interrupting the fall-back path:

```c
int data[100]; // shared
int mutex;
void update(int idx, int value) {
  if (_xbegin() == -1) {
    data[idx] += value;
    _xend();
  } else {
    wait(mutex);
    data[idx] += value
    signal(mutex);
  }
}
```

- fall-back path may not run in parallel with others ✔
- transactional region may not run in parallel with fall-back path
Implementing RTM using the Cache

Transactional operation:

- augment each cache line with an extra bit $T$
- use a nesting counter $C$ and a backup register set

Additional transaction logic:

- **XBEGIN** increment $C$ and, if $C = 0$, back up registers
- read or write access to a cache line sets $T$ if $C > 0$
- applying an *invalidate* message from *invalidate queue* to a cache line with $T = 1$ issues **XABORT**
- observing a *read* message for a *modified* cache line with $T = 1$ issues **XABORT**

**XABORT** clears all $T$ flags, sets $C = 0$ and restores CPU registers.
Implementing RTM using the Cache

Transactional operation:
- augment each cache line with an extra bit $T$
- use a nesting counter $C$ and a backup register set
  ~ additional transaction logic:
  - $\text{XBEGIN}$ increment $C$ and, if $C = 0$, back up registers
  - read or write access to a cache line sets $T$ if $C > 0$
  - applying an invalidate message from invalidate queue to a cache line with $T = 1$ issues $\text{XABORT}$
  - observing a read message for a modified cache line with $T = 1$ issues $\text{XABORT}$
  - $\text{XABORT}$ clears all $T$ flags, sets $C = 0$ and restores CPU registers
  - $\text{XCOMMIT}$ decrement $C$ and, if $C = 0$, clear all $T$ flags

Common Code Pattern for Mutexes

Using HTM in order to implement mutex:

```c
int data[100]; // shared
int mutex;
void update(int idx, int val) {
    if (_xbegin()==-1) {
        if (mutex>0) _xabort();
        data[idx] += value;
        _xend();
    }
    else {
        wait(mutex);
        data[idx] += value
        signal(mutex);
    }
}
```

- the critical section may be executed with an elided lock
- as soon as one thread conflicts, the mutex will be taken, thereby aborting all other transactions that have read mutex

Illustrating Transactions

Augment MESI state with extra bit $T$ per cache line. CPU A: E5, CPU B: I

Thread A
```c
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

Thread B
```c
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

Hardware Lock Elision

Observation: Using HTM to implement lock elision is a common pattern
- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
## Hardware Lock Elision

**Observation:** Using HTM to implement lock elision is a common pattern

<table>
<thead>
<tr>
<th>~ ~</th>
<th>provide special handling in hardware: HLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock</td>
<td></td>
</tr>
</tbody>
</table>
| • 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 |
| • for a successful elision, all signal/wait operations of a lock must be annotated |
| • the memory location of the lock is locally visible as 0 ("taken") |

## Hardware Lock Elision

**Observation:** Using HTM to implement lock elision is a common pattern

<table>
<thead>
<tr>
<th>~ ~</th>
<th>provide special handling in hardware: HLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock</td>
<td></td>
</tr>
</tbody>
</table>
| • 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 |
| • for a successful elision, all signal/wait operations of a lock must be annotated |
| • the memory location of the lock is locally visible as 0 ("taken") |
| • other processor see the lock as 1 ("not taken") |
Hardware Lock Elision

**Observation:** Using HTM to implement lock elision is a common pattern
~ provide special handling in hardware: HLE
- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with \texttt{XACQUIRE}
  - instruction that increments the semaphore must be prefixed with \texttt{XRELEASE}
  - these prefixes are ignored on older platforms
- for a successful elision, all signal/wait operations of a lock must be annotated
- the memory location of the lock is locally visible as 0 ("taken")
- other processor see the lock as 1 ("not taken")
- only a finite number of locks can be elided
- all but one elided lock may abort ~

---

Implementing Lock Elision

Transaction operation:
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer
- \texttt{XACQUIRE} of lock ensures shared/exclusive cache line state, issues \texttt{XBEGIN} and stores written value in \texttt{elided lock} buffer

```
    CPU A
       |   register
       |    bank
       |      C
```

```
    store
    buffer
    elided
    locks
```

```
    cache
    T
```

```
    invalidate
    queue
```

```
    Memory
```

---

Hardware Lock Elision

**Observation:** Using HTM to implement lock elision is a common pattern
~ provide special handling in hardware: HLE
- provides a way to execute a critical section without the overhead of the atomic updates necessary to acquire and release the lock
- requires annotations:
  - instruction setting the semaphore to 0 must be prefixed with \texttt{XACQUIRE}
  - instruction that increments the semaphore must be prefixed with \texttt{XRELEASE}
  - these prefixes are ignored on older platforms
- for a successful elision, all signal/wait operations of a lock must be annotated
- the memory location of the lock is locally visible as 0 ("taken")
- other processor see the lock as 1 ("not taken")
- only a finite number of locks can be elided
- all but one elided lock may abort ~
  - progress guarantee since lock is taken on abort
  - no back up path is required

```
    CPU A
       |   register
       |    bank
       |      C
```

```
    store
    buffer
    elided
    locks
```

```
    cache
    T
```

```
    invalidate
    queue
```

```
    Memory
```
Implementing Lock Elision

Transactional operation:
- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer
  - \texttt{XACQUIRE} of lock ensures \texttt{shared/exclusive} cache line state, issues \texttt{XBEGIN} and stores written value in \texttt{elided lock} buffer
  - \texttt{r/w} access to a cache line sets \texttt{T}
  - applying an \texttt{invalidate} message from \texttt{invalidate queue} to an address in the elided lock buffer issues \texttt{XABORT}
  - a \texttt{read} message for a \texttt{modified} cache line or an \texttt{invalidate} message makes the transaction \texttt{irrevocable}

References

- D. Dice, O. Shalev, and N. Shavit.
  Transactional Locking II.

- T. Harris, J. Larus, and R. Rajwar.
  Transactional memory, 2nd edition.

Online blog entries on Intel HTM:
  fun-with-intel-transactional-synchronization-extensions