Choices for Optimistic Concurrency Control

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

Semantics of Transactions

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

- 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 interfere with 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 concurrent 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 \[ \rightsquigarrow \text{zombie transaction} \]
- This is usually ok since it will be aborted eventually
- But transactions may cause havoc when run on inconsistent states

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

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

\[ \rightsquigarrow \text{failing transactions still sees a consistent view of memory} \]

Weak- and Strong Isolation

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

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

- \texttt{no conflict detection} for non-transactional accesses
- Standard race problems as in unlocked shared accesses

```c
// Thread 1
atomic {
    int tmp = x;
    x = 42;
    y = 10;
}
```
- Give programs with races the same semantics as if using a single global lock for all \texttt{atomic} blocks
- \texttt{strong isolation}: retain order between accesses to TM and non-TM

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

\[ \rightsquigarrow \text{like sequential consistency} \]
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
   while (true) {
     atomic {
       int tmp = x; // x in TM
     }
   }
   // Thread 2
   ```
2. SLA correctness is too strong in practice
   ```
   // Thread 1
   int tmp = data;
   if (ready) {
     // use tmp
   }
   // Thread 2
   atomic {
   }
   ```
   - under the SLA model, atomic {} acts as barrier
   - intuitively, the two transactions should be independent rather than synchronize

need a weaker model for more flexible implementation of strong isolation

Transactional Sequential Consistency

How about a more permissive view of transaction semantics?
- TM should not have the blocking behaviour of locks
- the programmer cannot rely on synchronization

Definition (TSC)

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

- 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

**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:

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

```c
====>
```

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

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

**Transactional Memory for the Queue**

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

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

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

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

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

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

- provides opacity: zombie transactions do not see inconsistent state
- uses lazy versioning: writes are stored in a redo-log and done on commit
- validating conflict detection: accessing a modified address aborts

TL2 stores a global version counter and:

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

Principles of TL2

The idea: obtain a version tx.RV from the global clock when starting the transaction, the read-version, and set the versions of all written cells to a new version on commit.

A read from a field at offset of object obj is implemented as follows:

```c
int ReadTx(TMDesc tx, object obj, int offset) {
    if (k(obj[offset]) in tx.redoLog) {
        return tx.redoLog[k(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:

```
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.

Properties of TL2

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

```
StartTx  ReadTx  WriteTx  ReadTx  CommitTx
memory state seems to be consistent
validate read set
increment global clock
```

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`

Committing a Transaction

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

```
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;

General Challenges when using TM

Executing atomic blocks by repeatedly trying to execute 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:
  - Thread 1
  - Thread 2
```
atomic {
    // clock=12
    ...
}
```
  - atomic {
      // clock=13
```
    int r = ReadTx(&x, 0);
```
  - // tx.RV=12/=clock
- 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, ...

---

**Hardware Transactional Memory**
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

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 TSX in Broadwell/Skylake microarchitecture (since Aug 2014):

- *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

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 implementations of HTM:

1. 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

2. Implicit Transactional HTM: only the beginning and end of a transaction are marked
   - same instructions can be used, hardware interprets them as transactional
   - only instructions affecting memory that can be cached can be executed transactionally
   - hardware access, OS calls, page table changes, etc. all abort a transaction
   - provides *strong isolation*

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 TSX in Broadwell/Skylake microarchitecture (since Aug 2014):

- *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
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
- XABORT aborts the current transaction with an error code
- XTEST checks if the processor is executing transactionally

The instruction XBEGIN can be implemented as a C function:

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

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

---

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
- need to ensure that the fall-back path is executed atomically
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
- **XCOMMIT** decrement $C$ and, if $C = 0$, clear all $T$ flags