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 depends on the ability to abstract from details.
Abstraction and Concurrency

Two fundamental concepts to build larger software are:

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

Both, abstraction and composition are closely related, since the ability to compose depends 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 `push()` and `forAll()`
- a set object may internally use the list object and expose a set of operations, including `push()`

The `insert()` operation uses the `forAll()` operation to check if the element already exists and uses `push()` 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 depends 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 `push()` and `forAll()`
- a set object may internally use the list object and expose a set of operations, including `push()`

The `insert()` operation uses the `forAll()` operation to check if the element already exists and uses `push()` 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

Transactional Memory [2]

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

```plaintext
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
    // code
}
```
**Transactional Memory [2]**

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

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

Execute code as transaction:
- execute the code of an atomic block
- nested atomic blocks act like a single atomic block
- check that it runs without conflicts due to accesses from another thread
- if another thread interferes through conflicting updates:
  - undo the computation done so far
  - re-start the transaction
- 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:
- **optimistic vs. pessimistic concurrency control**:
  - pessimistic: detection/resolution when the conflict is about to occur
    - resolution here is usually delaying one transaction
    - can be implemented using locks: deadlock problem
  - optimistic: detection and resolution happen after a conflict occurs
    - resolution here must be aborting one transaction
    - need to repeat 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
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 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

**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 {
    int tmp1 = x;  // preserved invariant: x == y
    int tmp2 = y;  atomic {
        int 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 see a consistent view of memory
Weak- and Strong Isolation

If guarantees are only given about memory accessed inside atomic, a TM implementation provides weak isolation. Can we mix transactions with code accessing memory non-transactionally?
- no conflict detection for non-transactional accesses
- standard race problems as in unlocked shared accesses

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

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

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

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

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

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

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

Properties of Single-Lock Atomicity

Observation:
- SLA enforces order between TM and non-TM accesses
  - this guarantees strong isolation between TM and non-TM accesses
- within one transaction, 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

---

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

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

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

2. SLA correctness is too strong in practice.
   ```
   // Thread 1
   data = 1;
   atomic {
   }
   ready = 1;
   // Thread 2
   atomic {
       int tmp = data;
       // Thread 1 not in atomic
       if (ready) {
           // use tmp
       }
   }
   ```
   - under the SLA model, atomic {} acts as barrier
   - 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.

```latex
\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {$A$};
\node (i) at (0,1) {$i$};
\node (j) at (1,0) {$j$};
\node (k) at (1,1) {$k$};
\node (B) at (2,0) {$B$};
\node (k42) at (2,1) {$k=42$};
\node (atomici) at (0,0.5) {atomic \{ $k = i+j$; \}};
\node (atomicj) at (1,0.5) {atomic \{ $k = i+j$; \}};
\node (atomici42) at (0,1.5) {atomic \{ $k = i+j$; \}};
\node (atomicj42) at (1,1.5) {atomic \{ $k = i+j$; \}};
\draw[->] (A) -- (i);
\draw[->] (A) -- (j);
\draw[->] (A) -- (k);
\draw[->] (i) -- (j);
\draw[->] (i) -- (k);
\draw[->] (j) -- (k);
\draw[->] (i) -- (atomici);
\draw[->] (j) -- (atomicj);
\draw[->] (k) -- (atomici42);
\draw[->] (k42) -- (atomicj42);
\end{tikzpicture}
\end{center}
```

- 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 \( \text{ReadTx}(kx) \)
- convert every write access \( x^w \) to a shared variable to \( \text{WriteTx}(kx,e) \)

Convert atomic blocks as follows:

```
atomic {
    \( \text{StartTx}() \);
    // 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 \( \text{retry} \) 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

**Software Transactional Memory**

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

- \( \text{undo-log} \) of writes if writes have to be undone if a commit fails
- \( \text{redo-log} \) of writes if writes are postponed until a commit
- \( \text{read-} \) and \( \text{write-set} \): locations accessed so far
- \( \text{read-} \) and \( \text{write-version} \): 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 \( \text{redo-log} \) and done on commit
- validating conflict detection: accessing a modified address aborts
A Software TM Implementation

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

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

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

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

Committing a Transaction

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

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

Properties of TL2

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

```
StartTx → ReadTx → WriteTx → ReadTx → CommitTx
```

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, 1fence, read value, 1fence, read version
  - there might be contention on the global clock
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
    atomic {
        // clock=12
        ...
        int r = ReadTx(kx, 0);
    } // tx.RV=12 // clock
    
    // Thread 2
    atomic {
        WriteTx(kx, 0) = 42; // clock=13
    }
    ```
- 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

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, RedTx, 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

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

Intel provides two software interfaces to TM:
1. **Restricted Transactional Memory (RTM)**
2. **Hardware Lock Elision (HLE)**
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;
    }
}
```

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
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) {
    if (_mutex > 0) _xabort();
    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

- 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
- r/w access to cache lines 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_ transition from all T? flags to T, sets C = 0 and restores CPU registers
- _XCOMMIT_ decrement C and, if C = 0, clear all T flags

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();
```

![Diagram showing the interaction between threads A and B]