Transactional Memory

Dave Clarke
dave.clarke@it.uu.se

Suggested Reading

T. Harris, J. Larus, R. Rajwar: **Transactional Memory (2nd edition)**. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010. 263 pages.

Especially Chapters 1-3.

Abstractions for Concurrent/Parallel Programming

Multi-core hardware is incredibly complex. What abstractions do we use to write concurrent/parallel programs?

Two common approaches are:

- Data parallelism
- Task-based parallelism

Data Parallelism

Data parallelism: the same operation is performed on different pieces of data (cf. SIMD).

+ Simple programming model
+ Convenient for certain numeric computations (e.g., matrix operations)
+ Parallelization (e.g., synchronization and load-balancing) can be left to the compiler and run-time system

– Not a universal programming model: only applicable to certain data structures and programming problems

Task-based Parallelism

Task-based parallelism: operations are performed on separate threads that are coordinated with explicit synchronization (fork/join, locks, etc.)

This model places no restrictions on the code that each thread executes, when and how threads communicate, etc.

+ Universal programming model: capable of expressing all forms of parallel computation

– Low level of abstraction, close to hardware
– Very difficult to write correct programs

Locks Are Bad

We have seen some of the difficulties with locks:

- Taking too few locks

  . . . can lead to race conditions and data races.
- Taking too many locks

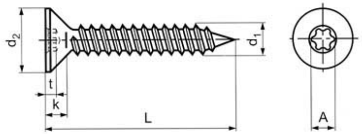
  . . . can inhibit concurrency or lead to deadlocks.
- Taking the wrong locks

  . . . because the connection between data and lock is only implicit.
- Taking locks in the wrong order

  . . . can lead to deadlocks, and is difficult to avoid.
- Error recovery

  . . . is tedious: inconsistent states must be avoided, locks released.

What is Abstraction?



Abstraction (in Computer Science)

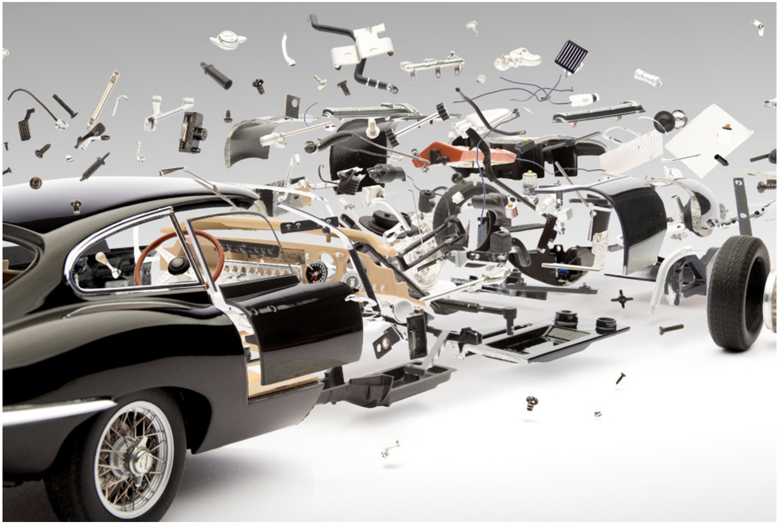
Abstraction is a way to introduce new concepts that are meaningful to humans.

Abstraction tries to reduce and factor out details so that, e.g., the programmer can focus on a few concepts at a time.

Examples: files, data structures, procedure calls, . . .

(We think of files as something real, but files don't really exist—they are a bunch of bits on a hard drive. Come to think of it, bits don't really exist either—they are magnetic fluctuations on the surface of disk platters.)

What is Composition?



Composition (in Computer Science)

Composition is the ability to put entities together to build a larger, more complex entity.

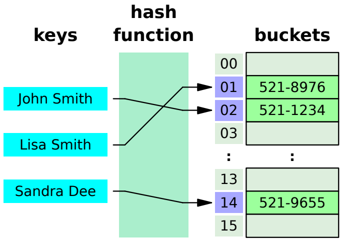
Examples: logic circuits, functions, objects, . . .

(These can be composed into more complex circuits/functions/objects, respectively.)

Abstraction allows us to think about the composition as a single entity, rather than about its constituent parts.

Locks Don't Compose

Consider a hash table that provides operations `Insert` and `Remove`. Suppose that these functions use locks to achieve thread-safety.



How would you implement a function `Move` that moves an item from one hash table to another? The intermediate state, in which neither table contains the item, must not be visible to concurrent threads.

Locks Don't Compose (cont.)

`Move` cannot be implemented (in a thread-safe manner) by simply composing `Insert` and `Remove`.

Instead, new methods such as `LockTable` and `UnlockTable` are needed.

Note that these break the hash table abstraction by exposing an implementation detail.

Moreover, avoiding deadlock is tricky.

Locks Don't Compose: Another Example

Suppose a library function `WaitAny` takes a list of input queues and blocks until (at least) one of the queues is non-empty.

A function `f1` uses `WaitAny` to wait for one of the queues `A` and `B`.

A function `f2` uses `WaitAny` to wait for one of the queues `C` and `D`.

How would you write a function that waits on any of the four queues, and performs the appropriate action in `f1` or `f2`, depending on which queue becomes non-empty?

Locks Don’t Compose: Another Example (cont.)

Simply applying `WaitAny` to `f1` and `f2` is not possible (since these are functions and not queues).

A typical (but awkward) solution is to use `WaitAny` on `A`, `B`, `C` and `D`, and then dispatch to the appropriate function (`f1` or `f2`), depending on which queue has become non-empty.

Here, the abstractions `f1` and `f2` cannot be composed, but we need to know some of their implementation details before they can be merged.

Database Systems

Are there better approaches to general-purpose concurrent/parallel programming?

Let's look at (and take inspiration from) databases. These have supported concurrent access and exploited parallel hardware for many decades.

What is the programming model for databases?

Database Transactions

A database transaction is a sequence of actions (e.g., look-ups, updates) that appears to execute as if there was no concurrent database access.

Thus, different transactions appear to execute one after another—they can be serialized.

The database system ensures that this property is preserved even when transactions are actually executed in parallel.

ACID

A database transaction, by definition, satisfies atomicity, consistency, isolation, and durability (ACID).

Atomicity

Atomicity: Either all actions in a transaction complete successfully (the transaction commits), or there is no evidence at all that the transaction began executing (the transaction aborts).

"All or nothing."

Consistency

Consistency: Transactions start in a consistent state and should produce a consistent state.

The meaning of “consistent” is entirely application-dependent. Typically, data structures must satisfy a number of invariants.

When a transaction aborts, consistency is trivially maintained.

Isolation

Isolation: Each transaction appears to execute on its own. It is not affected by other transactions, even if those execute in parallel.

Durability

Durability: Once a transaction commits, all changes are persistent (i.e., written to disk) and will not be lost in case of a system crash.

Transactions for Concurrent/Parallel Programming

Idea: Transactions provide a convenient abstraction also for coordinating reads and writes of shared data in a concurrent (or parallel) system.

If we could wrap a computation in a transaction, we would get atomicity, consistency and isolation—without having to worry about locking!

Since programs typically access shared data in memory, this approach to concurrency control is known as transactional memory.

Transactional Memory: A Brief History

Transactional memory was first proposed by Lomet in 1977.

Practical implementations became available in the 1990s.

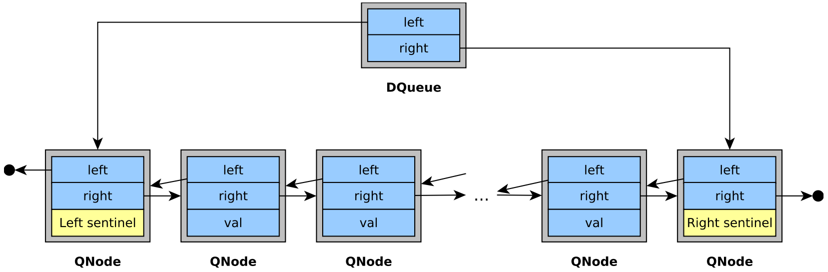
Today, TM is (still) a very active research area with many proposed hard- and software implementations.

Intel Haswell processors (released in 2013) offer hardware transactional memory support.

Differences Between Database Transactions And TM

- Data is held in memory, not on disk (much faster access times).
  - TM implementations must have little overhead for each transaction.
  - Hardware support is more attractive for TM.
- Data is lost when the program terminates (no durability).
- Data may be accessed also through normal memory operations.
- TM must coexist with existing languages, libraries, etc.

Running Example: A Double-Ended Queue



Exercise: implement a (sequential) function

```
void PushLeft(DQueue *q, int val)
```

PushLeft

```
void PushLeft(DQueue *q, int val) {
  QNode *qn = malloc(sizeof(QNode));
  qn->val = val;
  QNode *leftSentinel = q->left;
  QNode *oldLeftNode = leftSentinel->right;
  qn->left = leftSentinel;
  qn->right = oldLeftNode;
  leftSentinel->right = qn;
  oldLeftNode->left = qn;
}
```

Basic Transactional Memory: Example

```
void PushLeft(DQueue *q, int val) {
  QNode *qn = malloc(sizeof(QNode));
  qn->val = val;
  do {
    StartTx();
    QNode *leftSentinel = ReadTx(&(q->left));
    QNode *oldLeftNode = ReadTx(&(leftSentinel->right));
    WriteTx(&(qn->left), leftSentinel);
    WriteTx(&(qn->right), oldLeftNode);
    WriteTx(&(leftSentinel->right), qn);
    WriteTx(&(oldLeftNode->left), qn);
  } while (!CommitTx());
}
```

Basic Transactional Memory: Remarks

Memory accesses are performed by `ReadTx`/`WriteTx` operations. These accesses must be managed by the transactional memory implementation.

The `StartTx`/`CommitTx` pair indicates the scope of a transaction.

Note that the level of abstraction is fundamentally different from manual locking: we have not indicated where to acquire/release locks, or which operations may execute concurrently with `PushLeft`.

Implementation of TM: Versioning

The TM implementation must execute the (tentative) operations performed by a transaction.

- Eager versioning: updates are directly written to memory, maintaining information about overwritten values
- Lazy versioning: updates are collected in a private buffer and written to memory when the transaction commits

Implementation of TM: Conflict Detection

The TM implementation must ensure isolation between transactions. This is typically achieved by detecting conflicts (cf. Bernstein's condition[1]) between transactions that run in parallel.

- Eager conflict detection: conflicts are identified while transactions are running; (at least) one transaction is aborted
- Lazy conflict detection: conflicts are identified when transactions commit; (at least) one commit fails

[1]Bernstein: $S_1; S_2 \equiv S_1 \| S_2$ provided $WR(S_1) \cap WR(S_2) = \emptyset$, $RD(S_1) \cap WR(S_2) = \emptyset$ and $WR(S_1) \cap RD(S_2) = \emptyset$.

Building on Basic Transactions

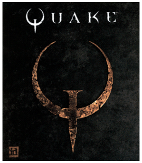
ReadTx/WriteTx and StartTx/CommitTx operations introduce a lot of boilerplate. Transactions are frequently integrated into high-level languages via atomic blocks. For instance:

```
void PushLeft(DQueue *q, int val) {
  QNode *qn = malloc(sizeof(QNode));
  qn->val = val;
  atomic {
    QNode *leftSentinel = q->left;
    QNode *oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
  }
}
```

What is Transactional Memory Good For?

Scalability: locking is pessimistic (every thread needs to acquire the lock before it can do anything), while transactions are optimistic (a commit fails only in the case of a conflict).

Algorithms that operate on **dynamically selected parts** of a larger data structure (e.g., graph traversals): locks are overly conservative (e.g., locking the entire data structure at once) or very difficult to get right.



Quake game server: to model the effect of a player's move, a lock-based implementation first needs to simulate the move to find out which objects to lock, then lock and update the affected objects. Transactions can avoid the simulation step.

Transactions Are Not a Panacea

It is still up to the programmer to divide the work into tasks that can be executed concurrently.

- Transactions might be too short
  . . . exposing inconsistent intermediate states.
- Transactions might be too long
  . . . inhibiting concurrency or even blocking concurrent threads.
- Transactions might be used incorrectly
  . . . e.g., `StartTx` without a matching `CommitTx`.
- Transactions might cause a decline in performance
  . . . potentially rendering parallelization pointless.