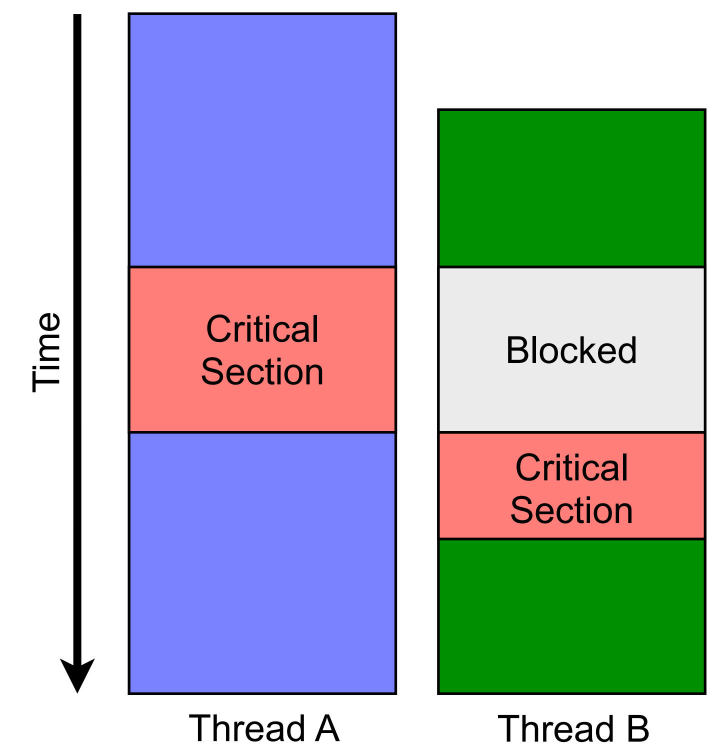
University of Glasgow | School of Computing Science

Transactional Memory and Concurrency

Advanced Operating Systems (M)
Lecture 18

Concurrency, Threads, and Locks

- Operating systems expose concurrency via *processes* and *threads*
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory
- The processor/language memory models specify how concurrent access to shared memory works
  - Generally enforce synchronisation via explicit locks around *critical sections* (e.g. Java `synchronized` methods and statements; `pthread` mutexes)
  - Very limited guarantees about unlocked concurrent access to shared memory



Limitations of Locking

- Limitations of locks for managing concurrency:
  - Difficult to enforce locking
    - Users of shared data must acquire and release the locks
    - Encapsulating shared data in objects that manage the lock can help
  - Difficult to guarantee freedom from deadlocks
    - Usual solution: acquire and release locks in a fixed order
    - But, conflicts with encapsulation of locks within objects to enforce locking
  - Failures are silent
    - Race conditions due to incorrect locking generally only show under load
    - Extremely difficult to locate and debug
  - Balancing performance and correctness is difficult
    - Too many locks inhibit concurrency and reduce performance; too few lead to subtle bugs
- Implication: ensuring correct use of locks is difficult

Composition of Lock-based Code

- Correctness of small-scale code using locks can be ensured by careful coding (at least in theory)
- A more fundamental issue: lock-based code does not compose to larger scale
  - Assume a correctly locked bank account class, with methods to credit and debit money from an account
  - Want to take money from `a1` and move it to `a2`, without exposing an intermediate state where the money is in neither account
  - Can't be done without locking all other access to `a1` *and* `a2` while the transfer is in progress
  - The individual operations are correct, but the combined operation is not

```
a1.debit(v)
a2.credit(v)
```

Preemption exposes intermediate state

- This is lack of abstraction a limitation of the lock-based concurrency model, and cannot be fixed by careful coding
- Locking requirements form part of the API of an object

Transactions for Managing Concurrency

- An alternative approach: use *atomic transactions* to manage concurrency
  - A program is a sequence of concurrent atomic actions
  - Atomic actions succeed or fail in their entirety, and intermediate states are not visible to other threads
  - The runtime must ensure actions have the usual ACID properties:
    - Atomicity – all changes to the data are performed, or none are
    - Consistency – data is in a consistent state when a transaction starts, and when it ends
    - Isolation – intermediate states of a transaction are invisible to other transactions
    - Durability – once committed, results of a transaction persist

```
atomic {
  a1.debit(v)
  a2.credit(v)
}
```

- Advantages:
  - Transactions can be composed arbitrarily, without affecting correctness
  - Avoid deadlock due to incorrect locking, since there are no locks

Transactional Memory Programming Model

- Simple programming model:
  - Blocks of code can be labelled `atomic {…}`
  - Run concurrently and atomically with respect to every other `atomic {…}` blocks – controls concurrency and ensures consistent data structures
- Implemented via optimistic synchronisation
  - A thread-local *transaction log* is maintained, records every memory read and write made by the atomic block
  - When an atomic block completes, the log is *validated* to check that it has seen a consistent view of memory
  - If validation succeeds, the transaction *commits* its changes to memory; if not, the transaction is rolled-back and retried from scratch

Limitations of the Programming Model

- Transactions may be re-run automatically, if their transaction log fails to validate
- Places restrictions on transaction behaviour:
  - Transactions must be referentially transparent
    - They produce the same answer each time they're executed
  - Transactions must do nothing irrevocable

```
atomic {
  if (n > k) then launchMissiles();
  doMoreStuff;
}
```

    - Might launch the missiles multiple times, if it gets re-run due to validation failure caused by `doMoreStuff`
    - Might accidentally launch the missiles if a concurrent thread modifies `n` or `k` while the transaction is running (this will cause a transaction failure, but too late to stop the launch)
  - These restrictions *must* be enforced, else we trade hard-to-find locking bugs for hard-to-find transaction bugs

Managing Communication and I/O

- Communication and I/O must be limited during a transaction
  - Pure functions can be executed normally
  - Functions that only perform memory actions can be executed normally, provided transaction log tracks the memory actions and validates them before the transaction commits
  - Functions that perform I/O are prohibited within a transaction

- Difficult to ensure through programmer discipline – needs language support

Implementations of Transactional Memory

- Transactions can be implemented in hardware or software
  - Need to track memory accesses, and potentially perform rollback
  - Can be done by a run-time support library, or using dedicated hardware
  - To date, have used software-based implementations; hardware-based implementations likely in future
    - e.g., Intel has announced support in their Haswell platform, due in 2013
- Difficulty *enforcing* transactions are side-effect free, so they can safely be rolled-back
  - Requires programming language (type-system) support

Controlling Side Effects

- Monads → well-defined way to control side-effects in functional languages
  - A monad `M a` describes an action (i.e., a function) that, when executed, produces a result of type `a`; along with rules for chaining actions
  - A common use is for controlling I/O operations:
    - The `putChar` function takes a character, and returns an I/O action that can display the character when performed
    - The `getChar` function is an I/O action; when performed it reads and returns a character
    - The `main` function is itself an I/O action, which wraps and performs the other actions

```
putChar :: Char -> IO ()
getChar :: IO Char
```

  - The definition of the I/O monad type ensures that a function that is not passed an I/O action cannot perform I/O
    - This is one part of the puzzle for transactional memory: define `atomic {…}` to so that it doesn't take an I/O action

Controlling Side Effects in Transactions

- How to track side-effecting memory actions?
  - Use another monad *STM a* to wrap the transaction

```
-- The STM monad itself
data STM a
instance Monad STM
```

  - Manage side-effect via a *TVar* type
    - The *newTVar* function takes a value of type *a*, returns a new *TVar* to hold the value, wrapped in an STM monad action
    - *readTVar* takes a *TVar* and returns an STM monad action; when performed this returns the value of that *TVar*; *writeTVar* function takes a *TVar* and a value, and returns an STM action that assigns the value to the *TVar*

```
-- Transactional variables
data TVar a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

  - Define `atomic {…}` to perform an STM transaction, and return an I/O action that performs the I/O and side effects that run the transaction

```
atomic :: STM a -> IO a
```

    - The *newTVar*, *readTVar*, and *writeTVar* functions need an STM action, and so can only run in the context of an atomic block that provides such an action

Transactional Memory in Haskell

- Transactional memory is a good fit with Haskell
  - Pure functions and monads ensure transaction semantics are preserved
  - I/O and side-effects contained in `STM` action of an `atomic {…}` block
    - The `TVar` implementation is responsible for tracking side effects
    - The `atomic {…}` block validates, then commits the transaction (by returning an IO action to perform the transaction)
  - Untracked I/O or side-effects cannot be performed within an `atomic {…}` block, since there is no way to access an `IO` action directly
    - There is no `IO` action in scope, so code requiring one will not compile
    - Only way to access to an `IO` action is via the `STM` action passed to the `atomic {…}` block
    - A `TVar` requires an `STM` action, but these are only available in an `atomic {…}` block; hence can't update a `TVar` outside a transaction (and hence can't break atomicity guidelines)

STM Haskell Example: Resource Manager

- Implement a resource manager, granting access to integral chunks of the resource, enforcign access control between threads
  - `getR r n` should return `n` units of the resource `r` blocking until it is available
  - `putR r n` should return `n` units of the resource `r` to the available pool – implementation on the right

```
type Resource = TVar Int

putR :: Resource -> Int -> STM ()
putR r i = do {
  v <- readTVar r;
  writeTVar r (v + i)
}
```

- The use of the STM monad requires that the `putR` function be called from within an `atomic {…}` block (this is enforced by the compiler)

```
main = do {
  ...;
  atomic (putR r 3);
  ...;
}
```

Blocking Memory Transactions

- Transactions control access to resources, they do not provide synchronisation
  - Address by providing a `retry` operation for atomic blocks e.g., consider the `getR` implementation on the right

```
getR :: Resource -> Int -> STM ()
getR r i = do {
  v <- readTVar r;
  if (v < i) then
    retry
  else
    writeTVar r (v - i)
}
```

- The `retry` function has type `STM a`, so must run within an STM action
  - That is, it must run within an `atomic` block

```
retry :: STM a
```

- Calling `retry` function aborts and restarts the current transaction, but blocks until one of its associated TVars has been modified
  - The system tracks access to the TVars to maintain the transaction log, so this is easily implemented
- The `retry` function is generally called when other concurrency approaches would block waiting for a signal

Sequential Composition

- The entire set of operations surrounded in an atomic block appears to take place indivisibly
  - e.g., the operation on the right atomically gets 3 units of `r1` then 7 of `r2`, the `do` notation provides for sequential composition of STM actions

```
atomic (do {getR r1 3; getR r2 7})
```

- Note:
  - The type system ensures STM actions can *only* be executed in an atomic block
  - Actions accumulated over the *entire* atomic block execute or are rolled back when the transaction log for that block is validated
  - Either call to `getR` can invoke `retry`, causing the *entire* atomic block to be restarted
  - Any STM action can be robustly composed with other STM actions, and the resulting sequence of actions will still execute atomically

Composition of Alternatives

- May want to try one operation, and if that fails, try something else
  - Useful for error handling
- STM Haskell defines the `orElse` function which takes two STM actions, and returns one

```
orElse :: STM a -> STM a -> STM a
```

  - Calling `` s1 `orElse` s2 `` first tries to run `s1`; if `s1` calls `retry` then it's abandoned without effect and `s2` is run instead
  - If `s2` also calls `retry`, then the entire action is restarted

```
atomic (getR r1 3 `orElse` getR r2 7)
```

- An alternative error handling method is throwing an exception
  - Throwing an exception causes the transaction abort and be validated;
  - If the transaction validates, the exception propagates; else the exception is caught and the transaction retried (the exception might be due to the inconsistency that caused validation to fail)

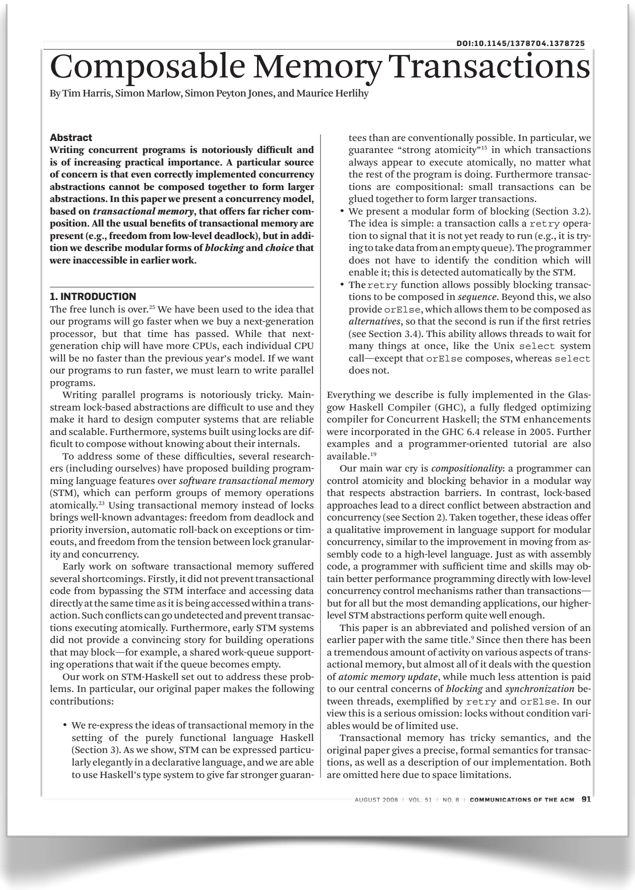
Transactional Memory in Other Languages

- STM Haskell is very powerful – but relies on the type system to ensure safe composition and retry
- Integration into mainstream languages is difficult
  - Most languages cannot require use of pure functions
  - Most languages cannot limit the use of I/O and side effects
  - Transaction memory can be used without these, but requires programmer discipline to ensure correctness – and has silent failure modes
- Unclear that the approach generalises to other languages

Discussion and Further Reading

- T. Harris, S. Marlow, S. Peyton Jones and M. Herlihy, “Composable Memory Transactions”, CACM, 51(8), August 2008. DOI:10.1145/1378704.1378725

- Is software transactional memory a realistic technique?
- Do its requirements for a purely functional language, with monadic I/O, restrict it to being a research toy?
- How much benefit can be gained from transactional memory in more traditional languages?



DOI:10.1145/1378704.1378725

Composable Memory Transactions

By Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy

Abstract
Writing concurrent programs is notoriously difficult and is of increasing practical importance. A particular source of concern is that even correctly implemented concurrency abstractions cannot be composed together to form larger abstractions. In this paper we present a concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g., freedom from low-level deadlock), but in addition we describe modular forms of *blocking* and *choice* that were inaccessible in earlier work.

1. INTRODUCTION
The free lunch is over.[25] We have been used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs.

Writing parallel programs is notoriously tricky. Mainstream lock-based abstractions are difficult to use and they make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals.

To address some of these difficulties, several researchers (including ourselves) have proposed building programming language features over *software transactional memory* (STM), which can perform groups of memory operations atomically.[23] Using transactional memory instead of locks brings well-known advantages: freedom from deadlock and priority inversion, automatic roll-back on exceptions or timeouts, and freedom from the tension between lock granularity and concurrency.

Early work on software transactional memory suffered several shortcomings. Firstly, it did not prevent transactional code from bypassing the STM interface and accessing data directly at the same time as it is being accessed within a transaction. Such conflicts can go undetected and prevent transactions executing atomically. Furthermore, early STM systems did not provide a convincing story for building operations that may block—for example, a shared work-queue supporting operations that wait if the queue becomes empty.

Our work on STM-Haskell set out to address these problems. In particular, our original paper makes the following contributions:

- We re-express the ideas of transactional memory in the setting of the purely functional language Haskell (Section 3). As we show, STM can be expressed particularly elegantly in a declarative language, and we are able to use Haskell's type system to give far stronger guarantees than are conventionally possible. In particular, we guarantee “strong atomicity”[15] in which transactions always appear to execute atomically, no matter what the rest of the program is doing. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.
- We present a modular form of blocking (Section 3.2). The idea is simple: a transaction calls a `retry` operation to signal that it is not yet ready to run (e.g., it is trying to take data from an empty queue). The programmer does not have to identify the condition which will enable it; this is detected automatically by the STM.
- The `retry` function allows possibly blocking transactions to be composed in *sequence*. Beyond this, we also provide `orElse`, which allows them to be composed as *alternatives*, so that the second is run if the first retries (see Section 3.4). This ability allows threads to wait for many things at once, like the Unix `select` system call—except that `orElse` composes, whereas `select` does not.

Everything we describe is fully implemented in the Glasgow Haskell Compiler (GHC), a fully fledged optimizing compiler for Concurrent Haskell; the STM enhancements were incorporated in the GHC 6.4 release in 2005. Further examples and a programmer-oriented tutorial are also available.[19]

Our main war cry is *compositionality*: a programmer can control atomicity and blocking behavior in a modular way that respects abstraction barriers. In contrast, lock-based approaches lead to a direct conflict between abstraction and concurrency (see Section 2). Taken together, these ideas offer a qualitative improvement in language support for modular concurrency, similar to the improvement in moving from assembly code to a high-level language. Just as with assembly code, a programmer with sufficient time and skills may obtain better performance programming directly with low-level concurrency control mechanisms rather than transactions—but for all but the most demanding applications, our higher-level STM abstractions perform quite well enough.

This paper is an abbreviated and polished version of an earlier paper with the same title.[9] Since then there has been a tremendous amount of activity on various aspects of transactional memory, but almost all of it deals with the question of *atomic memory update*, while much less attention is paid to our central concerns of *blocking* and *synchronization* between threads, exemplified by `retry` and `orElse`. In our view this is a serious omission: locks without condition variables would be of limited use.

Transactional memory has tricky semantics, and the original paper gives a precise, formal semantics for transactions, as well as a description of our implementation. Both are omitted here due to space limitations.

AUGUST 2008 | VOL. 51 | NO. 8 | COMMUNICATIONS OF THE ACM 91