Managing Concurrency Using Transactions

- Programming model
- Integration into Haskell
- Integration into other languages
- Discussion

Transactions for Managing Concurrency

- An alternative to locking: 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:
    - **A**tomicity – all changes to the data are performed, or none are
    - **C**onsistency – data is in a consistent state when a transaction starts, and when it ends
    - **I**solation – intermediate states of a transaction are invisible to other transactions
    - **D**urability – once committed, results of a transaction persist
- Advantages:
  - Transactions can be composed arbitrarily, without affecting correctness
  - Avoid deadlock due to incorrect locking, since there are no locks

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

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 transactions
  - 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
  - Progress may be slow if *conflicting* transactions cause repeated validation failures, but will eventually occur

Programming Model – Consequences

- Transactions may be re-run automatically, if their transaction log fails to validate
- Places restrictions on transaction behaviour:
  - Transactions must be referentially transparent
  - Transactions must do nothing irrevocable:
    - 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 transaction modifies **`n`** or **`k`** while the transaction is running (will cause 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

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

Controlling I/O

- Unrestricted I/O breaks transaction isolation
  - Reading and writing files
  - Sending and receiving data over the networks
  - Taking mouse or keyboard input; changing the display
- Require language control of when I/O is performed
  - Remove global functions to perform I/O from the standard library
  - Add an **I/O context** object, local to **`main()`**, passed explicitly to functions that need to perform I/O
    - Compare sockets, that behave in this manner, with file I/O that typically does not
    - I/O functions (e.g., **`printf()`** and friends) then become methods on the I/O context object
    - The I/O context is not passed to transactions, so they cannot perform I/O
    - Example: the IO monad in Haskell

Controlling Side Effects

- Code that has side effects must be controlled
  - Pure and referentially transparent 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 – and can potentially roll them back
    - A *memory action* is an operation that manipulates data on the heap, that could be seen by other threads
    - Tracking memory actions can be done by language runtime (STM; software transactional memory), or via hardware enforced transactional memory behaviour and rollback
- Similar principle as controlling I/O
  - Disallow unrestricted heap access – only see data in transaction context
  - Pass transaction context explicitly to transactions; this has operations to perform transactional memory operations, and rollback if the transaction fails to commit
  - Very similar to the state monad in Haskell

Monadic STM Implementation (1)

- Monads → way to control side-effects in functional languages
  - A monad **`M a`** describes an action (i.e., a function) that, produces a result of type **`a`**, that can be performed in the context **`M`**
    - Along with rules for chaining operations in that context
  - A common use is for controlling I/O operations:
    - The **`putChar`** function takes a character, operates on the **`IO`** context to add the character, and returns nothing
    - The **`getChar`** operates on the **`IO`** context to return a character
    - The main function has an IO context, that wraps and performs other actions

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

  - The definition of the I/O monad type ensures that a function that is not passed the IO context cannot perform I/O operations
  - One part of a software transactional memory implementation: ensure type of the **`atomic {…}`** function does not allow it to be passed an IO context, hence preventing I/O

Monadic STM Implementation (2)

- How to track side-effecting memory actions?
  - Define an STM monad to wrap transactions
  - Based on the state monad; manages side-effects via a **`TVar`** type
    - The **`newTVar`** function takes a value of type a, returns new **`TVar`** to hold the value, wrapped in an STM monad

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

    - **`readTVar`** takes a **`TVar`** and returns an STM context; when performed this returns the value of that **`TVar`**; **`writeTVar`** function takes a **`TVar`** and a value, returns an STM context that can validate the transaction and commit the value to the **`TVar`**
  - The **`atomic {…}`** function operates in an STM context and returns an IO context that performs the operations needed to validate and commit 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
    - I/O prohibited within transactions, since operations in **`atomic {…}`** don't have access to I/O context

Integration into Haskell

- Transactional memory is a good fit with Haskell
  - Pure functions and monads ensure transaction semantics are preserved
  - Side-effects are contained in **`STM`** context of an **`atomic {…}`** block
    - The **`TVar`** implementation is responsible for tracking side effects
    - The **`atomic {…}`** block validates, then commits the transaction (by returning an action to perform in the IO context)
    - A **`TVar`** requires an **`STM`** context, but these are only available in an **`atomic {…}`** block; can't update a **`TVar`** outside a transaction, so can't break atomicity guidelines – Haskell doesn't allow unrestricted heap access via pointers, so can't subvert
  - I/O cannot be performed within an **`atomic {…}`** block
    - The transaction is not in the IO context

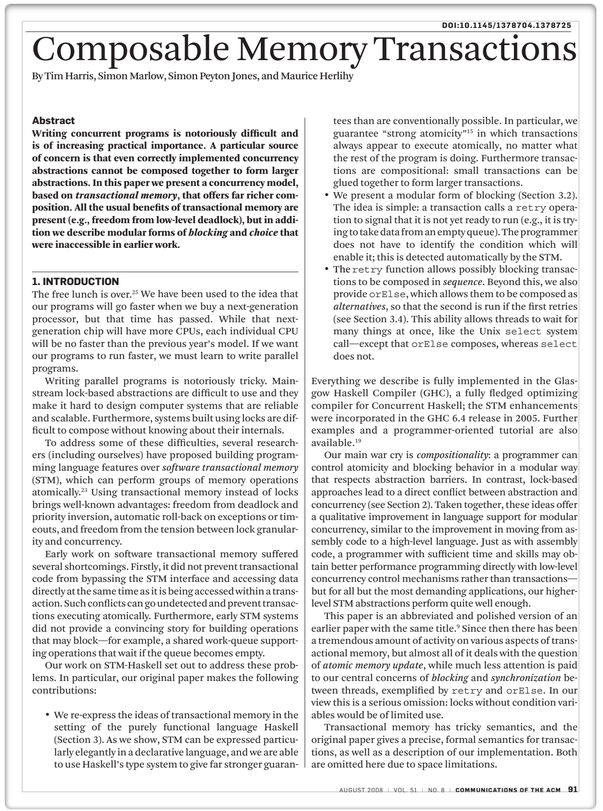
Integration into Other Languages

- Atomic transactions Haskell are very powerful – but rely on the type system to ensure safe composition and retry
- Integration into mainstream languages is difficult
  - Most languages cannot enforce 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 if the transactional approach generalises to other languages

Further Reading

- Is transactional memory a realistic technique?
- Do its requirements for a purely functional language, with controlled 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.

T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, "Composable Memory Transactions", Communications of the ACM, 51(8), August 2008. DOI:10.1145/1378704.1378725.

http://www.cmi.ac.in/~madhavan/courses/pl2009/reading-material/harris-et-al-cacm-2008.pdf

Managing Concurrency Using Transactions

- Programming model
- Integration into Haskell
- Integration into other languages
- Discussion