Managing Concurrency Using Transactions

Advanced Operating Systems
Lecture 13
Lecture Outline

- Transactions for managing concurrency
- 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:
    • **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

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

```plaintext
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 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
  • 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
    - They produce the same answer each time they’re executed
  - 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 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

```java
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 (software transactional memory, STM), or using dedicated hardware to enforce 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 → 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 \( a \) 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

• 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 a new `TVar` to hold the value, wrapped in an STM monad
    - `readTVar` takes a `TVar` and returns an STM monad; when performed this returns the value of that `TVar`; `writeTVar` function takes a `TVar` and a value, and returns an STM monad 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
    - 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 is prohibited within the transaction, since operations in `atomic` {...} don’t have access to the I/O context
Integration into 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 monad 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 monad 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 the IO monad directly
    • There is no IO monad in scope within the transaction, so code requiring one will not compile
    • A TVar requires an STM monad, 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
Integration into Other Languages

• STM in 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 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
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 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?