Using STM for modular concurrency

An industrial experience report on Software Transactional Memory

Duncan Coutts

Haskell Symposium 2020 —

Well-Typed
The Haskell Consultants

Summary

Concurrency is still hard

STM does make it easier

STM enables some useful and interesting concurrency patterns

Design motivation

Overload design and backpressure

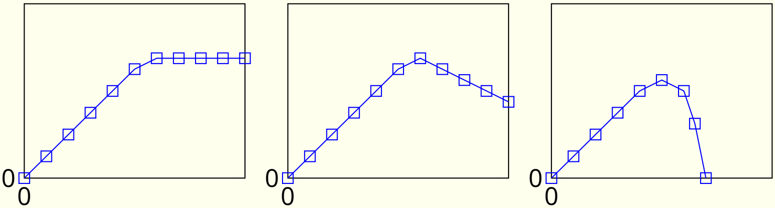
As a consultant I ask:

Q: what is your design for system overload?

Ummm. . .

Q: what does your demand vs throughput curve look like?

The good   The bad   The ugly



Wouldn't it be nice if our basic design patterns gave us good results?

Project context

CARDANO

Commercial context

- a blockchain and a crypto-currency
- a top 10 crypto-currency (by market capitalisation)

Technical context

- a **from-scratch** blockchain implementation in Haskell
- interacting networked nodes, **lots of concurrency**
- design assumption that **‘they really are out to get you’**

Background ideas

Ideas from previous projects working with networking experts

- Queues often make things worse in overload situations and are a source of timing **variability**
- **Pull**-based designs are often better than **push**-based
- Aim for designs that do not become **less efficient** under load
- 'ΔQ' performance algebra as a intellectual framework

Initial design ideas for Cardano

- Exclusively use STM for concurrency
- Aim for a mostly-queueless design
- Worry about worst-case resource consumption, not average-case

An STM refresher

The STM primitives

```
data STM a
instance Monad STM

data TVar

newTVar    :: a → STM (TVar a)
readTVar   :: TVar a → STM a
writeTVar  :: TVar a → a → STM ()

atomically :: STM a → IO a
```

Operations on transactional variables

Concurrent atomic transactions are serialisable

The STM primitives

Blocking is fundamental to communication between threads.

```haskell
retry  :: STM a
orElse :: STM a → STM a → STM a

instance Alternative STM where
  empty = retry
  (<|>) = orElse
```

- Using `retry` we can block on **any condition**, depending on variables we have read.
- Using `orElse` we can block on **alternative** STM actions.

This combination is very flexible and allows modularity.

Blocking on conditions

Using `retry` we can block on **any condition**, depending on variables we have read.

```
do x ← readTVar xv
   guard (p x)    -- uses retry via Alternative's empty
   y ← readTVar yv
   return (x, y)
```

The `retry` suspends the thread until **any** of the variables read up to this point in the transaction are written to by other threads.

The transaction will be re-run **any time** after any variable is written.

Corollary

- Defer reads not needed for blocking conditions.
- No guarantee of observing every change in a variable.

Blocking on alternatives

Using `orElse` we can block on **alternative** STM actions.

$$
\begin{aligned}
\text{firstToFinish} \; = \; & \text{waitForThis} \\
<\!|\!> \; & \text{waitForThat} \\
<\!|\!> \; & \text{waitForTheOther}
\end{aligned}
$$

Each of these can read variables and use conditions.

Allows building up complex conditions in a **modular** way.

Similarities to guarded alternatives from process calculi.

Blocking on all the things!

Most languages and OSs do not have a good unified framework for waiting on any combination of events.
(libraries like `libev` try to paper over the cracks.)

In Haskell, STM **should be** that unified framework

- ✓ inter-thread synchronisation
- ∼ waiting on timeouts
- ∼ waiting on I/O

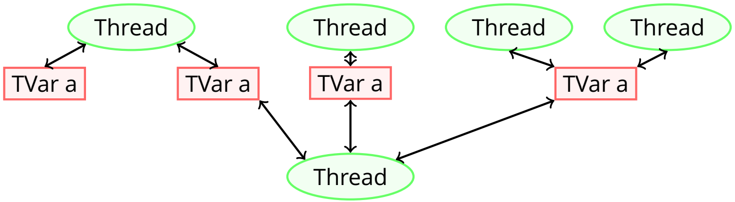
Little-known STM feature to allow waiting on timeouts

```
registerDelay :: Int → IO (TVar Bool)
```

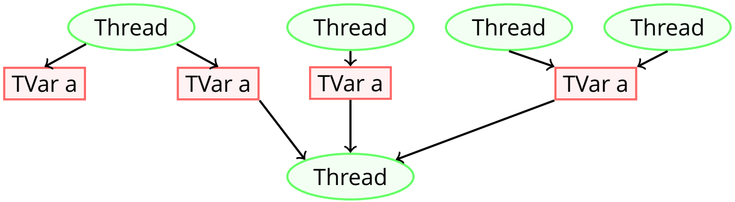
Waiting on I/O needs an extra thread and inter-thread synchronisation

STM based concurrency patterns

Design thought process

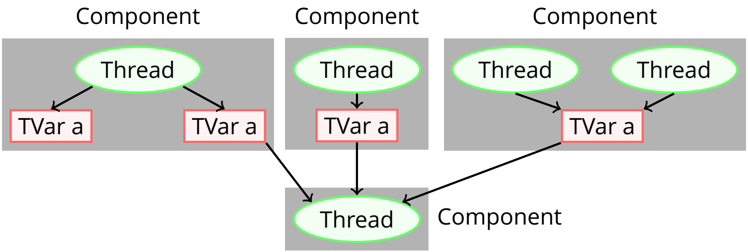


Design thought process



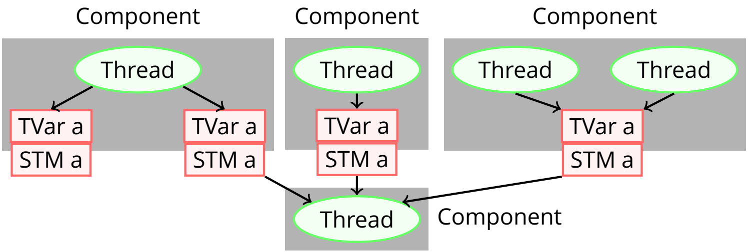
- Unidirectional data flow for each TVar

Design thought process



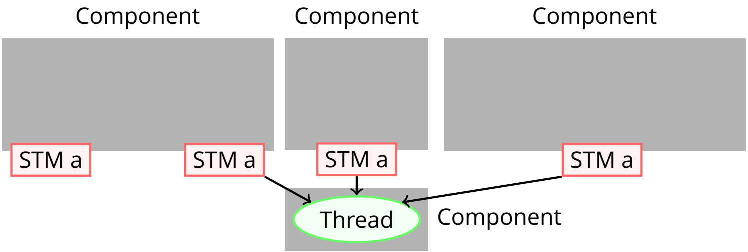
- Unidirectional data flow for each TVar
- Associate TVars with the components that write to them

Design thought process



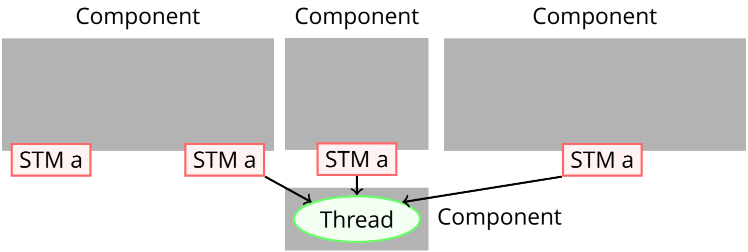
- Unidirectional data flow for each TVar
- Associate TVars with the components that write to them
- Expose `TVar` reads as opaque `STM` queries
  Think of such `STM` queries as time-varying observations

Design thought process



- Unidirectional data flow for each TVar
- Associate TVars with the components that write to them
- Expose `TVar` reads as opaque `STM` queries
  Think of such `STM` queries as time-varying observations
- Does not matter if components are ‘active’ or ‘passive’

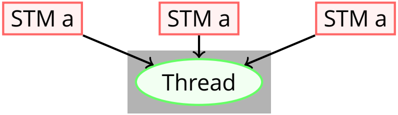
State observation pattern



Expose `STM a` observables for other components

- No need to know about, or coordinate, with consumers
- No need to expose any `TVar` hence read-only
- Preserves abstraction boundaries
- Can read multiple variables and project only public parts
- Example: `TVar (Map Id (TVar X))` exposed as `STM (Map Id X)`

Observing relevant changes



Use combinations of `STM a` observables and act on relevant changes

- No implicit notion of change. It is not a queue of diffs.
- Use an explicit **fingerprint** to identify changes of interest
- Not all changes are relevant
  - read relevant vars;
  - select relevant parts to form the fingerprint.
- May want to read and return extra observations **after** establishing the fingerprint has changed
  - not needed to establish there is a change
  - but used later in acting on the change
- Observe current state, not all intermediate changes.

Observing relevant changes

```
readStateSnapshot fingerprint = do
    -- Read all the trigger state variables
    a ← readA
    b ← readB
    -- Construct the change detection fingerprint
    let fingerprint′ = Fingerprint (f a) (g b)
    -- Check the fingerprint changed, or block and wait until it does
    guard (fingerprint′ ≢ fingerprint)
    -- Read all the non-trigger state variables
    c ← readC
    d ← readD
    -- Construct the overall snapshot of the state
    let stateSnapshot = StateSnapshot a b c d
    return (stateSnapshot, fingerprint′)
```

Acting on the current state

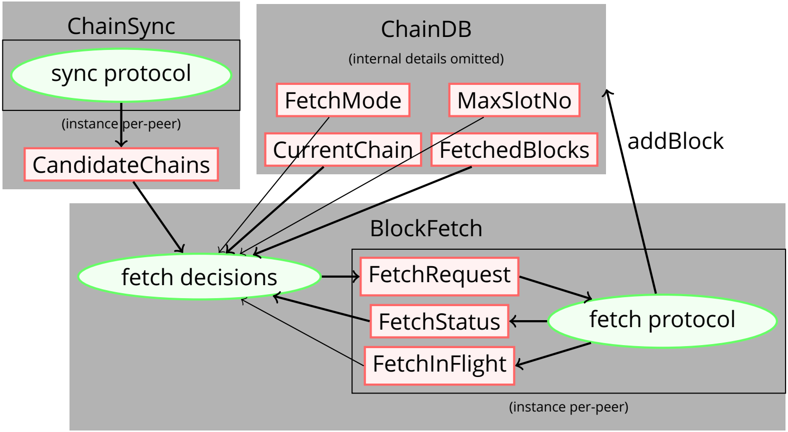
We observe the **current** state, not all intermediate changes.

This encourages a pattern where we act based on the current state.

- Irrespective of how many changes there have been
- Can miss intermediate states if there are frequent changes
- Can become **more efficient** as we get more overloaded

A real example: block fetch

A component for fetching blocks: deciding which ones, and executing



A modular variation on the state observation pattern

The previous example made one big (complicated) decision based on many observables.

Other examples have many possible **alternative** actions.

- each action **guarded** by conditions
- conditions on internal state
- conditions on external observables

Would like some degree of modularity in writing such examples

- perfect use for `orElse` / `(<|>)`

Modular guarded actions

```
loop :: State → IO ()
loop st = do
  Action jobs st′ ← atomically (guardedActions st)
  mapM_ (JobPool.forkJob jobPool) jobs
  loop st′

data Action         = Action (Job Completion) State
type Completion = State → Action

guardedActions :: STM Action
guardedActions st   =   this st
                       <|> that st
                       <|> jobCompletion
  where
    jobCompletion = do
      completion ← JobPool.collect jobPool
      return (completion st)
```

Well-Typed

A real example: Peer-to-Peer control loop

Cardano node's P2P network peer selection control loop

- Internal state tracks 'cold', 'warm' and 'hot' peers
- Targets for numbers of each class
- Actions guarded on internal state only:
    - below target, for each class
    - above target, for each class
    - several of these actions complete asynchronously
- Actions guarded on STM observables:
    - root set of peers changed
    - changed targets
    - connection failures
    - async action completion

Testing

Testing

Concurrency is still hard! Testing is especially important.

Strategy

- deterministic simulation
- property-based testing
- properties over execution traces
- properties via state-machine models

Simulation

Type classes to abstract over selected IO effects

- threads, STM, sync & async exceptions, time, timers
- allows running the **same code in IO and simulation**

```
class (Monad stm, Alternative stm) ⇒ MonadSTMTx stm where
  type TVar stm :: * → *

  newTVar   :: a → stm (TVar stm a)
  readTVar  :: TVar stm a → stm a
  writeTVar :: TVar stm a → a → stm ()
  retry     :: stm a
  orElse    :: stm a → stm a → stm a

class (Monad m, MonadSTMTx (STM m)) ⇒ MonadSTM m where
  type STM m :: * → *

  atomically :: STM m a → m a
```

Simulation

Simulator implementation

- pure & deterministic
- simple thread scheduler
- full STM and async exceptions behaviour
- ‘faster than real-time’ execution for timeouts
- monotonic clock and (adjustable) wall-clock
- produces an execution trace, including custom events

$$\text{runSimTrace} :: \forall a.\ (\forall s.\ \text{SimM s a}) \rightarrow \text{Trace a}$$

$$\text{runSim} :: \forall a.\ (\forall s.\ \text{SimM s a}) \rightarrow \text{Either Failure a}$$

Testing via simulation within Cardano

Many uses of QuickCheck + simulation

- some use state machines
- some use properties over traces

Examples

- file system fault injection for chain database
- simulated full-cluster consensus testing
- protocol performance testing with simulated network delays
- live-lock avoidance in the P2P control loop
  by checking progress within time limits
- planned: clock-skew testing

Conclusions

Use of STM within Cardano

The use of STM within Cardano has been a **clear success**

- Allowed a modular design by appropriate use of concurrency
- Used with explicit (pull-based) protocols for distributed concurrency
- Handles overload well: slows down asking for more work
- Concurrency testing found **lots** of bugs, very few found in production
- Did not hit any STM weak spots
  - no long-running STM transactions
  - no fairness problems
  - no low level performance problems

Contrast with message-passing

Message passing

- push-based
- act on individual change events
- implicit queues
- resource control is implicit (size of queues)
- no natural backpressure

State observation

- pull-based
- act on changed state eventually
- no queues
- resource control is explicit (content of state variables)
- natural backpressure by slowdown

Conclusion

Concurrency is still hard

STM does make it easier

STM enables some useful and interesting concurrency patterns

- A plausible alternative to message-passing for many applications
- Works for internal concurrency
- For distributed concurrency, use in combination with additional patterns, e.g. explicit protocols

Acknowledgements

INPUT | OUTPUT — Marcin Szamotulski and Karl Knutsson

PREDICTABLE NETWORK SOLUTIONS — Neil Davies and Peter Thompson

Well-Typed — Edsko de Vries and Thomas Winant