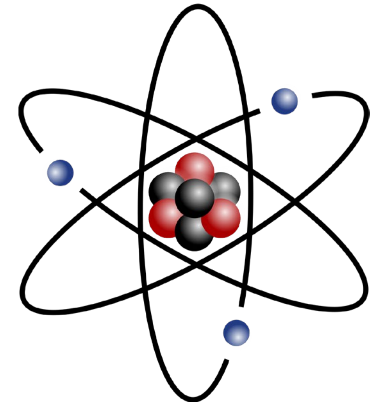
Composable Memory Transactions in Haskell

Parallel Functional Programming

John Hughes

Recall Race Conditions…

x := x + 1 || x := x + 1



0

1 **1**

Transaction

”a group of related actions that need to be performed as a single action”

- **Single global lock**
  Inefficient!

- **Multiple locks**
  Inefficient!
  Error prone!

Database solution: optimistic concurrency

- Transactions are often *independent*
- Run them in parallel, and *detect conflicts*
  - A conflicted transaction is automatically retried until it succeeds

Software Transactional Memory

- Optimistic concurrency for updates to memory

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '05). Association for Computing Machinery, New York, NY, USA, 48-60

Composable Memory Transactions

Tim Harris Simon Marlow Simon Peyton Jones Maurice Herlihy

Microsoft Research
7 J J Thomson Avenue, Cambridge, UK, CB3 0FB
{tharris,simonmar,simonpj,t-maherl}@microsoft.com

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 new concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g. freedom from deadlock), but in addition we describe new modular forms of *blocking* and *choice* that have been inaccessible in earlier work.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming – Parallel programming

General Terms: Algorithms, Languages

Keywords: Non-blocking algorithms, locks, transactions

1. INTRODUCTION

Concurrent programming is notoriously tricky. Current lock-based abstractions are difficult to use and 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.

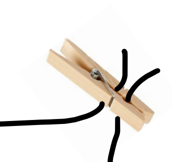
- We re-express the ideas of transactional memory in the setting of Concurrent Haskell (Section 3). This is much more than a routine "port" into a new setting. As we show, STM can be expressed particularly elegantly in a declarative language, and we are able to use Haskell's type system to give far stronger guarantees than are conventionally possible. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.

- We present a new, modular form of blocking, which appears to the programmer as a simple function called `retry` (Section 3.2). Unlike most existing approaches, the programmer does not have to ide... the condition under which the transaction ...omple- tion: `retry` can occu... tion, blocking it until ... becomes possible.

- The `retry` functi... tions to be composed ... provide `orElse`, which allow... *alternatives*, so that the secon... (Section 3.4). This ability al...s threads to wait for many things at once, like the Unix `select` system call – except that `orElse` composes well, whereas `select`

1112
citations

STM in Haskell



x := x + 1

IO Monad	**STM Monad**
x :: IORef Int	x :: TVar Int
do n <- readIORef x writeIORef x (n+1)	**do** n <- readTVar x writeTVar x (n+1)

atomically

STM Types

- newIORef :: a -> **IO** (IORef a)
- readIORef :: IORef a -> **IO** a
- writeIORef :: IORef a -> a -> **IO** ()

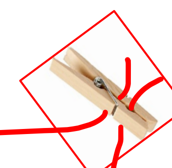
*put together into **IO** actions*

- forkIO :: **IO** () -> **IO** ThreadId

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

*put together into **STM** actions*

- atomically :: **STM** a -> **IO** a



x := x + 1

Example: transferring money

```
newtype Account = Account (TVar Int)

transfer n (Account a) (Account b) = do
   modifyTVar' b (+n)
   modifyTVar' a (+(-n))
```

add n *to contents of* b

evaluate the contents of the reference

transfer :: Int -> Account -> Account -> **STM** ()

Example: exchanging money

```
atomically $ do
  transfer 1000  aSEK  bSEK
  transfer 97    bUSD aUSD
```

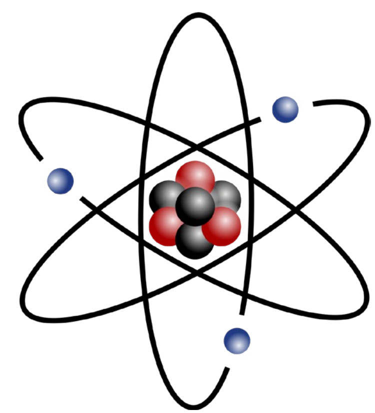
STM

IO

Implementation

aSEK	*4000*
bSEK	*21000*
aUSD	*97*
bUSD	*1903*

Log
bSEK = 20000
bSEK := 21000
aSEK = 5000
aSEK := 4000
bUSD = 2000
bUSD := 1903
aUSD = 0
aUSD := 97



Example: exchanging money

```
atomically $ do
    transfer 1000  aSEK  bSEK
    transfer 97      bUSD aUSD
```

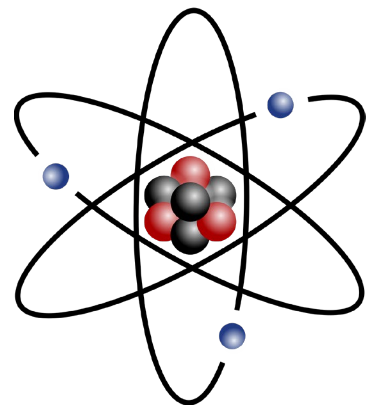
STM

IO

Implementation

aSEK	**6000**
bSEK	20000
aUSD	0
bUSD	2000

Log
bSEK = 20000
bSEK := 21000
aSEK = 5000
aSEK := 4000
bUSD = 2000
bUSD := 1903
aUSD = 0
aUSD := 97



Blocking transactions

```
newtype Account = Account (TVar Int)

transfer n (Account a) (Account b) = do
  m <- readTVar a
  if m<n then retry else do
    modifyTVar' b (+n)
    writeTVar a (m-n)
```

*Discard this execution & **block** the transaction to be retried later…*

…once one of the variables read in the log changes

Example: when would this block?

```
atomically $ do
  transfer 1000  aSEK  bSEK
  transfer 97    bUSD  aUSD
```

If bUSD **contains too little, *when is the transaction retried?***

Example: paying from two accounts

```
atomically $ do
   transfer 500   aSEK  bSEK
   transfer 500   cSEK  bSEK
   transfer 97    bUSD aUSD
```

Example: paying from one of two accounts

```
atomically $ do
  transfer 1000  aSEK bSEK `orElse` transfer 1000  cSEK bSEK
  transfer 97     bUSD aUSD
```

Do the first, but if it retries, then

- *Undo its writes*
- *Try the second*
- *If that retries too, wake up when* ***any*** *Tvar read changes*

`orElse` generalises **`select`**

```
wait for A…
`orElse`
wait for B…
`orElse`
wait for C…
…
```

*Wait for **any** combination of events, not just a file descriptor*

Algebra of **`` `orElse` ``**

- **``(`orElse`)``** is associative, but *not* commutative

```
return 1 `orElse` return 2 /=
return 2 `orElse` return 1
```

- **``retry `orElse` m``**

  = **`m`**

  = **``m `orElse` retry``**

Algebra of `retry`

- `retry >>= f` = `retry`

  Nothing after a `retry` matters

- `m >>= (\x -> retry)` = `retry`

  Nothing *before* a `retry` matters either

  (except to determine when a transaction needs rescheduling)

STM can simulate other primitives

- E.g. one-place buffer

new type, same representation

```
newtype Buffer a = Buffer (TVar (Maybe a))

newBuffer = Buffer <$> newTVar Nothing
```

STM can simulate other primitives

- E.g. one-place buffer

```
send (Buffer t) a = do
  m <- readTVar t
  case m of
    Nothing -> do
      writeTVar t (Just a)
      return a
    Just _ ->
      retry
```

STM can simulate other primitives

- E.g. one-place buffer

```
receive (Buffer t) = do
  m <- readTVar t
  case m of
    Nothing ->
      retry
    Just a -> do
      writeTVar t Nothing
      return a
```

What does this do?

```
do
  buf1 <- atomically newBuffer
  buf2 <- atomically newBuffer
  atomically (send buf1 "hello")
  atomically (send buf2 "goodbye")
  x <- atomically (send buf1 "hohoho”
          `orElse` receive buf2)
  y <- atomically (receive buf1
          `orElse` receive buf2)
  return (x,y)
```

What does this do?

```
atomically $ do
    transfer 25 wallet cokeMachine
    Just <$> receive cokeMachineSlot
  `orElse`
    return Nothing
```

Example: Futures

```
newtype Future a = Future (TVar (Maybe a))
```

start a task whose result will be available in the future

```
future io = do
  v <- atomically $ newTVar Nothing
  forkIO $ do a <- io
              atomically $ writeTVar v (Just a)
  return (Future v)
```

the future is now

```
await (Future t) =
  atomically $ do m <- readTVar t
                  case m of
                    Nothing -> retry
                    Just a  -> return a
```

Example: Snapshotting a state

```
snapShot :: Eq f => STM s -> (s -> f) -> IO (TVar s)
snapShot getState fingerprint = do
  s <- atomically $ getState
  v <- atomically $ newTVar s
  forkIO $ loop v (fingerprint s)
  return v
  where loop v print = do
     print' <- atomically $ do
       s' <- getState
       let print' = fingerprint s'
       if print'==print then retry else do
         writeTVar v s'
         return print'
     loop v print'
```

Benchmark: binary trees

strict—to simplify benchmarking

```
data Tree k v = Leaf
              | Branch !(Tree k v) k v !(Tree k v)
  deriving (Eq, Show, Generic)

find k Leaf = Nothing
find k (Branch l k' v' r)
  | k < k' = find k l
  | k ==k' = Just v'
  | k > k' = find k r

insert k v Leaf = Branch Leaf k v Leaf
insert k v (Branch l k' v' r)
  | k < k' = Branch (insert k v l) k' v' r
  | k ==k' = Branch l k v r
  | k > k' = Branch l k' v' (insert k v r)
```

Benchmark: binary trees

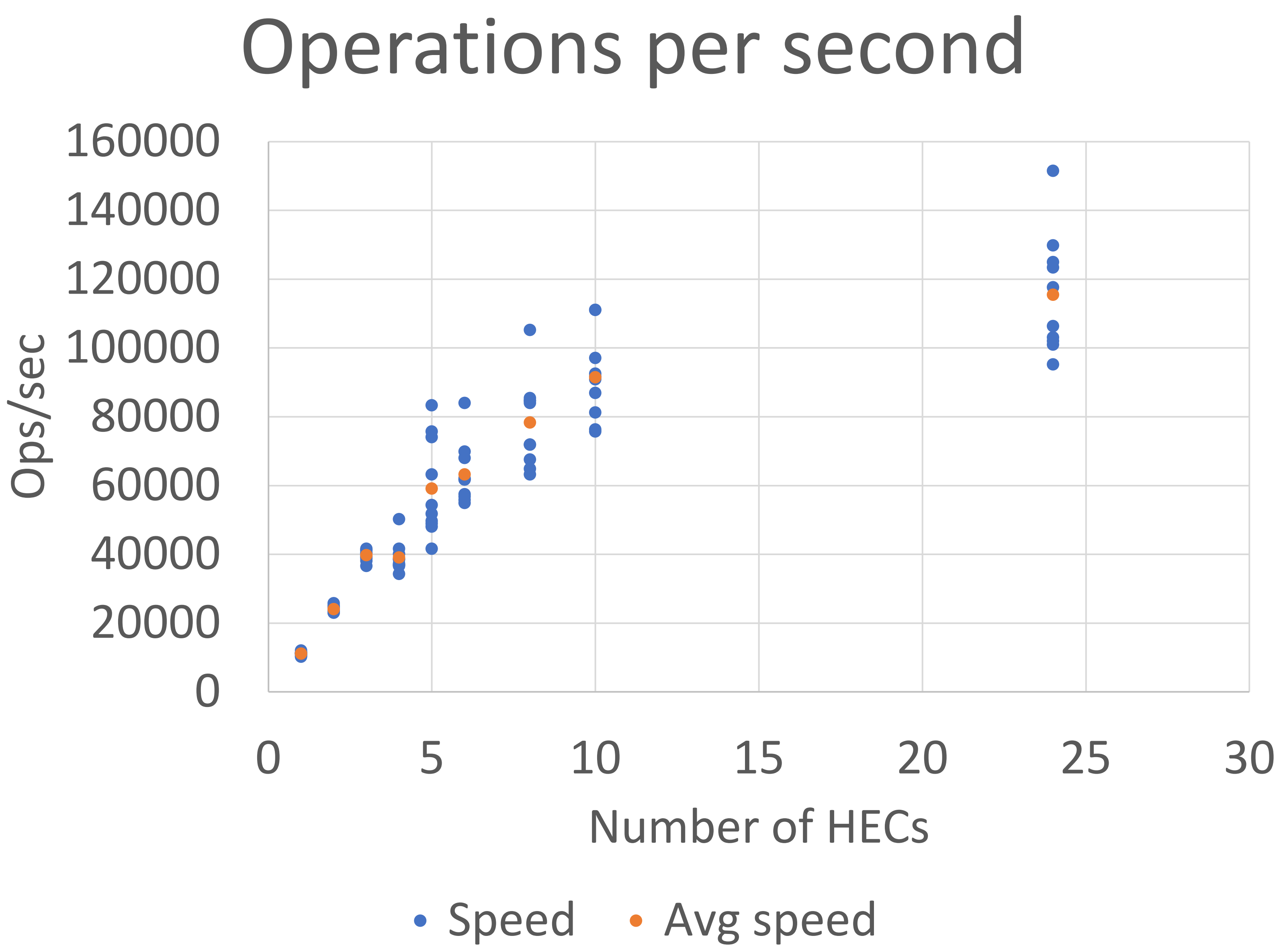
```
delete k Leaf = Leaf
delete k (Branch l k' v' r)
  | k < k' = Branch (delete k l) k' v' r
  | k ==k' = merge l r
  | k > k' = Branch l k' v' (delete k r)

merge Leaf t = t
merge t Leaf = t
merge (Branch l k v r) (Branch l' k' v' r') =
  Branch l k v (Branch (merge r l') k' v' r')
```

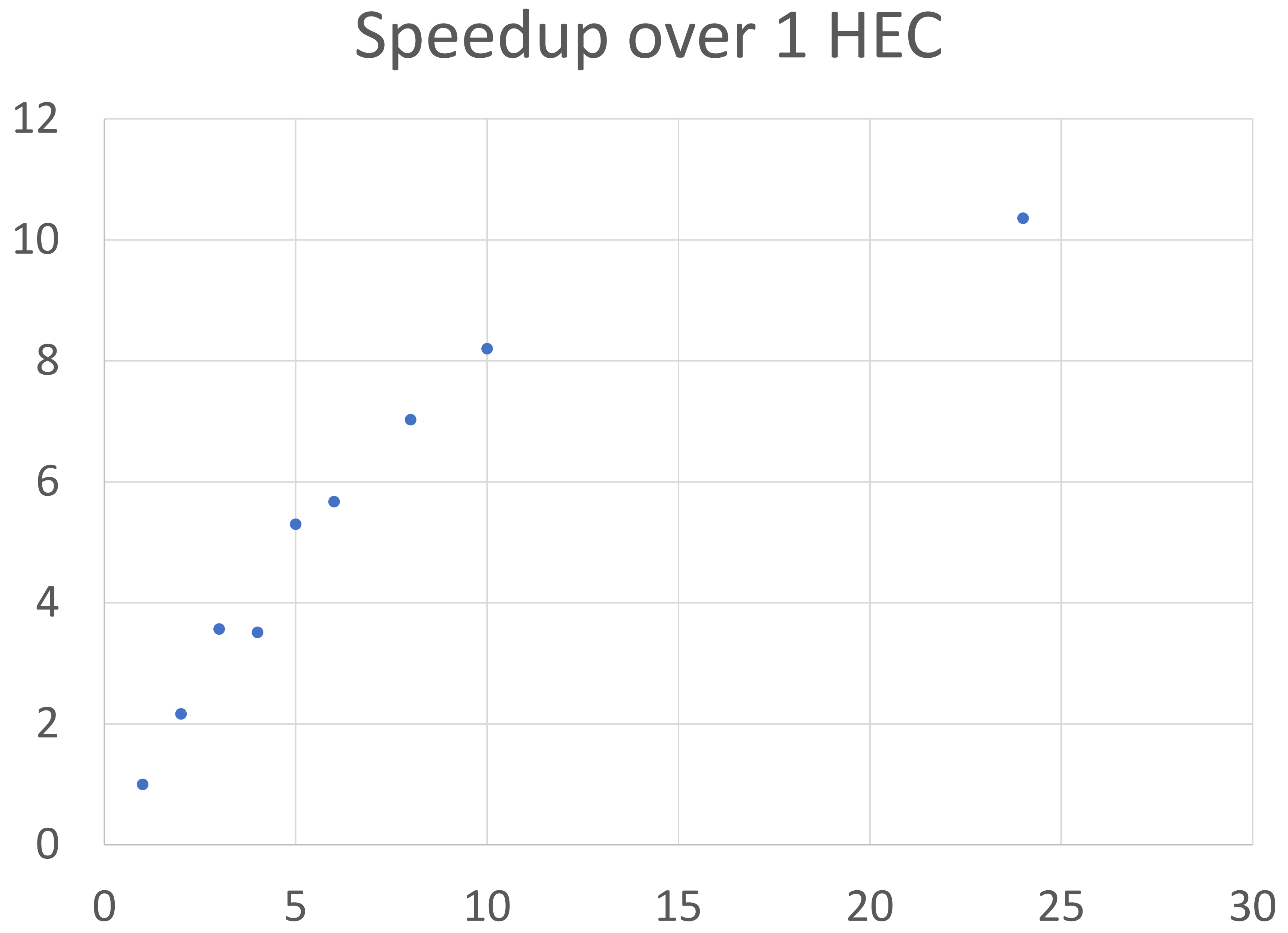
Simplest imperative version

```
newTree = atomically $ newTVar Leaf

insertI t k v = atomically $
                  modifyTVar' t (insert k v)

deleteI t k   = atomically $
                  modifyTVar' t (delete k)

findI   t k   = find k <$> atomically $
                  readTVar t
```

Benchmark

- 500,000 random operations
  (all performed in parallel)
- find::insert::delete is 1::1::1
- Keys selected uniformly from the range 1..1000
  (each key appears ~500 times)
- Total Haskell run-time is the measure
- Average of 10 runs with each number-of-cores
- Hardware: core i9 with 16 cores (8+8)/24 threads


Operations per second
Ops/sec
160000
140000
120000
100000
80000
60000
40000
20000
0
0
5
10
15
20
25
30
Number of HECs
Speed
Avg speed



Speedup over 1 HEC
12
10
8
6
4
2
0
0
5
10
15
20
25
30

A transactional version

```
type TTree k v = TVar (TNode k v)

data TNode k v = TLeaf
               | TBranch (TTree k v) k v (TTree k v)


findT t k = do
  n <- readTVar t
  case n of
    TLeaf ->
      return Nothing
    TBranch l k' v' r
      | k < k' -> findT l k
      | k ==k' -> return (Just v')
      | k > k' -> findT r k
```

```haskell
insertT t k v = do
  n <- readTVar t
  case n of
    TLeaf -> do
      l <- newTVar TLeaf
      r <- newTVar TLeaf
      writeTVar t (TBranch l k v r)
    TBranch l k' v' r
      | k < k' -> insertT l k v
      | k ==k' -> writeTVar t (TBranch l k v r)
      | k > k' -> insertT r k v
```

```
deleteT t k = do
  n <- readTVar t
  case n of
    TLeaf ->
      return ()
    TBranch l k' v' r
      | k < k' -> deleteT l k
      | k ==k' -> mergeT t l r
      | k > k' -> deleteT r k
```

```
mergeT dest l r = do
  ln <- readTVar l
  rn <- readTVar r
  case ln of
    TLeaf ->
      writeTVar dest rn
    TBranch ll lk lv lr ->
      case rn of
        TLeaf ->
          writeTVar dest ln
   TBranch rl rk rv rr -> do
     writeTVar dest (TBranch ll lk lv l)
     writeTVar l    (TBranch r rk rv rr)
     mergeT r lr rl
```

```
merge (Branch l k v r) (Branch l' k' v' r') =
  Branch l k v (Branch (merge r l') k' v' r')
```

Does this work?

List of 'actions'

corresponding pure Tree

TVar tree

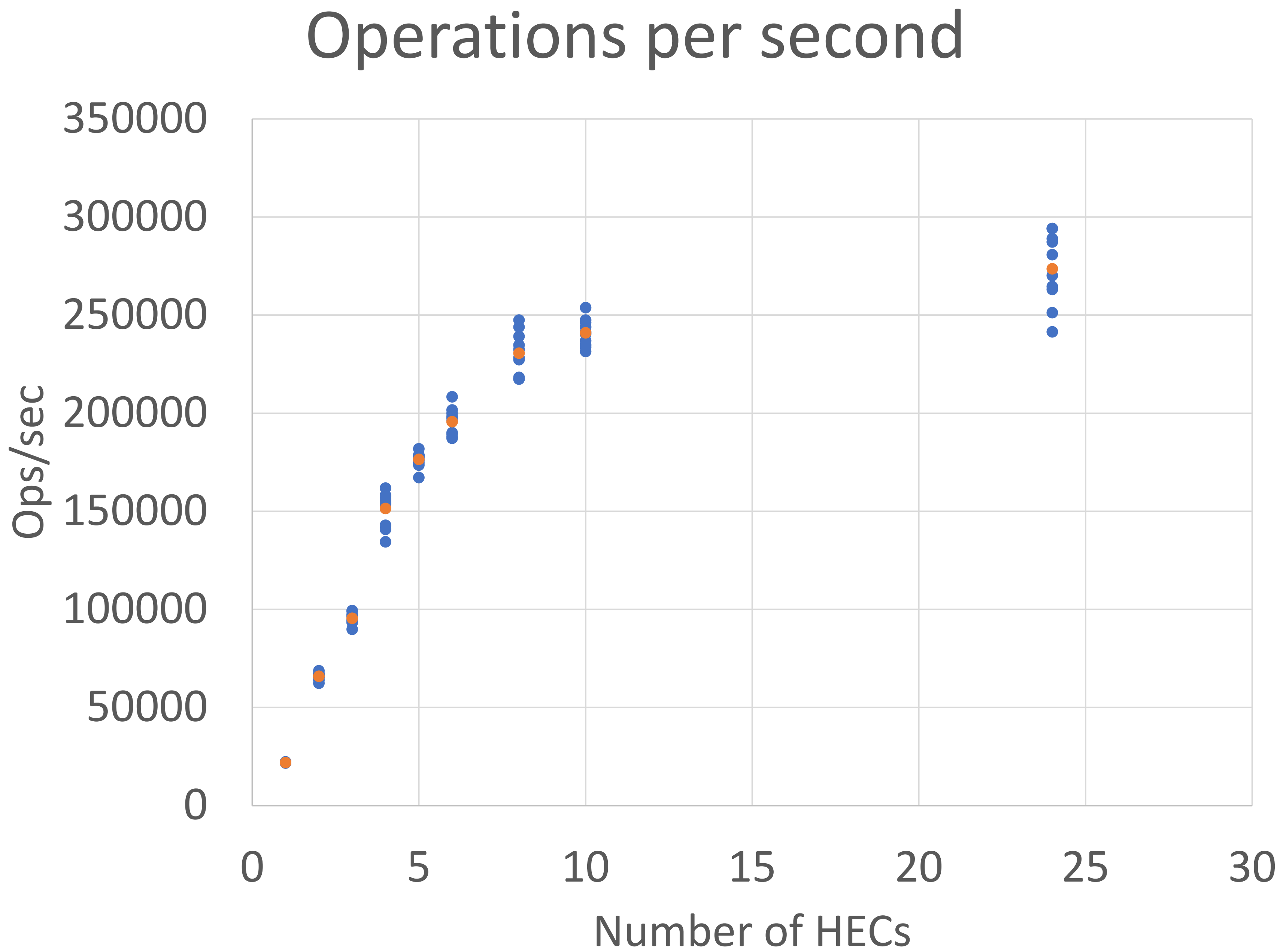
convert a TVar tree into a pure one

```
prop_TTree acts = ioProperty $ do
  t <- atomically $ newTVar TLeaf
  agrees t Leaf acts
  where agrees t t' [] = do
          t_ <- atomically $ treeOf t
          return (t_ === t')
        agrees t t' (Find k:acts) = do
          v <- atomically $ findT t k
          if v==find k t'
          then agrees t t' acts
          else return (v===find k t')
        agrees t t' (Insert k v:acts) = do
          atomically $ insertT t k v
          agrees t (insert k v t') acts
        agrees t t' (Delete k:acts) = do
          atomically $ deleteT t k
          agrees t (delete k t') acts
```

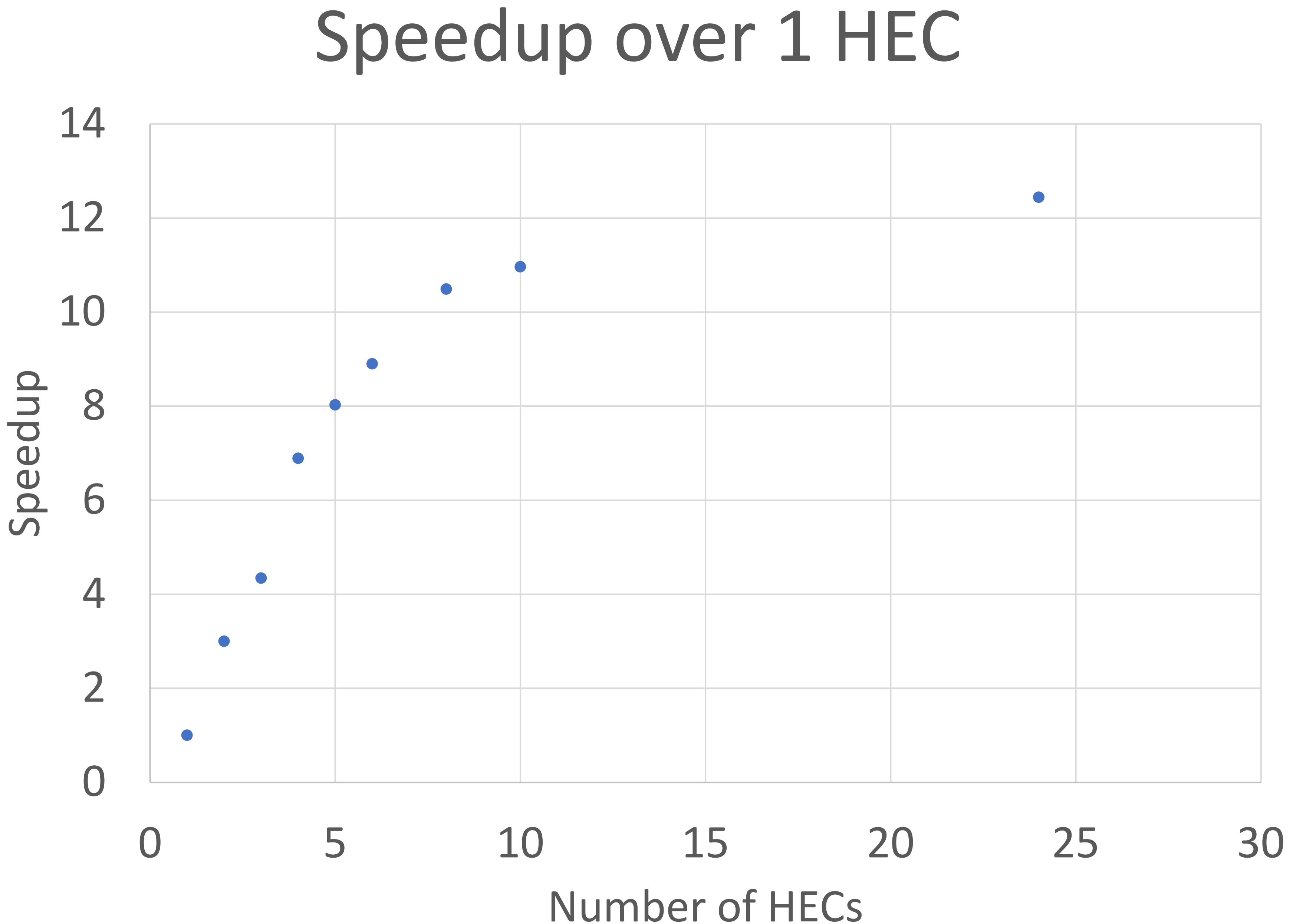
One bug...

```
*Main> quickCheck prop_TTree
*** Failed! Falsified (after 22 tests and 11 shrinks):
[Insert 0 0,Insert 1 0,Delete 1]
Branch Leaf 0 0 (Branch Leaf 1 0 Leaf)
  /= Branch Leaf 0 0 Leaf
```

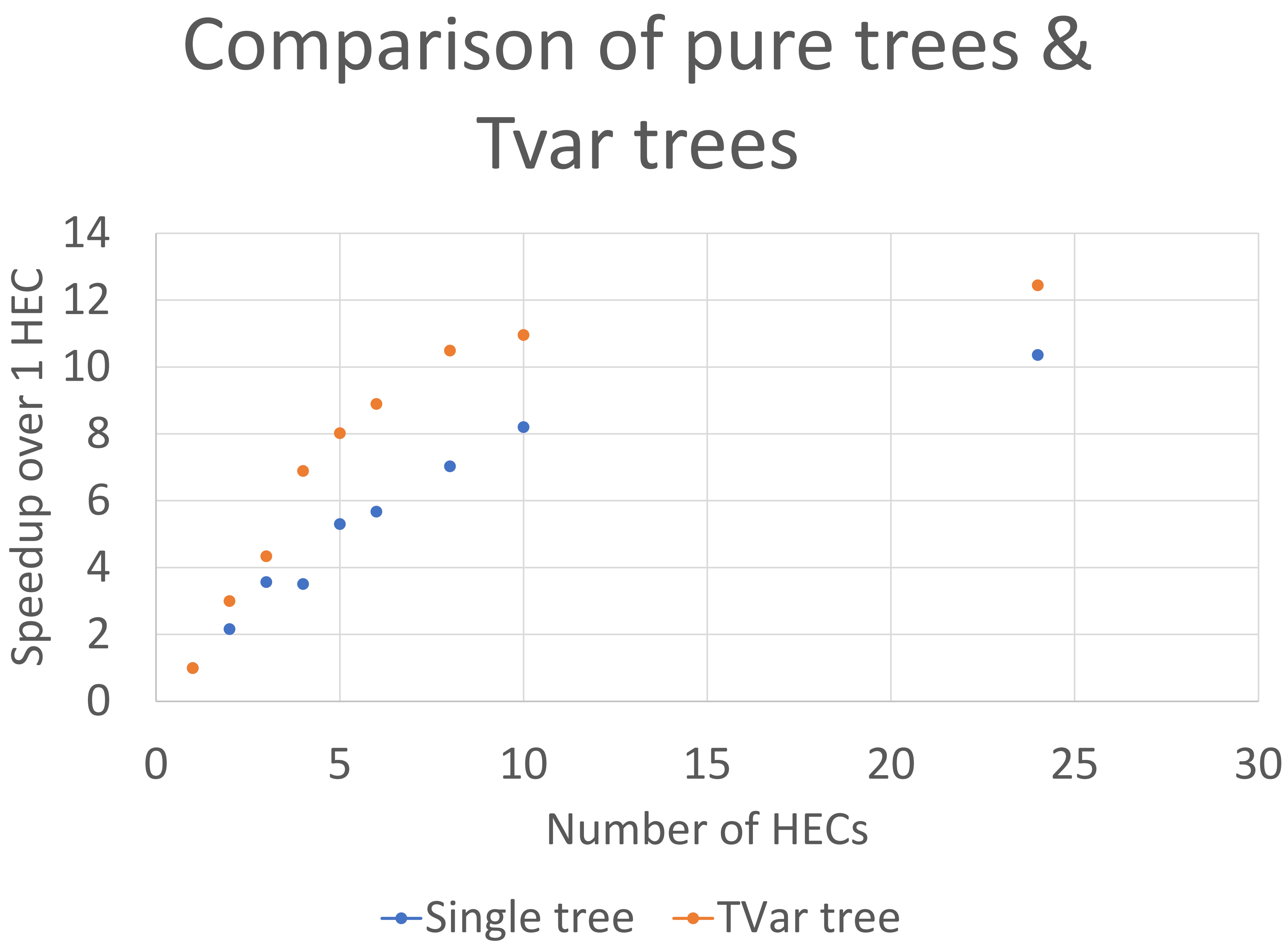
```
deleteT t k = do
  n <- readTVar t
  case n of
    TLeaf ->
      return ()
    TBranch l k' v' r
      | k < k' -> deleteT l k'
      | k ==k' -> mergeT t l r
      | k > k' -> deleteT r k'
```



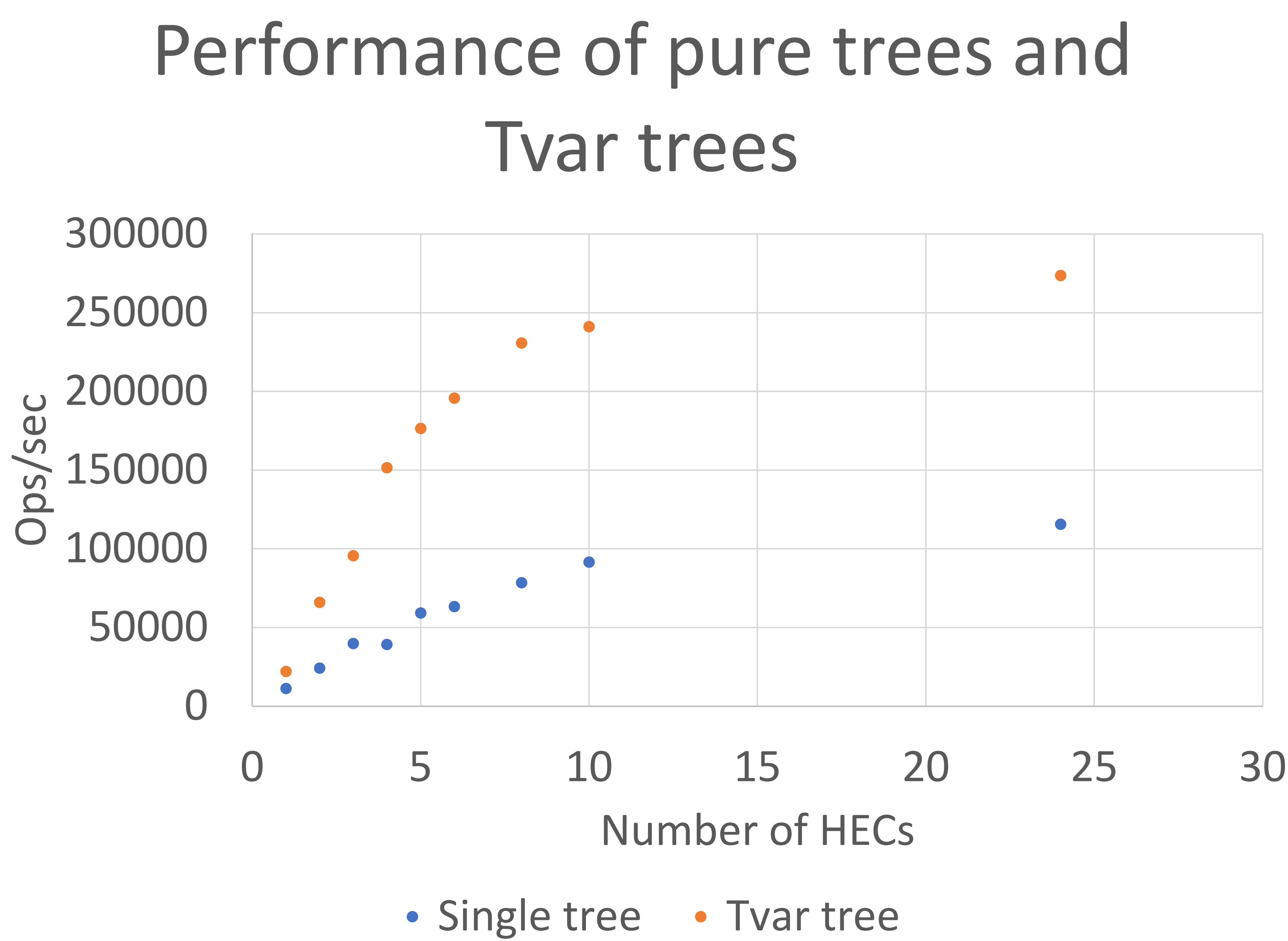
Operations per second
Ops/sec
Number of HECs
0
50000
100000
150000
200000
250000
300000
350000
0
5
10
15
20
25
30



Speedup over 1 HEC
Speedup
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
Number of HECs

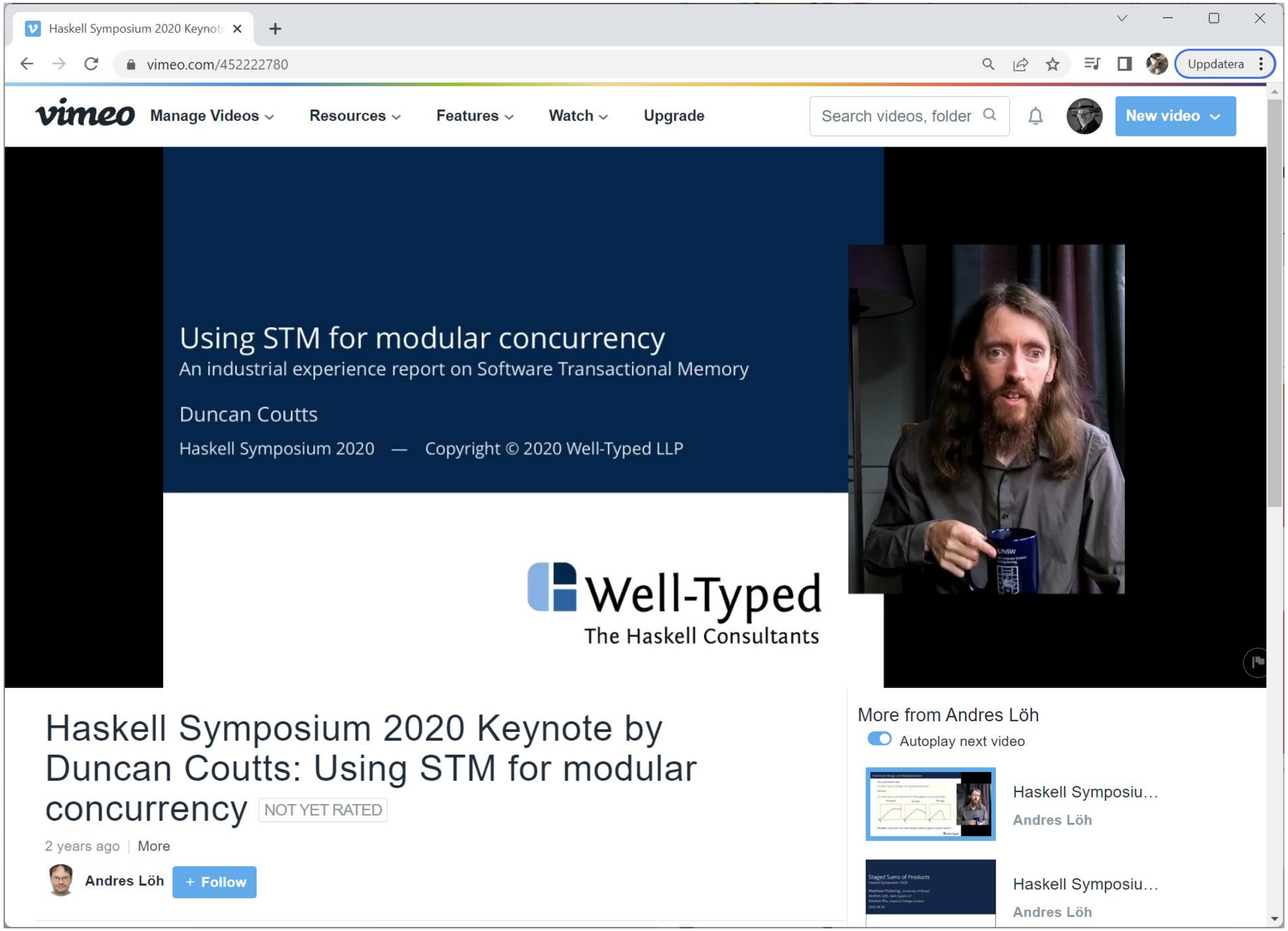


Comparison of pure trees & Tvar trees
Speedup over 1 HEC
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
Number of HECs
Single tree
TVar tree



Performance of pure trees and Tvar trees
Ops/sec
300000
250000
200000
150000
100000
50000
0
0
5
10
15
20
25
30
Number of HECs
Single tree
Tvar tree

System Design with STM



Overload design and backpressure

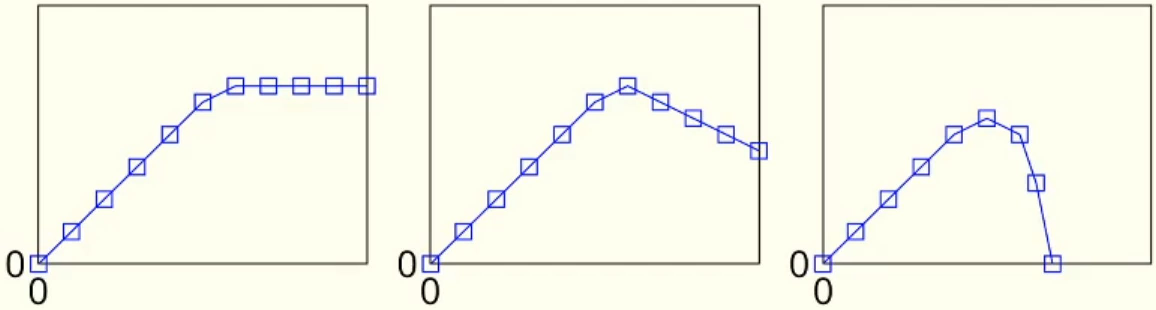
As a consultant I ask:

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

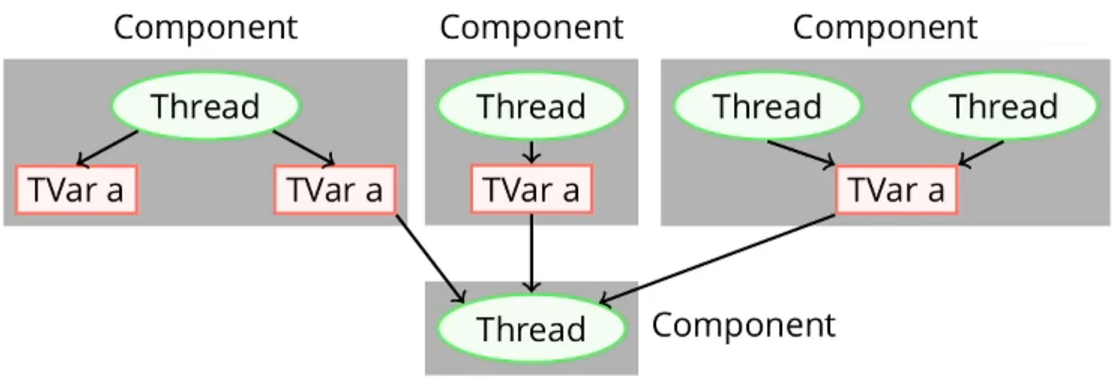
- a **from-scratch** blockchain implementation in Haskell
- interacting networked nodes, **lots of concurrency**
- design assumption that **'they really are out to get you'**

Background ideas

Ideas from previous projects working with networking experts

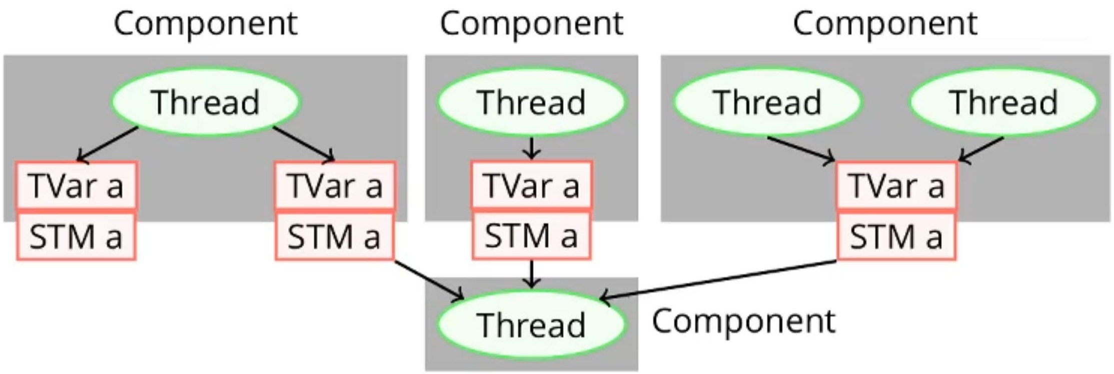
- Queues often make things worse in overload situations and are a source of timing **variability**
- **Pull**-based designs are often better than **push**-based
- Aim for designs that do not become **less efficient** under load

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

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

Well-Typed

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

Well-Typed

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

Well-Typed

Haskell + STM =  ?

- STM is available for
  - Scala, Rust, .NET, the JVM, Go, C, C++…

- Haskell uses types to *guarantee* transactions only use TVars.

- Most Haskell memory accesses are *not* to Tvars, and need not be `transactionalized'.

Summary

- STM is an approach to *concurrent programming* that reduces pain
- STM is *compositional*; transactions can be combined in arbitrary ways
- STM is *expressive*; many other concurrency primitives can be expressed in terms of STM
- STM *performs well*; it can help reduce contention
- STM enables ’state observation design’, with built-in ability to withstand overload.