The Problem

• GHC has rich support for concurrency & parallelism:
  – Lightweight threads (*fast*)
  – Transparent scaling on a multiprocessor
  – STM
  – par/seq
  – Multithreaded FFI
  – Asynchronous exceptions

• But...
The Problem

• ... it is *inflexible*.
  – The implementation is entirely in the runtime
  – Written in C
  – Modifying the implementation is hard: it is built using OS threads, locks and condition variables.
  – Can only be updated with a GHC release
why do we care?

• The concurrency landscape is changing.
  – New abstractions are emerging; e.g. we might want to experiment with variants of STM
  – We might want to experiment with scheduling policies: e.g. STM-aware scheduling, or load-balancing algorithms
  – Our scheduler doesn’t support everything: it lacks priorities, thread hierarchies/groups
  – Certain applications might benefit from application-specific scheduling
  – For running the RTS on bare hardware, we want a new scheduler
The Idea

Haskell Code

forkIO, MVar, STM, ...

RTS

Haskell Code

forkIO, MVar, STM, ...

UltimateConcurrency™

???

RTS
What is ???

• We call it the *substrate interface*
• The Rules of the Game:
  – as small as possible: mechanism, not policy
  – We must have lightweight threads
  – Scheduling, “threads”, blocking, communication, CPU affinity etc. are the business of the library
  – The RTS provides:
    • GC
    • multi-CPU execution
    • stack management
  – Must be enough to allow GHC’s concurrency support to be implemented as a library
The substrate

-------- (1) Primitive Transaction Memory

data PTM a
data PVar a
instance Monad PTM
newPVar :: a -> PTM (PVar a)
readPVar :: PVar a -> PTM a
writePVar :: PVar a -> a -> PTM ()
catchPTM :: PTM a -> (Exception->PTM a)
         -> PTM a
atomicPTM :: PTM a -> IO a

-------- (2) Haskell Execution Context

data HEC
instance Eq HEC
instance Ord HEC
getHEC :: PTM HEC
waitCond :: PTM (Maybe a) -> IO a
wakeupHEC :: HEC -> IO a

-------- (3) Stack Continuation

data SCont
newSCont :: IO () -> IO SCont
switch :: (SCont -> PTM SCont) -> IO ()

-------- (4) Thread Local States

data TLSKey a
newTLSKey :: a -> IO (TLSKey a)
getTLS :: TLSKey a -> PTM a
setTLS :: TLSKey a -> a -> IO ()
initTLS :: SCont -> TLSKey a -> a
         -> IO ()

-------- (5) Asynchronous Exceptions

raiseAsync :: Exception -> IO ()
deliverAsync :: SCont -> Exception
            -> IO ()

-------- (6) Callbacks

rtsInitHandler :: IO ()
inCallHandler :: IO a -> IO a
outCallHandler :: IO a -> IO a
timerHandler :: IO ()
blockedHandler :: IO Bool -> IO ()
In the beginning...

Haskell

外国导出函数
“haskell_main”
main :: IO ()
main = do ...

C

...haskell_main()
...
Haskell execution context

- Haskell code executes inside a HEC
- HEC = OS thread (or CPU) + state needed to run Haskell code
  - Virtual machine state
  - Allocation area, etc.

```haskell
data HEC
instance Eq HEC
instance Ord HEC
getHEC :: PTM HEC
```

- A HEC is created by (and only by) a foreign in-call.
- Where is the scheduler? I’ll come back to that.
Synchronisation

- There may be multiple HECs running simultaneously. They need a way to synchronise access to shared data: scheduler data structures, for example.
- Use locks & condition variables?
  - Too hard to program with
  - Bad interaction with laziness:

```
do { takeLock lk
    ; rq <- read readyQueueVar
    ; rq' <- if null rq then ...
        else ...
    ; write readyQueueVar rq'
    ; releaseLock lk }
```
  - (MVars have this problem already)
PTM

• Transactional memory?
  – A better programming model: compositional
  – Sidesteps the problem with laziness: a transaction holds no locks while executing
  – We don’t need *blocking* at this level (STM’s retry)

```haskell
data PTM a
data PVar a
instance Monad PTM
newPVar :: a -> PTM (PVar a)
readPVar :: PVar a -> PTM a
writePVar :: PVar a -> a -> PTM ()
catchPTM :: PTM a -> (Exception -> PTM a) -> PTM a
atomicPTM :: PTM a -> IO a
```
Stack continuations

- Primitive threads: the RTS provides multiple *stacks*, and a way to switch execution from one to another.

```
data SCont
newSCont :: IO () -> IO SCont
switch     :: (SCont -> PTM SCont) -> IO ()
```

Switches control to a new stack. Can decide not to switch, by returning the current stack.

PTM very important!
Stack Continuations

• Stack continuations are *cheap*
• Implementation: just a stack object and a stack pointer.
• Using a stack continuation multiple times is an (un)checked runtime error.
• If we want to check that an SCont is not used multiple times, need a separate object.
Putting it together: a simple scheduler

Design a scheduler supporting threads, cooperative scheduling and MVars.

```haskell
runQueue :: [SCont]
runQueue <- newPVar []

addToRunQueue :: SCont -> PTM ()
addToRunQueue sc = do
    q <- readPVar runQueue
    writePVar runQueue (q++[sc])

data ThreadId = ThreadId SCont

forkIO :: IO () -> IO ThreadId
forkIO action = do
    sc <- newSCont action
    atomicPTM (addToRunQueue sc)
    return (ThreadId sc)
```
yield

• Voluntarily switches to the next thread on the run queue

```haskell
popRunQueue :: IO SCont
popRunQueue = do
  scs <- readPVar runQueue
  case scs of
    [] -> error "deadlock!"
    (sc:scs) -> do
      writePVar runQueue scs
      return sc

yield :: IO ()
yield =
  switch $ \sc -> do
    addToRunQueue sc
    popRunQueue
```
MVar: simple communication

- MVar is the original communication abstraction from Concurrent Haskell

```haskell
data MVar a
 takeMVar :: MVar a -> IO a
 putMVar  :: MVar a -> a -> IO ()
```

- takeMVar blocks if the MVar is empty
- takeMVar is fair (FIFO), and single-wakeup
- resp. putMVar
data MVar a = MVar (PVar (MVState a))
data MVState a = Full a [(a,    SCont)]
  | Empty  [(PVar a, SCont)]

takeMVar :: MVar a -> IO a
takeMVar (MVar mv) = do
  buf ⟨ atomicPTM $ newPVar undefined
  switch $ \
c    -> do
  state ⟨ readPVar mv
  case state of
    Full x []  -> do
      writePVar mv $ Empty []
      writePVar buf x
      return c
    Full x l@((y,wakeup):ts)  -> do
      writePVar mv $ Full y ts
      writePVar buf x
      addToRunQueue wakeup
      return c
    Empty ts  -> do
      writePVar mv $ Empty (ts++[(buf,c)])
      popRunQueue
      atomicPTM $ readPVar buf

This will hold the result

MVar is full, no other

MVar is full, there are other
threads waiting to put.
Wake up one thread and
return.

MVar is empty: add this
thread to the end of the
queue, and yield.
When switch returns, buf
will contain the value we
read.
PTM Wins

• This implementation of takeMVar still works in a multiprocessor setting!
• The tricky case:
  – one CPU is in takeMVar, about to sleep, putting the current thread on the queue
  – another CPU is in putMVar, taking the thread off the queue and running it
  – but switch hasn’t returned yet: the thread is not ready to run. BANG!
• This problem crops up in many guises. Existing runtimes solve it with careful use of locks, e.g. a lock on the thread, or on the queue, not released until the last minute (GHC). Another solution is to have a flag on the thread indicating whether it is ready to run (CML).
• With PTM and switch this problem just doesn’t exist: when switch’s transaction commits, the thread is ready to run.
Semantics

• The substrate interface has an operational semantics (see paper)

\[
\begin{align*}
  & s \text{ fresh} \quad M \, s; \Theta; \emptyset; \xrightarrow{\star_{D, h}} \text{return } s'; \Theta'[s' \mapsto (M', D')] \quad s \neq s' \\
  \hline
  & S \mid (E[\text{switch } M], D, h); \Theta \quad \xrightarrow{} \quad S \mid (M', D', h); \Theta'[s \mapsto (E[\text{return }()], D)]
\end{align*}
\]

(Switch)

• Now to flesh out the design...
Pre-emption

- The concurrency library should provide a `callback handler`:

```haskell
timerHandler :: IO ()
```

- the RTS causes each executing HEC to invoke `timerHandler` at regular intervals.

- We can use this in our simple scheduler to get pre-emption:

```haskell
timerHandler :: IO ()
timerHandler = yield
```
Thunks

• If two HECs are evaluating the same thunk (suspension), the RTS may decide to suspend one of them\(^1\)

• The current RTS keeps a list of threads blocked on thunks, and periodically checks whether any can be awakened.

• The substrate provides another callback:

```
blockedHandler :: IO Bool -> IO ()
```

• Simplest implementation:

```
blockedHandler :: IO ()
blockedHandler = yield
```

\(^1\) Haskell on a Shared-Memory Multiprocessor (Tim Harris, Simon Marlow, Simon Peyton Jones)
Thread-local state

- In a multiprocessor setting, one global run queue is a bad idea. We probably want one scheduler per CPU.
- A thread needs to ask “what is my scheduler?”: *thread-local state*
- Simple proposal:

```haskell
data TLSKey a
newTLSKey :: a -> IO (TLSKey a)
getTLS  :: TLSKey a -> PTM a
setTLS  :: TLSKey a -> a -> IO ()
initTLS :: SCont -> TLSKey a -> a -> IO ()
```
Multiprocessors: sleeping HECs

- On a multiprocessor, we will have multiple HECs, each of which has a scheduler.
- When a HEC has no threads to run, it must idle somehow. Busy waiting would be bad, so we provide more functionality to put HECs to sleep:
  
  ```hs
  waitCond :: PTM (Maybe a) -> IO a
  wakeupHEC :: HEC -> IO ()
  ```
  
  “execute the PTM transaction repeatedly until it returns Just a, then deliver a”

- A bit like STM’s retry, but less automatic
Multiprocessor scheduler

- One scheduler (run queue) per CPU
- Scheduler has its own SCont

```
yield =
  switch $ \sc -> do
    addToRunQueue sc
  sched_var <- readTLS mySchedulerKey
  sched <- readPVar sched_var
  return sched

schedule sched_var = do
  thread <- waitCond popRunQueue
  switch $ \sc -> do
    writePVar sched_var sc
  return thread
```
Foreign calls

• Foreign calls and concurrency interact:
  – in-calls from multiple OS threads (Haskell as a multithreaded foreign API)
  – an out-call may block, we want to schedule another Haskell thread when this happens
  – out-calls can make in-calls (callbacks)
  – sometimes, out-calls need to be made in a particular OS thread ("bound threads")

• All of the above can be implemented in the concurrency library, all we need are some small additions to the substrate...
Foreign calls, cont.

- Two concurrency library callbacks:

  \[
  \text{inCallHandler} :: \text{IO} ~ a \rightarrow \text{IO} ~ a \\
  \text{outCallHandler} :: \text{IO} ~ a \rightarrow \text{IO} ~ a
  \]

- When an in-call happens, the RTS
  - makes a new HEC,
  - executes inCallHandler (f args...)
  - inCallHandler can e.g. create a new scheduler, or add this thread to the run queue of an existing scheduler (GHC currently does the latter)

- For each out-call
  - the compiler generates outCallHandler (f args...)
  - outCallHandler can e.g. arrange to switch to another HEC to make the call, or wake up another HEC to schedule more Haskell threads.

- The scheduler support for the full FFI is complex, but the substrate is simple.
Asynchronous exceptions

- Phew
Performance

• Time in (s):

<table>
<thead>
<tr>
<th></th>
<th>ghc-6.6</th>
<th>fake-ptm</th>
<th>real-ptm</th>
</tr>
</thead>
<tbody>
<tr>
<td>spawn-test</td>
<td>18</td>
<td>32</td>
<td>46</td>
</tr>
<tr>
<td>producer-consumer</td>
<td>4.3</td>
<td>7.0</td>
<td>16.2</td>
</tr>
<tr>
<td>cheap-concurrency</td>
<td>6.5</td>
<td>7.1</td>
<td>12.6</td>
</tr>
<tr>
<td>chameneos</td>
<td>6.3</td>
<td>4.8</td>
<td>26</td>
</tr>
</tbody>
</table>

• spawn-test: benchmarks forkIO
• the others benchmark MVar performance
• fake PTM: PTM implementation with no atomicity
• real PTM: based on existing STM implementation
• Prototype concurrency library is 2-4 times slower than existing RTS.
Gentoo : Intel® Pentium® 4
Computer Language Benchmarks Game

chameleon benchmark

Read the benchmark rules. Symmetrical thread rendez-vous requests N=5,000,000 (Check that Error or Timeout happened at other values of N with chameleon full data).

<table>
<thead>
<tr>
<th></th>
<th>Full CPU Time</th>
<th>Memory Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full CPU Time</td>
<td>Memory Use</td>
</tr>
<tr>
<td></td>
<td>sort</td>
<td>sort</td>
</tr>
<tr>
<td>x</td>
<td>Program &amp; Logs</td>
<td>Full CPU Time</td>
</tr>
<tr>
<td>1.0</td>
<td>Haskell GHC</td>
<td>4.43</td>
</tr>
<tr>
<td>1.0</td>
<td>Haskell GHC #2</td>
<td>4.54</td>
</tr>
<tr>
<td>1.1</td>
<td>D Digital Mars</td>
<td>5.06</td>
</tr>
<tr>
<td>1.8</td>
<td>Erlang HIPE</td>
<td>7.80</td>
</tr>
<tr>
<td>2.1</td>
<td>Smalltalk VisualWorks</td>
<td>9.42</td>
</tr>
<tr>
<td>2.5</td>
<td>Forth bigForth</td>
<td>11.05</td>
</tr>
<tr>
<td>3.4</td>
<td>Eiffel SmartEffi</td>
<td>14.97</td>
</tr>
<tr>
<td>4.8</td>
<td>Mozart/Oz</td>
<td>21.07</td>
</tr>
<tr>
<td>4.9</td>
<td>Pascal Free Pascal #2</td>
<td>21.64</td>
</tr>
<tr>
<td>6.0</td>
<td>SML MLton</td>
<td>26.38</td>
</tr>
<tr>
<td>6.0</td>
<td>C gcc</td>
<td>26.76</td>
</tr>
<tr>
<td>6.2</td>
<td>Ada 95 GNAT #2</td>
<td>27.29</td>
</tr>
<tr>
<td>6.4</td>
<td>Lua</td>
<td>28.15</td>
</tr>
<tr>
<td>9.4</td>
<td>C++ g++</td>
<td>41.60</td>
</tr>
<tr>
<td>9.8</td>
<td>Java 6 -server #2</td>
<td>43.29</td>
</tr>
</tbody>
</table>
The (lack of a) conclusion

- We get a great research platform...
- Is a factor of 2-4 a reasonable price to pay for the extra flexibility?
  - For concurrent programs, performance of concurrency is not usually the bottleneck
  - but the scheduler might be critical for parallel performance
  - STM on top of PTM is possible, but hairy
- Most users don’t care about the extra flexibility
- better reliability (maybe), but is it really easier to debug?
- Have to worry about: the scheduler being pre-empted, blocking, running out of stack (non-issues with the C version)
- The “scheduler tax” is high: a scheduler must implement blocking, MVars, STM, FFI, asynchronous exceptions, par. Few people will write a scheduler, most likely we’ll provide an extensible one.
  - could we just make the existing scheduler extensible?
- Major issues for users are debugging concurrency, and debugging parallel performance. Does this enable improvements there?