Lightweight Concurrency in GHC

KC Sivaramakrishnan
Tim Harris
Simon Marlow
Simon Peyton Jones
GHC: Concurrency and Parallelism

- forkIO
- MVars
- Bound threads
- Par Monad
- STM
- Safe foreign calls
- Asynchronous exceptions
Concurrency landscape in GHC

- Haskell Code
- LWT Scheduler
- RTS (C Code)
- STM
- MVar
- Safe FFI
- Black Holes
- and more...
- OS Thread pool
- Capability 0
- Capability N

preemptive, round-robin scheduler + work-sharing
Idea

Haskell Code

RTS (C Code)

LWT Scheduler

STM

MVar

Safe FFI

Black Holes

and more...

Concurrency Substrate

Haskell Code

LWT Scheduler+

MVar+

STM+

OS Thread pool

Capability 0

Capability N

Safe FFI

Black Holes

OS Thread pool

Capability 0

Capability N
Contributions

Where do these live in the new design?

What should this be?

How to unify these?

Haskell Code

LWT Scheduler

OS Thread pool

RTS (C Code)

Concurrency Substrate

Haskell Code

LWT Scheduler+

MVar+

STM+

RTS (C Code)

Safe FFI

Black Holes

and more…

Capability 0  Capability N
Concurrency Substrate

- One-shot continuations (SCont) and primitive transactional memory (PTM)
- PTM is a bare-bones TM
  - Better composability than CAS

---

```haskell
-- PTM -------------------
data PTM a
data PVar a
instance Monad PTM
  atomically :: PTM a -> IO a
  newPVar :: a -> PTM (PVar a)
  readPVar :: PVar a -> PTM a
  writePVar :: PVar a -> a -> PTM ()

-- SCont -------------------
data SCont -- Stack Continuations
newSCont :: IO () -> IO SCont
switch :: (SCont -> PTM SCont) -> IO ()
getCurrentSCont :: PTM SCont
switchTo :: SCont -> PTM ()
```
Switch

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

Current SCont

PTM!

SCont to switch to
Abstract Scheduler Interface

- Primitive scheduler actions
  - SCont \{ scheduleSContAction :: SCont \rightarrow PTM (), yieldControlAction :: PTM () \}
  - Expected from every user-level thread
scheduleSContAction :: SCont -> PTM ()
scheduleSContAction sc = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  writePVar $ contents ++ [sc]

yieldControlAction :: PTM ()
yieldControlAction = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  case contents of
    x:tail -> do {
      writePVar $ contents tail;
      switchTo x -- DOES NOT RETURN
    }
  otherwise -> ...
**Primitive Scheduler Actions (2)**

```
scheduleSContAction :: SCont -> PTM ()
scheduleSContAction sc = do
    sched :: PVar [SCont] <- -- get sched
    contents :: [SCont] <- readPVar sched
    writePVar $ contents ++ [sc]
```

```
yieldControlAction :: PTM ()
yieldControlAction = do
    sched :: PVar [SCont] <- -- get sched
    contents :: [SCont] <- readPVar sched
    case contents of
        x:tail -> do {
            writePVar $ contents tail;
            switchTo x -- DOES NOT RETURN
        }
    otherwise -> …
```

```
getScheduleSContAction
    :: SCont -> PTM (SCont -> PTM())
setScheduleSContAction
    :: SCont -> (SCont -> PTM()) -> PTM()
```

```
getYieldControlAction
    :: SCont -> PTM (PTM ())
setYieldControlAction
    :: SCont -> PTM () -> PTM ()
```
Primitive Scheduler Actions (3)

\[
\begin{align*}
schedule\text{SContAction} & : \text{SCont} \rightarrow \text{PTM}() \\
schedule\text{SContAction} \text{ sc} &= \text{do} \\
& \quad \text{sched} : \text{PVar [SCont]} \leftarrow -- \text{get sched} \\
& \quad \text{contents} : : [\text{SCont}] \leftarrow \text{readPVar sched} \\
& \quad \text{writePVar $ contents } ++ [\text{sc}]
\end{align*}
\]

\[
\begin{align*}
yield\text{ControlAction} & : \text{PTM}() \\
yield\text{ControlAction} = \text{do} \\
& \quad \text{sched} : \text{PVar [SCont]} \leftarrow -- \text{get sched} \\
& \quad \text{contents} : : [\text{SCont}] \leftarrow \text{readPVar sched} \\
& \quad \text{case contents of} \\
& \quad \quad x:tail \rightarrow \text{do } \{ \\
& \quad \quad \quad \text{writePVar $ contents tail;} \\
& \quad \quad \quad \text{switchTo x -- DOES NOT RETURN} \\
& \quad \quad \} \\
& \quad \text{otherwise } \rightarrow \ldots
\end{align*}
\]

\[
\begin{align*}
\text{getScheduleSContAction} & : \text{SCont} \rightarrow \text{PTM (SCont} \rightarrow \text{PTM}()) \\
\text{setScheduleSContAction} & : \text{SCont} \rightarrow (\text{SCont} \rightarrow \text{PTM}()) \rightarrow \text{PTM}() \\
\text{getSSA} &= \text{getScheduleSContAction} \\
\text{setSSA} &= \text{setScheduleScontAction}
\end{align*}
\]

\[
\begin{align*}
\text{getYieldControlAction} & : \text{PTM \} (PTM \} (\text{PTM}()) \\
\text{setScheduleSContAction} & : \text{SCont} \rightarrow \text{PTM (}) \rightarrow \text{PTM}() \\
\text{getYCA} &= \text{getYieldControlAction} \\
\text{setYCA} &= \text{setYieldControlAction}
\end{align*}
\]
yield :: IO ()
yield = atomically $ do
  s :: SCont <- getCurrentSCont
  -- Add current SCont to scheduler
  ssa :: (SCont -> PTM ()) <- getSSA s
  enqueue :: PTM () <- ssa s
  enqueue
  -- Switch to next scont from scheduler
  switchToNext :: PTM () <- getYCA s
  switchToNext
Building Concurrency Primitives (2)

```haskell
forkIO :: IO () -> IO SCont
forkIO f = do
    ns <- newSCont f
    atomically $ do {  
        s :: SCont <- getCurrentSCont;
        -- Initialize new sconts scheduler actions
        ssa :: (SCont -> PTM ()) <- getSSA s;
        setSSA ns ssa;
        yca :: PTM () <- getYCA s;
        setYCA ns yca;
        -- Add to new scont current scheduler
        enqueueAct :: PTM () <- ssa ns;
        enqueueAct
    }
    return ns
```
An MVar is either empty or full and has a single hole.

```haskell
newtype MVar a = MVar (PVar (ST a))
data ST a = Full a [(a, PTM())]
  | Empty [(PVar a, PTM())]
takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  hole <- atomically $ newPVar undefined
  atomically $ do
    st <- readPVar ref
    case st of
      Empty ts -> do
        s <- getCurrentSCont
        ssa :: (SCont -> PTM ()) <- getSSA s
        wakeup :: PTM () <- ssa s
        writePVar ref $ v
        where v = Empty $ ts++[(hole, wakeup)]
        switchToNext <- getYCA s
        switchToNext
      Full x ((x', wakeup :: PTM ()):ts) -> do
        writePVar hole x
        writePVar ref $ Full x' ts
        wakeup
      otherwise -> ...
  atomically $ readPVar hole
```
Building MVars

newtype MVar a = MVar (PVar (ST a))
data ST a = Full a [(a, PTM())] |
                   Empty [(PVar a, PTM())]

takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  hole <- atomically $ newPVar undefined
  atomically $ do
    st <- readPVar ref
    case st of
      Empty ts -> do
        s <- getCurrentSCont
        ssa :: (SCont -> PTM ()) <- getSSA s
        wakeup :: PTM () <- ssa s
        writePVar ref $ v
          where v = Empty $ ts++[(hole, wakeup)]
        switchToNext <- getYCA s
        switchToNext
      Full x ((x', wakeup :: PTM ()):ts) -> do
        writePVar hole x
        writePVar ref $ Full x' ts
        wakeup
      otherwise -> …
  atomically $ readPVar hole

An MVar is either empty or full and has a single hole

Result will be here
Building MVars

newtype MVar a = MVar (PVar (ST a))
data ST a = Full a [(a, PTM())] | Empty [(PVar a, PTM())]

takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  hole <- atomically $ newPVar undefined
  atomically $ do
    st <- readPVar ref
    case st of
      Empty ts -> do
        s <- getCurrentSCont
        ssa :: (SCont -> PTM()) <- getSSA s
        wakeup :: PTM() <- ssa s
        writePVar ref $ v
        where v = Empty $ ts++[(hole, wakeup)]
        switchToNext <- getYCA s
        switchToNext
      Full x ((x', wakeup :: PTM()):ts) -> do
        writePVar hole x
        writePVar ref $ Full x' ts
        wakeup
      otherwise -> ...
  atomically $ readPVar hole
Building MVars

An MVar is either empty or full and has a single hole

Result will be here

If the mvar is empty
(1) Append hole & wakeup info to mvar list (getSSA!)
(2) Yield control to scheduler (getYCA!)

Wake up a pending writer, if any. wakeup is a PTM ()!

MVars is scheduler agnostic!
Interaction of C RTS and User-level scheduler

- Many “Events” that necessitate actions on the scheduler become apparent only in the C part of the RTS

```
Haskell Code

LWT Scheduler+  MVar+  STM+

Concurrency Substrate

Safe FFI  Black Hole  Asynchronous exceptions  Finalizers
```
Interaction of C RTS and User-level scheduler

- Many “Events” that necessitate actions on the scheduler become apparent only in the C part of the RTS

Haskell Code

Concurrency Substrate

Safe FFI
Black Hole
Asynchronous exceptions
Finalizers

Re-use primitive scheduler actions!

Pending upcall queue :: [PTM ()]

Capability X

Upcall Thread
Blackholes

Capability 0

T1

evaluating...

Thunk

Capability 1

T2

T3

T → Running

T → Suspended

T → Blocked
Blackholes

Capability 0

T1

thunk "blackholed"

BH

Capability 1

T2

T3

→ Running

→ Suspended

→ Blocked
Blackholes

Capability 0

T1

Capability 1

T2

T3

BH enters blackhole

T → Running

T → Suspended

T → Blocked
Blackholes

Capability 0

- T1

Capability 1

- T2
- T3

BH

T → Running
T → Suspended
T → Blocked
Blackholes

Capability 0

T1

BH

Capability 1

T2

T3

Yield control action

T → Running

T → Suspended

T → Blocked

24
Blackholes

Capability 0

- T1 finishes evaluation
- V

Capability 1

- T2
- T3

Schedule SCont action

T → Running
T → Suspended
T → Blocked
Blackholes: The Problem

Capability 0

T1

Switch $ \text{T1} \rightarrow \text{do}$
--
--
return T2

T2

BH

T
→ Running

T
→ Suspended

T
→ Blocked
Blackholes: The Problem

- In order to make progress, we need to resume to T2
- But, in order to resume to T2, we need to resume T2 (Deadlocked!)
  - Can be resolved through runtime system tricks (Work in Progress!)
Conclusions

• Status
  – Mostly implemented (SConts, PTM, Simple schedulers, MVars, Safe FFI, bound threads, asynchronous exceptions, finalizers, etc.)
  – 2X to 3X slower on micro benchmarks (programs only doing synchronization work)

• To-do
  – Re-implement Control.Concurrent with LWC
  – Formal operational semantics
  – Building real-world programs

• Open questions
  – Hierarchical schedulers, Thread priority, load balancing, Fairness, etc.
  – STM on top of PTM
  – PTM on top of SpecTM
  – Integration with par/seq, evaluation strategies, etc.
  – and more...