Under consideration for publication in J. Functional Programming

Composable Scheduler Activations for Haskell

KC SIVARAMAKRISHNAN
University of Cambridge

TIM HARRIS*
Oracle Labs

SIMON MARLOW*
Facebook UK Ltd.

SIMON PEYTON JONES
Microsoft Research, Cambridge

Abstract

The runtime for a modern, concurrent, garbage collected language like Java or Haskell is like an operating system: sophisticated, complex, performant, but alas very hard to change. If more of the runtime system were in the high level language, it would be far more modular and malleable. In this paper, we describe a novel concurrency substrate design for the Glasgow Haskell Compiler (GHC) that allows multicore schedulers for concurrent and parallel Haskell programs to be safely and modularly described as libraries in Haskell. The approach relies on abstracting the interface to the user-implemented schedulers through scheduler activations, together with the use of Software Transactional Memory (STM) to promote safety in a multicore context.

1 Introduction

High performance, multicore-capable runtime systems (RTS) for garbage-collected languages have been in widespread use for many years. Examples include virtual machines for popular object-oriented languages such as Oracle's Java HotSpot VM (HotSpotVM, 2014), IBM's Java VM (IBM, 2014), Microsoft's Common Language Runtime (CLR) (Microsoft Corp., 2014), as well as functional language runtimes such as Manticore (Fluet *et al.*, 2008), MultiMLton (Sivaramakrishnan *et al.*, 2014) and the Glasgow Haskell Compiler (GHC) (GHC, 2014).

These runtime systems tend to be complex monolithic pieces of software, written not in the high-level source language (Java, Haskell, etc), but in an unsafe, systems programming language (usually C or C++). They are highly concurrent, with extensive use of locks, condition variables, timers, asynchronous I/O, thread pools, and other arcana. As a result, they are extremely difficult to modify, even for their own authors. Moreover, such modifications typically require a rebuild of the runtime, so it is not an easy matter to make changes on a program-by-program basis, let alone within a single program.

This lack of malleability is particularly unfortunate for the *thread scheduler*, which governs how the computational resources of the multi-core are deployed to run zillions of lightweight high-level language threads. A broad range of strategies are possible, including ones using priorities, hierarchical scheduling, gang scheduling, and work stealing. *The goal*

* This work was done at Microsoft Research, Cambridge.

of this paper is, therefore, to allow programmers to write a User Level Scheduler (ULS), as a library written in the high level language itself. Not only does this make the scheduler more modular and changeable, but it can readily be varied between programs, or even within a single program.

The difficulty is that the scheduler interacts intimately with other aspects of the runtime such as transactional memory or blocking I/O. Our main contribution is the design of an interface that allows expressive user-level schedulers to interact cleanly with these low-level communication and synchronisation primitives:

- We present a new concurrency substrate design for Haskell that allows application programmers to write schedulers for Concurrent Haskell programs in Haskell (Section 3). These schedulers can then be plugged-in as ordinary user libraries in the target program.
- By abstracting the interface to the ULS through *scheduler activations* (Anderson *et al.*, 1991), our concurrency substrate seamlessly integrates with the existing RTS concurrency support such as `MVars`, asynchronous exceptions (Marlow *et al.*, 2001), safe foreign function interface (Marlow *et al.*, 2004), software transactional memory (Harris *et al.*, 2005a), resumable black-holes (Reid, 1999), etc. The RTS makes *upcalls* to the activations whenever it needs to interact with the ULS. This design absolves the scheduler writer from having to reason about the interaction between the ULS and the RTS, and thus lowering the bar for writing new schedulers.
- Concurrency primitives and their interaction with the RTS are particularly tricky to specify and reason about. An unusual feature of this paper is that we precisely formalise not only the concurrency substrate primitives (Section 5), but also their interaction with the RTS concurrency primitives (Section 6).
- We present an implementation of our concurrency substrate in GHC. Experimental evaluation indicate that the performance of ULS's is comparable to the highly optimised default scheduler of GHC (Section 7).

2 Background

To understand the design of the new concurrency substrate for Haskell, we must first give some background on the existing RTS support for concurrency in our target platform – the Glasgow Haskell Compiler (GHC). We then articulate the goals of our concurrency substrate.

2.1 The GHC runtime system

GHC has a sophisticated, highly tuned RTS that has a rich support for concurrency with advanced features such as software transactional memory (Harris *et al.*, 2005a), asynchronous exceptions (Marlow *et al.*, 2001), safe foreign function interface (Marlow *et al.*, 2004), and transparent scaling on multicores (Harris *et al.*, 2005b). The Haskell programmer can use very lightweight Haskell *threads*, which are executed by a fixed number of Haskell execution contexts, or *HECs*. Each HEC is in turn animated by an operating system thread; in this paper we use the term *tasks* for these OS threads, to distinguish them from Haskell

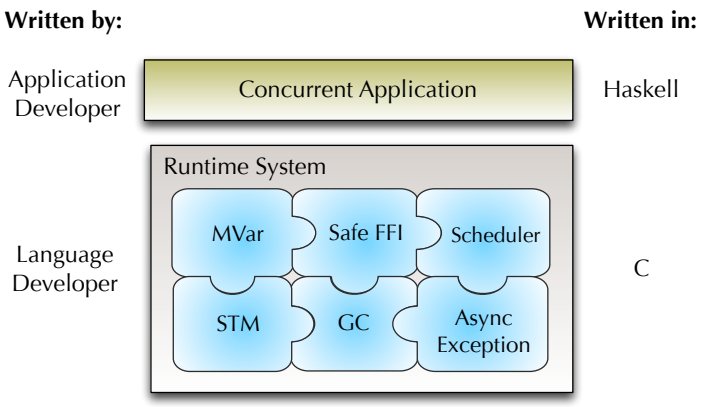


Fig. 1: The anatomy of the Glasgow Haskell Compiler runtime system

threads. The choice of which Haskell thread is executed by which HEC is made by the *scheduler*.

GHC's current scheduler is written in C, and is hard-wired into the RTS (Figure 1). It uses a single run-queue per processor, and has a single, fixed notion of work-sharing to move work from one processor to another. There is no notion of thread priority; nor is there support for advanced scheduling policies such as gang or spatial scheduling. From an application developer's perspective, the lack of flexibility hinders deployment of new programming models on top of GHC such as data-parallel computations (Chakravarty *et al.*, 2007; Lippmeier *et al.*, 2012), and applications such as virtual machines (Galois, 2014) and web-servers (Haskell, 2014) that can benefit from the ability to define custom scheduling policies.

2.2 The challenge

Because there is such a rich design space for schedulers, our goal is to allow a user-level scheduler (ULS) to be written in Haskell, giving programmers the freedom to experiment with different scheduling or work-stealing algorithms. Indeed, we would like the ability to combine *multiple* ULS's in the same program. For example, in order to utilise the best scheduling strategy, a program could dynamically switch from a priority-based scheduler to gang scheduling when switching from general purpose computation to data-parallel computation. Applications might also combine the schedulers in a *hierarchical* fashion; a scheduler receives computational resources from its parent, and divides them among its children.

This goal is not not easy to achieve. The scheduler interacts intimately with other RTS components including

- `MVars` and transactional memory (Harris *et al.*, 2005a) allow Haskell threads to communicate and synchronise; they may cause threads to block or unblock.
- The garbage collector must somehow know about the run-queue on each HEC, so that it can use it as a root for garbage collection.

- Lazy evaluation means that if a Haskell thread tries to evaluate a thunk that is already under evaluation by another thread (it is a "black hole"), the former must block until the thunk's evaluation is complete (Harris *et al.*, 2005b). Matters are made more complicated by asynchronous exceptions, which may cause a thread to abandon evaluation of a thunk, replacing the thunk with a "resumable black hole".
- A foreign-function call may block (e.g., when doing I/O). GHC's RTS can schedule a fresh task (OS thread) to re-animate the HEC, blocking the in-flight Haskell thread, and scheduling a new one (Marlow *et al.*, 2004).

All of these components do things like "block a thread" or "unblock a thread" that require interaction with the scheduler. One possible response, taken by Li *et al* (Li *et al.*, 2007) is to program these components, too, into Haskell. The difficulty is that they all are intricate and highly-optimised. Moreover, unlike scheduling, there is no call from Haskell's users for them to be user-programmable.

Instead, our goal is to tease out the scheduler implementation from the rest of the RTS, establishing a clear API between the two, and leaving unchanged the existing implementation of `MVars`, STM, black holes, FFI, and so on.

Lastly, schedulers are themselves concurrent programs, and they are particularly devious ones. Using the facilities available in C, they are extremely hard to get right. Given that the ULS will be implemented in Haskell, we would like to utilise the concurrency control abstractions provided by Haskell (notably transactional memory) to simplify the task of scheduler implementation.

3 Design

In this section, we describe the design of our concurrency substrate and present the concurrency substrate API. Along the way, we will describe how our design achieves the goals put forth in the previous section.

3.1 Scheduler activation

Our key observation is that the interaction between the scheduler and the rest of the RTS can be reduced to two fundamental operations:

1. **Block operation.** The currently running thread blocks on some event in the RTS. The execution proceeds by switching to the next available thread from the scheduler.
2. **Unblock operation.** The RTS event that a blocked thread is waiting on occurs. After this, the blocked thread is resumed by adding it to the scheduler.

For example, in Haskell, a thread might encounter an empty `MVar` while attempting to take the value from it. This operation is analogous to attempting to take a lock that is currently held by some other thread. In this case, the thread performing the `MVar` read operation should block. Eventually, the `MVar` might be filled by some other thread (analogous to lock release), in which case, the blocked thread is unblocked and resumed with the value from the `MVar`. As we will see, all of the RTS interactions (as well as the interaction with the concurrency libraries) fall into this pattern.

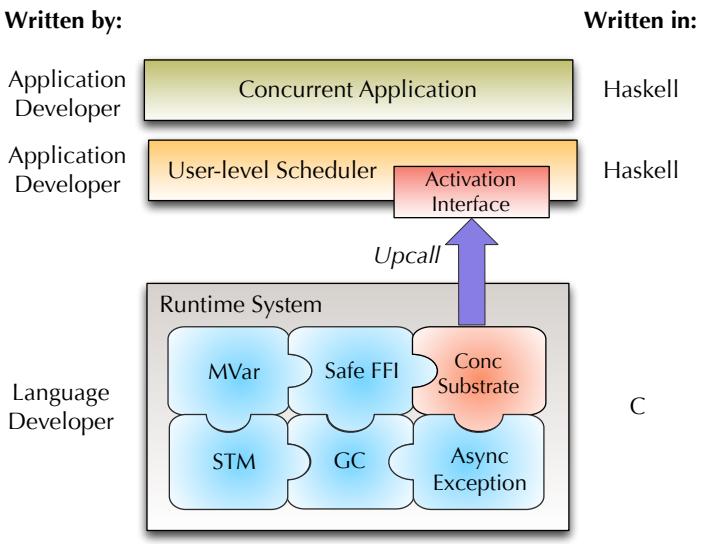


Fig. 2: New GHC RTS design with Concurrency Substrate.

Notice that the RTS blocking operations enqueue and dequeue threads from the scheduler. But the scheduler is now implemented as a Haskell library. So how does the RTS find the scheduler? We could equip each HEC with a fixed scheduler, but it is much more flexible to equip each Haskell *thread* with its own scheduler. That way, different threads (or groups thereof) can have different schedulers.

But what precisely is a "scheduler"? In our design, the scheduler is represented by two function values, or scheduler *activations* [1]. Every user-level thread has a *dequeue activation* and an *enqueue activation*. The activations provide an abstract interface to the ULS to which the thread belongs to. The activations are function values closed over the shared datastructure representing the scheduler. At the very least, the dequeue activation fetches the next available thread from the ULS encapsulated in the activation, and the enqueue activation adds the given thread to the encapsulated ULS. The activations are stored at known offsets in the thread object so that the RTS may find them. The RTS makes *upcalls* to the activations to perform the enqueue and dequeue operations on a ULS.

Figure 2 illustrates the modified RTS design that supports the implementation of ULS's. The idea is to have a minimal concurrency substrate which is implemented in C and is a part of the RTS. The substrate not only allows the programmer to implement schedulers as Haskell libraries, but also enables other RTS mechanisms to interface with the user-level schedulers through upcalls to the activations.

Figure 3 illustrates the steps associated with blocking on an RTS event. Since the scheduler is implemented in user-space, each HEC in the RTS is aware of only the currently running thread, say `t`. Suppose thread `t` waits for an abstract event `e` in the RTS, which is currently disabled. Since the thread `t` cannot continue until `e` is enabled, the RTS adds `t`

[1] The idea of an "activation" comes from the operating systems literature (Anderson *et al.*, 1991)

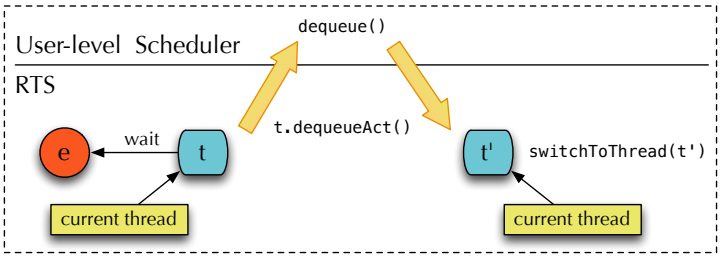


Fig. 3: Blocking on an RTS event.

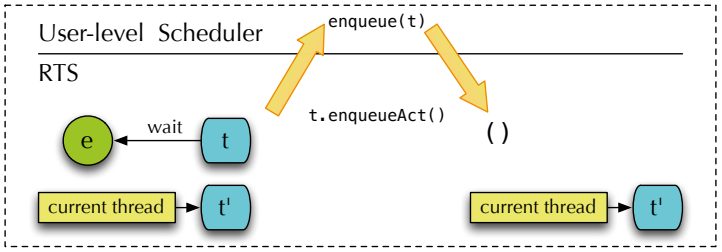


Fig. 4: Unblocking from an RTS event.

to the queue of threads associated with `e`, which are currently waiting for `e` to be enabled. Notice that the RTS "owns" `t` at this point. The RTS now invokes the dequeue activation associated with `t`, which returns the next runnable thread from `t`'s scheduler queue, say `t'`. This HEC now switches control to `t'` and resumes execution. The overall effect of the operation ensures that although the thread `t` is blocked, `t`'s scheduler (and the threads that belong to it) is not blocked.

Figure 4 illustrates the steps involved in unblocking from an RTS event. Eventually, the disabled event `e` can become enabled. At this point, the RTS wakes up all of the threads waiting on event `e` by invoking their enqueue activation. Suppose we want to resume the thread `t` which is blocked on `e`. The RTS invokes `t`'s enqueue activation to add `t` to its scheduler. Since `t`'s scheduler is already running, `t` will eventually be scheduled again.

3.2 Software transactional memory

Since Haskell computations can run in parallel on different HECs, the substrate must provide a method for safely coordinating activities across multiple HECs. Similar to Li's substrate design (Li *et al.*, 2007), we adopt *transactional memory* (STM), as the sole multiprocessor synchronisation mechanism exposed by the substrate. Using transactional memory, rather than locks and condition variables make complex concurrent programs much more modular and less error-prone (Harris *et al.*, 2005a) – and schedulers are prime candidates, because they are prone to subtle concurrency bugs.

3.3 Concurrency substrate

Now that we have motivated our design decisions, we will present the API for the concurrency substrate. The concurrency substrate includes the primitives for instantiating and switching between language level threads, manipulating thread local state, and an abstraction for scheduler activations. The API is presented below:

```
data SCont
type DequeueAct = SCont -> STM SCont
type EnqueueAct = SCont -> STM ()

-- activation interface
dequeueAct :: DequeueAct
enqueueAct :: EnqueueAct

-- SCont manipulation
newSCont     :: IO () -> IO SCont
switch       :: (SCont -> STM SCont) -> IO ()
runOnIdleHEC :: SCont -> IO ()

-- Manipulating local state
setDequeueAct :: DequeueAct -> IO ()
setEnqueueAct :: EnqueueAct -> IO ()
getAux        :: SCont -> STM Dynamic
setAux        :: SCont -> Dynamic -> STM ()

-- HEC information
getCurrentHEC :: STM Int
getNumHECs    :: IO Int
```

3.3.1 Activation interface

Rather than directly exposing the notion of a "thread", the substrate offers *linear continuations* (Bruggeman *et al.*, 1996), which is of type `SCont`. An `SCont` is a heap-allocated object representing the current state of a Haskell computation. In the RTS, `SConts` are represented quite conventionally by a heap-allocated Thread Storage Object (TSO), which includes the computations stack and local state, saved registers, and program counter. Unreachable `SConts` are garbage collected.

The call (`dequeueAct s`) invokes *s*'s `dequeue` activation, passing *s* to it like a "self" parameter. The return type of `dequeueAct` indicates that the computation encapsulated in the `dequeueAct` is transactional (under `STM` monad(STMLibrary, 2014)) which when discharged, returns an `SCont`. Similarly, the call (`enqueueAct s`) invokes the `enqueue` activation transactionally, which enqueues *s* to its ULS.

Since the activations are under `STM` monad, we have the assurance that the ULS' cannot be built with low-level unsafe components such as locks and condition variables. Such low-level operations would be under `IO` monad, which cannot be part of an `STM` transaction. Thus, our concurrency substrate statically prevents the implementation of potentially unsafe schedulers.

3.3.2 SCont *management*

The substrate offers primitives for creating, constructing and transferring control between SConts. The call (newSCont M) creates a new SCont that, when scheduled, executes M. By default, the newly created SCont is associated with the ULS of the invoking thread. This is done by copying the invoking SCont's activations.

An SCont is scheduled (i.e., is given control of a HEC) by the switch primitive. The call (switch M) applies M to the current continuation s. Notice that $(M\,s)$ is an STM computation. In a single atomic transaction switch performs the computation $(M\,s)$, yielding an SCont s', and switches control to s'. Thus, the computation encapsulated by s' becomes the currently running computation on this HEC.

Our continuations are linear; resuming a running SCont raises an exception. Our implementation enforces this property by attaching a transactional status variable to each SCont. The status of an SCont is updated from suspended to running, if the control switches to the SCont. Correspondingly, the suspending SCont's status is updated from running to suspended. Since our continuations are always used linearly, capturing a continuation simply fetches the reference to the underlying TSO object. Hence, continuation capture involves no copying, and is cheap. Using the SCont interface, a thread yield function can be built as follows:

```
yield :: IO ()
yield = switch (\s -> enqueueAct s >> dequeueAct s)
```

Calling the yield function switches control to the next available thread that is available to run on the yielding thread's scheduler.

3.4 *Parallel* SCont *execution*

When the program begins execution, a fixed number of HECs (N) is provided to it by the environment. This signifies the maximum number of *parallel* computations in the program. Of these, one of the HEC runs the main IO computation. All other HECs are in idle state. The call runOnIdleHEC s initiates parallel execution of SCont s on an idle HEC. Once the SCont running on a HEC finishes evaluation, the HEC moves back to the idle state. The primitive getNumHECs and getCurrentHEC returns the number of HECs and the HEC number of the current HEC.

Notice that the upcall from the RTS to the dequeue activation as well as the body of the switch primitive return an SCont. This is the SCont to which the control would switch to subsequently. But what if such an SCont cannot be found? This situation can occur during multicore execution, when the number of available threads is less than the number of HECs. If a HEC does not have any work to do, it better be put to sleep.

Notice that the result of the dequeue activation and the body of the switch primitive are STM transactions. GHC today supports blocking operations under STM. When the programmer invokes retry inside a transaction, the RTS blocks the thread until another thread writes to any of the transactional variables read by the transaction; then the thread is re-awoken, and retries the transaction (Harris *et al.*, 2005a). This is entirely transparent to the programmer. Along the same lines, we interpret the use of retry within a switch

or dequeue activation transaction as putting the whole HEC to sleep. We use the existing RTS mechanism to resume the thread when work becomes available on the scheduler.

3.5 `SCont` *local state*

The activations of an `SCont` can be read by `dequeueAct` and `enqueueAct` primitives. In effect, they constitute the `SCont`-local state. Local state is often convenient for other purposes, so we also provide a single dynamically-typed[2] field, the "aux-field", for arbitrary user purposes. The aux-field can be read from and written to using the primitives `getAux` and `setAux`. The API additionally allows an `SCont` to change its own scheduler through `setDequeueAct` and `setEnqueueAct` primitives.

4 Developing concurrency libraries

In this section, we will utilise the concurrency substrate to implement a multicore capable, round-robin, work-sharing scheduler and a user-level MVar implementation.

4.1 User-level scheduler

The first step in designing a scheduler is to describe the scheduler data structure. We shall build a first-in-first-out (FIFO) scheduler that schedules that `SCont` that arrived in the scheduler queue earliest. FIFO scheduling is useful for applications such as a web-server that minimises the overall latency for pending requests. We utilise an array of runqueues, with one queue per HEC. Each runqueue is represented by a transactional variable (a `TVar`), which can hold a list of `SConts`.

```
newtype Sched = Sched (Array Int (TVar[SCont]))
```

The next step is to provide an implementation for the scheduler activations.

```
dequeueActivation:: Sched -> SCont -> STM SCont
dequeueActivation (Sched pa) _ = do
  cc <- getCurrentHEC -- get current HEC number
  l <- readTVar $ pa!cc
  case l of
    [] -> retry
    x:tl -> do
      writeTVar (pa!cc) tl
      return x

enqueueActivation:: Sched -> SCont -> STM ()
enqueueActivation (Sched pa) sc = do
  dyn <- getAux sc
  let (hec::Int, _::TVar Int) = fromJust $ fromDynamic dyn
  l <- readTVar $ pa!hec
  writeTVar (pa!hec) $ l++[sc]
```

[2] `http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-Dynamic.html`

`dequeueActivation` either returns the `SCont` at the front of the runqueue and updates the runqueue appropriately, or puts the HEC to sleep if the queue is empty. Recall that performing `retry` within a dequeue activation puts the HEC to sleep. The HEC will automatically be woken up when work becomes available i.e., queue becomes non-empty. Although we ignore the `SCont` being blocked in this case, one could imagine manipulating the blocked `SCont`'s aux state for accounting information such as time slices consumed for fair-share scheduling. Enqueue activation (`enqueueActivation`) finds the `SCont`'s HEC number by querying its `SCont`-local state (the details of which is presented along with the next primitive). This HEC number (`hec`) is used to fetch the correct runqueue, to which the `SCont` is appended to.

4.1.1 Scheduler initialisation

The next step is to initialise the scheduler. This involves two steps: (1) allocating the scheduler (`newScheduler`) and initialising the main thread and (2) spinning up additional HECs (`newHEC`). We assume that the Haskell program wishing to utilise the ULS performs these two steps at the start of the `main` IO computation. The implementation of these functions are given below:

```
newScheduler :: IO ()
newScheduler = do
  -- Initialise Auxiliary state
  myS <- switch $ \s -> do
           counter <- newTVar (0::Int)
           setAux s $ toDyn $ (0::Int,counter)
           return s
  -- Allocate scheduler
  nc <- getNumHECs
  sched <- (Sched . listArray (0,nc-1)) <$>
           replicateM nc (newTVar [])
  -- Initialise activations
  setDequeueAct myS $ dequeueActivation sched
  setEnqueueAct myS $ enqueueActivation sched

newHEC :: IO ()
newHEC = do
 -- Initial task
 s <- newSCont $ switch dequeueAct
 -- Run in parallel
 runOnIdleHEC s
```

First we will focus on initialising a new ULS (`newScheduler`). For load balancing purposes, we will spawn threads in a round-robin fashion over the available HECs. For this purpose, we initialise a `TVar` counter, and store into the auxiliary state a pair (c,t) where c is the `SCont`'s home HEC and t is the counter for scheduling. Next, we allocate an empty scheduler data structure (`sched`), and register the current thread with the scheduler activations. This step binds the current thread to participate in user-level scheduling.

All other HECs act as workers (`newHEC`), scheduling the threads that become available on their runqueues. The initial task created on the HEC simply waits for work to become available on the runqueue, and switches to it. Recall that allocating a new `SCont` copies the current `SCont`'s activations to the newly created `SCont`. In this case, the `main` `SCont`'s activations, initialised in `newScheduler`, are copied to the newly allocated `SCont`. As a result, the newly allocated `SCont` shares the same ULS with the main `SCont`. Finally, we run the new `SCont` on a free HEC. Notice that scheduler data structure is not directly accessed in `newHEC`, but accessed through the activation interface.

The Haskell program only needs to prepend the following snippet to the `main` IO computation to utilise the ULS implementation.

```
main = do
  newScheduler
  n <- getNumHECs
  replicateM_ (n-1) newHEC
  ... -- rest of the main code
```

How do we create new user-level threads in this scheduler? For this purpose, we implement a `forkIO` function that spawns a new user-level thread as follows:

```
forkIO :: IO () -> IO SCont
forkIO task = do
  numHECs <- getNumHECs
  -- epilogue: Switch to next thread
  newSC <- newSCont (task >> switch dequeueAct)
  -- Create and initialise new Aux state
  switch $ \s -> do
    dyn <- getAux s
    let (_::Int, t::TVar Int) = fromJust $ fromDynamic dyn
    nextHEC <- readTVar t
    writeTVar t $ (nextHEC + 1) `mod` numHECs
    setAux newSC $ toDyn (nextHEC, t)
    return s
  -- Add new thread to scheduler
  atomically $ enqueueAct newSC
  return newSC
```

`forkIO` function spawns a new thread that runs concurrently with its parent thread. What should happen after such a thread has run to completion? We must request the scheduler to provide us the next thread to run. This is captured in the epilogue `e`, and is appended to the given IO computation `task`. Next, we allocate a new `SCont`, which implicitly inherits the current `SCont`'s scheduler activations. In order to spawn threads in a round-robin fashion, we create a new auxiliary state for the new `SCont` and prepare it such that when unblocked, the new `SCont` is added to the runqueue on HEC `nextHEC`. Finally, the newly created `SCont` is added to the scheduler using its enqueue activation.

The key aspect of this `forkIO` function is that it does not directly access the scheduler data structure, but does so only through the activation interface. As a result, aside from the auxiliary state manipulation, the rest of the code pretty much can stay the same for any

user-level `forkIO` function. Additionally, we can implement a `yield` function similar to the one described in Section 3.3.2. Due to scheduler activations, the interaction with the RTS concurrency mechanisms come for free, and we are done!

4.2 Scheduling algorithms

4.2.1 Last-in-first-out scheduler

A last-in-first-out (LIFO) scheduler is useful for applications such as parallel depth-first search. We can easily modify this FIFO scheduler to perform LIFO scheduling by changing the enqueue activation logic as follows:

```
enqueueActivation:: Sched -> SCont -> STM ()
enqueueActivation (Sched pa) sc = do
  dyn <- getAux sc
  let (hec::Int, _::TVar Int) = fromJust $ fromDynamic dyn
  l <- readTVar $ pa!hec
  writeTVar (pa!hec) $ sc:l
```

The only change is in the last line where we add the given `SCont` to the head of the scheduler instead of the tail to get the necessary behaviour.

4.2.2 Priority-based scheduler

Let us now build a FIFO priority scheduler with low and high priorities. We will save the thread priority in the `SCont`-local state and use two scheduler queues in each HEC for each of the priorities. The modification to the scheduler datastructure is as follows:

```
data Prio = High | Low

data PrioQueues =
  PrioQueues { high :: [SCont], low  :: [SCont] }

newtype Sched = Sched (Array Int (TVar PrioQueues))
```

We modify `dequeueActivation` such that it first examines the high priority queue for `Sconts` before examining the low priority queue.

```
dequeueActivation:: Sched -> SCont -> STM SCont
dequeueActivation (Sched pa) _ = do
  cc <- getCurrentHEC -- get current HEC number
  q <- readTVar $ pa!cc
  case high q of
    [] -> case low q of
          [] -> retry
          x:tl -> do
              let newQ = q { low = tl }
              writeTVar (pa!cc) newQ
              return x
    x:tl -> do
```

```
      let newQ = q { high = tl }
      writeTVar (pa!cc) newQ
      return x
```

We modify `enqueueActivation` such that the given SCont is enqueued to the correct queue based on its priority.

```
enqueueActivation:: Sched -> SCont -> STM ()
enqueueActivation (Sched pa) sc = do
  dyn <- getAux sc
  let (hec::Int, prio::Prio, _::TVar Int) =
    fromJust $ fromDynamic dyn
  q <- readTVar $ pa!hec
  let newQ = case prio of
               High -> q { high = (high q)++[sc] }
               Low  -> q { low  = (low q)++[sc]  }
  writeTVar (pa!hec) newQ
```

Observe that the priority of the `SCont` is obtained from its local state.

The function `forkIO` and `newScheduler` remain the same except for change to `setAux` invocations, which take an additional member in the tuple for the priority. We initialise the newly forked threads to have low priority, and provide a function for updating the priority of an `SCont`:

```
setPriority :: SCont -> Prio -> STM ()
setPriority sc prio = do
  dyn <- getAux sc
  let (h::Int, _::Prio, c::TVar Int) =
    fromJust $ fromDynamic dyn
  setAux sc (toDyn (h, prio, c))
```

Using this function, an `SCont` can update its own priority as follows:

```
setPrioritySelf :: Prio -> IO ()
setPrioritySelf prio =
  switch (\s -> setPriority s prio >> return s)
```

4.2.3 Work-stealing scheduler

Let us now implement a work-stealing scheduler by modifying the FIFO scheduler. The scheduler works by first looking for `SCont` in its own HEC's runqueue before stealing work from other HEC's runqueues. The only modification is to the `dequeueActivation` behaviour.

```
dequeueActivation:: Sched -> SCont -> STM SCont
dequeueActivation (Sched pa) _ = do
  cc <- getCurrentHEC -- get current HEC number
  let (_,end) = bounds pa
```

```
  let targets = cc:(filter (\i -> i /= cc) [0..end])
  res <- foldM checkNext Nothing targets
  case res of
    Nothing -> retry
    Just x -> return x
  where
    checkNext mx hec =
      case mx of
        Nothing -> checkQ hec
        Just x -> return $ Just x
    checkQ hec = do
      l <- readTVar $ pa!hec
      case l of
        [] -> return Nothing
        x:tl -> do
          writeTVar (pa!hec) tl
          return $ Just x
```

We create a list of HEC numbers `targets` where the head of the list is the current HEC number followed by the other HEC numbers. We use the `checkNext` function to check the next HEC for available work. The function `checkQ` checks the given HEC for available work. Thus, by folding over the `targets` list with `checkNext` function, we first check the current HEC for available work, followed by examining other HECs. If an `SCont` is found either in the current HEC or stolen from another HEC, we switch to that `SCont`. Otherwise, the switch transaction retries, putting the current HEC to sleep until work arrives. The rest of the functions remain the same.

Observe that since the entire computation is in a transaction, if the HEC goes to sleep, then none of the HECs have any available `SCont` ready to be scheduled. This avoids the tricky sleep bugs associated with work-stealing multicore schedulers where one has to atomically check all the available work-stealing queues, before atomically putting the HEC to sleep. Implementing the sleep correctly without spurious or lost wakeups is quite tricky. Using an STM for the scheduler simplifies program logic and avoids subtle concurrency bugs.

4.3 Scheduler agnostic user-level MVars

Our scheduler activations abstracts the interface to the ULS's. This fact can be exploited to build scheduler agnostic implementation of user-level concurrency libraries such as MVars. The following snippet describes the structure of an MVar implementation:

```
newtype MVar a = MVar (TVar (MVPState a))
data MVPState a = Full a [(a, SCont)]
                | Empty [(TVar a, SCont)]
```

An MVar is either empty with a list of pending takers, or full with a value and a list of pending putters. An implementation of the `takeMVar` function is presented below:

```
takeMVar :: MVar a -> IO a
```

```
takeMVar (MVar ref) = do
  h <- atomically $ newTVar undefined
  switch $ \s -> do
    st <- readTVar ref
    case st of
      Empty ts -> do
        writeTVar ref $ Empty $ enqueue ts (h,s)
        dequeueAct s
      Full x ts -> do
        writeTVar h x
        case deque ts of
          Nothing -> do
            writeTVar ref $ Empty emptyQueue
          Just ((x', s'), ts') -> do
            writeTVar ref $ Full x' ts'
            enqueueAct s'
        return s
  atomically $ readTVar h
```

If the MVar is empty, the `SCont` enqueues itself into the queue of pending takers. If the MVar is full, `SCont` consumes the value and unblocks the next waiting putter `SCont`, if any. The implementation of `putMVar` is the dual of this implementation. Notice that the implementation only uses the activations to block and resume the `SConts` interacting through the MVar. This allows threads from different ULS's to communicate over the same MVar, and hence the implementation is scheduler agnostic.

5 Semantics

In this section, we present the formal semantics of the concurrency substrate primitives introduced in Section 3.3. We will subsequently utilise the semantics to formally describe the interaction of the ULS with the RTS in Section 6. Our semantics closely follows the implementation. The aim of this is to precisely describe the issues with respect to the interactions between the ULS and the RTS, and have the language to enunciate our solutions.

5.1 Syntax

Figure 5 shows the syntax of program states. The program state P is a soup S of HECs, and a shared heap Θ. The operator $\parallel$ in the HEC soup is associative and commutative. Each HEC is either idle (`Idle`) or a triple $\langle s, M, D \rangle_t$ where s is a unique identifier of the currently executing `SCont`, M is the currently executing term, D represents `SCont`-local state. Each HEC has an optional subscript t, which ranges over $\{Sleeping, Outcall\}$, and represents its current state. The absence of the subscript represents a HEC that is running. As mentioned in Section 3.4, when the program begins execution, the HEC soup has the following configuration:

$$\text{Initial HEC Soup } S = \langle s, M, D \rangle \parallel \texttt{Idle}_1 \parallel \ldots \parallel \texttt{Idle}_{N-1}$$

$$x,y \in \mathit{Variable} \quad r,s \in \mathit{Name}$$

$$\begin{array}{lcl}
\mathit{Md} & ::= & \mathsf{return}\ M \mid M \gg\!= N \\
\mathit{Ex} & ::= & \mathsf{throw}\ M \mid \mathsf{catch}\ M\ N \mid \mathsf{catchSTM}\ M\ N \\
\mathit{Stm} & ::= & \mathsf{newTVar}\ M \mid \mathsf{readTVar}\ r \mid \mathsf{writeTVar}\ r\ M \\
& \mid & \mathsf{atomically}\ M \mid \mathsf{retry} \\
\mathit{Sc} & ::= & \mathsf{newSCont}\ M \mid \mathsf{switch}\ M \mid \mathsf{runOnIdleHEC}\ s \\
\mathit{Sls} & ::= & \mathsf{getAux}\ s \mid \mathsf{setAux}\ s\ M \\
\mathit{Act} & ::= & \mathsf{dequeueAct}\ s \mid \mathsf{enqueueAct}\ s \\
& \mid & \mathsf{setDequeueAct}\ M \mid \mathsf{setEnqueueAct}\ M
\end{array}$$

Term

$$\begin{array}{lcl}
M,N & ::= & r \mid x \mid \lambda x \mathrel{->} M \mid M\ N \mid \ldots \\
& \mid & \mathit{Md} \mid \mathit{Ex} \mid \mathit{Stm} \mid \mathit{Sc} \mid \mathit{Sls} \mid \mathit{Act}
\end{array}$$

$$\begin{array}{rcll}
\text{Program state} & P & ::= & S;\Theta \\
\text{HEC soup} & S & ::= & \emptyset \mid H \parallel S \\
\text{HEC} & H & ::= & \langle s,M,D\rangle \mid \langle s,M,D\rangle_{\mathit{Sleeping}} \\
& & \mid & \langle s,M,D\rangle_{\mathit{Outcall}} \mid \texttt{Idle} \\
\text{Heap} & \Theta & ::= & r \mapsto M \oplus s \mapsto (M,D) \\
\text{SLS Store} & D & ::= & (M,N,r) \\
\text{IO Context} & \mathbb{E} & ::= & \bullet \mid \mathbb{E} \gg\!= M \mid \mathsf{catch}\ \mathbb{E}\ M \\
\text{STM Context} & \mathbb{P} & ::= & \bullet \mid \mathbb{P} \gg\!= M
\end{array}$$

Fig. 5: Syntax of terms, states, contexts, and heaps

where M is the main computation, and all other HECs are idle. We represent the `SCont`-local state D as a tuple with two terms and a name (M,N,r). Here, M, N, and r are the dequeue activation, enqueue activation, and a `TVar` representing the auxiliary storage of the current `SCont` on this HEC. For perspicuity, we define accessor functions as shown below.

$$\mathsf{deq}(M,\_,\_) = M \quad \mathsf{enq}(\_,M,\_) = M \quad \mathsf{aux}(\_,\_,r) = r$$

The primitives under the collection *Sls* and *Act* in Figure 5 read from and write to the `SCont`-local states. The semantics of these primitives is discussed in Section 5.5. The heap Θ is a disjoint finite map of:

- $(r \mapsto M)$, maps the identifier r of a transactional variable, or `TVar`, to its value.
- $(s \mapsto (M,D))$, maps the identifier s of an `SCont` to its current state.

In a program state $(S;\Theta)$, an `SCont` with identifier s appears *either* as the running `SCont` in a HEC $\langle s,M,D\rangle_t \in S$, *or* as a binding $s \mapsto (M,D)$ in the heap Θ, but never in both. The distinction has direct operational significance: an `SCont` running in a HEC has part

of its state loaded into machine registers, whereas one in the heap is entirely passive. In both cases, however, the term M has type `IO()`, modelling the fact that concurrent Haskell threads can perform I/O.

The number of HECs remains constant, and each HEC runs one, and only one `SCont`. The business of multiplexing multiple `SConts` onto a single HEC is what the scheduler is for, and is organised by Haskell code using the primitives described in this section. Finally, $\mathbb{E}$ and $\mathbb{P}$ represent the evaluation context for reduction under IO and STM monads.

$$s \in \mathit{Name} \qquad M \in \mathit{Term}$$

$$\begin{array}{rrcl} \text{RTS actions} & a & ::= & \mathit{Tick} \mid \mathit{STMBlock}\ s \mid \mathit{RetrySTM}\ s \mid \mathit{OC}\ s \\ & & \mid & \mathit{OCSteal}\ s \mid \mathit{OCRet}\ s\ M \mid \mathit{BlockBH}\ s \mid \mathit{ResumeBH}\ s \\ \text{Upcalls} & u & ::= & \mathit{enq}\ s \mid \mathit{deq} \end{array}$$

$$\begin{array}{rl} \text{Top-level} & S;\Theta \stackrel{a}{\Longrightarrow} S';\Theta' \\ \text{HEC} & H;\Theta \Longrightarrow H';\Theta' \\ \text{Purely functional} & M \to N \\ \text{STM} & s,M,D;\Theta;\twoheadrightarrow M';\Theta' \\ \text{Upcall} & H;\Theta \stackrel{u}{\hookrightarrow} H';\Theta' \end{array}$$

Fig. 6: Transition relations

Our semantics uses different transition relations to describe the operational behaviour under different contexts. The program makes a transition from one state to another through the top-level program small-step transition relation: $S;\Theta \stackrel{a}{\Longrightarrow} S';\Theta'$. This says that the program makes a transition from $S;\Theta$ to $S';\Theta'$, possibly interacting with the underlying RTS through action a. We return to these RTS interactions in Section 6, and we omit a altogether if there is no interaction. The HEC transitions enable one of the HECs to perform a step, and possibly modifying the heap as a result. Purely functional transitions simply reduce a term, and by definition do not touch the heap.

The STM transitions are used to capture the behaviour of the HEC running a transaction. An STM transition is of the form $s;M;D;\Theta \twoheadrightarrow M';\Theta'$, where M is the current monadic term under evaluation, and the heap Θ binds transactional variables to their current values. The current `SCont` s and its local state D are read-only. Finally, upcall transitions capture the behaviour of handling upcalls at a HEC, and u ranges over the kind of activation. We will come back to RTS interactions in Section 6.

5.2 Basic transitions

The basic transitions are presented in Figure 7. Rule OneHEC says that if one HEC H can take a step with the single-HEC transition relation, then the whole machine can take a step. As usual, we assume that the soup S is permuted to bring a runnable HEC to the left-hand end of the soup, so that OneHEC can fire. Similarly, Rule PureStep enables

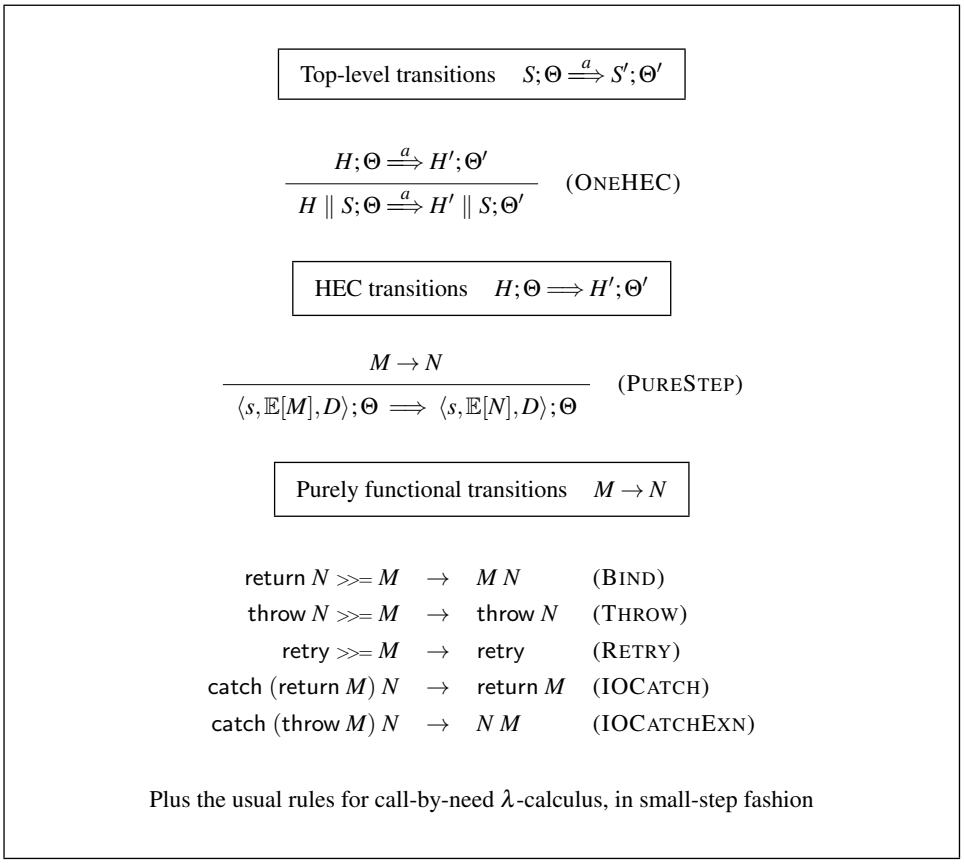


Fig. 7: Operational semantics for basic transitions

one of the HECs to perform a purely functional transition under the evaluation context $\mathbb{E}$ (defined in Figure 5). There is no action a on the arrow because this step does not interact with the RTS. Notice that PureStep transition is only possible if the HEC is in running state (with no subscript). The purely functional transitions $M \to N$ include β-reduction, arithmetic expressions, case expressions, monadic operations return, bind, throw, catch, and so on according to their standard definitions. Bind operation on the transactional memory primitive retry simply reduces to retry (Figure 7). These primitives represent blocking actions under transactional memory and will be dealt with in Section 6.2.

5.3 Transactional memory

Since the concurrency substrate primitives utilise STM as the sole synchronisation mechanism, we will present the formal semantics of basic STM operations in this section. We will build upon the basic STM formalism to formally describe the behaviour of concurrency substrate primitives in the following sections. The semantics of existing STM primitives in GHC is preserved in the new implementation.

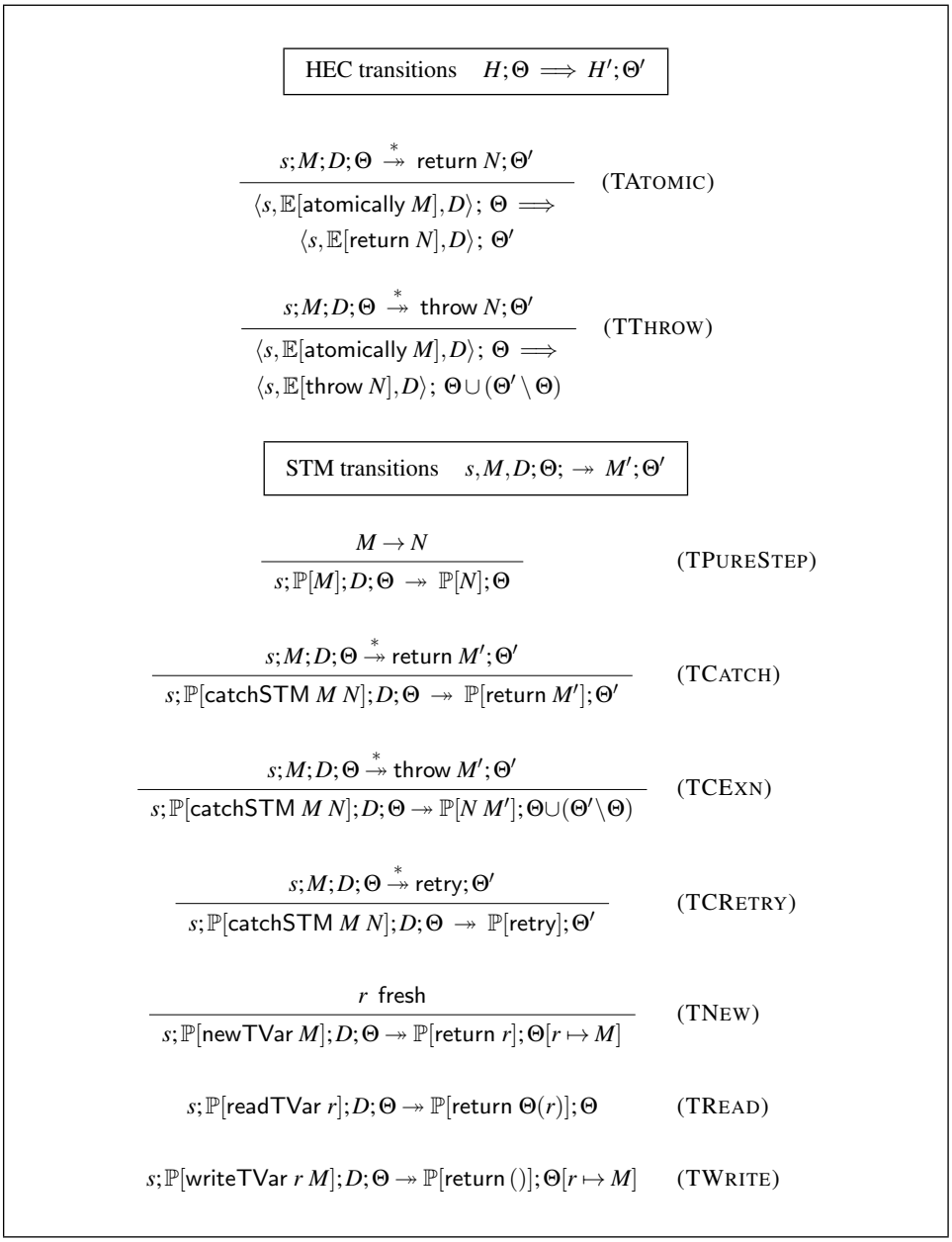


Fig. 8: Operational semantics for software transactional memory

Figure 8 presents the semantics of non-blocking STM operations. The semantics of blocking operations is deferred until Section 6.2. Recall that an STM transition is of the form $s;M;D;\Theta \twoheadrightarrow M';\Theta'$, where M is the current monadic term under evaluation, and the heap Θ binds transactional variables to their current values. The current `SCont` s and its local state D are read-only, and are not used at all in this section, but will be needed when

manipulating SCont-local state. The reduction produces a new term M' and a new heap Θ'. Rule TPureStep is similar to PureStep rule in Figure 7. STM allows creating (TNew), reading (TRead), and writing (TWrite) to transactional variables.

The most important rule is TAtomic which combines multiple *STM* transitions into a single *program* transition. The notation $\overset{*}{\twoheadrightarrow}$ stands for the transitive closure of $\twoheadrightarrow$. The antecedent in the rule says that if the term M can be reduced in multiple steps to return N, possibly modifying the heap from Θ to Θ', then M is *atomically* evaluated with the same result. Thus, other HECs are not allowed to witness the intermediate effects of the transaction.

The semantics of exception handling under STM is interesting (rules TCExn and TThrow). Since an exception can carry a TVar allocated in the aborted transaction, the effects of the current transaction are undone except for the newly allocated TVars. Otherwise, we would have dangling pointer corresponding to such TVars. Rule TCRetry simply propagates the request to retry the transaction through the context. The act of blocking, wake up and undoing the effects of the transaction are handled in Section 6.2.

5.4 SCont *semantics*

The semantics of SCont primitives are presented in Figure 9. Each SCont has a distinct identifier s (concretely, its heap address). An SCont's state is represented by the pair (M, D) where M is the term under evaluation and D is the local state.

Rule NewSCont binds the given IO computation and a new SCont-local state pair to a new SCont s', and returns s'. Notice that the newly created SCont inherits the activations of the calling SCont. This implicitly associates the new SCont with the invoking SCont's scheduler.

The rules for switch (SwitchSelf, Switch, and SwitchExn) begin by atomically evaluating the body of switch M applied to the current SCont s. If the resultant SCont is the same as the current one (SwitchSelf), then we simply commit the transaction and there is nothing more to be done. If the resultant SCont s' is different from the current SCont s (Switch), we transfer control to the new SCont s' by making it the running SCont and saving the state of the original SCont s in the heap. If the switch primitive happens to throw an exception, the updates by the transaction are discarded (SwitchExn).

Observe that the Switch rule gets stuck if the resultant SCont s' does not have a binding in Θ'. This can happen if the target SCont s' is already running on a different HEC. This design enforces linear use of SConts. As mentioned earlier, our implementation enforces this property by attaching an transactional status variable to each SCont and updating it accordingly during context switches.

The alert reader will notice that the rules for switch duplicate much of the paraphernalia of an atomic transaction (Figure 8), but that is unavoidable because the switch to a new continuation must form part of the same transaction as the argument computation.

5.5 Semantics of local state manipulation

In our formalisation, we represent local state D as a tuple with two terms and a name (M, N, r) (Figure 5), where M, N and r are dequeue activation, enqueue activation, and a

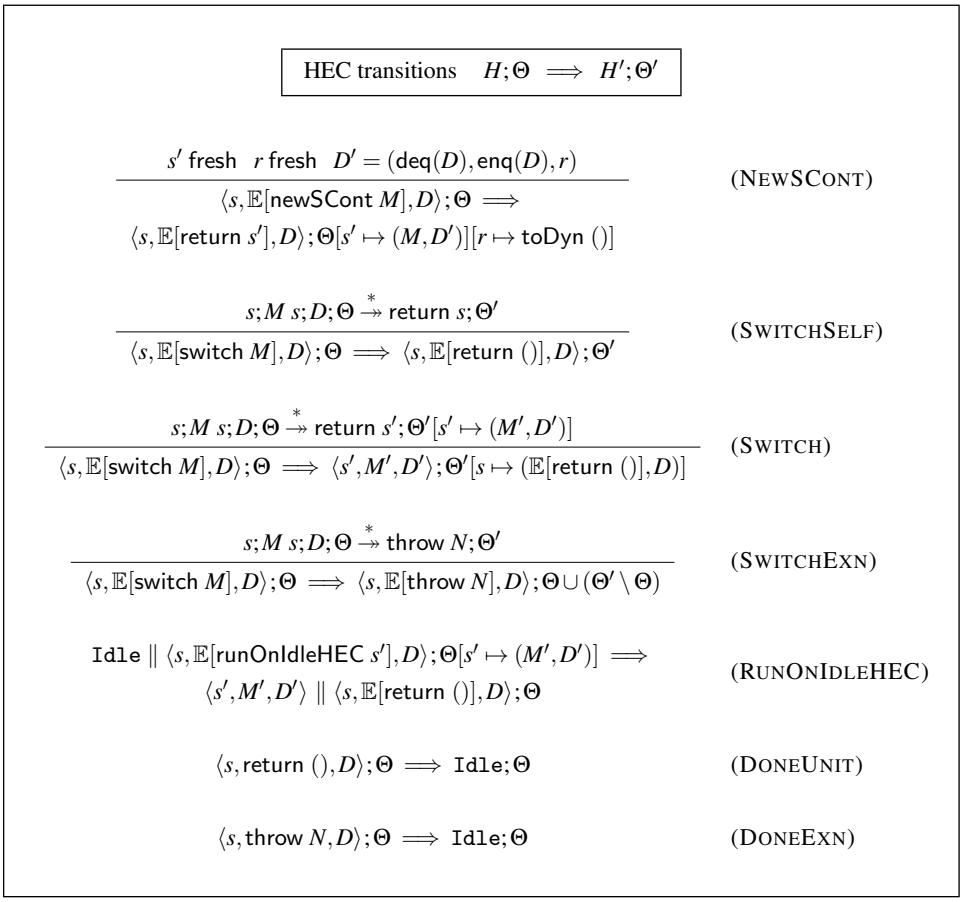


Fig. 9: Operational semantics for `SCont` manipulation

`TVar` representing auxiliary storage, respectively. The precise semantics of activations and stack-local state manipulation is given in Figure 10.

Our semantics models the auxiliary field in the `SCont`-local state as a `TVar`. It is initialised to a dynamic unit value toDyn () when an new `SCont` is created (rule NEWSCONT in Figure 9). The rules SETAUXSELF and SETAUXOTHER update the aux state of a `SCont` by writing to the `TVar`. There are two cases, depending on whether the `SCont` is running in the current HEC, or is passive in the heap. The aux-state is typically used to store scheduler accounting information, and is most likely to be updated in the activations, being invoked by some other `SCont` or the RTS. This is the reason why we model aux-state as a `TVar` and allow it to be modified by some other `SCont`. If the target of the `setAux` is running in another HEC, no rule applies, and we raise a runtime exception. This is reasonable: one HEC should not be poking into another running HEC's state. The rules for `getAux` also have two cases.

An `SCont`'s activations can be invoked using the dequeueAct and enqueueAct primitives. Invoking an `SCont`'s own activation is straight-forward; the activation is fetched

HEC transitions $H;\Theta \implies H';\Theta'$

(SETDEQUEUEACT)

$$\langle s, \mathbb{E}[\mathsf{setDequeueAct}\ M], (b,u,r)\rangle;\Theta \implies \langle s, \mathbb{E}[\mathsf{return}\ ()], (M,u,r)\rangle;\Theta$$

(SETENQUEUEACT)

$$\langle s, \mathbb{E}[\mathsf{setEnqueueAct}\ M], (b,u,r)\rangle;\Theta \implies \langle s, \mathbb{E}[\mathsf{return}\ ()], (b,M,r)\rangle;\Theta$$

STM transitions $s;M;D;\Theta \twoheadrightarrow M';\Theta'$

(GETAUXSELF)

$$s;\mathbb{P}[\mathsf{getAux}\ s];D;\Theta \twoheadrightarrow \mathbb{P}[\mathsf{return}\ \mathsf{aux}(D)];\Theta$$

(SETAUXSELF)

$$s;\mathbb{P}[\mathsf{setAux}\ s\ M];D;\Theta \twoheadrightarrow \mathbb{P}[\mathsf{return}\ ()];\Theta[\mathsf{aux}(D) \mapsto M]$$

(INVOKEDEQUEUEACTSELF)

$$s;\mathbb{P}[\mathsf{dequeueAct}\ s];D;\Theta \twoheadrightarrow \mathbb{P}[\mathsf{deq}(D)\ s];\Theta$$

(INVOKEENQUEUEACTSELF)

$$s;\mathbb{P}[\mathsf{enqueueAct}\ s];D;\Theta \twoheadrightarrow \mathbb{P}[\mathsf{enq}(D)\ s];\Theta$$

(GETAUXOTHER)

$$s;\mathbb{P}[\mathsf{getAux}\ s'];D;\Theta[s' \mapsto (M',D')] \twoheadrightarrow \mathbb{P}[\mathsf{return}\ \mathsf{aux}(D')];\Theta[s' \mapsto (M',D')]$$

(SETAUXOTHER)

$$s;\mathbb{P}[\mathsf{setAux}\ s'\ M];D;\Theta[s' \mapsto (M',D')] \twoheadrightarrow \mathbb{P}[\mathsf{return}\ ()];\Theta[s' \mapsto (M',D')][\mathsf{aux}(D') \mapsto M]$$

(INVOKEDEQUEUEACTOTHER)

$$s;\mathbb{P}[\mathsf{dequeueAct}\ s'];D;\Theta[s' \mapsto (M',D')] \twoheadrightarrow \mathbb{P}[\mathsf{deq}(D')\ s'];\Theta[s' \mapsto (M',D')]$$

(INVOKEENQUEUEACTOTHER)

$$s;\mathbb{P}[\mathsf{enqueueAct}\ s'];D;\Theta[s' \mapsto (M',D')] \twoheadrightarrow \mathbb{P}[\mathsf{enq}(D')\ s'];\Theta[s' \mapsto (M',D')]$$

Fig. 10: Operational semantics for manipulating activations and auxiliary state.

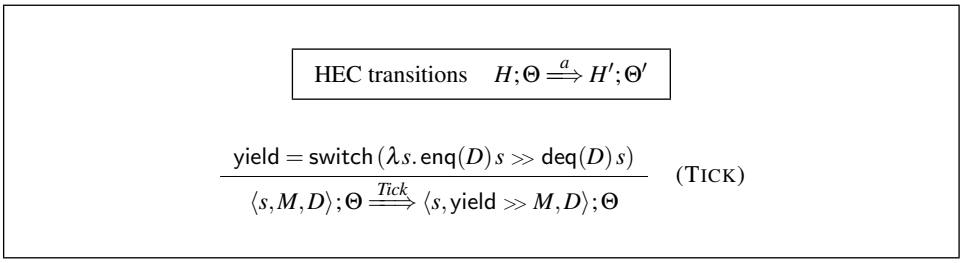


Fig. 11: Handling timer interrupts

from the local state and applied to the current `SCont` (rules INVOKEDEQUEUEACTSELF). We do allow activations of an `SCont` other than the current `SCont` to be invoked (rule INVOKEDEQUEUEACTOTHER and INOKEENQUEUEACTOTHER). Notice that in order to invoke the activations of other `SConts`, the `SCont` must be passive on the heap, and currently not running.

We allow an `SCont` to modify its *own* activations, and potentially migrate to another ULS. In addition, updating own activations allows the initial thread evaluating the `main` IO computation to initialise its activations, and to participate in user-level scheduling. In the common use case, once an `SCont`'s activations are initialised, we do not expect them to change. Hence, we do not store the activations in a `TVar`, but rather directly in the underlying TSO object field. The avoids the overheads of transactional access of activations.

6 Interaction with the RTS

The key aspect of our design is composability of ULS's with the existing RTS concurrency mechanisms (Section 3.1). In this section, we will describe in detail the interaction of RTS concurrency mechanisms and the ULS's. The formalisation brings out the tricky cases associated with the interaction between the ULS and the RTS.

6.1 Timer interrupts

In GHC, concurrent threads are preemptively scheduled. The RTS maintains a timer that ticks, by default, every 20ms. On a tick, the current `SCont` needs to be de-scheduled and a new `SCont` from the scheduler needs to be scheduled. The semantics of handling timer interrupts is shown in Figure 11.

The *Tick* label on the transition arrow indicates an interaction with the RTS; we call such a label an *RTS-interaction*. In this case the RTS-interaction *Tick* indicates that the RTS wants to signal a timer tick[3]. The transition here injects `yield` into the instruction stream of the `SCont` running on this HEC, at a GC safe point, where `yield` behaves just like the definition in Section 3.3.2.

[3] Technically we should ensure that every HEC receives a tick, and of course our implementation does just that, but we elide that here.

6.2 STM blocking operations

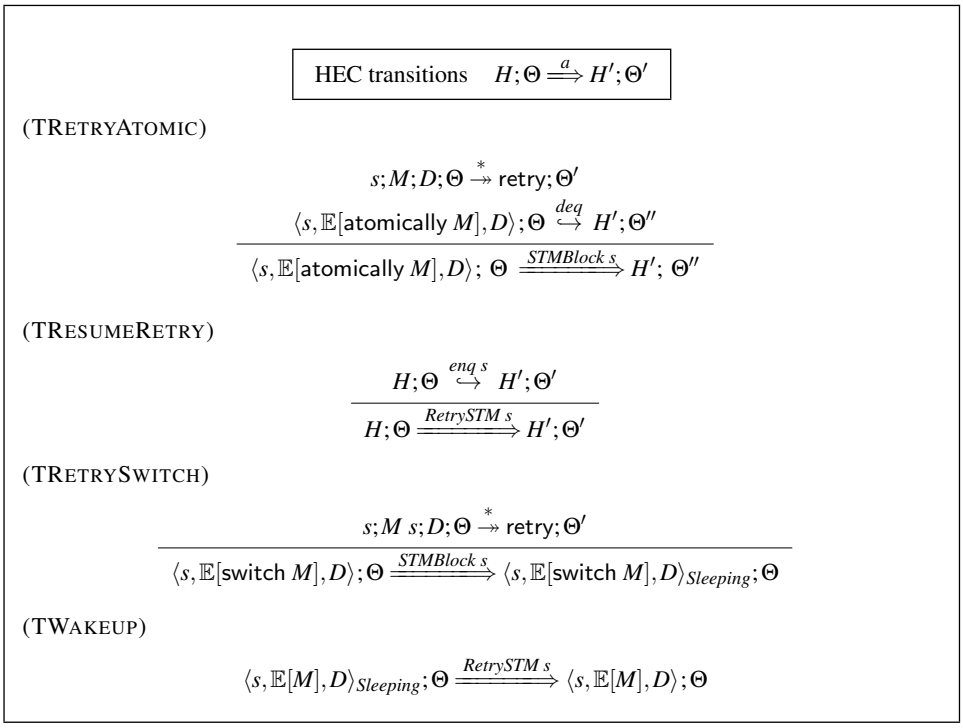


Fig. 12: STM Retry

As mentioned before (Section 3.4), STM supports blocking operations through the `retry` primitive. Figure 12 gives the semantics for STM `retry` operation.

6.2.1 Blocking the `SCont`

Rule TRETRYATOMIC is similar to TTHROW in Figure 8. It runs the transaction body M; if the latter terminates with `retry`, it abandons the effects embodied in Θ', reverting to Θ. But, unlike TTHROW it then uses an auxiliary rule $\overset{deq}{\hookrightarrow}$, defined in Figure 13, to fetch the next `SCont` to switch to. The transition in TRETRYATOMIC is labelled with the RTS interaction *STMBlock s*, indicating that the RTS assumes responsibility for s after the reduction.

The rules presented in Figure 13 are the key rules in abstracting the interface between the ULS and the RTS, and describe the invocation of upcalls. In the sequel, we will often refer to these rules in describing the semantics of the RTS interactions. Rule UPDEQUEUE in Figure 13 stashes s (the `SCont` to be blocked) in the heap Θ, instantiates an ephemeral `SCont` that fetches the `dequeue` activation b from s's local state D, and switches to the `SCont` returned by the `dequeue` activation. s' is made the running `SCont` on this HEC.

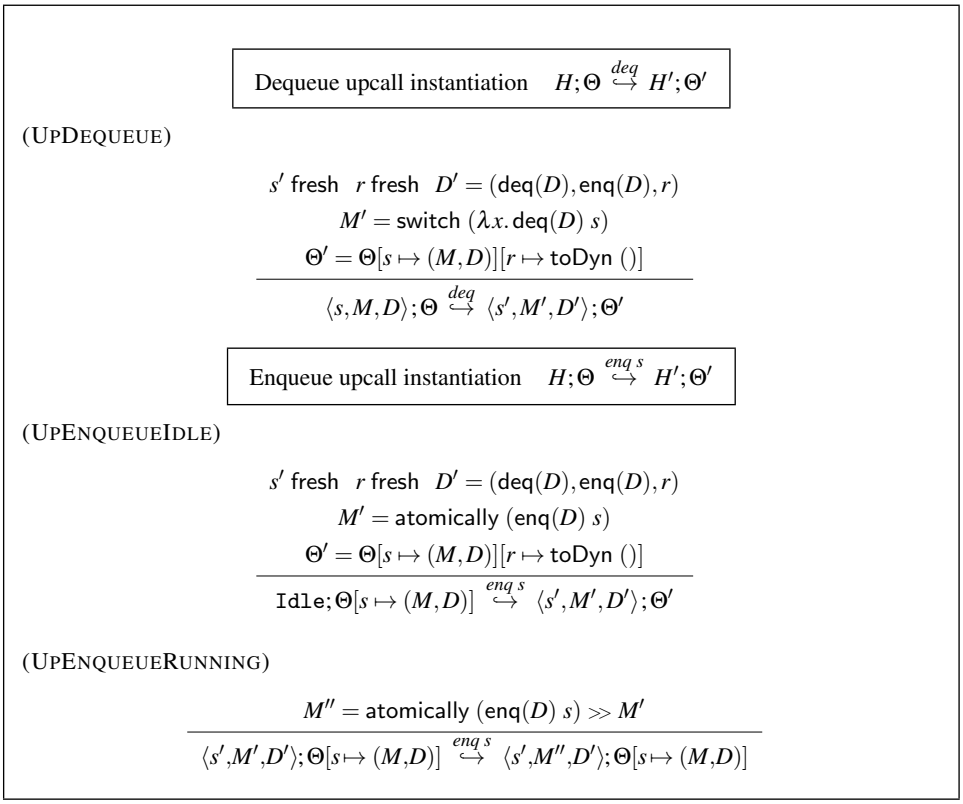


Fig. 13: Instantiating upcalls

It is necessary that the dequeue upcall be performed on a new `SCont` s', and not on the `SCont` s being blocked. At the point of invocation of the dequeue upcall, the RTS believes that the blocked `SCont` s is completely owned by the RTS, not running, and available to be resumed. Invoking the dequeue upcall on the blocked `SCont` s can lead to a race on s between multiple HECs if s happens to be unblocked and enqueued to the scheduler before the `switch` transaction is completed.

6.2.2 Resuming the `SCont`

Some time later, the RTS will see that some thread has written to one of the `TVars` read by s's transaction, so it will signal an *RetrySTM s* interaction (rule TRESUMERETRY). Again, we use an auxiliary transition $\overset{enq\ s}{\hookrightarrow}$ to enqueue the `SCont` to its scheduler (Figure 13).

Unlike $\overset{deq}{\hookrightarrow}$ transition, unblocking an `SCont` has nothing to do with the computation currently running on any HEC. If we find an idle HEC (rule UPENQUEUEIDLE), we instantiate a new ephemeral `SCont` s' to enqueue the `SCont` s. The actual unblock operation is achieved by fetching `SCont` s's `enqueue` activation, applying it to s and atomically performing the resultant STM computation. If we do not find any idle HECs (rule UPEN-

QUEUERUNNING), we pick one of the running HECs, prepare it such that it first unblocks the `SCont` *s* before resuming the original computation.

6.2.3 HEC sleep and wakeup

Recall that invoking `retry` within a `switch` transaction or `dequeue` activation puts the HEC to sleep (Section 3.4). Also, notice that the `dequeue` activation is always invoked by the RTS from a `switch` transaction (Rule UPDEQUEUE). This motivates rule TRETRYSWITCH: if a `switch` transaction blocks, we put the whole HEC to sleep. Then, dual to TRESUMERETRY, rule TWAKEUP wakes up the HEC when the RTS sees that the transaction may now be able to make progress.

6.2.4 Implementation of upcalls

Notice that the rules UPDEQUEUE and UPENQUEUEIDLE in Figure 13 instantiate a fresh `SCont`. The freshly instantiated `SCont` performs just a single transaction; `switch` in UPDEQUEUE and `atomically` in UPENQUEUEIDLE, after which it is garbage-collected. Since instantiating a fresh `SCont` for every upcall is unwise, the RTS maintains a dynamic pool of dedicated *upcall SConts* for performing the upcalls. It is worth mentioning that we need an "upcall `SCont` pool" rather than a single "upcall `SCont`" since the upcall transactions can themselves get blocked synchronously on STM `retry` as well as asynchronously due to optimisations for lazy evaluation (Section 6.5).

6.3 Safe foreign function calls

Foreign calls in GHC are highly efficient but intricately interact with the scheduler (Marlow *et al.*, 2004). Much of the efficiency owes to the RTS's task model. Each HEC is animated by one of a *pool* of tasks (OS threads); the current task may become blocked in a foreign call (e.g., a blocking I/O operation), in which case another task takes over the HEC. However, at most only one task ever has exclusive access to a HEC.

GHC's task model ensures that a HEC performing a safe-foreign call only blocks the Haskell thread (and the task) making the call but not the other threads running on the HEC's scheduler. However, it would be unwise to switch the thread (and the task) on every foreign call as most invocations are expected to return in a timely fashion. In this section, we will discuss the interaction of safe-foreign function calls and the ULS. In particular, we restrict the discussion to outcalls — calls made from Haskell to C.

Our decision to preserve the task model in the RTS allows us to delegate much of the work involved in safe foreign call to the RTS. We only need to deal with the ULS interaction, and not the creation and coordination of tasks. The semantics of foreign call handling is presented in Figure 14. Rule OCBLOCK illustrates that the HEC performing the foreign call moves into the *Outcall* state, where it is no longer runnable. In the fast path (rule OCRETFAST), the foreign call returns immediately with the result M, and the HEC resumes execution with the result plugged into the context.

In the slow path, the RTS may decide to pay the cost of task switching and resume the scheduler (rule OCSTEAL). The scheduler is resumed using the dequeue upcall. Once

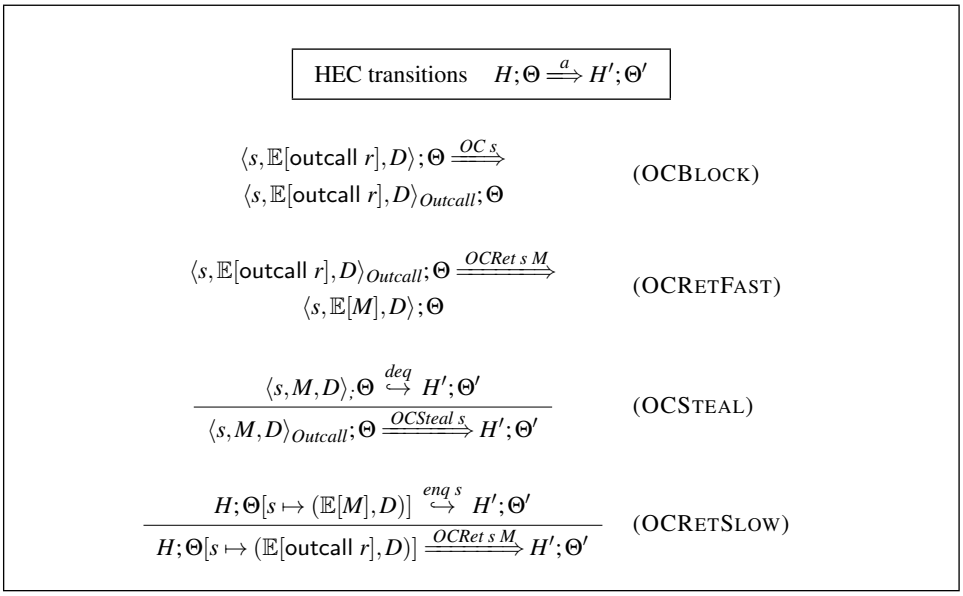


Fig. 14: Safe foreign call transitions

the foreign call eventually returns, the `SCont` *s* blocked on the foreign call can be resumed. Since we have already resumed the scheduler, the correct behaviour is to prepare the `SCont` *s* with the result and add it to its ULS. Rule OCRETSLOW achieves this through enqueue upcall.

6.4 Timer interrupts and transactions

What if a timer interrupt occurs during a transaction? The (TICK) rule of Section 6.1 is restricted to *HEC transitions*, and says nothing about *STM transitions*. One possibility (Plan A) is that transactions should not be interrupted, and ticks should only be delivered at the end. This is faithful to the semantics expressed by the rule, but it does mean that a rogue transaction could completely monopolise a HEC.

An alternative possibility (Plan B) is for the RTS to roll the transaction back to the beginning, and then deliver the tick using rule (TICK). That too is implementable, but this time the risk is that a slightly-too-long transaction would always be rolled back, so it would never make progress.

Our implementation behaves like Plan B, but gives better progress guarantees, while respecting the same semantics. Rather than rolling the transaction back, the RTS suspends the transaction mid-flight. None of its effects are visible to other `SConts`; they are confined to its `SCont`-local transaction log. When the `SCont` is later resumed, the transaction continues from where it left off, rather than starting from scratch. Of course, time has gone by, so when it finally tries to commit there is a higher chance of failure, but at least uncontended access will go through.

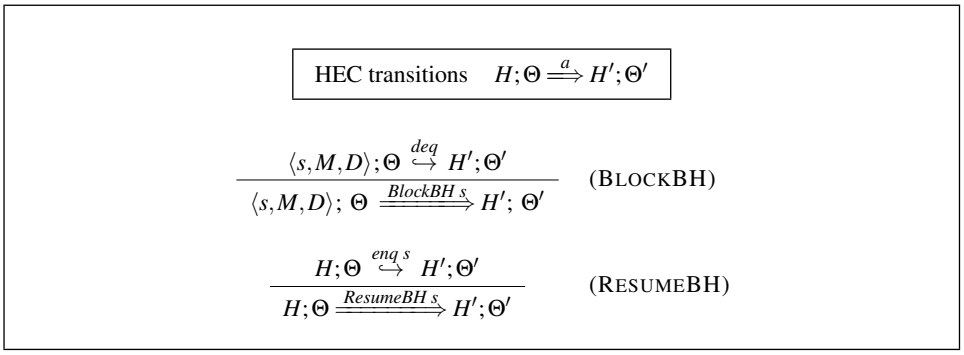


Fig. 15: Black holes

That is fine for vanilla `atomically` transactions. But what about the special transactions run by `switch`? If we are in the middle of a `switch` transaction, and suspend it to deliver a timer interrupt, rule (TICK) will initiate ... a `switch` transaction! And that transaction is likely to run the very same code that has just been interrupted. It seems much simpler to revert to Plan A: the RTS does not deliver timer interrupts during a `switch` transaction. If the scheduler has rogue code, then it will monopolise the HEC with no recourse.

6.5 Black holes

In a concurrent Haskell program, a thread `A` may attempt to evaluate a thunk `x` that is already being evaluated by another thread `B`. To avoid duplicate evaluation the RTS (in intimate cooperation with the compiler) arranges for `B` to *blackhole* the thunk when it starts to evaluate `x`. Then, when `A` attempts to evaluate `x`, it finds a black hole, so the RTS enqueues `A` to await its completion. When `B` finishes evaluating `x` it updates the black hole with its value, and makes any queued threads runnable. This mechanism, and its implementation on a multicore, is described in detail in earlier work (Harris *et al.*, 2005b).

Clearly this is another place where the RTS may initiate blocking. We can describe the common case with rules similar to those of Figure 12, with rules shown in Figure 15. The RTS initiates the process with a *BlockBH s* action, taking ownership of the `SCont` *s*. Later, when the evaluation of the thunk is complete, the RTS initiate an action *ResumeBH s*, which returns ownership to *s*'s scheduler.

But these rules only apply to *HEC transitions*, outside transactions. What if a black hole is encountered during an STM transaction? We addressed this same question in the context of timer interrupts, in Section 6.4, and we adopt the same solution. The RTS behaves as if the black-hole suspension and resumption occurred just before the transaction, but the implementation actually arranges to resume the transaction from where it left off.

Just as in Section 6.4, we need to take particular care with `switch` transactions. Suppose a `switch` transaction encounters a black-holed thunk under evaluation by some other `SCont B`; and suppose we try to suspend the transaction (either mid-flight or with roll-back) using rule (BLOCKBH). Then the very next thing we will do (courtesy of $\overset{deq}{\hookrightarrow}$) is a `switch` transaction; and that is very likely to encounter the very same thunk. Moreover, it

is just possible that the thunk is under evaluation by an `SCont` in this very scheduler's run-queue, so the black hole is preventing us from scheduling the very `SCont` that is evaluating it. Deadlock beckons!

In the case of timer interrupts we solved the problem by switching them off in `switch` transactions, and it turns out that we can effectively do the same for thunks. Since we cannot sensibly suspend the `switch` transaction, we must find a way for it to make progress. Fortunately, GHC's RTS allows us to *steal* the thunk from the `SCont` that is evaluating it, and that suffices. The details are beyond the scope of this paper, but the key moving parts are already part of GHC's implementation of asynchronous exceptions (Marlow *et al.*, 2001; Reid, 1999).

6.6 Interaction with RTS MVars

An added advantage of our scheduler activation interface is that we are able to reuse the existing MVar implementation in the RTS. Whenever an `SCont` s needs to block on or unblock from an MVar, the RTS invokes the $\overset{deq}{\hookrightarrow}$ or $\overset{enq\ s}{\hookrightarrow}$ upcall, respectively. This significantly reduces the burden of migrating to a ULS implementation.

6.7 Asynchronous exceptions

GHC's supports *asynchronous exceptions* in which one thread can send an asynchronous interrupt to another (Marlow *et al.*, 2001). This is a very tricky area; for example, if a thread is blocked on a user-level `MVar` (Section 4.3), and receives an exception, it should wake up and do something — even though it is linked onto an unknown queue of blocked threads. Our implementation does in fact handle asynchronous exceptions. However, we are not yet happy with the details of the design, and elide presenting the design in this paper.

6.8 On the correctness of user-level schedulers

While the concurrency substrate exposes the ability to build ULS's, the onus is on the scheduler implementation to ensure that it is sensible. The invariants such as not switching to a running thread, or a thread blocked in the RTS, are not statically enforced by the concurrency substrate, and care must be taken to preserve these invariants. Our implementation dynamically enforces such invariants through runtime assertions. We also expect that the activations do not raise an exception that escape the activation. Activations raising exceptions indicates an error in the ULS implementation, and the substrate simply reports an error to the standard error stream.

The fact that the scheduler itself is now implemented in user-space complicates error recovery and reporting when threads become unreachable. A thread suspended on an ULS may become unreachable if the scheduler data structure holding it becomes unreachable. A thread indefinitely blocked on an RTS `MVar` operation is raised with an exception and added to its ULS. This helps the corresponding thread from recovering from indefinitely blocking on an `MVar` operation.

However, the situation is worse if the ULS itself becomes unreachable; there is no scheduler to run this thread! Hence, salvaging such a thread is not possible. In this case,

immediately after garbage collection, our implementation logs an error message to the standard error stream along with the unreachable `SCont` (thread) identifier.

7 Results

Our implementation, which we call Lightweight Concurrency (LWC) GHC, is a fork of GHC,[4] and supports all of the features discussed in the paper. We have been very particular not to compromise on any of the existing features in GHC. As shown in Section 4, porting existing concurrent Haskell program to utilise a ULS only involves few additional lines of code.

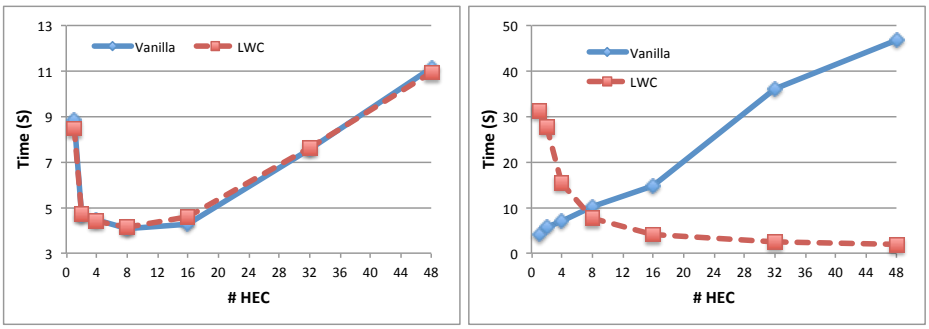
In order to evaluate the performance and quantify the overheads of the LWC substrate, we picked the following Haskell concurrency benchmarks from The Computer Language Benchmarks Game (Shootout, 2014): `k-nucleotide`, `mandelbrot`, `spectral-norm` and `chameneos-redux`. We also implemented a concurrent prime number generator using sieve of Eratosthenes (`primes-sieve`), where the threads communicate over the MVars. For our experiments, we generated the first 10000 primes. The benchmarks offer varying degrees of parallelisation opportunity. `k-nucleotide`, `mandelbrot` and `spectral-norm` are computation intensive, while `chameneos-redux` and `primes-sieve` are communication intensive and are specifically intended to test the overheads of thread synchronisation.

The LWC version of the benchmarks utilised the scheduler and the MVar implementation described in Section 4, except that instead of utilising a list to represent the queue, we use a functional double-ended queue similar to the one in `Data.Sequence`. For comparison, the benchmark programs were also implemented using `Control.Concurrent` on a vanilla GHC implementation. The default thread scheduler of GHC uses one scheduler queue per HEC, where threads are initially spawned on the same HEC and work is shared between the HECs at the end of minor collections. Recall that vanilla GHC's thread scheduler is implemented in the RTS and incorporates a number of heuristics for improving throughput, whereas the ULS is a simple round-robin scheduler. Experiments were performed on a 48-core AMD Opteron server, and the GHC version was 7.7.20130523.

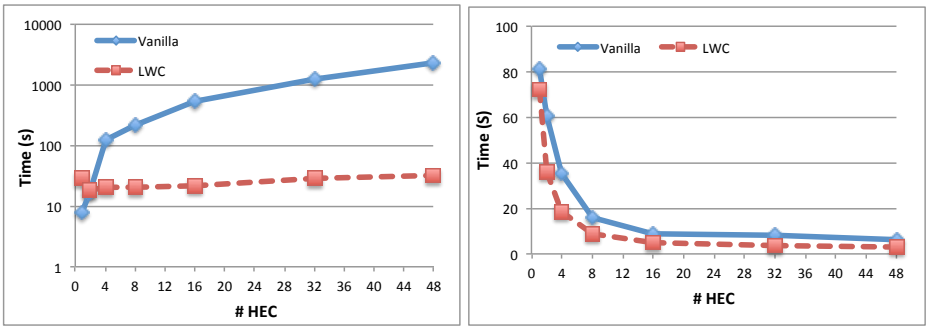
The performance results are presented in Figure 16. We vary the number of HECs and measure the running time. All the times reported are in seconds. For each benchmark, we also measure the baseline numbers using the vanilla GHC program compiled without the `-threaded` option. This non-threaded baseline does not support multi-processor execution of concurrent programs, but also does not include the mechanisms necessary for (and the overheads included in) multi-processor synchronisation. Hence, the non-threaded version of a program running on 1 processor is faster than the corresponding threaded version.

In `k-nucleotide` and `spectral-norm` benchmarks, the performance of the LWC version was indistinguishable from the vanilla version. The threaded versions of the benchmark programs were fastest on 8 HECs and 48 HECs on `k-nucleotide` and `spectral-norm`, respectively. In `mandelbrot` benchmark, the LWC version was 2× *faster* than the vanilla version. While the vanilla version was 12× faster than the baseline, the LWC version was 24× faster. In the vanilla GHC, the RTS thread scheduler by default spawns

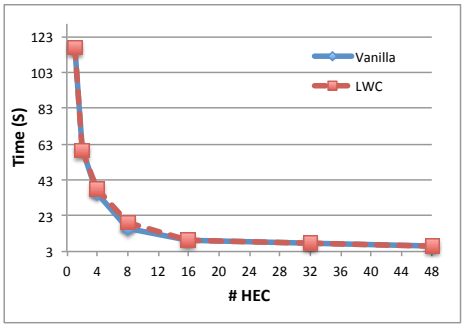
[4] The development branch of LWC substrate is available at `https://github.com/ghc/ghc/tree/ghc-lwc2`



(a) `k-nucleotide` (b) `primes-sieve`



(c) `chameneos-redux` (d) `mandelbrot`



(e) `spectral-norm`

Fig. 16: Performance comparison of vanilla GHC vs LWC GHC.

a thread on the current HEC and only shares the thread with other HECs if they are idle. The LWC scheduler (described in Section 4) spawns threads by default in a round-robin fashion on all HECs. This simple scheme happens to work better in `mandelbrot` since the program is embarrassingly parallel.

In `chameneos-redux` benchmark (note that the y-axis is in logarithmic scale), the LWC version was 3.9× slower than the baseline on 1 HEC and 2.6× slower on 2 HECs, and slows down with additional HECs as `chameneos-redux` does not present much parallelisation opportunity. The vanilla `chameneos-redux` program was fastest on 1 HEC, and

was 1.24× slower than the baseline. With additional HECs, the vanilla version the default work-stealing algorithm baked into the runtime system brings all of the threads to a single HEC and performs sub-optimal scheduling. The performance continues to worsen with additional HECs.

In `primes-sieve` benchmark, while the LWC version was 6.8× slower on one HEC, the vanilla version was 1.3X slower, when compared to the baseline. The vanilla version is fastest on 1 HEC and slows down with additional HECs similar to `chameneos-redux` benchmark. In `chameneos-redux` and `primes-sieve`, we observed that the LWC implementation spends around half of its execution running the transactions for invoking the activations or `MVar` operations. Additionally, in these benchmarks, LWC version performs 3X-8× more allocations than the vanilla version. Most of these allocations are due to the data structure used in the ULS and the MVar queues. In the vanilla `primes-sieve` implementation, these overheads are negligible. This is an unavoidable consequence of implementing concurrency libraries in Haskell.

Luckily, these overheads are parallelisable. In `primes-sieve` benchmark, while the vanilla version was fastest on 1 HEC, LWC version scaled to 48 HECs, and was 2.37× *faster* than the baseline program. This gives us the confidence that with careful optimisations and application specific heuristics for the ULS and the MVar implementation, much of the overheads in the LWC version can be eliminated.

8 Related Work

8.1 Language Runtimes

The idea of continuation based concurrency was initially explored in the context of Lisp (Wand, 1980) and Scheme, a lexically scoped dialect of Lisp that supports first-class continuations. Shivers (Shivers, 1997) proposed exposing hardware concurrency using continuations (Shivers, 1997). The Engines (Dybvig & Hieb, 1989; Haynes & Friedman, 1987) construct incorporated into some Scheme implementations allow preemptive multithreading to be incorporated into languages that support first-class continuations and timer interrupts. The idea is to re-define the lambda binder to maintain a timer (essentially counting the number of function calls made) that is then used to interrupt a thread when a certain number of reductions have occurred. These early works serve as the basis of several contemporary parallel and concurrent programming language implementations (Reppy, 2007; Reppy *et al.*, 2009; Sivaramakrishnan *et al.*, 2014; GHC, 2014). Among these, ConcurrentML (Reppy, 2007) implementations on SML/NJ and MLton, and MultiMLton (Sivaramakrishnan *et al.*, 2014) do not expose the ability to describe alternative ULS's. Fluet et al. (Fluet *et al.*, 2008) propose a scheduling framework for a strict parallel functional language on Manticore (Reppy *et al.*, 2009). However, unlike our system, the schedulers are described in an external language of the compiler's internal representation, and not the source language.

The proposed multicore extension to OCaml (Dolan *et al.*, 2015) incorporate one-shot unchecked algebraic effects and handlers into the language. Algebraic effects and their handlers behave similar to *restartable exceptions*, and can be used to implement interesting control flow operations. One of the key use cases of the proposed extension is to allow

thread schedulers to be written in OCaml, using a one-shot continuation interface. The continuations are managed as heap objects similar to our system. However, unlike our system, the synchronisation between multiple cores is achieved using low-level hardware synchronisation primitives such as compare-and-swap. Unlike GHC, which implements safe FFI support in the runtime, the standard solution for asynchronous IO in OCaml is to use monadic concurrency libraries such as Lwt (Vouillon, 2008) and Async (Async, 2015). Such libraries use an event loop in the backend, and are completely managed in the user-space. With the introduction of continuation-based user-level threading, such libraries need to be updated individually to reconcile their use of monadic concurrency with continuation-based user-level threads. On the other hand, with our system, the asynchronous IO libraries in Haskell need not be rewritten as the behaviour GHC's FFI is preserved with scheduler activations. Finally, OCaml being a strict language, unlike our system, does not have to deal with blackholding issues under user-level scheduling.

Of the meta-circular implementations of Java, Jikes RVM (Frampton *et al.*, 2009) is perhaps the most mature. Jikes does not support user-level threads, and maps each Java thread directly onto a native thread, which are arbitrarily scheduled by the OS. This decision is partly motivated to offer better compatibility with Java Native Interface (JNI), the foreign function interface in Java. Thread-processor mapping is also transparent to the programmer. Jikes supports unsafe low-level operations to block and synchronise threads in order to implement other operations such as garbage collection. Compared to Jikes, our concurrency substrate *only* permits safe interaction with the scheduler through the STM interface. The ULS's also integrates well with GHC's safe foreign-function interface through the activation interface (Section 6.3).

While Manticore (Reppy *et al.*, 2009), and MultiMLton (Sivaramakrishnan *et al.*, 2014) utilise low-level compare-and-swap operation as the core synchronisation primitive, Li et al.'s concurrency substrate (Li *et al.*, 2007) for GHC was the first to utilise transactional memory for multiprocessor synchronisation for in the context of ULS's. Our work borrows the idea of using STM for synchronisation. Unlike Li's substrate, we retain the key components of the concurrency support in the runtime system. Not only does this alleviate the burden of implementing the ULS, but enables us to safely handle the issue of blackholes that requires RTS support, and perform blocking operations under STM. In addition, Li's substrate work uses explicit wake up calls for unblocking sleeping HECs. This design has potential for bugs due to forgotten wake up messages. Our HEC blocking mechanism directly utilises STM blocking capability provided by the runtime system, and by construction eliminates the possibility of forgotten wake up messages.

While we argue that scheduler activations are a good fit for coarse-grained concurrent programs, where application specific tuning is beneficial, it is unclear whether the technique is useful for fine-grained parallelism found in GHC's Evaluation Strategies (Marlow, Simon and Maier, Patrick and Loidl, Hans-Wolfgang and Aswad, Mustafa K. and Trinder, Phil, 2010) and Parallel ML (Fluet *et al.*, 2010). The goal here is to annotate programs with potential opportunities for parallelism, and let the runtime automatically decide to run the fine-grained computations on idle cores. Since the goal with implicit parallelism is to utilise idle cores for parallel speedup, this decision is best left to the runtime system, which is capable of making this decision, without much bookkeeping overhead.

8.2 Operating Systems

Scheduler activations were first conceived by Anderson et al. (Anderson *et al.*, 1991) to allow user-level threads to safely interact with kernel services. With the help of an interrupt mechanism, the kernel notifies the user-level thread of any changes such as blocking events. Scheduler activations have successfully been demonstrated to interface kernel with the user-level process scheduler (Williams, 2002; Baumann *et al.*, 2009). Similar to scheduler activations, Psyche (Marsh *et al.*, 1991) allows user-level threads to install event handlers for scheduler interrupts and implement the scheduling logic in user-space. Unlike these works, our system utilises scheduler activations in the language runtime rather than OS kernel. Moreover, our activations being STM computations allow them to be composed with other language level transactions in Haskell, enabling scheduler-agnostic concurrency library implementations.

The idea of scheduler activations recurs frequently in the case of virtual machines and operating systems implemented in high-level languages. Sting (Philbin, 1993) is an operating system designed to serve as a highly efficient substrate for high-level programming languages such as Scheme, SmallTalk, ML, Modula3, or Haskell. Sting uses a *thread controller* that handles the virtual processor's interaction with other system components such as physical processors and threads. Sting uses continuations to model suspended threads that are blocked on some kernel events, and uses activations to notify blocking and unblocking events. A thread policy manager dictates the thread scheduling policy, and is highly customizable. This design is analogous to the user-level thread scheduler in our system.

SPIN (Bershad *et al.*, 1995) is an operating system that can be customised through dynamic installation of kernel extensions written in the high-level language Modula-3. By suitably restricting Modula-3 code and performing static and dynamic verification, SPIN allows safe kernel extensions. While SPIN allowed user-level scheduler extensions that are executed in kernel space, it is not possible to replace the default scheduler.

House (Hallgren *et al.*, 2005) describes a monadic interface to low-level hardware features that is a suitable basis for building operating systems in Haskell. At its core is the `H` monad, which can be thought of as a specialised version of a Haskell IO monad, suitable for supporting programs that run in the machine's privileged (supervisor) mode. House supports two modes of concurrency – implicit and explicit. While implicit concurrency uses concurrent Haskell as the kernel, resorting to GHC's threading primitives for concurrency, explicit concurrency is supported by maintaining a queue of *context* objects that capture the state of the virtual processor. The contexts are analogous to continuations in our system, and capture the state of the suspended virtual processor. With explicit concurrency, the programmer has the ability to control scheduling but interrupts must be polled for. While monadic concurrency is explicit and suitable for simple programs, one has to resort to monad transformers with larger programs, which can quickly become unwieldy. Unlike the `H` monad, the execution of user-space programs with scheduler activations avoids abstraction leaks; programs execute transparently on top of the kernels and interface through the activations. Moreover, House does not support multiprocessor execution, which can complicate the execution of interrupt handlers with shared resources.

Recently, there has been a lot of interest in Unikernels (Madhavapeddy & Scott, 2014) – specialised, single address space machine images constructed by using library operating systems, typically implemented in high-level languages. HalVM (Galois, 2014) is a Haskell-based unikernel that run directly on top of Xen hypervisor. HalVM uses GHC's native threads for concurrency and thus cannot do custom scheduling and thread priorities. MirageOS (Madhavapeddy *et al.*, 2013) is a unikernel implemented in OCaml, and uses monadic Lwt threads (Vouillon, 2008) for cooperative concurrency. Lwt allows plugging in custom engines for defining specialised scheduling policies. However, neither HalVM nor MirageOS supports multiprocessor execution.

9 Conclusions and Future Work

We have presented a concurrency substrate design for Haskell that lets programmers write schedulers for Haskell threads as ordinary libraries in Haskell. Through an activation interface, this design lets GHC's runtime system to safely interact with the user-level scheduler, and therefore tempering the complexity of implementing full-fledged schedulers. The fact that many of the RTS interactions such as timer interrupts, STM blocking operation, safe foreign function calls, etc,. can be captured through the activation interface reaffirms the idea that we are on the right track with the abstraction.

Our precise formalisation of the RTS interactions served as a very good design tool and a validation mechanism, and helped us gain insights into subtle interactions between the ULS and the RTS. Through the formalisation, we realised that the interaction of black holes and timer interrupts with a scheduler transaction is particularly tricky, and must be handled explicitly by the RTS in order to avoid livelock and deadlock. As for the implementation, we would like to explore the effectiveness of user-level gang scheduling for Data Parallel Haskell (Chakravarty *et al.*, 2007) workloads, and priority scheduling for Haskell based web-servers (Haskell, 2014) and virtual machines (Galois, 2014).

References

Anderson, Thomas E., Bershad, Brian N., Lazowska, Edward D., & Levy, Henry M. (1991). Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. *Pages 95–109 of: Proceedings of the 13th ACM Symposium on Operating Systems Principles*. SOSP '91. Pacific Grove, California, USA: ACM, New York, NY, USA.

Async. (2015). *Jane Street Capitalś asynchronous execution library*. `https://github.com/janestreet/async`.

Baumann, Andrew, Barham, Paul, Dagand, Pierre-Evariste, Harris, Tim, Isaacs, Rebecca, Peter, Simon, Roscoe, Timothy, Schüpbach, Adrian, & Singhania, Akhilesh. (2009). The Multikernel: A New OS Architecture for Scalable Multicore Systems. *Pages 29–44 of: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, New York, NY, USA.

Bershad, Brian N., Chambers, Craig, Eggers, Susan, Maeda, Chris, McNamee, Dylan, Pardyak, Przemyslaw, Savage, Stefan, & Sirer, Emin Gün. (1995). SPIN – an Extensible Microkernel for Application-specific Operating System Services. *Sigops oper. syst. rev.*, **29**(1), 74–77.

Bruggeman, Carl, Waddell, Oscar, & Dybvig, R. Kent. (1996). Representing Control in the Presence of One-shot Continuations. *Pages 99–107 of: Proceedings of the ACM SIGPLAN 1996 Conference*

on Programming Language Design and Implementation. PLDI '96. Philadelphia, Pennsylvania, USA: ACM, New York, NY, USA.

Chakravarty, Manuel M. T., Leshchinskiy, Roman, Peyton Jones, Simon, Keller, Gabriele, & Marlow, Simon. (2007). Data Parallel Haskell: A Status Report. *Pages 10–18 of: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*. DAMP '07. Nice, France: ACM, New York, NY, USA.

Dolan, Stephen, White, Leo, Sivaramakrishnan, KC, Yallop, Jeremy, & Madhavapeddy, Anil. (2015). Effective Concurrency through Algebraic Effects. *The OCaml Users and Developers Workshop*. OCaml '15.

Dybvig, R. K., & Hieb, R. (1989). Engines from Continuations. *Computer languages*, **14**(2), 109–123.

Fluet, Matthew, Rainey, Mike, & Reppy, John. (2008). A Scheduling Framework for General-purpose Parallel Languages. *Pages 241–252 of: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: ACM, New York, NY, USA.

Fluet, Matthew, Rainey, Mike, Reppy, John, & Shaw, Adam. (2010). Implicitly Threaded Parallelism in Manticore. *Journal of functional programming*, **20**(5-6), 537–576.

Frampton, Daniel, Blackburn, Stephen M., Cheng, Perry, Garner, Robin J., Grove, David, Moss, J. Eliot B., & Salishev, Sergey I. (2009). Demystifying Magic: High-level Low-level Programming. *Pages 81–90 of: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '09. Washington, DC, USA: ACM, New York, NY, USA.

Galois. (2014). *Haskell Lightweight Virtual Machine (HaLVM)*. `http://corp.galois.com/halvm`.

GHC. (2014). *Glasgow Haskell Compiler*. `http://www.haskell.org/ghc`.

Hallgren, Thomas, Jones, Mark P., Leslie, Rebekah, & Tolmach, Andrew. (2005). A Principled Approach to Operating System Construction in Haskell. *Pages 116–128 of: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. New York, NY, USA: ACM.

Harris, Tim, Marlow, Simon, Peyton-Jones, Simon, & Herlihy, Maurice. (2005a). Composable Memory Transactions. *Pages 48–60 of: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL, USA: ACM, New York, NY, USA.

Harris, Tim, Marlow, Simon, & Jones, Simon Peyton. (2005b). Haskell on a Shared-memory Multiprocessor. *Pages 49–61 of: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. Haskell '05. Tallinn, Estonia: ACM, New York, NY, USA.

Haskell. (2014). *Haskell Web Development*. `http://www.haskell.org/haskellwiki/Web/Servers`.

Haynes, Christopher T., & Friedman, Daniel P. (1987). Abstracting Timed Preemption with Engines. *Computer languages*, **12**(2), 109–121.

HotSpotVM. (2014). *Java SE HotSpot at a Glance*. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-137187.html`.

IBM. (2014). *Java Platform Standard Edition (Java SE)*. `http://www.ibm.com/developerworks/java/jdk/`.

Li, Peng, Marlow, Simon, Peyton Jones, Simon, & Tolmach, Andrew. (2007). Lightweight Concurrency Primitives for GHC. *Pages 107–118 of: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: ACM, New York, NY, USA.

Lippmeier, Ben, Chakravarty, Manuel, Keller, Gabriele, & Peyton Jones, Simon. (2012). Guiding Parallel Array Fusion with Indexed Types. *Pages 25–36 of: Proceedings of the 2012 Haskell Symposium*. Haskell '12. Copenhagen, Denmark: ACM, New York, NY, USA.

Madhavapeddy, Anil, & Scott, David J. (2014). Unikernels: The Rise of the Virtual Library Operating System. *Communication of the ACM*, **57**(1), 61–69.

Madhavapeddy, Anil, Mortier, Richard, Rotsos, Charalampos, Scott, David, Singh, Balraj, Gazagnaire, Thomas, Smith, Steven, Hand, Steven, & Crowcroft, Jon. (2013). Unikernels: Library Operating Systems for the Cloud. *Pages 461–472 of: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. New York, NY, USA: ACM.

Marlow, Simon, Jones, Simon Peyton, Moran, Andrew, & Reppy, John. (2001). Asynchronous Exceptions in Haskell. *Pages 274–285 of: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: ACM, New York, NY, USA.

Marlow, Simon, Jones, Simon Peyton, & Thaller, Wolfgang. (2004). Extending the Haskell Foreign Function Interface with Concurrency. *Pages 22–32 of: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. Haskell '04. Snowbird, Utah, USA: ACM, New York, NY, USA.

Marlow, Simon and Maier, Patrick and Loidl, Hans-Wolfgang and Aswad, Mustafa K. and Trinder, Phil. (2010). Seq No More: Better Strategies for Parallel Haskell. *Pages 91–102 of: Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. New York, NY, USA: ACM.

Marsh, Brian D., Scott, Michael L., LeBlanc, Thomas J., & Markatos, Evangelos P. (1991). First-class User-level Threads. *Pages 110–121 of: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. SOSP '91. Pacific Grove, California, USA: ACM, New York, NY, USA.

Microsoft Corp. (2014). *Common Language Runtime (CLR)*. `http://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx`.

Philbin, James Francis. (1993). *The design of an operating system for modern programming languages*. Ph.D. thesis, Yale University, New Haven, CT, USA. UMI Order No. GAX93-29376.

Reid, Alastair. (1999). Putting the Spine Back in the Spineless Tagless G-Machine: An Implementation of Resumable Black-Holes. *Pages 186–199 of: Selected Papers from the 10th International Workshop on 10th International Workshop*. IFL '98. London, UK, UK: Springer-Verlag.

Reppy, J.H. (2007). *Concurrent programming in ml*. Cambridge University Press.

Reppy, John, Russo, Claudio V., & Xiao, Yingqi. (2009). Parallel Concurrent ML. *Pages 257–268 of: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: ACM, New York, NY, USA.

Shivers, Olin. (1997). Continuations and Threads: Expressing Machine Concurrency Directly in Advanced Languages. *Continuations Workshop*.

Shootout. (2014). *The Computer Language Benchmarks Game*. `http://benchmarksgame.alioth.debian.org/`.

Sivaramakrishnan, KC, Ziarek, Lukasz, & Jagannathan, Suresh. (2014). MultiMLton: A Multicore-aware Runtime for Standard ML. *Journal of Functional Programming*.

STMLibrary. (2014). *Control.Concurrent.STM*. `http://hackage.haskell.org/package/stm-2.1.1.0/docs/Control-Concu`

Vouillon, Jérôme. (2008). Lwt: A Cooperative Thread Library. *Pages 3–12 of: Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML '08. New York, NY, USA: ACM.

Wand, Mitchell. (1980). Continuation-based Multiprocessing. *Pages 19–28 of: Proceedings of the 1980 ACM Conference on LISP and Functional Programming*. LFP '80. Stanford University, California, USA: ACM, New York, NY, USA.

Williams, Nathan J. (2002). An Implementation of Scheduler Activations on the NetBSD Operating System. *Pages 99–108 of: Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association.