Abstract
The runtime for a modern, concurrent, garbage collected language like Haskell is like an operating system: sophisticated, complex, performant, but alas very hard to change. If more of the runtime system were in Haskell it would become far more modular and malleable. In this paper we describe a new concurrency design that allows the scheduler for concurrent and parallel programs to be written in Haskell. In particular, this substrate allows new primitives to be constructed modularly, obviating the need to re-engineer or reason about the interactions with GHC’s existing concurrency support.

1. Introduction
GHC has a sophisticated, highly tuned runtime system (RTS) and rich support for concurrency (forkIO, MVars, STM, asynchronous exceptions, bound threads, safe foreign function interface, transparent scaling on multicore, etc.). But the performance benefit comes at the cost of lack of flexibility; the scheduler is baked into the runtime system and is implemented in a mixture of low-level C and C++ code. Not only is the RTS code base large, but also involves non-trivial interaction between the thread schedulers and concurrency libraries through a cascade of locks and condition variables. This hinders the research and development of new concurrency primitives, by forcing one to reason, not just about the new primitive, but also its non-trivial interaction with existing features in the RTS. As a result, the language practitioners end up relying on the language implementors for any improvements or modifications to the thread scheduler.

Why should the language practitioners be interested in the thread scheduling strategy? There are several good reasons to believe that a particular concurrent programming model, or a scheduling policy would not suit every application. With the emergence of many-core processors, we see NUMA effects becoming more prominent, and applications may benefit from NUMA aware scheduling and load balancing policies. Moreover, an application might have a better knowledge of the scheduling requirements – a thread involved in user-interaction is expected to be given more priority over threads performing background processing. We might want to experiment with various work-stealing or work-sharing policies. With the emergence of new programming models such as data parallelism [4, 11], it behoves the RTS to provide greater control of the scheduler to the programmer.

Our goal with this work is to allow the language practitioners to safely gain control over the thread scheduler without compromising the rich concurrency support and performance. In this paper, we present a new concurrency substrate that allows Haskell programmers using GHC to build schedulers as libraries written in Haskell, while allowing seamless interaction with RTS concurrency mechanisms. Building user-level schedulers is hardly a novel endeavor, and indeed, our design is inspired by Peng Li et al. [10]’s earlier attempt at building a lightweight concurrency substrate, but we make several new contributions:

• Our concurrency substrate design relies on abstracting the interface to the user-level scheduler through scheduler activations [2] (Section 4.3). While scheduler activations have previously been utilized to interface the OS kernel with the user-level process scheduler, our system is the first to utilize scheduler activations to interface the language runtime system with a scheduler implemented in the source language.

• We utilize the same activation abstraction to allow scheduler-agnostic implementations of concurrency libraries such as MVars (Section 5). The activation abstraction not only admits modular implementation of concurrency libraries, but also enables threads belonging to different user-level schedulers to seamlessly interact through a shared concurrency abstraction.

• We retain the implementation of key, performance-critical functionalities such as safe foreign calls (Section 5.5), transactional memory blocking operations (Section 5.2), asynchronous exceptions (Section 5.8), blackhole handling (Section 5.6) in the RTS. The RTS makes upcalls to scheduler activations whenever it needs to interact with the user-level scheduler. As we will illustrate, this significantly reduces the burden of building a customized scheduler.

• Concurrency primitives and their interaction with RTS are particularly tricky to specify and reason about. An unusual feature of this paper is that we precisely formalize not only the concurrency substrate primitives, but also their interaction with the RTS concurrency primitives.

Everything we describe is implemented in (a fork of) GHC; we present the implementation details and preliminary results in Section 7.

2. The challenge
GHC’s current scheduler is written in C, and is hard-wired into the runtime system. The goal of this paper is a simple one: to allow

---

1 This work was done at Microsoft Research, Cambridge.
the scheduler for a parallel Haskell program, running on a multi-core computer, to be written in Haskell itself, and embodied in the program as an ordinary user library. There are several motivations for this goal:

- Modifying the current scheduler is a major exercise, involving detailed knowledge of the runtime system and its intimate interaction with the running Haskell code. Moreover, it requires a complete new version of the compiler and its runtime system, so deployment is difficult.

- Schedulers are themselves concurrent programs, and they are particularly devious ones. Using the facilities available in C, they are extremely hard to get right. Writing a scheduler in Haskell using (as we shall see) transactional memory, is much easier.

- Schedulers, especially for parallel architectures, have a rich design space, and no single scheduler is good for all applications. For example, GHC’s built-in scheduler has no notion of thread priority; nor does it support “gang scheduling”, needed for data-parallel computation. It has a single, fixed notion of work-stealing to move work from one processor to another. The ability for Haskell users to write their own schedulers, perhaps for particular classes of applications, is potentially very useful.

3. Design overview

In this section we give an overview of our design, which is depicted in Figure 1. We divide the world into:

- A fixed runtime system or RTS, written in a mixture of C and C++. We sometimes call this the “substrate” because it is the foundation on which the rest is built.

- A concurrency library (or libraries) that implement a scheduler (or schedulers), written in ordinary Haskell.

- The client Haskell program.

Our main design focus is on exactly what the substrate does, and the interface between it and the concurrency libraries it serves.

3.1 HECs, tasks, and SConts

Our concurrency substrate executes a concurrent Haskell program on a multi-core processor with a shared heap. The Haskell programmer can use very lightweight Haskell threads. These threads are executed by a fixed number of Haskell Execution Contexts, or HECs. A HEC is a virtual processor, and we typically create one HEC for each physical processor; this number is initially determined by the RTS argument -N provided by the programmer, though it can be changed at runtime.

A HEC is in turn animated by an operating system thread, but we use the term task for these OS threads, to distinguish them from Haskell threads. In fact, each HEC is animated by one of a pool of tasks; the current task may become blocked in a foreign call (eg a blocking I/O operation), in which case another task takes over the HEC. However, only at most one task can execute a HEC at once.

The choice of which (Haskell) thread is executed by which HEC is made by a scheduler. Our goal is to allow the scheduler to be written in Haskell, giving programmers the freedom to experiment with different scheduling or work-stealing algorithms. So the substrate does not directly support the notion of a “thread” at all; instead, it offers one-shot continuations, of type SCont.

A thread is an active thing, whereas an SCont is a passive value that must be scheduled explicitly before it can do anything. Nevertheless, in this paper we sometimes sloppily use the term “thread” where the “SCont” would be more accurate.

3.2 Blocking and scheduler activations

GHC’s existing runtime system already supports many interactions among parallel threads. Many involve blocking. For example,

- Two threads may evaluate the same thunk at the same time; the second should block until the first has completed evaluation.

- A thread might read an empty MVar; then it should block until the MVar is filled.

- An STM transaction might retry, in which case the thread should block until one of the TVars read by the transaction changes.

The details are not important, but the key point is this: re-implementing these mechanisms is not part of our goal. They are intricate, highly-optimised, and (unlike scheduling) there is no call for rapid design exploration. In earlier work [10], we attempted to move all these mechanisms into Haskell, but we stalled. Doing so was tricky, carried a heavy performance penalty, and gained little in flexibility. So a key aspect of our design is that we continue to use all these existing RTS mechanisms unchanged.

We describe the details in the next section, but meanwhile the following vocabulary is useful. An SCont s may be in one of three situations:

- Running on a HEC.

- Blocked in the RTS. The RTS “owns” s exclusively. When s becomes runnable, the RTS will enable it, by transferring ownership to its scheduler.

- Ready in a scheduler. The SCont s is ready to run, and the RTS has transferred ownership to its scheduler. This does not mean that s is running, merely that it can be run when the scheduler chooses to do so. The RTS has no further interest in s.

The details of this ownership transfer, and how a thread moves from enabled to running, are discussed in Section 4.2.

4. Concurrency Substrate

It is hard to give English-language descriptions of concurrency primitives that are truly precise. We make our descriptions precise
x, y ∈ Variable \ r, s ∈ Name

\textbf{Md} ::= return M | M >> N
\textbf{Ex} ::= throw M | catch M N | catchSTM M N
\textbf{Stm} ::= newTVar M | readTVar r | writeTVar r M |
atomically M | retry
\textbf{Sc} ::= newSCont M | switch M | runOnIdleHEC s
\textbf{Sls} ::= getAux s | setAux s M
\textbf{Act} ::= blockAct s | unblockAct s |
setBlockAct M | setUnblockAct M

\textbf{Term} ::= r | x | λ.x → M | M N | . . .
\textbf{Program state} P ::= S; Θ
\textbf{HEC} H ::= (s,M,D) | (s,M,D)_{Sleeping} |
(s,M,D)_{Deadlock} | idle
\textbf{Heap} Θ ::= r → M ⊕ s → (M,D)
\textbf{SLS Store} D ::= ⟨ M, N, r ⟩
\textbf{IO Context} E ::= * | E >> M | catch E M
\textbf{STM Context} F ::= * | F >> M

Figure 2: Syntax of terms, states, contexts, and heaps

\textbf{Top-level transitions} \ S; Θ \xrightarrow{a} S'; Θ'

\textbf{OneHEC} \ S \parallel H; Θ \xrightarrow{a} S \parallel H'; Θ'

\textbf{HEC transitions} \ H; Θ → H'; Θ'

\textbf{PURESTEP} \ ⟨ s, E[M], D ; Θ ⟩ → ⟨ s, E[N], D ; Θ' ⟩

\textbf{Purely functional transitions} \ M → N

\textbf{Return} N >> M → M N \ (BIND)
\textbf{Throw} N >> M → throw N \ (THROW)
\textbf{Retry} >> M → retry \ (RETRY)
\textbf{Catch} (throw M) N → return M \ (IOCATCH)
\textbf{Catch} (throw M) N → N M \ (IOCATCHEXP)

Plus the usual rules for call-by-need λ-calculus, in small-step fashion

Figure 3: Operational semantics for basic transitions

by giving an operational semantics of the substrate and its primitives.

Figure 2 shows the syntax of program states. The program state P is a soup S of Haskell execution contexts, HECs, and a shared heap Θ. Each HEC is either idle (1Idle) or a triple (s,M,D), where

- M is a unique identifier of the currently executing SCont, M is the currently executing term, D represents stack-local state. We defer the details of stack-local states till Section 4.5. Each HEC has an optional subscript t representing its current state, and the absence of the subscript represents a HEC that is running. The heap is a disjoint finite map of:
  - (r → M), maps the identifier r of a transactional variable, or
  - TVar, to its value.
  - (s → (M,D)), maps the identifier s of an SCont to its current state.

In a program state (S; Θ), a SCont with identifier s appears either as the running SCont in a HEC (s,M,D)_t, ∈ S, or as a binding (s → (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 τ for some type τ, modelling the fact that concurrent Haskell threads can perform I/O.

The number of HECs remains constant; remember that each models a processor, and we cannot make new processors! 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.

The program makes a transition from one state to another through the top-level program small-step transition relation:

\[ S; Θ \xrightarrow{a} S'; Θ' \]

This says that the program makes a transition from S; Θ to S'; Θ’, possibly interacting with the underlying RTS through action a. We return to these RTS interactions in Section 5 and we omit a altogether if there is no interaction.

Some basic transitions are presented in Figure 3. 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 right-hand end of the soup, so that OneHEC can fire. Similarly, Rule PURESTEP enables one of the HECs to perform a purely functional transition under the evaluation context E (defined in Figure 2). 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 → 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 3). These primitives represent blocking actions under transactional memory and will be dealt with in Section 4.4.

4.1 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 [10], 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 [3] — and schedulers are prime candidates, because they are
refactoring. Θ
where STM based concurrency libraries such as 
are presented in Figure 4. Unlike Li’s design, however, 
particularly prone to subtle concurrency bugs. The STM primitives 
are undone except for the newly allocated TVar due to reasons 
explained in [8]. Rules TCRETRY and TCSLEEP simply propagate the 
request to retry the transaction or putting the HEC to sleep through the context. The act of blocking, wake up and undoing the 
effects of the transaction are handled in Section 5.2.

4.2 One-shot continuations
HECs provide a fixed number (roughly, one per processor) of execution engines. To implement the programmer’s model of very lightweight Haskell threads we use a one-shot continuation to represent the state of a Haskell thread [19]. The core interface for creating and scheduling one-shot continuations is presented below:

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

An SCont (stack-continuation) is an IO computation that has been suspended mid-execution. The call (newSCont M) creates a new SCont that, when scheduled, executes M. In the RTS, SConts are represented quite conventionally by a heap-allocated Thread Storage Object (TSO), which includes the computation’s stack and local state, saved registers, and program counter. Unreachable SConts are garbage collected.

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.

Since our continuations are one-shot, capturing a continuation simply fetches the reference to the underlying TSO object. Hence, continuation capture involves no copying, and is cheap. Moreover, by using an STM operation as the body of the switch primitive we avoid, by construction, an important class of concurrency bugs associated with utilising one-shot continuations to implement schedulers in a multicores context [10].

Using the SCont interface, a cooperative scheduler can be built as follows:

switch $ \text{\$ \text{\$ s -> do}}$
...save current SCont “s” somewhere...
$s’ = \ldots\text{\$ fetch a new SCont from somewhere...}$
return s’

We address the question of where “somewhere” is in Section 4.5.

4.3 SCont semantics
The semantics of SCont primitives are presented in Figure 6. 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. For the current discussion, it is safe to ignore D; we return to it in Section 4.5.

Rule NEWSCont binds the given IO computation and a new stack-local state pair to a new SCont s’, and returns s’.

The rules for switch (SWITCHSELF, SWITCH, and SWITCH-EXP) 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 (SWITCHEXP).

The alert reader will notice that the rules for switch duplicate much of the paraphernalia of an atomic transaction (Figure 5), but that is unavoidable because the switch to a new continuation must form part of the same transaction as the argument computation.

4.4 Parallel SCont execution

When the program begins execution, the HEC soup consists of \( N \) HECs where \( N \) is the number of HECs provided by the OS, and has the following configuration:

Initial HEC Soup \( S = (s, M, D) \parallel \text{Idle}_1 \parallel \ldots \parallel \text{Idle}_{N-1} \)

Of these, one of the HEC runs the main IO computation \( M \) with initial SCont identifier \( s \). All other HECs are in idle state. The substrate exposes the primitive to start evaluation of an SCont on an idle HEC.

\[
\text{runOnIdleHEC} :: \text{SCont} \to \text{IO} ()
\]

Rule RUNONIDLEHEC illustrates the behaviour of parallelly instantiating evaluation of the given SCont \( s' \) on an idle HEC. In the implementation, if there are no idle HECs, an exception is raised to inform the programmer that no free HECs are available to run this SCont on.

Once the SCont running on a HEC finishes evaluation, it is removed from the HEC soup and replaced by an idle HEC. Due to the type of new SCont, any completed SCont either produces a unit value (rule DoneUnit), or an exceptional value (rule DoneExp). Exceptional values signify an error condition, and is printed to the standard output. The implementation marks the HEC on which an SCont runs to completion as being idle.

4.5 Scheduler activations

Suppose that a Haskell thread is running, and a clock tick happens. The RTS must capture the thread’s continuation and hand it off to the scheduler. But 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 SCont 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 activation \(^2\) called block and unblock. Each SCont has its own block and unblock activations; it is like an object with two methods, accessible via this interface:

\[
\begin{align*}
\text{blockAct} & :: \text{SCont} \to \text{STM} () \\
\text{unblockAct} & :: \text{SCont} \to \text{STM} \text{ SCont}
\end{align*}
\]

- The call (unblockAct \( s \)) invokes \( s \)'s unblock scheduler activation, passing \( s \) to it like a “self” parameter. It is invoked when some agent (often the RTS) wants to hand an SCont over to its scheduler. The scheduler typically maintains some mutable state, where it stashes \( s \).
- The call (blockAct \( s \)) invokes \( s \)'s block scheduler activation, again passing \( s \) to it. It is invoked by some agent (often, but not always, the RTS) when it wants to ask a scheduler for the SCont that it would like to run next. The scheduler consults its mutable state, mutates it (e.g. to remove an SCont from the ready queue) and returns the SCont. The argument to block is expected to be the SCont that has been running to this point.

We give a more precise semantics for blockAct and unblockAct in Section 4.7. Now we can fill out the “…” in yield from Section 4.2.

\[
\begin{align*}
\text{yield} & :: \text{IO} () \\
\text{yield} = \text{switch} (\langle s \rightarrow \text{unblockAct} s \gg \text{blockAct} s \rangle)
\end{align*}
\]

The unblockAct hands the current SCont to its scheduler, which will presumably put it in a queue of runnable threads. The blockAct asks \( s \)'s scheduler what SCont it would like to run next, after which switch will load the returned SCont into the current HEC. This hand-off must always be done with care. For example, it would be bad if the unblockAct published \( s \) in a run-queue, and another hungry HEC picked it up and started running it — because \( s \) is still running in this HEC. That is why the switch body is transactional: the effects on the scheduler’s run-queue become visible only after the hand-off to the new SCont is complete.

\(^2\) The term “activation” comes from the operating systems literature [2].
4.6 User-level blocking
Scheduler activations already allow us to write concurrency abstractions that would have been tricky before. For example, suppose we wanted to implement a quantity semaphore, with this API:

```haskell
data QSem
getQ :: Int -> IO ()
putQ :: Int -> IO ()
```

The QSem holds an integer-valued resource. putQ adds to the resource, and getQ takes from it. If there is not enough resource to satisfy getQ the thread should block. When putQ finds blocked threads it makes a policy decision about which blocked thread(s) to reawaken, based on their requests. We might implement QSem like this:

```
data QSem = QS (TVar (Int, [(Int, SCont)]))
getQ req (Q v) = switch $ \s ->
do { (n,bq) <- readTVar v }
    ; if n >= req
        then do { writeTVar v (n-req, bq)
            ; return s }
    else do { writeTVar v (n, (req,s):bq)
            ; blockAct s }
```

The QSem is implemented as a TVar containing a pair (n, bq), indicating n units of resource and a queue containing blocked threads, each indicating their request (presumably all bigger than n). The function getQ then captures the continuation (which, recall, is very cheap), and reads the TVar. If there is enough resource, things are easy, otherwise getQ adds the thread to the blocked queue, and invokes the block scheduler activation to find another SCont to run.

Notice that the blocked SCont is still owned by the scheduler, not the RTS, even though it is not runnable.

4.7 Semantics of local state
The scheduler activations of an SCont can be read by blockAct and unblockAct primitives. In effect, they constitute SCont-local state. Local state is often convenient for other purposes, so we also provide a single dynamically-typed field, the “aux-field”, for arbitrary user purposes. The API is presented below.

```
type BlockAct = SCont -> STM SCont
type UnblockAct = SCont -> STM ()
blockAct :: BlockAct
unblockAct :: UnblockAct
setBlockAct :: BlockAct -> IO ()
setUnblockAct :: UnblockAct -> IO ()
getAux :: SCont -> STM Dynamic
setAux :: SCont -> Dynamic -> STM ()
```

The API additionally allows an SCont to change its own scheduler through setBlockAct and setUnblockAct primitives.

```
newBoundSCont :: IO () -> IO SCont
```

Bound SConts have the property that they are always executed by its corresponding bound task (OS thread) to which they are bound to. Moreover, no other SCont can run on this bound task. The semantics of newBoundSCont primitive is the same as newSCont primitive, except for the creation of a new suspended task in RTS. Whenever control switches to a bound SCont, RTS ensures that this SCont gets to run on its bound task. Similarly, when switching away from a bound SCont, its bound task is suspended and an appropriate task resumes the target SCont.

5. Interaction with the RTS
The key aspect of our design is composability of user-level schedulers with the existing RTS concurrency mechanisms (Section 3.2). In this section, we will describe in detail the interaction of RTS concurrency mechanisms and their interaction with the user-level schedulers.

5.1 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
interrupts is shown in Figure 8

 scheduler needs to be scheduled. The semantics of handling timer

 The Tick label on the transition arrow indicates an interaction

 The big question is this: how should the RTS block the thread?

 GHC today supports blocking operations in STM. When

 The big question is this: how should the RTS block the thread?

 Rule UpBlock in Figure 10 instantiates an ephemeral thread that fetches

 The transition in TryEntry is labelled with the RTS interaction

 3 Technically we should ensure that every HEC receives a tick, and of
 course our implementation does just that, but we elide that here.
s' fresh r fresh \( D' = (\text{blk}(D), \text{unblk}(D), r) \)

\[ M' = \text{switch}(x. \text{blk}(D) s) \]

\[ \Theta' = \Theta[s \rightarrow (M, D)][r \rightarrow \text{toDyn}()] \]

\[ \langle s, M, D \rangle; \Theta \stackrel{\text{blk}}{\rightarrow} \langle s', M', D' \rangle; \Theta' \]

(UpBlock)

Unblock upcall instantiation \( H; \Theta \stackrel{\text{unblk} s}{\rightarrow} H'; \Theta' \)

(UpUnblock)

\[ s' \stackrel{\text{fresh}}{\rightarrow} r \stackrel{\text{fresh}}{\rightarrow} \]

\[ \langle s', M', D' \rangle; \Theta[s \rightarrow (M, D) \rightarrow \text{toDyn}()] \]

(UpUnblockRunning)

\[ M'' = \text{atomically} \ (\text{unblk}(D) s) \Rightarrow M' \]

\[ \langle s', M', D' \rangle; \Theta[s \rightarrow (M, D) \rightarrow \text{toDyn}()] \]

\[ \Theta' \rightarrow \Theta[s \rightarrow (M, D)] \]

Figure 11: Instantiating upcalls

Figure 5.4: Safe foreign call transitions

5.4 Timer interrupts and transactions

What if a timer interrupt occurs during a transaction? The (Tick) rule of Section 5.3 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 threads; they are confined to its SCont-local transaction log. When the thread 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.

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.

5.5 Safe foreign function calls

Foreign calls in GHC are highly efficient but intricately interact with the scheduler [13]. Much of it owes to the the RTS’s task model which 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 (Requirement 1). 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 (Requirement 2). In this section, we will discuss the interaction of safe-foreign function calls and the user-level scheduler. In particular, we restrict the discussion to outcalls — calls made from Haskell to C.
starts to evaluate
with the compiler) arranges for B to
B. To avoid duplicate evaluation the RTS (in intimate cooperation
evaluate a thunk
5.6 Black holes

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 user-level scheduler interaction. The
semantics of foreign call handling is presented in Figure 11. Rule
OCB orn nr illustrates that the HEC performing the foreign call
moves into the Outcall state, where it is no longer runnable. In the
first path (rule OCRT FAST), 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 pay the cost of task
switching and resume the scheduler (rule OCST EA L). The sche-
duler is resumed using the block upcall. Once the foreign call eventu-
ally returns, the S Cont s blocked on the foreign call can be re-
sumed. Since we have already resumed the scheduler, the correct
behaviour is to prepare the S Cont s with the result and add it to
its user-level scheduler. Rule OCRT S LOW achieves this through
unblock upcall.

5.6 Black holes

In a concurrent Haskell program, a thread A may attempt to eval-
uate 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 black hole 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
[7].

Clearly this is another place where the RTS may initiate block-
ing. We can describe the common case with rules similar to those
of Figure 9 with rules shown in Figure 12. The RTS initiates the
process with a BlockBH s action, taking ownership of the S Cont s.
Later, when the evaluation of the thunk is complete, the RTS initiates
an action ResumeBH s, which returns ownership to s’s scheduler.

Just as in Section 5.4, we need to take particular care with
switch transactions. Suppose a switch transaction encounters a
black-holed thunk under evaluation by some other thread 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 blk ) is a switch transaction; and that is
very likely to encounter the same thunk. Moreover, it is just
possible that the thunk is under evaluation by an S Cont in this
very scheduler’s run-queue, so the black hole is preventing us from
scheduling the very S Cont that is evaluating it. Deadlock beckons!

In the case of timer interrupts we solved the problem by switch-
ning them off in switch transactions, and it turns out that we can
effectively do the same for thunks. Since we cannot sensibly sus-
pend 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
S Cont 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 [12].

5.7 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 S Cont s needs to block on or unblock from
an MVar, the RTS invokes the blk or unblk upcall, respectively.
This significantly reduces the burden of migrating to a user-level
scheduler implementation.

5.8 Asynchronous exceptions

GHC’s supports asynchronous exceptions in which one thread can
send an asynchronous interrupt to another [12]. This is a very
tricky area; for example, if a thread is blocked on a user-level
QSem (Section 4.6), 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, but we are not yet happy with the details
of the design, and in any case space precludes presenting them here.

5.9 On the correctness of user-level schedulers

While the concurrency substrate exposes the ability to build user-
level schedulers, the onus is on the scheduler implementation to
ensure that it is sensible. The invariants such as not switching
to a running S Cont, or an S Cont 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 block and unblock activations do not
raise an exception that escape the activation. Activations raising
exceptions indicate that the user-level scheduler implementation is
faulty, and the substrate raises an error.

An S Cont suspended on a user-level scheduler may become
unreachable if the scheduler data structure holding it becomes
unreachable. While an S Cont indefinitely blocked on an RTS MVar
operation is raised with an exception and added to its user-level
scheduler, the situation is worse if the user-level scheduler itself
becomes unreachable. There is no scheduler to run this S Cont! In
this case, immediately after garbage collection, our implementation
logs an error message to the standard error stream along with the
unreachable S Cont identifier.

6. Developing concurrency libraries

We have so far been dealing with the concurrency substrate prim-
itives more or less in isolation. In this section, we will bring it all
together, by building a multicore capable, round-robin scheduler
and a user-level MVar implementation.

6.1 User-level scheduler

The first step in designing a scheduler is to describe the scheduler
data structure. We utilise an array of runqueues, with one queue per
HEC. Each runqueue is represented by a TVar list of S Conts.

newtype Sched = Sched (Array Int (TVar [S Cont]))
The next step is to provide an implementation for the scheduler activations.

```hs
blockActivation :: Sched -> ST SCont
blockActivation (Sched pa) = do
  cc <- getCurrentHEC
  l <- readTVar $ pa!cc
  case l of
    [] -> retry
    x:tl -> do
      writeTVar (pa!cc) tl
      return x

unblockActivation :: Sched -> ST ()
unblockActivation (Sched sc) = do
  l <- readTVar $ sc!cc
  writeTVar (sc!cc) $ l++[sc]
```

blockActivation 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 block activation is always evaluated by a `switch` primitive, and performing `retry` within a `switch` puts the capability 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.

unblockActivation adds the given SCont to the back of the runqueue. Unblock activation finds the SCont’s HE by querying its stack-local state, the details of which are explained along with the next primitive.

Next question to answer is how to initialise the user-level 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 user-level scheduler performs these two steps at the start of the main IO computation. The implementation of these primitives are given below:

```hs
newScheduler :: IO ()
newScheduler = do
  -- Initialise Auxiliary state
  switch $ \s -> do
    counter <- newTVar (0::Int)
    setAux s $ toDyn (0::Int, counter)
    return s
  -- Allocate scheduler
  nc <- getNumHECs
  let sched = ...
  -- Initialise activations
  setBlockAct s $ blockActivation sched
  setUnblockAct s $ unblockActivation sched

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

First we will focus on initialising a new user-level scheduler (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, 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 user-level scheduler 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 user-level scheduler implementation.

```hs
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` primitive that spawns a new user-level thread as follows:

```hs
forkIO :: IO () -> IO SCont
forkIO task = do
  numHECs <- getNumHECs
  -- epilogue: Switch to next thread
  let e = switch blockAct
  newSC <- newSCont (task >> e)
  -- Create and initialise new Aux state
  switch $ \s -> do
    dyn <- getAux s
    nextHEC <- readTVar t
    writeTVar t $(nextHEC + 1) 'mod' numHECs
    setAux newSC $ toDyn (nextHEC, t)
    return s
  -- Add new thread to scheduler
  atomically $ unblockAct newSC
  return newSC
```

`forkIO` primitive 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 unblock activation.

The key aspect of this `forkIO` primitive is that it does not directly access the scheduler data structure, but does so through only 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` primitive. Additionally, we can implement a `yield` primitive similar to the one described in Section 4.5. Due to scheduler activations, the interaction with the RTS concurrency mechanisms come for free, and we are done!

### 6.2 Scheduler agnostic user-level MVars

Our scheduler activations abstracts the interface to the user-level schedulers. 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:
### Implementation and Results

Our implementation 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. The lightweight concurrency (LWC) substrate co-exists with the vanilla GHC, and the programmer can freely mix programs using Control.Concurrent threads and SConts. Although such a program would be safe, the scheduling guarantees are hard to enforce; the concurrency substrate does have control over the RTS scheduling decisions. Hence, mixing Control.Concurrent threads and SConts is generally not recommended.

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 \(^1\): k-nucleotide, mandelbrot, and chameneos. We also implemented a concurrent prime number generator using sieve of Eratosthenes (primes-sieve), where the threads communicate over MVars. For our experiments, we generated the first 10000 primes. The benchmarks offer varying degrees of parallelisation opportunity. The vanilla chameneos program was fastest on 1 HEC, and was 1.24X slower than the baseline. In primes-sieve benchmark, while the vanilla version was 29X faster than the baseline, LWC version was 38X faster. In the vanilla GHC, the RTS thread scheduler by default spawns a thread on the current HEC and only shares the thread with other HECs if they are idle. The LWC scheduler (described in Section 6) 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 benchmark, the LWC version was 3.9X slower than the baseline on 1 HEC and 2.6X slower on 2 HECs, and slows down with additional HECs as chameneos does not present much parallelisation opportunity. The vanilla chameneos program was fastest on 1 HEC, and was 1.24X slower than the baseline. In primes-sieve benchmark, while the vanilla version was 6.8X slower on one HEC, the vanilla version was 1.3X slower, when compared to the baseline.

In chameneos 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-6X more allocations than the vanilla version. Most of these allocations are due to the data structure used in the user-level scheduler and 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 2.37X faster than the baseline program. This gives us the confidence that with careful optimisations and application specific heuristics for user-level scheduler and MVar implementation, much of the overheads in the LWC version can be eliminated.

### Related Work

Continuation based concurrency libraries have been well studied \(^{18, 19}\) and serve as the basis of several parallel and concurrent programming language implementations \(^{16, 17, 22}\). While these systems utilise low-level compare-and-swap operation as the core synchronisation primitive, Li et al.’s concurrency substrate \(^{10}\) for GHC was the first to utilise transactional memory for multiprocessor synchronisation for in the context of user-level schedulers. Our work borrows the idea of using STM for synchronisation. LightHouse \(^6\) proposes an extension of Haskell-based House operating system which integrates Li’s concurrency substrate framework.

Unlike Li’s substrate, we retain the key components of the concurrency support in the runtime system. Not only does alleviate the burden of implementing user-level scheduler, but enables us to safely handle blackholes and asynchronous exceptions, and perform blocking operations under STM. Li’s substrate work uses explicit wake up calls for unlocking sleeping HECs. This design has potential for bugs due to forgotten wake up messages. Our HEC blocking mechanism directly utilises STM blocking capability pro-

#### Table: MVar Implementation

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

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

```haskell
takeMVar :: MVar a -> IO a
h <- atomically $ newPVar undefined
switch $ \x \rightarrow do
st <- readTVar ref
| Empty ts -> do
writeTVar h x
case deque ts of
  (_, Nothing) \rightarrow do
writeTVar ref $ Empty emptyQueue
(ts', Just (x', s)) \rightarrow do
writeTVar ref $ Full x' ts'
unblockAct 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 scheduler activations to block and schedule the SConts interacting through the MVar. This allows threads from different user-level schedulers to communicate over the same MVar, and hence the implementation is scheduler agnostic.

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

---

The results are presented in Figure 13. For each benchmark, the baseline is the non-threaded (not compiled with -threaded) vanilla GHC version. For the vanilla and LWC versions (both compiled with -threaded), we report the running times on 1 HEC as well as the fastest running time observed with additional HECs. Additionally, we report the HEC count corresponding to the fastest configuration. All the times are reported in seconds.

In k-nucleotide benchmark, the performance of LWC version was indistinguishable from the vanilla version. Both threaded versions of the benchmark program were fastest on 8 HECs. In mandelbrot benchmark, LWC version was faster than the vanilla version. While the vanilla version was 29X faster than the baseline, LWC version was 38X faster. In the vanilla GHC, the RTS thread scheduler by default spawns a thread on the current HEC and only shares the thread with other HECs if they are idle. The LWC scheduler (described in Section 6) 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 benchmark, the LWC version was 3.9X slower than the baseline on 1 HEC and 2.6X slower on 2 HECs, and slows down with additional HECs as chameneos does not present much parallelisation opportunity. The vanilla chameneos program was fastest on 1 HEC, and was 1.24X slower than the baseline. In primes-sieve benchmark, while the vanilla version was 6.8X slower on one HEC, the vanilla version was 1.3X slower, when compared to the baseline.

In chameneos 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-6X more allocations than the vanilla version. Most of these allocations are due to the data structure used in the user-level scheduler and 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 2.37X faster than the baseline program. This gives us the confidence that with careful optimisations and application specific heuristics for user-level scheduler and MVar implementation, much of the overheads in the LWC version can be eliminated.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline (Secs)</th>
<th>Vanilla (Secs)</th>
<th>LWC (Secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 HEC</td>
<td>Fastest (# HECs)</td>
</tr>
<tr>
<td>k-nucleotide</td>
<td>10.60</td>
<td>10.62</td>
<td>4.82 (8)</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>85.30</td>
<td>90.83</td>
<td>3.21 (48)</td>
</tr>
<tr>
<td>chamomone</td>
<td>4.62</td>
<td>5.71</td>
<td>5.71 (1)</td>
</tr>
<tr>
<td>primes-sieve</td>
<td>32.52</td>
<td>36.33</td>
<td>36.33 (1)</td>
</tr>
</tbody>
</table>

Figure 13: Benchmark results.

9. Conclusions and Future Work

We have presented a concurrency substrate design that allows key runtime concurrency mechanisms of GHC to safely interact with Haskell scheduler implementation. This is achieved by abstracting the interface to a user-level scheduler through scheduler activations. The runtime system invokes the activations whenever it wishes to block the currently running thread and eventually unblock the thread. The fact that many of the RTS interactions such as timer interrupts, STM blocking operation, safe foreign function calls, etc., only required activations to interface with the user-level schedulers reaffirms the idea that we are on the right track with the activation 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 user-level scheduler and the RTS. 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. Such scheduler interactions are better not handled purely in Haskell code.

As the next step, we plan to improve upon our current, less-desirable solution for handling asynchronous exceptions. A part of this solution involves making the SCont reference a bona fide one-shot continuation, and not simply a reference to the underlying TSO object. Switching to such an SCont more than once raises an exception under STM, and must be handled by the scheduler implementation. As a result, concurrency substrate would be able to better handle erroneous scheduler behaviours rather than raising an error and terminating.

As for the implementation, we would like to explore the effectiveness user-level "gang-scheduling" for Data Parallel Haskell workloads, and priority scheduling for Haskell based web-servers and virtual machines.

References

