Lightweight Concurrency Primitives for GHC

Peng Li
University of Pennsylvania
lipeng@cis.upenn.edu

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Andrew Tolmach
Portland State University
apt@cs.pdx.edu

Abstract

The Glasgow Haskell Compiler (GHC) has quite sophisticated support for concurrency in its runtime system, which is written in low-level C code. As GHC evolves, the runtime system becomes increasingly complex, error-prone, difficult to maintain and difficult to add new concurrency features.

This paper presents an alternative approach to implement concurrency in GHC. Rather than hard-wiring all kinds of concurrency features, the runtime system is a thin substrate providing only a small set of concurrency primitives, and the remaining concurrency features are implemented in software libraries written in Haskell. This design improves the safety of concurrency support; it also provides more customizability of concurrency features, which can be developed as Haskell library packages and deployed modularly.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.10 [*Software Engineering*]: Design—Methodologies; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.4.1 [*Operating Systems*]: Process Management—Concurrency,Scheduling,Synchronization,Threads

General Terms Design, Experimentation, Languages, Performance, Measurement.

Keywords Haskell, Concurrency, Thread, Transactional Memory.

1. Introduction

In any programming language supporting concurrency, a great deal of complexity is hidden inside the implementation of the concurrency abstractions. Much of this support takes the form of a *runtime system* that supports threads, primitives for thread communication (e.g. locks, condition variables, transactional memory), a scheduler, and much else besides. This runtime system is usually written in C; it is large, hard to debug, and cannot be altered except by the language implementors.

That might not be so bad if the task were cut-and-dried. But it isn't: in these days of multicores the concurrency landscape is changing fast. For example, a particular application might benefit from an application-specific thread scheduling strategy; or, one might wish to experiment with a variety of concurrency abstractions; new challenges, such as multi-processor support or data parallelism [4], place new demands on the runtime system.

An attractive alternative to a monolithic runtime system written by the language implementators is to support concurrency using a *library* written in the language itself. In this paper we explore doing exactly this for the language Haskell and its implementation in the Glasgow Haskell Compiler (GHC). Although concurrency-as-a-library is hardly a new idea, we make several new contributions:

- We describe in detail the interface between the *concurrency library* written in Haskell, and the underlying *substrate*, or runtime system (RTS), written in C. Whilst the basic idea is quite conventional, the devil is in the details, especially since we want to support a rich collection of features, including: foreign calls that may block, bound threads [16], asynchronous exceptions [15], transactional memory [12], parallel sparks [24] and multiprocessors [11].
- Concurrency primitives are notoriously slippery topic, so we provide a precise operational semantics for our implementation.
- A key decision is what synchronization primitives are provided by the substrate. We propose a simplified transactional memory as this interface in Section 3.2, a choice that fits particularly well with a lazy language.
- The substrate follows common practice, offering continuations as a mechanism from which concurrency can be built. However our continuations, which we call *stack continuations* are, by construction, much cheaper than full continuations. Furthermore, capturing a continuation and transferring control to another continuation are elegantly combined in a single `switch` primitive introduced in Section 3.4.
- The whole issue of thread-local state becomes pressing in user-level threads library, because a computation must be able to ask "what is my scheduler?". We propose a robust interface that supports local state in Section 3.5.
- Interfacing Haskell code to foreign functions, especially if those functions may themselves block, is particularly tricky. We build on earlier work to solve this problem in an elegant way.
- We illustrate our interface by describing a scheduler written entirely in Haskell in Section 5.
- We have implemented most features we describe, in a mature Haskell compiler, which gives a useful reality check on our claims.

2. Setting the scene

Our goal is to design a *substrate interface*, on top of which a variety of *concurrency libraries*, written in Haskell, can be built (Figure 1). The substrate is implemented by ourselves and hence, so far as possible, it should implement *mechanism*, leaving *policy*

Haskell'07, September 30, 2007, Freiburg, Germany.

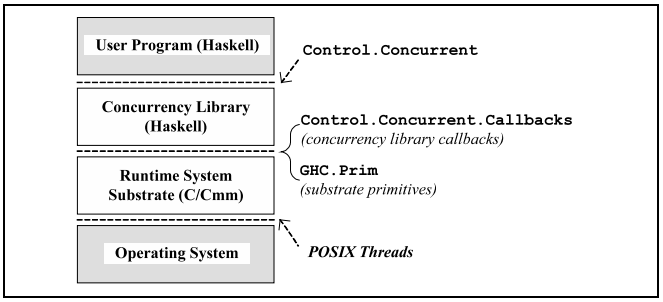

Figure 1: Components of the new RTS design

to the library. In general, we strive to put as little as possible in the substrate, and as much as possible in the concurrency libraries.

The *substrate interface* consists of two parts:

1. A set of *substrate primitives* in Haskell, including primitive *data types* and *operations* over these types (Section 3).
2. A set of *concurrency library callbacks*, specifying interfaces that the concurrency library must implement (Section 4).

The key choices of our design are embodied in the substrate interface: once you know this interface, everything else follows. A good way to think of the substrate interface is that it encapsulates the virtual machine (or operating system) on which the Haskell program runs.

We intend that a single fixed substrate should support a variety of concurrency libraries. Haskell's existing concurrency interface (`forkIO`, `MVars`, `STM`) is one possibility. Another very intriguing one is a compositional (or "virtualizable") concurrency interface [19], in which a scheduler may run a thread that itself is a scheduler... and so on. Another example might be a scheduler for a Haskell-based OS [10] or virtual machine (e.g. HALVM) that needs to give preferential treatment to threads handling urgent interrupts.

In addition to multiple clients, we have in mind multiple implementations of the concurrency substrate. The primary implementation will be based on OS-threads and run atop the ordinary OS. Another possibility is that the RTS runs directly on the hardware, or as a virtualized machine on top of a hypervisor, and manages access to multiple CPUs.

Although written in Haskell, the concurrency library code may require the author to undertake some crucial proof obligations that Haskell will not check; for example, "you may use this continuation at most once, and a checked runtime error will result if you use it twice". This is still (much) better than writing it in C!

We take as our starting point the following design choices:

- It must be possible to write a concurrency library that supports *pre-emptive* concurrency of *very light-weight threads*, perhaps thousands of them. It would be too expensive to use a whole CPU, or a whole OS thread, for each Haskell thread. Instead, a scheduler must multiplex many Haskell fine-grain threads onto a much smaller number of coarse-grain computational resources provided by the substrate.
- Scheduling threads — indeed the very notion of a "thread" — is the business of the concurrency library. The substrate knows nothing of threads, instead supporting (a flavor of) passive continuations. Here we simply follow the path blazed by Mitch Wand [25].
- Since the substrate does not know about Haskell threads, it cannot deal with *blocking* of threads. Hence, any communication mechanisms that involve *blocking*, such as `MVars` and Software Transactional Memory (STM), are also the business of the concurrency library.

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

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

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

data SLSKey a
newSLSKey  :: a -> IO (SLSKey a)
getSLS     :: SLSKey a -> PTM a
setSLS     :: SLSKey a -> a -> IO ()

raiseAsync :: Exception -> IO ()
```

Figure 2: The substrate primitives

- Garbage collection is the business of the substrate, and requires no involvement from the concurrency library.
- The system should run on a shared-memory multi-processor, in which each processor can independently run Haskell computations against a shared heap.
- Because we are working in a lazy language, two processors may attempt to evaluate the same suspended computation (thunk) at the same time, and something sensible should happen.
- The design must be able to accommodate a scheduler that implements the current FFI design [16], including making an out-call that blocks (on I/O, say) without blocking the other Haskell threads, out-calls that re-enter Haskell, and asynchronous in-calls.

3. Substrate primitives

We are now ready to embark on the main story of the paper, beginning with the substrate primitives. The type signatures of these primitives are shown in Figure 2, and the rest of this section explains them in detail.

The details of concurrency primitives are notoriously difficult to describe in English, so we also give an operational semantics that precisely specifies their behavior. The syntax of the system is shown in Figure 3, while the semantic rules appear in Figures 4, 5, 6, 7, 8 and 10. These figures may look intimidating, but we will explain them as we go.

3.1 Haskell Execution Context (HEC)

The first abstraction is a *Haskell Execution Context* or HEC. A HEC should be thought of as a virtual CPU; the substrate may map it to a real CPU, or to an operating system thread (OS thread). For the sake of concreteness we usually assume the latter.

Informally, a HEC has the following behavior:

- A HEC is always in one of three states: *running* on a CPU or OS thread, *sleeping*, or making an *out-call*.

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

SLS Keys $k ::= (r, M)$

Terms

$$
\begin{array}{rcl}
M, N & ::= & r \mid x \mid \texttt{\textbackslash}x \texttt{->} M \mid M\ N \mid \ldots \\
& \mid & \texttt{return}\ M \mid M \texttt{>>=} N \\
& \mid & \texttt{throw}\ M \mid \texttt{catch}\ M\ N \mid \texttt{catchPTM}\ M\ N \\
& \mid & \texttt{newPVar}\ M \mid \texttt{readPVar}\ r \mid \texttt{writePVar}\ r\ M \\
& \mid & \texttt{getHEC} \mid \texttt{waitCond}\ M \mid \texttt{wakeupHEC}\ h \\
& \mid & \texttt{newSLSKey}\ M \mid \texttt{getSLS}\ k \mid \texttt{setSLS}\ k\ M \\
& \mid & \texttt{newSCont}\ M\ D \mid \texttt{switch}\ M
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Program state} & P & ::= & S; \Theta \\
\text{HEC soup} & S & ::= & \emptyset \mid (H \mid S) \\
\text{HEC} & H & ::= & (M, D, h) \mid (M, D, h)_{sleeping} \\
& & \mid & (M, D, h)_{outcall} \\
\text{Heap} & \Theta & ::= & r \hookrightarrow M \oplus s \hookrightarrow (M, D) \\
\text{SLS store} & D & ::= & r \hookrightarrow M \\
\text{Action} & a & ::= & \mathit{Init} \\
& & \mid & \mathit{InCall}\ M \mid \mathit{InCallRet}\ r \\
& & \mid & \mathit{OutCall}\ r \mid \mathit{OutCallRet}\ M \\
& & \mid & \mathit{Blackhole}\ M\ h \mid \mathit{Tick}\ h \\
\text{IO context} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \texttt{>>=} M \mid \texttt{catch}\ \mathbb{E}\ M \\
\text{PTM context} & \mathbb{E}_p & ::= & [\cdot] \mid \mathbb{E} \texttt{>>=} M
\end{array}
$$

Figure 3: Syntax of terms, states, contexts, and heaps

- A Haskell program initially begins executing on a single OS thread running a single HEC.
- When an OS thread enters the execution of Haskell code by making an in-call through the FFI, a fresh HEC is created in the *running* state, and the Haskell code is executed on this HEC. Note that this is the *only* way to create a new HEC.
- When the Haskell code being run by the HEC returns to its (foreign) caller, the HEC is deallocated, and its resources are returned to the operating system.
- When a running HEC makes a foreign out-call, it is put into the *outcall* state. When the out-call returns, the HEC becomes *running*, and the Haskell code continues to run on the same HEC.
- A HEC can enter the *sleeping* state voluntarily by executing `waitCond`. A sleeping HEC can be woken up by another HEC executing `wakeupHEC`. These two primitives are explained in Section 3.3.

Figure 3 shows the syntax of program states. The program state, P, is a "soup" S of HECs, and a heap Θ. A soup of HECs is simply an un-ordered collection of HECs $(H_1 \mid \ldots \mid H_n)$. Each HEC is a triple (M, D, h) where h is the unique identifier of the HEC, and M is the term that it is currently evaluating. The D component is the stack-local state, whose description we defer to Section 3.5. A sleeping HEC has a subscript "*sleeping*"; a HEC making a blocking foreign out-call has a subscript "*outcall*". The heap is a finite map from names to terms, plus a (disjoint) finite map from names to paused continuations represented by pairs (M, D).

A program makes a transition from one state to the next using a *program transition*

$$S; \Theta \implies S'; \Theta'$$

whose basic rules are shown in Figure 4. (We will introduce more rules in subsequent Figures.)

The $(IOAdmin)$ rule says that if any HEC in the soup has a term of form $\mathbb{E}[M]$, and M can make a purely-functional transition to N, then the HEC moves to a state with term $\mathbb{E}[N]$ without affecting any other components of the state. Here, $\mathbb{E}$ is an *evaluation context*, whose syntax is shown in Figure 3, that describes where in the term the next reduction must take place. A *purely-functional transition* includes β-reduction, arithmetic, `case` expressions and so on, which are not shown in Figure 4. However, we do show the purely-functional rules that involve the monadic operators `return`, `(>>=)`, `catch`, and `throw`. Notice also that a HEC in the *sleeping* state or the *outcall* state never takes a $(IOAdmin)$ transition.

Purely-functional transitions $M \rightarrow N$

$$
\begin{array}{rcll}
\texttt{return}\ N \texttt{>>=} M & \rightarrow & M\ N & (\mathit{Bind}) \\
\texttt{throw}\ N \texttt{>>=} M & \rightarrow & \texttt{throw}\ N & (\mathit{Throw}) \\
\texttt{catch}\ (\texttt{return}\ M)\ N & \rightarrow & \texttt{return}\ M & (\mathit{IOCatch}) \\
\texttt{catch}\ (\texttt{throw}\ M)\ N & \rightarrow & N\ M & (\mathit{IOCatchExp})
\end{array}
$$

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

Top-level program transitions $S; \Theta \implies S'; \Theta'$

$$\frac{M \rightarrow N}{S \mid (\mathbb{E}[M], D, h); \Theta \implies S \mid (\mathbb{E}[N], D, h); \Theta} \ (\mathit{IOAdmin})$$

Figure 4: Operational semantics (basic transitions)

In implementation terms, each HEC is executed by one, and only one, OS thread. However, a single OS thread may be responsible for more than one HEC, although all but one will be in the *outcall* state. For example suppose that OS thread `T` makes a foreign in-call to a Haskell function `f1`, creating a HEC `H1` to run the call. Then `f`, running on `H1` which is in turn running on `T`, makes a foreign out-call. Then the state of `H1` becomes *outcall*, and `T` executes the called C procedure. If that procedure in turn makes another foreign in-call to a Haskell procedure `f2`, a second HEC, `H2`, will be allocated, but it too will be executed by `T`. The process is reversed as the call stack unwinds.

To be even more concrete, a HEC can be represented by a data structure that records the following information:

- The identifier of the OS thread responsible for the HEC.
- An OS condition variable, used to allow the HEC to go sleep and be woken up later.
- Registers of the STG machine.
- The current Haskell execution stack.
- The current heap allocation area; each HEC allocates in a separate area to avoid bottlenecking on the allocator.
- A "remembered set" for the garbage collector. It is important for performance reasons that the generational garbage-collector's write barrier is lock-free, so we have a per-HEC remembered set. It is benign for an object to be in multiple remembered sets.

The live HECs (whether running, sleeping or making out-calls) are the roots for garbage collection.

3.2 Primitive transactional memory (PTM)

Since a program has multiple HECs, each perhaps executing on a different CPU, the substrate must provide a safe way for the HECs to communicate and synchronize with each other. The standard way to do so, and the one directly supported by most operating systems, is to use locks and other forms of low-level synchronization such as condition variables. However, while locks provide good performance, they are notoriously difficult to use. In particular, program modules written using locks are difficult to *compose* elegantly and correctly [12].

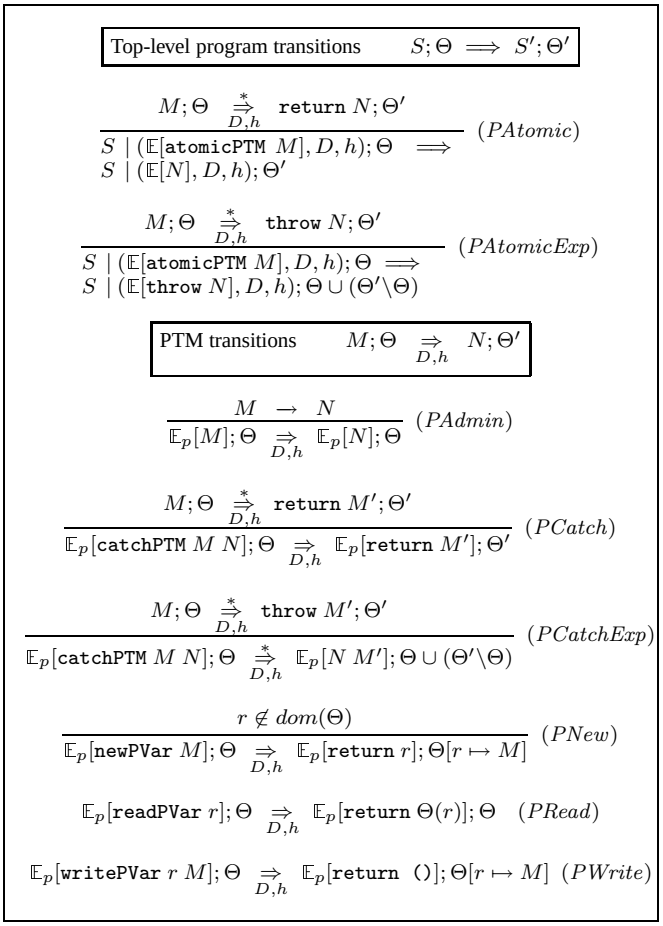

Figure 5: Operational semantics (PTM transitions)

Even ignoring all these difficulties, however, there is another Very Big Problem with using locks as the substrate's main synchronization mechanism in a lazy language like Haskell. A typical use of a lock is this: take a lock, modify a shared data structure (a global ready-queue, perhaps), and release the lock. The lock is used only to ensure that the shared data structure is mutated in a safe way. Crucially, a HEC never holds a lock for long, because blocking another HEC on the lock completely stops a virtual CPU.

Here is how we might realize this pattern in Haskell:

```
do { takeLock lk
   ; rq <- read readyQueueVar
   ; rq' <- if null rq then ...
                       else ...
   ; write readyQueueVar rq'
   ; releaseLock lk }
```

But *if* `rq` *is a thunk, the evaluation of* `(null rq)` *might take an arbitrarily long time*, so the lock `lk` might be held for a long time. That does not threaten correctness, but it does mean that all the other HECs might be held up waiting on `lk`! One could declare that the programmer should somehow ensure that this never happens, but it is far from easy for a programmer to be certain that a blob of code evaluates no thunks.

These observations motivated us to seek an alternative synchronization mechanism. One such alternative is *transactional memory* (TM), which is known to offer a more robust and modular basis for concurrency [12]. There is a dilemma, however, because the fully-featured software transactional memory supports blocking, and cannot therefore be part of the substrate!

Fortunately, all we require in terms of low-level synchronization is the ability to perform atomic transactions; the composable blocking and choice operators provided by STM can be safely omitted. Therefore, the substrate offers an interface that we call *primitive transactional memory* (PTM) [1], whose type signature is shown in Figure 2. Like STM, PTM is a monad, and its computations are fully compositional. Unlike STM, however, *a PTM computation is non-blocking*, so the question of blocking threads does not arise.

As Figure 2 shows, a PTM transaction may allocate, read, and write transactional variables of type `PVar a`. And that is about all, exceptions aside! Thus, a PTM transaction amounts to little more than an atomic multi-word read/modify/write operation. In operational terms, `atomicPTM` runs a PTM computation while buffering the reads and writes in a transaction log, and then commits the log all at once. If read-write conflicts are detected at the time of commit, the transaction is re-executed immediately.

How does this resolve the Big Problem mentioned earlier? The transaction runs without taking any locks and hence, if the transaction should happen to evaluate an expensive thunk, no other HECs are blocked. At the end of the transaction, the log must be committed by the substrate, in a truly-atomic fashion, *but doing so does not involve any Haskell computations*. It is as if the PTM computation generates (as slowly as it likes) a "script" (the log) which is executed (rapidly and atomically) by the substrate. It is likely that a long-running transaction will become invalid before it completes because it conflicted with another transaction. However in this case the transaction will be restarted, and any work done evaluating thunks during the first attempt is not lost, so the transaction will run more quickly the second and subsequent times.

3.2.1 The semantics of PTM

Figure 5 presents the semantics of PTM. A PTM transition takes the form

$$M;\Theta \underset{D,h}{\Rightarrow} N;\Theta'$$

The term M is, as usual, the current monadic term under evaluation. The heap Θ gives the mapping from `PVar` locations r to values M (Figure 3). The subscript D, h on the arrow says that these transitions are carried out by the HEC h, with stack-local state D. We will discuss stack-local state in Section 3.5, and D can be ignored until then.

The PTM transitions in Figure 5 are quite conventional. Rule $(PAdmin)$ is just like $(IOAdmin)$ in Figure 4. The three rules for `PVars` — $(PNew)$, $(PRead)$, and $(PWrite)$ — allow one to allocate, read, and write a `PVar`.

The semantics of exceptions is a little more interesting. In particular, $(PCatchExp)$ explains that if M throws and exception, then *the effects of* M *are undone*. To a first approximation that means simply that we abandon the modified Θ', reverting to Θ, but with one wrinkle: any `PVars` allocated by M must be retained, for reasons discussed by [12]. The heap $\Theta' \backslash \Theta$ is that part of Θ' whose domain is not in Θ.

The rules for `atomicPTM` in Figure 5 link the PTM transitions to the top-level IO transitions. The $(PAtomic)$ rule embodies the key idea, that *multiple* PTM transitions are combined into a *single* program transition. In this way, no HEC can observe another HEC half-way though a PTM operation.

3.3 HEC blocking

A PTM transaction allows a HEC safe access to mutable shared states between HECs. But what if a HEC wants to block? For

[1] Please do not confuse our PTM with "Paged-based Transactional Memory" by Chuang et. al., 2006.

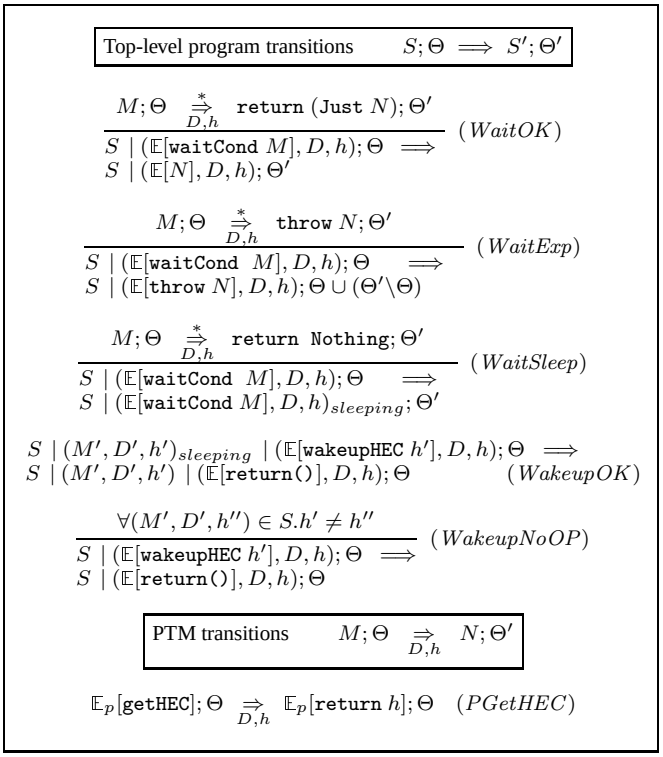

Figure 6: Operational semantics (HEC blocking)

example, suppose there are four HECs running, but the Haskell program has only one thread, so that there is nothing for the other three HECs to do. They could busy-wait, but that would be a poor choice if a HEC was mapped to an operating system thread in a multi-user machine, or in a power-conscious setting. Instead, we want some way for a HEC to *block*.

The common requirement is that we want to block a HEC until some conditions are met, for example, when tasks become available. Traditionally, such code is often implemented using *condition variables*, which themselves need to be protected using locks. Since we are now using PTM instead of locks, we design a *transactional* interface, `waitCond`, to perform blocking based on condition testing. The semantics is shown in Figure 6.

```
waitCond   :: PTM (Maybe a) -> IO a
wakeupHEC  :: HEC -> IO ()
```

The `waitCond` operation executes a transaction in nearly the same way as `atomicPTM`, except that it checks the resulting value of the transaction. If the transaction returns `Just` x, `waitCond` simply commits the transaction and returns x. Otherwise, if the result is `Nothing`, the HEC commits the transaction, and puts the HEC to sleep *at the same time*.

The `wakeupHEC` operation wakes up a sleeping HEC. After a HEC is woken up, it re-executes the `waitCond` operation which blocked it. If the HEC is not sleeping, `wakeupHEC` is simply a no-op. The atomicity of `waitCond` is important, otherwise a `wakeupHEC` might intervene between committing the transaction and the HEC going to sleep, and the wake-up would be missed.

As an example, suppose that the concurrency library uses a single shared run-queue for Haskell threads. A HEC uses `waitCond` to get work from the queue. If it finds the queue empty, it adds its own HEC identifier (gotten with `getHEC`) to a list of sleeping HECs attached to the empty run-queue, and goes to sleep.

When a running HEC adds a Haskell thread into the queue, it looks at the list of sleeping HECs and awakens one of them. Of course, by the time the sleeping HEC actually wakes up and runs, the queue may again be empty, but in that case the same sequence of events takes place again: the `waitCond` is re-run, and the HEC will go to sleep again. In effect, the classic error of forgetting to re-test the condition after blocking on a condition variable is eliminated by construction.

3.4 Stack continuations and context switching

A HEC is an abstraction of a virtual processor; in a given system we expect to have a handful of HECs running, roughly one for each physical CPU. To model fine-grain Haskell threads, we need an abstraction of a Haskell computation, together with a way to allow a HEC to multiplex its resources over such computations. Following Wand, we use a *continuation* to model a (suspended) Haskell computation [25]. Unlike Wand, our continuations are not first class — in particular, they can only be used once — in exchange for which they are dirt cheap to implement.

We provide one new data type and two new primitive operations (Figure 2):

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

An `SCont`, or *stack continuation*, should be thought of as passive value representing an I/O-performing Haskell computation that is suspended in mid-execution. The call `(newSCont io)` makes a new `SCont` that, when scheduled, will perform the action `io`. The primitive `switch` is the interesting part. The call (`switch` M) does the following:

- It captures the current computation as an `SCont`, say s. We call s the *current continuation*.
- Then it runs the primitive transaction $(M\ s)$. This transaction may read and write some `PVars` — for example, it may write `s` into a ready-queue — before returning an `SCont`, say s'. We call s' the *switch target*.
- Lastly, `switch` makes s' into the computation that the current HEC executes.

These steps are made precise by the rules of Figure 7. An `SCont` is represented by an `SCont` identifier (or *stack identifier*), s. The heap Θ maps a stack identifier to a pair (M, D) where M is the term representing the suspended computation, and D is its stack-local state. Again, we defer discussion of the stack-local state until Section 3.5. Rule $(NewSCont)$ simply allocates a new `SCont` in the heap, returning its identifier s.

All the rules for `switch` start the same way, by allocating a fresh identifier s and running $(M\ s)$ as a transaction. If the transaction completes normally, returning s', we distinguish two cases. In rule $(SwitchSelf)$, we have $s = s'$ so there is nothing to be done. In the more interesting case, rule $(Switch)$, we transfer control to the new continuation s', storing in the heap the current, but now suspended, continuation s. By writing $\Theta'[s' \mapsto (M', D')]$ on the top line of $(Switch)$ we mean that Θ' *does not include* s'. The computation proceeds without a binding for s' because s' is "used up" by the `switch`. Any further attempts to switch to the same s' will simply get stuck. (A good implementation should include a run-time test for this case.)

Figure 7 also describes precisely how `switch` behaves if its argument throws an exception: the `switch` is abandoned with no effect (allocation aside).

Note that, unlike many formulations of continuations, our stack continuation does not carry a returning value. This design makes it easier to have a well-typed `switch`. No expressiveness is lost, be-

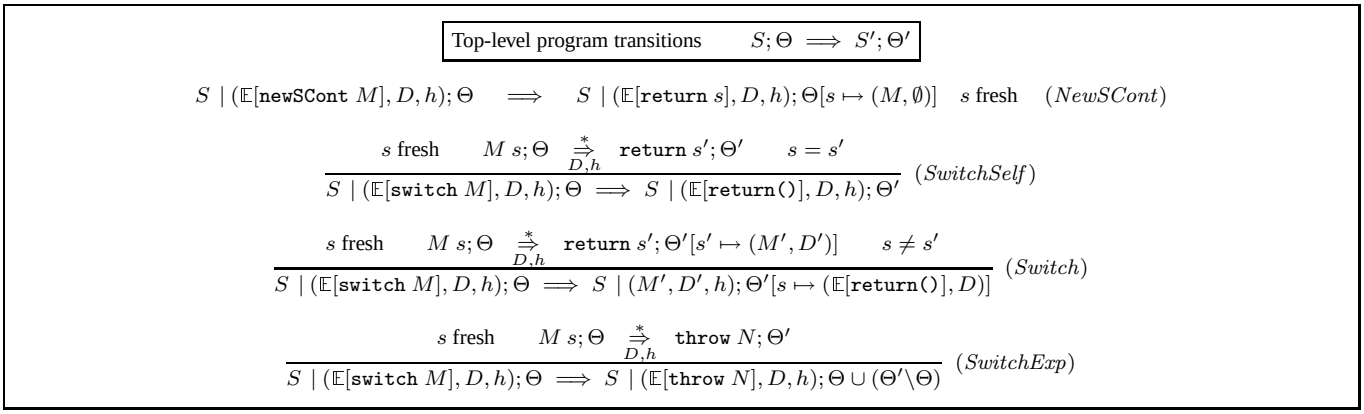

Figure 7: Operational semantics (stack continuations and context switching)

cause values can still be communicated using shared transactional variables (as we will show in Section 5.1).

3.4.1 Using stack continuations

With these primitives, a number of Haskell computations can be multiplexed on one HEC in a cooperative fashion: each computation runs for a while, captures and saves its continuation, and voluntarily switches to the continuation of another computation. More concretely, here is some typical code for the inner loop of a scheduler:

```
switch $ \s -> do
  ...
  save s in scheduler's data structure
  ...
  s' <- find the next thread to schedule
  ...
  return s'
```

It captures the current continuation `s`, saves `s` into the scheduler's data structure, finds the continuation of the next thread to be scheduled `s'`, and control is transferred to `s'`.

3.4.2 Implementing stack continuations

By design, `SConts` have a particularly cheap representation. In GHC, a Haskell computation runs on a stack, which itself is held in a *stack object* allocated in the run-time heap. Initially the stack object is small, but it can grow by being copied into a larger area if it overflows. An `SCont` is represented simply by a pointer to the stack object for its stack. When `switch` captures an `SCont`, it uses the pointer to the stack object; no copying is done, as is necessary for truly first-class continuations.

We are not, of course, the first to think of the idea of identifying stacks with second-class continuations [6]. However, our `switch` primitive deals rather neatly with a tiresome and non-obvious problem. Consider the call

```
switch (\s -> stuff)
```

The computation `stuff` must run on *some* stack, and it's convenient and conventional for it to run on the current stack. But suppose `stuff` writes `s` into a mutable variable (the ready queue, say) and then, while `stuff` is still running, another HEC picks up `s` and tries to run it. Disaster! Two HECs are running two different computations on the same stack. Fisher and Reppy recognized this problem and solved it by putting a flag on `s` saying "I can't run yet", and arranging that any HEC that picks up `s` would busy-wait until the flag is reset, which is done by `switch` when `stuff` finishes [6]. Although this works, it's a bit of a hack, and would complicate our semantics. The current GHC runtime deals with this by ensuring that there is always a lock that prevents the thread from being rescheduled until the switch has finished, and arranging to release the lock as the very last operation before switching - again this is fragile, and has been a rich source of bugs in the current implementation.

However, by integrating `switch` with PTM we can completely sidestep the issue, because the effects of `stuff` are not published to other HECs until `stuff` commits and control transfers to the new stack. To guarantee this, the implementation should commit the transaction and change the HEC's stack in a single, atomic operation.

The other error we must be careful of is when a stack continuation is the target of more than one `switch` — remember that stack continuations are "one-shot". To check for this error we need an indirection: an `SCont` is represented by a pair of a pointer to a stack and a bit to say when the `SCont` is used up. Another alternative would be to keep a sequence number in the stack object, incremented by every `switch`, and store the number in the `SCont` object.

3.5 Global and stack-local states

Because the concurrency library is written in an cooperative fashion, the code often needs to query for information like this:

- What is my thread identifier?
- Who is my scheduler?
- Where is the ready queue?

The code in Section 3.4.1 gives a more concrete example, in which the scheduler's data structure needs to be located. In principle there is nothing to prevent one adding a `ThreadId` parameter to every function that needs to know the thread identifier; and similarly for the other cases like the scheduler's task queues. However, doing so is extremely inconvenient and non-modular. We are already, in effect, passing the state of the world to every (effectful) function via the monad, and we would like all other state-passing to be implicit.

3.5.1 Global state

Suppose the concurrency library wanted a global, ready-queue of threads, shared among all HECs. Haskell provides no support for such a thing, so programmers use the well-known `unsafePerformIO` hack:

```
readyQueue :: PVar ReadyQueue
readyQueue = unsafePerformIO $ atomicPTM $
             newPVar emptyQueue
```

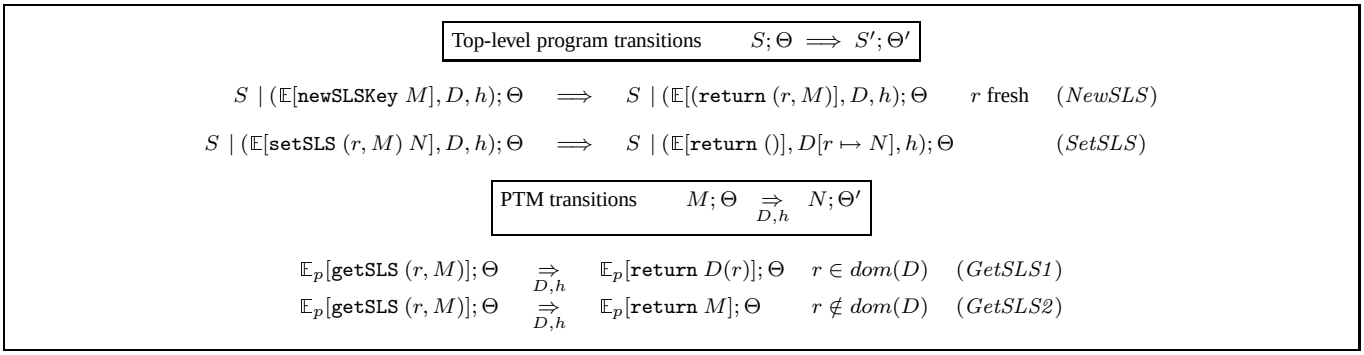

Figure 8: Operational semantics (stack-local state transitions)

This is obviously horrible, and the whole issue of accommodating effectful but benign top-level computations in Haskell has been frequently and heatedly discussed on the Haskell mailing list[2]. For the purposes of this paper we will simply assume that *some* decent solution is available, so that one can write something like this:

```
readyQueue :: PVar ReadyQueue
init readyQueue <- newPVar emptyQueue
```

Here the "`init`" keyword introduces a PTM transaction to be run once, at module initialization time or at some subsequent point. The effects permitted for such a transaction might be even more restricted than usual, perhaps involving only allocation. The binding should of course be monomorphic to avoid unsoundness, which is a well-known problem with `unsafePerformIO`.

3.5.2 Stack-local states

Now suppose we wanted to implement a *hierarchical* scheduler, in which any thread can be a scheduler for its child threads. Then there is no global ready queue; instead, each scheduler in the tree maintains its own. This is just one example of a well-known problem with multithreaded programming, namely the need for *thread-local state*. Other examples include: the seed for a random number generator (sharing a global one is a concurrency bottleneck); the `stdin` and `stdout` handles; and so on.

One might expect that the programmer could implement thread-local states entirely in Haskell, using globally shared data structures, such as hash tables, indexed by some form of thread identifier. But this approach has a few drawbacks. First, it may not be efficient: accessing a thread-local state could be much slower than performing a regular memory reference, especially if the implementation used purely functional data structures. More importantly, automatic garbage collection would not work for such states: the programmer would have to free them manually when their corresponding threads die, otherwise memory would be leaked.

Thus motivated, we propose to support *stack-local states* (SLS) directly in the substrate, using the following design shown in Figure 2:

```
data SLSKey a
newSLSKey :: a -> IO (SLSKey a)
getSLS    :: SLSKey a -> PTM a
setSLS    :: SLSKey a -> a -> IO ()
```

Each item of stack-local state is identified by a typed *SLS key*. For example, the key for `stdin` might be of type `SLSKey Handle`. The `getSLS` operation maps the key to its correspondingly-typed value, Each `SCont` carries a distinct mapping of keys to values, named D in our semantic rules, and this mapping persists across the suspensions and resumptions caused by `switch`; that is, an `SCont` now has an identity.

The detailed semantics are given in Figure 8. Several points are worth noticing:

- An `SCont` is represented by a pair (M, D) of a term M to be evaluated and a *dictionary* D that maps SLS keys to values (Figure 3).
- A running HEC (M, D, h) includes the dictionary of the running computation. When `switch` switches to a new computation, it loads its dictionary into the HEC (rule $(Switch)$ in Figure 7).
- The `newSCont` primitive makes a new `SCont` whose dictionary is empty (rule $(NewSCont)$).
- The `newSLSKey` primitive takes an *initial value* as its first argument, and a SLS Key is represented by a pair (r, M) of a unique identifier r and the the initial value M. Typically there will be a handful of SLS keys (`stdin`, the current scheduler, the random-number seed), but many stack continuations each with a potentially-different set of bindings for the keys. The SLS keys would usually be globally allocated; for example:

  ```
  stdinKey :: SLSKey Handle
  init stdinKey <- newSLSKey stdin
  ```

- If `getSLS` is given a key (r, M) whose identifier r is not present in the dictionary for the current computation, it returns the initial value M. This eliminates the necessity to initialize the dictionary with a binding for every SLS that could possibly be used.
- Stack-local state is manipulated only by the computation that owns it, and hence does not need to be transacted. Hence `getSLS` is a PTM operation, because it is convenient to be able to read it during a PTM transaction, while `setSLS` is an IO operation because we do not want the complication of having to undo `setSLS` operations if the transaction aborts. Note that `setSLS` operations are expected to be fairly rare.
- If the programmer wants to manipulate *shared* state accessed via the SLS mechanism, or to treat SLS state transactionally, the right thing to do is to make the SLS value a `PVar` and access it using PTM transactions.

In implementation terms, the identifier r of a SLS key (r, M) can be just a small integer, and the dictionary can be an array of slots in the stack object. Some overflow mechanism is needed for when there are more than a handful of SLS keys in use. Although not shown in the formal semantics, it is worth noting that the runtime system should automatically garbage-collect unused stack-local states: a stack and its local state are deallocated at the same

[2] `http://www.haskell.org/haskellwiki/Top_level_mutable_state`

time. An implementation is not required to reclaim unused SLS key values because such values are supposed to be globally-shared constants, and we don't expect there to be very many of them.

3.5.3 HEC-local states?

One might naively expect the substrate to support *HEC-local* states as well. A HEC could use local state to maintain its own scheduling data structures, such as task queues. But, in reality, such structures are almost always globally shared by all HECs so that load can be balanced using work stealing algorithms. In such cases global states are often more suitable. Also, HEC-local states only appear to be useful when writing the concurrency library. In contrast, stack-local states have broader applications: end-users can use them as *thread-local* states without much change.

More importantly, programming with HEC-local states can be tricky, because such states are *dynamically bound*: the execution of a sequential program can be interleaved on multiple HECs. A sequential code block can access one HEC's local state in one step, pause, be moved to a different HEC, and then access another HEC's state in the next step. In contrast, a sequential code block is always bound to a stack during its execution, so the programmer can safely assume that the SLS environment is fixed for a code block.

For these reasons, we do not currently plan to support HEC-local states, although they could be easily added via another set of primitives if desired.

4. Pre-emption, foreign calls, and asynchrony

```
rtsInitHandler   :: IO ()
inCallHandler    :: IO a -> IO a
outCallHandler   :: IO a -> IO a
timerHandler     :: IO ()
blackholeHandler :: IO Bool -> IO ()
```

Figure 9: The concurrency library callbacks

In addition to the substrate primitives shown in the previous section, the substrate interface also includes some callback functions, shown in Figure 9. These are functions supplied by the concurrency library, that are invoked by the RTS[3].

An external IO transition, $S;\Theta \stackrel{a}{\Longrightarrow} S';\Theta'$, is an IO transition tied to an action a; see Figure 10. The actions include FFI in-calls, timer events and blocking events.

4.1 Pre-emption

So far, the concurrency primitives introduced allow cooperative scheduling: a Haskell thread can only switch to another thread by voluntarily calling `switch`. This section introduces a mechanism for *pre-emptive* scheduling. This mechanism could be generalized to handle other asynchronous signals too.

The RTS substrate maintains a timer that ticks every 50ms by default. When a timer event is detected, the RTS substrate calls a timer handler function `timerHandler` exported by the concurrency library.[4] (This is the first time that the RTS calls the concurrency library; most of the calls work the other way around. Figure 9 summarizes all the call-backs we will discuss.)

[3] Note that this means the RTS is statically bound to a particular concurrency library when the program is linked. Nevertheless, we envisage that it will be possible to choose a concurrency library at link-time, or earlier. This design does not make it possible to compose concurrency libraries from different sources at runtime, however.

[4] More precisely, the handler is invoked at the first garbage collection point following the timer event.

The timer handler is triggered on every HEC that is running Haskell computation; i.e. is not *sleeping* or in an *outcall*. When the timer handler is triggered on a HEC, the state of the current computation is saved on the stack, and the timer handler uses the top of the stack to execute. The stack layout is set up in a way as if the timer handler is being explicitly called from the current Haskell computation, so when the timer handler finishes execution, the original computation is automatically resumed.

This semantics for the timer handler makes it easy to implement pre-emption, because a stack continuation captured inside the timer handler also contains the current computation on the HEC. Typically the timer handler will simply switch to the next runnable thread, as if the thread had invoked `yield` manually.

The RTS substrate must guarantee that timer handlers are called only at safe points. For example, the timer handler must not interrupt the final committing operation of a PTM transaction. Nevertheless, it is safe to call the timer handler during the script-building phase of a PTM transaction. The PTM implementation should allow the timer handler to run a new transaction, even if an old transaction is already running on the same HEC.

Pre-emption has a slightly tricky interaction with stack-local state. Because a SLS is initialized by the code running on that stack, it is possible that the interrupt handler is called before such initialization finishes. In such cases the interrupt handler will see the default initial value registered by `newSLSKey`, and the programmer must handle such cases explicitly.

4.2 Interrupting execution at thunks

In principle, any attempt to evaluate a thunk may see a *blackhole* because the thunk is already being evaluated by another thread [11]. If a blackhole is found, the best general policy is to pause the current thread until evaluation the thunk has completed (or at any rate until there is reason to believe that it *may* have completed). This implies that thunk evaluation sometimes needs to interact with the scheduler. In the old RTS design, the scheduler is built into the RTS, so it is easy to implement this policy. In our new design, however, implementing this policy requires a delicate communication between the substrate (which alone can detect when a thread evaluates a thunk that is already under evaluation) and the library (which alone can perform context switching and blocking of threads).

We propose to solve this problem using a special handler function `blackholeHandler` exported by the concurrency library. This function is called by the RTS whenever evaluation sees a blackhole; the execution model is the same as `timerHandler`.

The current runtime system design keeps track of the threads suspended on thunks in a global list. The list is periodically checked by the scheduler to see if any conflicting thunk evaluation has completed. To implement this polling design, the `blackholeHandler` takes an argument of type `(IO Bool)`, which is a function that can be called by the concurrency library to test whether the thread can be resumed. When evaluation enters a blackhole, the RTS substrate creates such a function closure and pass it to `blackholeHandler`.

The `(IO Bool)` polling action is purely to allow the thread's status to be polled without trying to switch to the thread. It is safe to switch to the thread at any time: if the thunk is still under evaluation, the thread will immediately call `blackholeHandler` again. So the simplest implementation of `blackholeHandler` just puts the current thread back on the run queue, where it will be tried again in due course.

A caveat of this design is that handlers can re-enter: if a blackhole is entered inside the `blackholeHandler`, the program may enter an infinite loop! One possible solution is that the programmer can use stack-local state to indicate whether the thread is already running a `blackholeHandler`, and `blackholeHandler` falls back to busy waiting if re-entrance occurs.

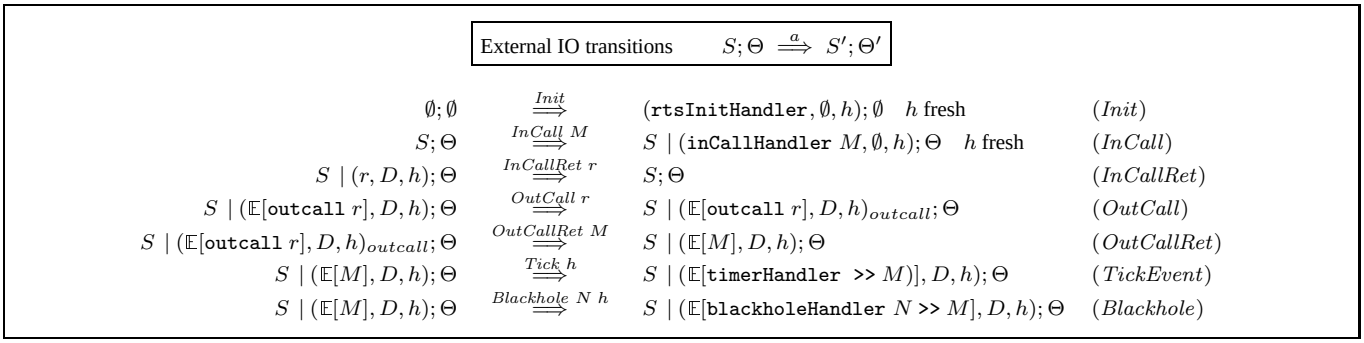

Figure 10: Operational semantics (external interactions)

4.3 Asynchronous exceptions

We would like to implement asynchronous exceptions [15] in the concurrency library. Asynchronous exceptions are introduced by the `throwTo` operation:

```
throwTo :: ThreadId -> Exception -> IO ()
```

which raises the given exception in the context of a target thread. Implementing asynchronous exceptions is tricky, particularly in a multi-processor context: the target thread may be running on another processor, it may be in the run queue waiting to run on some processor, or it may be blocked. The implementation of `throwTo` must avoid conflicting with any other operation that is trying to access the target thread, such as its scheduler, or a thread trying to wake it up.

We can divide the execution of an asynchronous exception into two steps:

1. the invoking thread communicates to the target thread that an exception should be raised; and
2. the target thread actually raises the exception.

Fortunately, only step (2) absolutely requires specialized substrate support, namely a single operation, given earlier in Figure 2:

```
raiseAsync :: Exception -> IO ()
```

The `raiseAsync` function raises an exception in the context of the current thread, but in a special way: any thunk evaluations currently under way will be suspended [20] rather than simply terminated as they would be by a normal, synchronous exception. If the suspended thunk is ever forced later, evaluation can be restarted without loss of work.

Step (1) can be implemented entirely in the concurrency library. One possible approach is to have the exception posted to the target thread via a PVar that is part of its local state and checked during a context-switch. Compared to the current implementation in GHC's RTS, this is not quite as responsive: the target thread may not receive the exception until its time-slice expires, or until it is next scheduled. We could improve this by providing an additional substrate primitive to interrupt a remote HEC at its next safe point. Such an interrupt could be delivered as a simulated timer interrupt or as a new, distinct signal with its own handler.

Compared to the implementation of `throwTo` in the current runtime system, implementing `throwTo` in Haskell on top of the substrate is a breeze. PTM means that many complicated locking issues go away, and the implementation is far more likely to be bug-free.

4.4 Foreign calls

Foreign calls and concurrency interact in delightfully subtle ways [16]. It boils down to the following requirements:

- The Haskell runtime should be able to process in-calls from arbitrary OS threads.
- An out-call that blocks or runs for a long time should not prevent execution of the other Haskell threads.
- An out-call should be able to re-enter Haskell by making an in-call.
- Sometimes we wish to make out-calls in a particular OS thread ("bound threads").

Fortunately the substrate interface that makes all this possible is rather small, and we can push most of the complexity into the concurrency library.

In-call handler Whenever the foreign code makes a FFI in-call to a Haskell function `hFunc`, the RTS substrate allocates a fresh HEC with a fresh stack, and starts executing Haskell code on the new HEC. But, instead of running the Haskell function `hFunc` directly, it needs to hand over this function to the concurrency library, and let the concurrency library *schedule* the execution of `hFunc`!

For this purpose, the concurrency library exports a callback function to accept in-calls from the substrate:

```
inCallHandler  :: IO a -> IO a
```

When an in-call to `hFunc` is made, the RTS substrate executes `(inCallHandler hFunc)` on a fresh HEC with a fresh stack, using the current OS thread. When `inCallHandler` returns, the HEC is deallocated and control is transferred back to foreign code, passing the return value.

The in-call handler is the entry point of the concurrency library: the schedulers accept jobs from the in-call handler. In a standalone Haskell program, the RTS makes an in-call to `Main.main` after the concurrency library is initialized (Section 4.5).

Out-call handler In order to give the concurrency library control over the way an out-call is made, the substrate arranges to invoke the callback `outCallHandler` for each safe out-call. For example, the following out-call:

```
foreign import ccall safe "stdio.h putchar"
  putChar :: CInt -> IO CInt
```

would be desugared into a call to `outCallHandler` at compile-time:

```
putChar arg = outCallHandler (putChar1 arg)
putChar1 arg = ... [the actual out-call] ...
```

The `outCallHandler` function can then decide how to schedule the execution of the actual out-call, `putChar1`.

The compiler implementation can choose to bypass the out-call handler for unsafe calls to improve performance.

4.5 Initialization handler

The concurrency library can be initialized through a callback function. When a Haskell program is started, the RTS will initialize itself, create a fresh HEC, and run the `rtsInitHandler` callback function. This function should create all the necessary data structures in the concurrency library, initialize the schedulers and make them ready to accept FFI in-calls.

5. Developing concurrency libraries

The main task of the concurrency library is to implement the notion of a Haskell *thread* and to provide application programming interfaces such as `forkIO`, `MVars` and STM. Given the underlying substrate interface, there are many design choices for the concurrency library. Here we discuss some possible designs.

The substrate design suggests that the concurrency library should be written in a cooperative fashion. A `SCont` represents the continuation of a Haskell thread. Threads can be created using `newSCont` and make context switches to each other. Thread-local information, such as thread identifiers, can be implemented straightforwardly using stack-local states.

The interesting question is how to design the scheduler. Naively, the simplest scheduler can consist of a globally shared data structure with some common procedures, such as adding a new thread, switching to the next thread, blocking and unblocking, etc. However, the scheduler can be quite complicated when many concurrency features are implemented. Besides the concurrency features that already exist in the current GHC, it would also be useful to make the scheduler code extensible by the end user, so new concurrency features can be readily added. Thus, the concurrency library needs a modular and extensible design. A promising design pattern is the concept of *hierarchical schedulers* discussed in Section 5.2.

5.1 A simple concurrency library

This section uses pseudo code to illustrate how to write a simple concurrency library. We assume that the scheduler's data structure is globally shared and initialized in `rtsInitHandler`. To create a Haskell thread, we simply create a stack continuation and submit it to the scheduler:

```
forkIO :: IO () -> IO ThreadId
forkIO action = do
  sc <- newSCont action
  atomicPTM $ do
    (put sc in scheduler's queue)
    id <- (create new thread id)
    (initialize the new thread's SLS)
    return $ ThreadId id
```

To make an context switch by voluntarily yielding control, we use the `switch` primitive together with a PTM transaction:

```
yield :: IO ()
yield = switch $ \c -> do
  (store c into scheduler's queue)
  n <- (get the next thread to run)
  (update scheduler's state and/or SLS)
  return n
```

To support pre-emptive scheduling, we can simply set the timer handler to be `yield`:

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

Every scheduler must provide a `blackholeHandler`, too. The simplest implementation of `blackholeHandler` is just this:

```
blackholeHandler :: IO Bool -> IO ()
blackholeHandler _ = yield
```

A thread suspended on a thunk will just go back on the run queue, but that's OK; next time it runs it will either immediately invoke `blackholeHandler` if the thunk is still under evaluation, or it will continue. This is a perfectly reasonable, if inefficient, implementation of `blackholeHandler`.

The code above forms the very basic skeleton of the concurrency library. Next, we implement the popular `MVar` synchronization interface. An `MVar` can be implemented as a `PVar` containing its state. If the `MVar` is full, it has a queue of pending write requests; if the `MVar` is empty, it has a queue of pending read requests. Each pending request is attached with a function closure (of type `PTM()`) that can be called to *unblock* the pending thread.

```
data MVar a = MVar (PVar (MVState a))
data MVState a = Full a [(a,      PTM ())]
               | Empty  [(PVar a, PTM ())]
```

The following code shows how to implement `takeMVar`; the `putMVar` operation is the dual case. A pending read request is implemented using a temporary `PVar`. If the `MVar` is empty, the current thread will be blocked, but a function closure is created to unblock the current thread later. If the `MVar` is full and there are additional threads waiting to write to the `MVar`, one of them is unblocked by executing its corresponding closure.

```
takeMVar :: MVar a -> IO a
takeMVar (MVar mv) = do
  buf <- atomicPTM $ newPVar undefined
  switch $ \c -> do
    state <- readPVar mv
    case state of
     Full x [] -> do
       writePVar mv $ Empty []
       writePVar buf x
       return c
     Full x l@((y,wakeup):ts) -> do
       writePVar mv $ Full y ts
       writePVar buf x
       wakeup
       return c
     Empty ts -> do
       let wakeup = (put c into scheduler's queue)
       writePVar mv $ Empty (ts++[(buf,wakeup)])
       n <- (get the next thread to run)
       (update scheduler's state and/or SLS)
       return n
  atomicPTM $ readPVar buf
```

In a real implementation, the above code can be optimized by using a non-transactional mutable state (such as IORef) for the `buf` variable, because its operations are guaranteed not to conflict. Also, we should use a double-ended queue to avoid the `++` in the `Empty` case.

5.2 Developing hierarchical schedulers

An important goal of the new RTS design is to implement *hierarchical scheduling* [19, 9] in concurrency libraries. The idea is that each thread can act as a parenting *scheduler* that divides its CPU cycles on its children threads (or, *schedulees*) and manages the interleaving of execution. If a child thread itself can also act as a parenting scheduler for other threads, all the threads in the system form a tree-like scheduling hierarchy.

It is not difficult to implement a specific system with some scheduling hierarchy; the challenge is to make the code of sched-

ulers *composable*: a child thread, without knowing all the implementation details of its parenting scheduler, can also act as a scheduler itself and have descendants.

A composable design of hierarchical schedulers can be beneficial to applications that have their own scheduling requirements. For example, to process a group of concurrent tasks with different priorities, a thread can act as a priority scheduler and run the tasks in children threads. Hierarchical scheduling also gives the programmer more control in concurrent programming: if a thread wants to run some tasks speculatively with a timeout limit and only needs the result of the task that finishes first, it can act as a scheduler and monitor the execution of individual tasks in its children threads.

It would also be appealing to make the scheduler code *reusable*, so some generic scheduling mechanisms, such as time sharing, priority scheduling, tentative computing and time-outs, can be implemented in library modules and employed by any thread.

The substrate design introduced in this paper suggests that such hierarchical schedulers can be developed in a cooperative fashion, in which a scheduler and its schedulees work together using a common interface. The common interface is *abstract* in the sense that the implementation details of the scheduler and the schedulee are kept hidden from each other: they can work together as long as they both respect the interface. Such an interface can consist of two parts:

- *Shared data structures* used to communicate between the scheduler and the schedulee. For example, a thread needs to know "how many time slices do I have". If the thread runs out of its allocated time slices, it needs to yield to its scheduler, so it also needs to know "who is my scheduler". Such shared data can be implemented using stack-local states and transactional variables.
- *Protocols* that specify how the scheduler and the schedulee should cooperate using the shared data structures and the `switch` primitive. For example, the protocol may specify that (i) the scheduler always assigns some time slices to the schedulee before switching to it, and (ii) the schedulee must yield to its scheduler as soon as its time slices are used up.

We explored a few possible designs of hierarchical scheduling in a prototyping environment that simulates the substrate interface using continuation monads. As a first step, we developed a round-robin scheduler as a library module. The round-robin scheduler can be parameterized by the size of a time slice, and multiple schedulers can be composed in a tree-like hierarchy. We then developed a top-level, SMP scheduler to distribute work on multiple OS threads using work stealing algorithms. The common scheduling interface is designed such that each round-robin scheduler treats its parenting scheduler abstractly, without knowing the parent's configuration. Thus, a user program can specify the scheduling hierarchy by composing the instances of schedulers at the top-level.

6. Implementation and performance

It is a substantial engineering task to modify the GHC RTS to support the substrate interface. Currently, our prototype implementation supports most of the substrate interface, except a few primitives such as asynchronous exceptions and blackhole handlers. Our prototype implementation is not yet optimized for multiprocessors and FFI.

Although there is still much work left to be done, our existing prototype already allows us to to develop simple concurrency libraries and obtain performance measurements. Building all concurrency features on top of a software transactional memory interface certainly adds more overheads to the system, and we hope it is not too much. On top of our substrate prototype, we developed a simple concurrency library that supports single-processor, round-robin scheduling and MVar operations. We then tested its performance and yielded some preliminary results.

	ghc-6.6	fake-ptm	real-ptm
spawn-test	18	32	46
producer-consumer	4.3	7.0	16.2
cheap-concurrency	6.5	7.1	12.6
chameneos	6.3	4.8	26

Figure 11: Benchmark results (program execution time in seconds)

For many multithreaded programs that are computation-intensive, context switching and synchronization is rarely the bottleneck, so the concurrency implementation has little impact on the overall performance. To reveal the actual overheads, we picked a few benchmarking programs: *spawn-test* performs a stress test on spawning new threads using `forkIO`; *producer-consumer* performs a stress test on synchronizing two threads using `takeMVar` and `putMVar`; *cheap-concurrency* and *chameneos* are concurrency benchmarks from the Computer Language Shootout Benchmarks [23].

The test results are shown in Figure 11. The table shows program execution time in three different configurations:

- *ghc-6.6*: the vanilla GHC 6.6 RTS.
- *real-ptm*: our modified GHC RTS substrate prototype with the concurrency library written in Haskell. The PTM implementation reused most code in the GHC 6.6 STM implementation.
- *fake-ptm*: same as *real-ptm*, except that the PTM implementation is fake. A `PVar` is implemented as an `IORef` and there is no transaction control. This configuration only works correctly on a single threaded RTS; the only purpose of this configuration is to reveal the overhead of PTM alone.

In these benchmarks, the new RTS design (column *real-ptm*) is 2-4 times slower than the existing GHC RTS (column *ghc-6.6*). By comparing *real-ptm* and *fake-ptm*, we can see that most of the additional overheads are caused by using PTM, which is a purely software implementation of transactional memory.

Although the new RTS design has a significant overhead, the overall synchronization performance still remains in roughly the same order of magnitude—it is still much better than using OS threads! On the other hand, these results suggest that the performance of software transactional memory needs to be improved to deliver performance comparable with that of locks.

7. Related work

The idea of using *continuations* to write a concurrency library in the language itself is not new [25, 22, 21]. There are two different strategies for implementing *lightweight* continuations in a language: first-class continuations can be made cheap for CPS-based runtime implementations [1] such as SML/NJ, and one-shot continuations [3] are more suitable for stack-based implementations. Our design uses the latter because the GHC runtime model is stack-based. In Haskell, the CPS monad can also be used to implement lightweight concurrency [5], but this approach cannot support preemption and it is thus limited to applications where cooperative scheduling is suitable.

Morrisett and Tolmach[17] extended continuation-based concurrency for SML/NJ to multiprocessors, by adding primitives types and operations for virtual processors and synchronization. The Sting language [13, 14] is a variant of Scheme that supports multiple parallel-programming constructs in a unified framework, which includes threads and virtual processors as primitive types.

Fisher and Reppy designed BOL as a compiler intermediate language to implement concurrency mechanisms [6]. They observed the problem we mentioned in Section 3.4.2, that a continuation shall not be used until the current thread has yielded control. BOL solves this problem by locking the current thread before publishing the continuation; our design of the `switch` primitive elegantly solves this problem by combining context switching with a memory transaction.

The recent Manticore project [8, 18, 7] is very similar to our work. The Manticore language is specifically designed to develop low-level runtime frameworks that support heterogeneous parallelism and complex scheduling policies. Manticore is based on a strict, ML-like language design; our design uses the Haskell language itself, which is pure and lazy, and also deals with special problems in Haskell such as thunk blackholing. Our HEC abstraction is similar to Manticore's notion of a *vproc* (virtual processor). Manticore uses the *compare-and-swap* operation and concurrent queues as synchronization primitives. In contrast, our substrate supports the higher-level notion of *transactional memory*. On the other hand, the Manticore substrate supports load-balancing and migration across vprocs, whereas we handle these entirely within the library.

Lastly, Berthold et. al. designed a run-time environment for implicitly parallel programs, using Concurrent Haskell and the existing GHC runtime system as a substrate [2].

8. Summary

This paper proposes the design of a substrate interface for developing concurrency libraries in Haskell. This design uses transactional memory as the synchronization primitive, and a special form of continuations for implementing lightweight concurrency. This design simplifies the GHC runtime system; it also improves the safety and customizability of concurrency implementation.

Up to now, we have a prototype implementation with preliminary performance results that look promising. Nevertheless, the design needs to be further validated (and improved as needed) through a full implementation of the existing concurrency features in GHC, and some performance tuning is definitely needed. There is still plenty of work left to be done.

Acknowledgments

We would like to thank Tim Harris, Norman Ramsey and Olin Shivers for their warmhearted participation in discussing the new RTS design. In addition, we thank all the PL club members at the University of Pennsylvania and the anonymous reviewers for their valuable comments. This work is supported by NSF grant CCF-0541040.

References

[1] A. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[2] J. Berthold, A. Al-Zain, and H.-W. Loidl. Adaptive High-Level Scheduling in a Generic Parallel Runtime Environment. In *Symposium on Trends in Functional Programming (TFP)*, New York, USA, April 2007.

[3] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. pages 99–107, May 1996.

[4] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.

[5] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[6] K. Fisher and J. Reppy. Compiler support for lightweight concurrency. Technical memorandum, Bell Labs, Mar. 2002.

[7] M. Fluet, M. Rainey, and J. Reppy. Nested schedulers for heterogeneous parallelism, submitted for publication, 2007.

[8] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007)*, pages 37–44, Jan. 2007.

[9] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[10] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in haskell. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 116–128, New York, NY, USA, 2005. ACM Press.

[11] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, September 2005.

[12] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, June 2005.

[13] S. Jagannathan and J. Philbin. A customizable substrate for concurrent languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 55–67, New York, NY, 1992. ACM Press.

[14] S. Jagannathan and J. Philbin. A foundation for an efficient multi-threaded scheme system. In *Proc. LISP and Functional Programming*, pages 345–357, 1992.

[15] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 274–285, Snowbird, Utah, June 2001. ACM Press.

[16] S. Marlow, S. Peyton Jones, and W. Thaller. Extending the Haskell foreign function interface with concurrency. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 57–68, Snowbird, Utah, USA, September 2004.

[17] J. G. Morrisett and A. Tolmach. Procs and locks: a portable multiprocessing platform for Standard ML of New Jersey. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–207, New York, NY, USA, 1993. ACM Press.

[18] M. Rainey. The Manticore runtime model, Master's paper, Departent of Computer Science, University of Chicago, 2007.

[19] J. Regehr. Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems, Ph.D. thesis, University of Virginia, 2001.

[20] A. Reid. Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable black-holes. volume 1595 of *Lecture Notes in Computer Science*, pages 186–199, 1999.

[21] J. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.

[22] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, January 1997.

[23] The Computer Language Shootout Benchmarks. http://shootout.alioth.debian.org/.

[24] P. Trinder, K. Hammond, J. Mattson, A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*. ACM Press, Philadelphia, May 1996.

[25] M. Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28, Aug. 1980.