Composable Scheduler Activations for Haskell

KC Sivaramakrishnan
Purdue University
chandras@cs.purdue.edu

Tim Harris[1]
Oracle Labs
timothy.l.harris@oracle.com

Simon Marlow[1]
Facebook UK Ltd.
smarlow@fb.com

Simon Peyton Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

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 multicores, 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--` [15] 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.

[1] This work was done at Microsoft Research, Cambridge.

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 4.5). 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 6). 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*

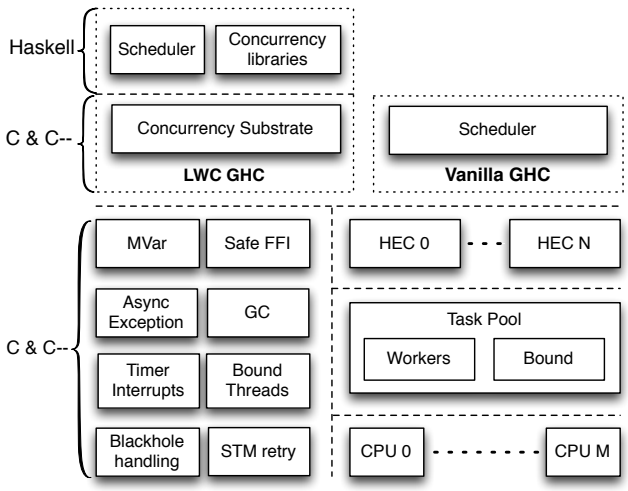

Figure 1: Anatomy of GHC's runtime system and concurrency substrate

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 `SCont` is a heap-allocated object representing the current state of a Haskell computation, and the substrate offers primitives for capturing and transferring control between `SConts` (Section 4.2).

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 4.5.

4. Concurrency Substrate

It is hard to give English-language descriptions of concurrency primitives that are truly precise. We make our descriptions precise

$$x, y \in Variable \quad r, s, \in Name$$

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

Term

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

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

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

Top-level transitions $S; \Theta \stackrel{a}{\Longrightarrow} S'; \Theta'$

$$\frac{H; \Theta \stackrel{a}{\Longrightarrow} H'; \Theta'}{S \parallel H; \Theta \stackrel{a}{\Longrightarrow} S \parallel H'; \Theta'}\ (\textsc{OneHEC})$$

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

$$\frac{M \to N}{\langle s, \mathbb{E}[M], D\rangle; \Theta \implies \langle s, \mathbb{E}[N], D\rangle; \Theta'}\ (\textsc{PureStep})$$

Purely functional transitions $M \to N$

$$\begin{array}{rcll}
\mathsf{return}\ N \gg\!= M & \to & M\ N & (\textsc{Bind}) \\
\mathsf{throw}\ N \gg\!= M & \to & \mathsf{throw}\ N & (\textsc{Throw}) \\
\mathsf{retry} \gg\!= M & \to & \mathsf{retry} & (\textsc{Retry}) \\
\mathsf{catch}\ (\mathsf{return}\ M)\ N & \to & \mathsf{return}\ M & (\textsc{IOCatch}) \\
\mathsf{catch}\ (\mathsf{throw}\ M)\ N & \to & N\ M & (\textsc{IOCatchExp})
\end{array}$$

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 (`Idle`) or a triple $\langle s, M, D\rangle_t$ where s is a unique identifier of the currently executing `SCont`, M is the currently executing term, D represents 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:

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

In a program state $(S; \Theta)$, a `SCont` with identifier s appears *either* as the running `SCont` in a HEC $\langle s, M, D\rangle_t \in S$, *or* as a binding $(s \mapsto (M, D)$ in the heap Θ, but never in both. The distinction has direct operational significance: an `SCont` running in a HEC has part of its state loaded into machine registers, whereas one in the heap is entirely passive. In both cases, however, the term M has type $\mathtt{IO}\ \tau$ 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 `SCont`s 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:

$$S; \Theta \stackrel{a}{\Longrightarrow} S'; \Theta'$$

This says that the program makes a transition from $S; \Theta$ to $S'; \Theta'$, possibly interacting with the underlying RTS through action a. We return to these RTS interactions in Section 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 $\mathbb{E}$ (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 $M \to N$ include β-reduction, arithmetic expressions, case expressions, monadic operations return, bind, throw, catch, and so on according to their standard definitions. Bind operation on the transactional memory primitive retry simply reduces to retry (Figure 3). These primitives represent blocking actions under transactional memory and will be dealt with in Section 5.2.

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 [8] — and schedulers are prime candidates, because they are

```
data STM a
atomically :: STM a -> IO a
retry      :: STM a
catchSTM:: Exception e => STM a -> (e->STM a)-> STM a

data TVar a
instance Monad STM
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Figure 4: API for transactional memory

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

$$\frac{s; M; D; \Theta \overset{*}{\twoheadrightarrow} \mathsf{return}\ N; \Theta'}{\langle s, \mathbb{E}[\mathsf{atomically}\ M], D\rangle;\ \Theta \implies \langle s, \mathbb{E}[\mathsf{return}\ N], D\rangle;\ \Theta'} \text{ (TATOMIC)}$$

$$\frac{s; M; D; \Theta \overset{*}{\twoheadrightarrow} \mathsf{throw}\ N; \Theta'}{\langle s, \mathbb{E}[\mathsf{atomically}\ M], D\rangle;\ \Theta \implies \langle s, \mathbb{E}[\mathsf{throw}\ N], D\rangle;\ \Theta \cup (\Theta' \setminus \Theta)} \text{ (TTHROW)}$$

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

$$\frac{M \to N}{s; \mathbb{P}[M]; D; \Theta\ \twoheadrightarrow\ \mathbb{P}[N]; \Theta} \text{ (TPURESTEP)}$$

$$\frac{s; M; D; \Theta \overset{*}{\twoheadrightarrow} \mathsf{return}\ M'; \Theta'}{s; \mathbb{P}[\mathsf{catchSTM}\ M\ N]; D; \Theta\ \twoheadrightarrow\ \mathbb{P}[\mathsf{return}\ M']; \Theta'} \text{ (TCATCH)}$$

$$\frac{s; M; D; \Theta \overset{*}{\twoheadrightarrow} \mathsf{throw}\ M'; \Theta'}{s; \mathbb{P}[\mathsf{catchSTM}\ M\ N]; D; \Theta \twoheadrightarrow \mathbb{P}[N\ M']; \Theta \cup (\Theta \setminus \Theta')} \text{ (TCEXP)}$$

$$\frac{s; M; D; \Theta \overset{*}{\twoheadrightarrow} \mathsf{retry}; \Theta'}{s; \mathbb{P}[\mathsf{catchSTM}\ M\ N]; D; \Theta\ \twoheadrightarrow\ \mathbb{P}[\mathsf{retry}]; \Theta'} \text{ (TCRETRY)}$$

$$\frac{r\ \ \mathsf{fresh}}{s; \mathbb{P}[\mathsf{newTVar}\ M]; D; \Theta \twoheadrightarrow \mathbb{P}[\mathsf{return}\ r]; \Theta[r \mapsto M]} \text{ (TNEW)}$$

$$s; \mathbb{P}[\mathsf{readTVar}\ r]; D; \Theta \twoheadrightarrow \mathbb{P}[\mathsf{return}\ \Theta(r)]; \Theta \quad \text{(TREAD)}$$

$$s; \mathbb{P}[\mathsf{writeTVar}\ r\ M]; D; \Theta \twoheadrightarrow \mathbb{P}[\mathsf{return}\,()]; \Theta[r \mapsto M] \quad \text{(TWRITE)}$$

Figure 5: Operational semantics for software transactional memory

particularly prone to subtle concurrency bugs. The STM primitives are presented in Figure 4. Unlike Li's design, however, *our transactional memory substrate supports blocking operations*. The ability to perform blocking operations in the scheduler allows us to utilise STM based concurrency libraries such as `TMVar` [8] with minimal refactoring.

Figure 5 presents the semantics of non-blocking STM operations. The semantics of blocking operations is deferred until Section 5.2. A STM transition is of the form:

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

where M is the current monadic term under evaluation, and the heap Θ binds transactional variables `TVar` locations r to their current values. The current `SCont` s and its local state D are read-only, and are not used at all in this section, but will be needed in Section 4.7. The reduction produces a new term M' and a new heap Θ'. Rule TPURESTEP is similar to PURESTEP rule in Figure 3. STM allows creating (TNEW), reading (TREAD), and writing (TWRITE) to transactional variables.

The most important rule is TATOMIC which combines multiple *STM* transitions into a single *program* transition. Thus, other HECs are not allowed to witness to intermediate effects of the transaction. The semantics of exception handling under STM is interesting (rules TCEXP and TTHROW). The effects of the current transaction are undone except for the newly allocated `TVars` 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 multicore context [10].

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

```
switch $ \s -> do
  ...save current SCont "s" somewhere...
  s' <- ...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 SWITCHEXP) begin by atomically evaluating the body of switch M applied to the current `SCont` s. If the resultant `SCont` is the same as the

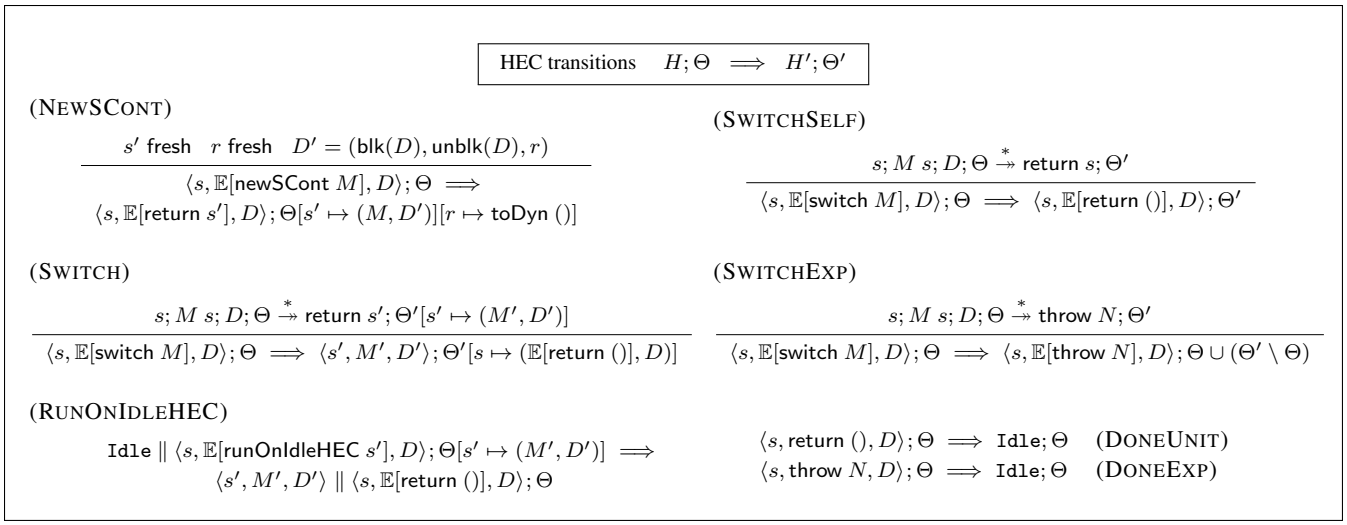

Figure 6: Operational semantics for `SCont` manipulation

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:

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

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 following primitive to start evaluation of an `SCont` on an idle HEC.

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

Rule RUNONIDLEHEC illustrates the behaviour of parallely 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 `newSCont`, 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 activations*[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:

```
blockAct   :: SCont -> STM ()
unblockAct :: SCont -> STM SCont
```

- 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.

```
yield :: IO ()
yield = switch (\s -> unblockAct s >> blockAct s)
```

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:

```
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 (QS 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 `bq` of 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`. 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.

In our formalisation, we represent local state D as a tuple with three terms and a name (M, N, r) (Figure 2), where M, N and r are block activation, unblock activation, and a `TVar` representing auxiliary storage, respectively. The precise semantics of activations and stack-local state manipulation is given in Figure 7. For perspicuity, we define accessor functions as shown below.

$$\mathsf{blk}(M, \_, \_) = M \quad \mathsf{unblk}(\_, M, \_) = M \quad \mathsf{aux}(\_, \_, M) = M$$

The auxiliary state is modelled as a `TVar` in the local state, that is initialised to a dynamic unit value `toDyn ()` when an new `SCont` is created (rule NEWSCONT in Figure 6). The rules SETAUXSELF and SETAUXOTHER update the aux state of a `SCont` by writing to the `TVar`. There are two cases, depending on whether the `SCont` is running in the current HEC, or is passive in the heap. The aux-state is typically used to store scheduler accounting information, and is most likely to be updated in the activations, being invoked by some other `SCont` or the RTS. This is the reason why we model aux-state as a `TVar` and allow it to be modified by some other `SCont`.

If the target of the `setAux` is running in another HEC, no rule applies, and we raise a runtime exception. This is reasonable: one HEC should not be poking into another running HEC's state. The rules for `getAux` also have two cases.

An `SCont`'s activations can be invoked using the `blockAct` and `unblockAct` primitives. Invoking an `SCont`'s own activation is straight-forward; the activation is fetched from the local state and applied to the current `SCont` (rules INVOKEBLOCKACTSELF and INVOKEBLOCKACTOTHER). We do allow activations of an `SCont` other than the current `SCont` to be invoked (rule INVOKEBLOCKACTOTHER and INOKEUNBLOCKACTOTHER). Notice that in order to invoke the activations of other `SConts`, the `SCont` must be passive on the heap, and currently not running.

A newly created `SCont`, by default, is attached to the same user-level scheduler as the parent `SCont`. This is achieved by inheriting the activations of the parent thread (rule NEWSCONT in Figure 6). We allow an `SCont` to modify its *own* activations, and potentially migrate to another user-level scheduler. In addition, updating own activations allows initial thread evaluating the `main` IO computation to initialise its activations, and participate in user-level scheduling.

In the common use case, once an `SCont`'s activations are initialised, we don't expect it to change. Hence, we do not store the activations in a `TVar`, but rather directly in the underlying TSO object field. The avoids the overheads of transactional access of activations.

4.8 Bound `SConts`

In GHC, *bound threads* provide a one-to-one mapping between a Haskell thread and an OS thread. Bound threads are needed because some foreign libraries must always run on the same OS thread; this issue is discussed at length in [13]. It is straightforward to extend our API to allow a *bound* `SCont` to be created:

```
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

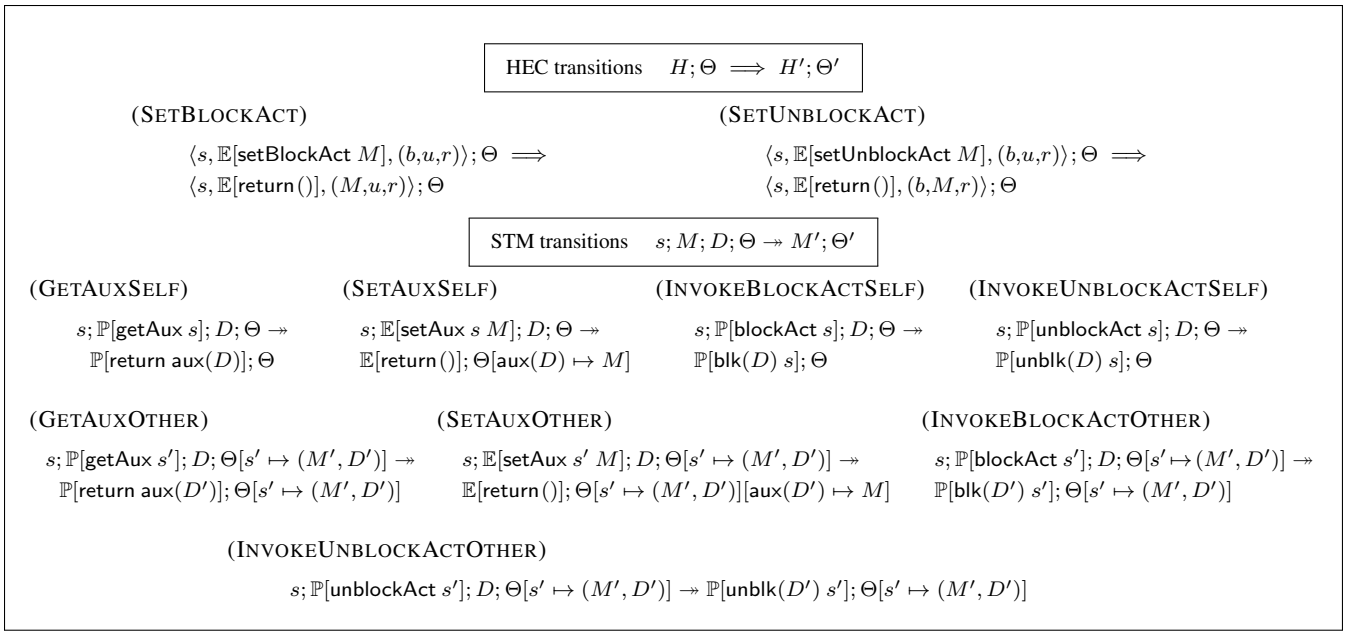

Figure 7: Operational semantics for activations and auxiliary state

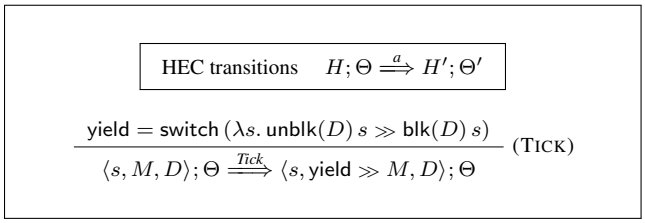

Figure 8: Handling timer interrupts

scheduler needs to be scheduled. The semantics of handling timer interrupts is shown in Figure 8

The *Tick* label on the transition arrow indicates an interaction with the RTS; we call such a label an *RTS-interaction*. In this case the RTS-interaction *Tick* indicates that the RTS wants to signal a timer tick[3]. The transition here injects `yield` into the instruction stream of the thread running on this HEC, at a GC safe point, where `yield` behaves just like the definition in Section 4.5.

5.2 STM blocking operations

GHC today supports blocking operations under STM. When the programmer invokes `retry` inside a transaction, the RTS blocks the thread until another thread writes to any of the `TVars` read by the transaction; then the thread is re-awoken, and retries the transaction [8]. This is entirely transparent to the programmer, and our goal is to take advantage of the existing RTS in our new substrate.

The big question is this: how should the RTS block the thread? By calling its `block` activation, of course! Figure 9 gives the semantics for `retry`. Rule TRETRYATOMIC is similar to TTHROW in Figure 5. It runs the transaction body M; if the latter terminates with `retry`, it abandons the effects embodied in Θ', reverting to Θ. But, unlike TTHROW it then uses an auxiliary rule $\overset{blk}{\hookrightarrow}$, defined in Figure 10, to block the thread.

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

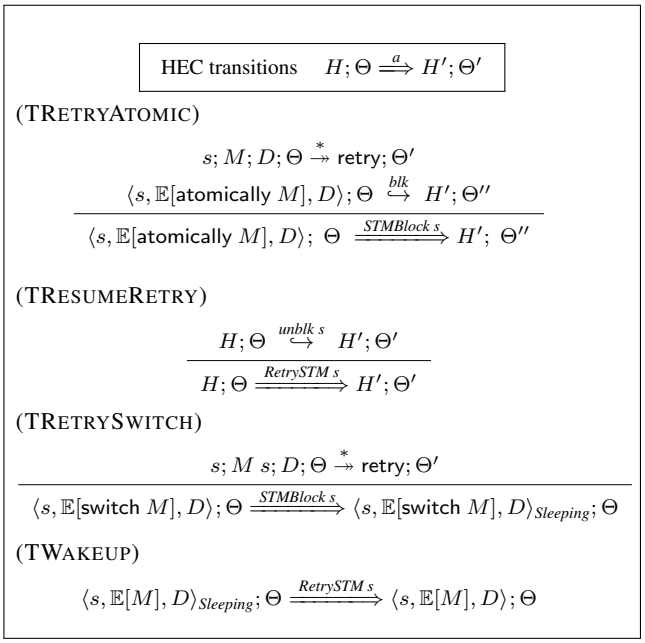

Figure 9: STM Retry

Rule UPBLOCK in Figure 10 stashes s (the thread to be blocked) in the heap Θ, instantiates an ephemeral thread that fetches the `block` activation b from s's state D, and switches to the thread returned by the `block` activation. s' is made the running `SCont` on this HEC. It is necessary that the block operation be performed on a new `SCont`. Recall that an `SCont` can either be running in a HEC, or be passive on the heap, but not both. Since $\overset{blk}{\hookrightarrow}$ saves the state of `SCont` s to the heap, the block activation is invoked on an ephemeral thread.

The transition in TRETRYATOMIC is labelled with the RTS interaction *STMBlock s*, indicating that the RTS now assumes respon-

Block upcall instantiation $H;\Theta \overset{blk}{\hookrightarrow} H';\Theta'$

(UPBLOCK)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{blk}(D), \mathsf{unblk}(D), r) \quad M' = \mathsf{switch}\ (\lambda x.\, \mathsf{blk}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M,D)][r \mapsto \mathsf{toDyn}\ ()]}{\langle s, M, D\rangle;\Theta \overset{blk}{\hookrightarrow} \langle s', M', D'\rangle;\Theta'}$$

Unblock upcall instantiation $H;\Theta \overset{unblk\ s}{\hookrightarrow} H';\Theta'$

(UPUNBLOCKIDLE)

$$\frac{s' \text{ fresh} \quad r \text{ fresh} \quad D' = (\mathsf{blk}(D), \mathsf{unblk}(D), r) \quad M' = \mathsf{atomically}\ (\mathsf{unblk}(D)\ s) \quad \Theta' = \Theta[s \mapsto (M,D)][r \mapsto \mathsf{toDyn}\ ()]}{\mathtt{Idle};\Theta[s \mapsto (M,D)] \overset{unblk\ s}{\hookrightarrow} \langle s', M', D'\rangle;\Theta'}$$

(UPUNBLOCKRUNNING)

$$\frac{M'' = \mathsf{atomically}\ (\mathsf{unblk}(D)\ s) \gg M'}{\langle s', M', D'\rangle;\Theta[s \mapsto (M,D)] \overset{unblk\ s}{\hookrightarrow} \langle s', M'', D'\rangle;\Theta[s \mapsto (M,D)]}$$

Figure 10: Instantiating upcalls

sibility for s. Some time later, the RTS will see that some thread has written to one of the `TVars` read by s's transaction, so it will signal an *RetrySTM s* interaction (rule TRESUMERETRY). Again, we use an auxiliary transition $\overset{unblk\ s}{\hookrightarrow}$ to unblock the thread (Figure 10).

Unlike $\overset{blk}{\hookrightarrow}$ transition, unblocking a thread has nothing to do with the computation currently running on any HEC. If we find an idle HEC (rule UPUNBLOCKIDLE), we instantiate a new ephemeral `SCont` s' to unblock `SCont` s. The actual unblock operation is achieved by fetching `SCont` s's `unblock` activation, applying it to s and atomically performing the resultant STM computation. If we do not find any idle HECs (rule UPUNBLOCKRUNNING), we pick one of the running HECs, prepare it such that it first unblocks the `SCont` s before resuming the original computation.

RTS always runs the block activation on the same HEC as the thread being blocked. Whenever possible, RTS ensures that unblock activation is run by the same HEC that performed the corresponding block activation. In addition, rather than creating a new ephemeral `SCont` for block and unblock operations, we utilise a dedicated *upcall thread* (one per HEC) for performing the unblock operation.

5.3 HEC blocking

When the RTS blocks a thread, it uses the `block` activation of the thread to get another thread to run; see rule UPBLOCK. But what if there *is* no other thread to run? Perhaps the run queue managed by the scheduler on this HEC has run dry, and perhaps even work stealing has failed, because the only other runnable threads are already running on other HECs. Then then there is nothing for this HEC to do but to go to sleep until something changes.

Recalling that the `block` activation is itself a STM transaction, this situation manifests itself by the `block` activation calling `retry`. This motivates rule TRETRYSWITCH: if a `switch` transaction blocks, we put the whole HEC to sleep. Then, dual to TRESUMERETRY, rule TWAKEUP wakes up the HEC when the RTS sees that the transaction may now be able to make progress.

HEC transitions $H;\Theta \overset{a}{\Longrightarrow} H';\Theta'$

$$\frac{}{\langle s, \mathbb{E}[\mathsf{outcall}\ r], D\rangle;\Theta \overset{OC\ s}{\Longrightarrow} \langle s, \mathbb{E}[\mathsf{outcall}\ r], D\rangle_{Outcall};\Theta} \quad \text{(OCBLOCK)}$$

$$\frac{}{\langle s, \mathbb{E}[\mathsf{outcall}\ r], D\rangle_{Outcall};\Theta \overset{OCRet\ s\ M}{\Longrightarrow} \langle s, \mathbb{E}[M], D\rangle;\Theta} \quad \text{(OCRETFAST)}$$

$$\frac{\langle s, M, D\rangle;\Theta \overset{blk}{\hookrightarrow} H';\Theta'}{\langle s, M, D\rangle_{Outcall};\Theta \overset{OCSteal\ s}{\Longrightarrow} H';\Theta'} \quad \text{(OCSTEAL)}$$

$$\frac{H;\Theta[s \mapsto (\mathbb{E}[M], D)] \overset{unblk\ s}{\hookrightarrow} H';\Theta'}{H;\Theta[s \mapsto (\mathbb{E}[\mathsf{outcall}\ r], D)] \overset{OCRet\ s\ M}{\Longrightarrow} H';\Theta'} \quad \text{(OCRETSLOW)}$$

Figure 11: Safe foreign call transitions

5.4 Timer interrupts and transactions

What if a timer interrupt occurs during a transaction? The (TICK) rule of Section 5.1 is restricted to *HEC transitions*, and says nothing about *STM transitions*. One possibility (Plan A) is that transactions should not be interrupted, and ticks should only be delivered at the end. This is faithful to the semantics expressed by the rule, but it does mean that a rogue transaction could completely monopolise a HEC.

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

Our implementation behaves like Plan B, but gives better progress guarantees, while respecting the same semantics. Rather than rolling the transaction back, the RTS suspends the transaction mid-flight. None of its effects are visible to other 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.

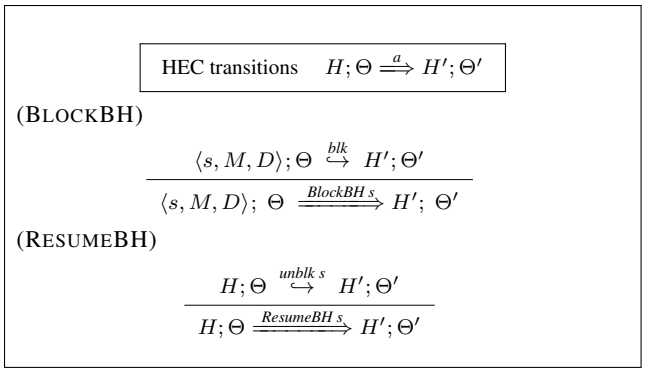

Figure 12: Black holes

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 OCBLOCK illustrates that the HEC performing the foreign call moves into the $Outcall$ state, where it is no longer runnable. In the fast path (rule OCRETFAST), the foreign call returns immediately with the result M, and the HEC resumes execution with the result plugged into the context.

In the slow path, the RTS may decide pay the cost of task switching and resume the scheduler (rule OCSTEAL). The scheduler is resumed using the block upcall. Once the foreign call eventually returns, the `SCont` s blocked on the foreign call can be resumed. Since we have already resumed the scheduler, the correct behaviour is to prepare the `SCont` s with the result and add it to its user-level scheduler. Rule OCRETSLOW achieves this through unblock upcall.

5.6 Black holes

In a concurrent Haskell program, a thread A may attempt to evaluate a thunk `x` that is already being evaluated by another thread B. To avoid duplicate evaluation the RTS (in intimate cooperation with the compiler) arranges for B to *blackhole* the thunk when it starts to evaluate `x`. Then, when A attempts to evaluate `x`, it finds a black hole, so the RTS enqueues A to await its completion. When B finishes evaluating `x` it updates the black hole with its value, and makes any queued threads runnable. This mechanism, and its implementation on a multicore, is described in detail in earlier work [7].

Clearly this is another place where the RTS may initiate blocking. 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 `SCont` s. Later, when the evaluation of the thunk is complete, the RTS initiate an action *ResumeBH s*, which returns ownership to s's scheduler.

But these rules only apply to *HEC transitions*, outside transactions. What if a black hole is encountered during an STM transaction? We addressed this same question in the context of timer interrupts, in Section 5.4, and we adopt the same solution. The RTS behaves as if the black-hole suspension and resumption occurred just before the transaction, but the implementation actually arranges to resume the transaction from where it left off.

Just as in Section 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 $\stackrel{blk}{\hookrightarrow}$) is a `switch` transaction; and that is very likely to encounter the very same thunk. Moreover, it is just possible that the thunk is under evaluation by an `SCont` in this very scheduler's run-queue, so the black hole is preventing us from scheduling the very `SCont` that is evaluating it. Deadlock beckons!

In the case of timer interrupts we solved the problem by switching them off in `switch` transactions, and it turns out that we can effectively do the same for thunks. Since we cannot sensibly suspend the `switch` transaction, we must find a way for it to make progress. Fortunately, GHC's RTS allows us to *steal* the thunk from the `SCont` 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 `SCont` s needs to block on or unblock from an MVar, the RTS invokes the $\stackrel{blk}{\hookrightarrow}$ or $\stackrel{unblk\ s}{\hookrightarrow}$ 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 `SCont`, or an `SCont` 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 `SCont` suspended on a user-level scheduler may become unreachable if the scheduler data structure holding it becomes unreachable. While an `SCont` 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 `SCont`! In this case, immediately after garbage collection, our implementation logs an error message to the standard error stream along with the unreachable `SCont` identifier.

6. Developing concurrency libraries

We have so far been dealing with the concurrency substrate primitives 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 `SConts`.

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

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

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

```
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.

```
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:

```
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
    let (_::Int, t::TVar Int) = fromJust $
        fromDynamic dyn
    nextHEC <- readTVar t
    writeTVar t $ (nextHEC + 1) `mod` numHECs
    setAux newSC $ toDyn (nextHEC, t)
    return s
  -- Add new thread to scheduler
  atomically $ 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 only through the activation interface. As a result, aside from the auxiliary state manipulation, the rest of the code pretty much can stay the same for any user-level `forkIO` 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:

```
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:

```
takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  h <- atomically $ newPVar undefined
  switch $ \s -> do
    st <- readTVar ref
    case st of
      Empty ts -> do
        writeTVar ref $ Empty $ enqueue ts (h,s)
        blockAct s
      Full x ts -> do
        writePVar h x
        case deque ts of
          (_, Nothing) -> do
            writeTVar ref $ Empty emptyQueue
          (ts', Just (x', s')) -> do
            writeTVar ref $ Full x' ts'
            unblockAct s'
        return s
  atomically $ readPVar 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.

7. 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 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. `k-nucleotide` and `mandelbrot` are computation intensive, while `chameneos` and `primes-sieve` are communication intensive and are specifically intended to test the overheads of thread synchronisation.

The LWC version of the benchmarks utilised the scheduler and MVar implementation described in Section 6. For comparison, the benchmark programs were also implemented using `Control.Concurrent` on a vanilla GHC implementation. Experiments were performed on a 48-core AMD Opteron server, and the GHC version was 7.7.20130523.

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 LWC 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-8X 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.

8. 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 unblocking sleeping HECs. This design has potential for bugs due to forgotten wake up messages. Our HEC blocking mechanism directly utilises STM blocking capability pro-

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

Benchmark	Baseline (Secs)	Vanilla (Secs)		LWC (Secs)	
		1 HEC	Fastest (# HECs)	1 HEC	Fastest (# HECs)
`k-nucleotide`	10.60	10.62	4.82 (8)	10.61	4.83 (8)
`mandelbrot`	85.30	90.83	3.21 (48)	87.06	2.19 (48)
`chameneos`	4.62	5.71	5.71 (1)	18.25	12.35 (2)
`primes-sieve`	32.52	36.33	36.33 (1)	223	13.7 (48)

Figure 13: Benchmark results.

vided by the runtime system, and by construction eliminates the possibility of forgotten wake up messages.

Scheduler activations [2] have successfully been demonstrated to interface kernel with the user-level process scheduler [3, 20]. Similar to scheduler activations, Psyche [14] allows user-level threads to install event handlers for scheduler interrupts and implement the scheduling logic in user-space. While we borrow the idea of upcalls for managing the scheduler from previous work, our system is unique in that we utilise activations to interface the GHC runtime system with Haskell scheduler. Moreover, our activations being STM computations enable composability of the activations with other transactions in Haskell. This enables us to implement scheduler agnostic concurrency libraries and thread schedulers only utilising the activations.

With respect to implementing traditionally low-level support to be implemented in the high-level language, Harris et al. [9] proposed extensible virtual machines in order to allow thread scheduling, runtime compilation and object placement within the heap to be implemented directly in the source language. Similarly, meta-circular Java virtual implementations such as JikesRVM [5] and Maxine [21] allow different components of the virtual machines, traditionally implemented in low-level code, to be implemented in Java.

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

[1] The Computer Language Benchmarks Game, 2013. URL `http://benchmarksgame.alioth.debian.org/`.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, Feb. 1992.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. SOSP '09, pages 29–44, 2009.

[4] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. DAMP '07, pages 10–18, 2007.

[5] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. VEE '09, pages 81–90, New York, NY, USA, 2009. ACM. doi: 10.1145/1508293.1508305. URL `http://doi.acm.org/10.1145/1508293.1508305`.

[6] K. W. Graunke. Extensible scheduling in a haskell-based operating system. Master's thesis, Portland State University, 2010.

[7] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *ACM Workshop on Haskell*, Tallin, Estonia, 2005. ACM.

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. PPoPP '05, pages 48–60, 2005.

[9] T. L. Harris. *Extensible Virtual Machines*. PhD thesis, University of Cambridge Computer Laboratory, 2001.

[10] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for ghc. Haskell '07, pages 107–118, 2007.

[11] B. Lippmeier, M. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. Haskell '12, pages 25–36, 2012.

[12] S. Marlow, S. P. Jones, A. Moran, and J. Reppy. Asynchronous exceptions in haskell. PLDI '01, pages 274–285, 2001.

[13] S. Marlow, S. P. Jones, and W. Thaller. Extending the haskell foreign function interface with concurrency. Haskell '04, pages 22–32, 2004.

[14] B. D. Marsh, M. L. Scott, T. J. Leblanc, and E. P. Markatos. First-class user-level threads. OSDI, pages 110–121, 1991.

[15] S. L. Peyton Jones, N. Ramsey, and F. Reig. C–: A portable assembly language that supports garbage collection. PPDP '99, pages 1–28, 1999.

[16] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

[17] J. Reppy, C. V. Russo, and Y. Xiao. Parallel concurrent ml. ICFP '09, pages 257–268, 2009.

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

[19] M. Wand. Continuation-based multiprocessing. LFP '80, pages 19–28, 1980.

[20] N. J. Williams. An implementation of scheduler activations on the netbsd operating system. Usenix ATC '02, pages 99–108, 2002.

[21] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013.

[22] L. Ziarek, K. Sivaramakrishnan, and S. Jagannathan. Composable asynchronous events. PLDI '11, pages 628–639, 2011.