Extending the Haskell Foreign Function Interface with Concurrency

Simon Marlow and Simon Peyton Jones
Microsoft Research Ltd., Cambridge, U.K.
{simonmar,simonpj}@microsoft.com

Wolfgang Thaller
wolfgang.thaller@gmx.net

Abstract

1 Introduction

Two of the most longest-standing and widely-used extensions to Haskell 98 are Concurrent Haskell [10] and the Haskell Foreign Function Interface [7]. These two features were specified independently, but their combination is quite tricky, especially when the FFI is used to interact with multi-threaded foreign programs.

The core question is this: what is the relationship between the native threads supported by the operating system (the *OS threads*), and the lightweight threads offered by Concurrent Haskell (the *Haskell threads*)? From the programmer's point of view, the simplest solution is to require that the two are in one-to-one correspondence. However, part of the design philosophy of Concurrent Haskell is to support extremely numerous, lightweight threads, which (in today's technology at any rate) is incompatible with a one-to-one mapping to OS threads. Instead, in the absence of FFI issues, the natural implementation for Concurrent Haskell is to multiplex all the Haskell threads onto a single OS thread.

In this paper we show how to combine the clean semantics of the one-to-one programming model with the performance of the multiplexed implementation. Specifically we make the following contributions:

- First we tease out a number of non-obvious ways in which this basic multiplexed model conflicts with the simple one-to-one programming model, and sketch how the multiplexed model can be elaborated to cope (Section 3).
- We propose a modest extension to the language, namely *bound threads*. The key idea is to distinguish the Haskell threads which must be in one-to-one correspondence with an OS thread for the purpose of making foreign calls, from those that don't need to be. This gives the programmer the benefits of one-to-one correspondence when necessary, while still allowing the implementation to provide efficient lightweight threads.

  We express the design as a concrete set of proposed extensions or clarifications to the existing designs for the Haskell FFI and Concurrent Haskell (Section 4).
- We give a precise specification of the extension in terms of an operational semantics (Section 5).
- We sketch three possible implementations, one of which has been implemented in a production compiler, the Glasgow Haskell Compiler[1] (Section 6).

2 Background

Concurrency and the foreign-function interface are two of the most long-standing and widely used extensions to Haskell 98. In this section we briefly introduce both features, establish our context and terminology.

2.1 The Foreign Function Interface

The Foreign Function Interface extends Haskell 98 with the ability to call, and be called by, external programs written in some other language (usually C). A `foreign import` declaration declares a foreign function that may be called from Haskell, and gives its Haskell type. For example:

```
foreign import ccall safe "malloc"
  malloc :: CSize -> IO (Ptr ())
```

declares that the external function `malloc` may be invoked from Haskell using the `ccall` calling convention. It takes one argument of type `CSize` (a type which is equivalent to C's `size_t` type), and returns a value of type `Ptr ()` (a pointer type parameterised with `()`, which normally indicates a pointer to an untyped value).

Similarly, a `foreign export` declaration identifies a particular Haskell function as externally-callable, and causes the generation of some impedance matching code to provide the Haskell function with the foreign calling convention. For example:

```
foreign export ccall "plus"
   addInt :: CInt -> CInt -> CInt
```

declares the Haskell function `addInt` (which should be in scope at this point) as an externally-callable function `plus` with the `ccall` calling convention.

A `foreign import` declaration may be annotated with the modifiers "`safe`" or "`unsafe`", where `safe` is the default and may be

Submitted to The Haskell Workshop, 2004

omitted. A foreign function declared as `safe` may indirectly invoke Haskell functions, whereas an `unsafe` one may not (that is, doing so results in undefined behaviour). The distinction is motivated purely by performance considerations: an unsafe call is likely to be faster than a safe call, because it does not need to save the state of the Haskell system in such a way that re-entry, and hence garbage collection, can be performed safely—for example, saving all temporary values on the stack such that the garbage collector can find them. It is intended that, in a compiled implementation, an unsafe foreign call can be implemented as a simple inline function call.

We use Haskell-centric terminology: the act of a Haskell program calling a foreign function (via `foreign import`) is termed a *foreign out-call* or sometimes just *foreign call*; and the act of a foreign program calling a Haskell function (via `foreign export`) is *foreign in-call*.

2.2 Concurrent Haskell

Concurrent Haskell is an extension to Haskell that offers lightweight concurrent threads that can perform input/output[10]. It aims to increase *expressiveness*, rather than *performance*; A Concurrent Haskell program is typically executed on a uni-processor, and may have dozens or hundreds of threads, most of which are blocked. Non-determinism is part of the specification; for example, two threads may draw to the same window and the results are, by design, dependent on the order in which they execute. Concurrent Haskell provides a mechanism, called `MVars`, through which threads may synchronise and cooperate safely, but we will not need to discuss `MVars` in this paper.

For the purposes of this paper, the only Concurrent Haskell facilities that we need to consider are the following:

```
data ThreadId -- abstract, instance of Eq, Ord
myThreadId :: IO ThreadId
forkIO :: IO () -> IO ThreadId
```

The abstract type `ThreadId` type represents the identity of a Haskell thread. The function `myThreadId` provides a way to obtain the `ThreadId` of the current thread. New threads are created using `forkIO`, which takes an `IO` computation to perform in the new thread, and returns the `ThreadId` of the new thread. The newly-created thread runs concurrently with the other Haskell threads in the system.

Details on the rest of the operations provided by Concurrent Haskell can be found in [10] and the documentation for the `Control.Concurrent` library distributed with GHC[1].

2.3 Haskell threads and OS threads

Every operating system natively supports some notion of "threads", so it is natural to ask how these threads map onto Concurrent Haskell's notion of a "thread". Furthermore, this mapping is intimately tied up with the FFI/concurrency interaction that we explore in this paper.

To avoid confusion in the following discussion, we define the following terminology:

- A *Haskell thread* is a thread in a Concurrent Haskell program.
- An operating system thread (or *OS thread*) is a thread managed by the operating system.

There are three approaches to mapping a system of lightweight threads, such as Haskell threads, onto the underlying OS threads:

One-to-one. Each Haskell thread is executed by a dedicated OS thread. This system is simple but expensive: Haskell threads are supposed to be lightweight, with hundreds or thousands of threads being entirely reasonable, but most operating systems struggle to support so many OS threads. Furthermore, any interaction between Haskell threads (using `MVars`) must use expensive OS-thread facilities for synchronisation.

Multiplexed. All Haskell threads are multiplexed, by the Haskell runtime system, onto a single OS thread, the *Haskell execution thread*. Context switching between Haskell threads occurs only at *yield points*, which the compiler must inject. This approach allows extremely lightweight threads with small stacks. Interaction between Haskell threads requires no OS interaction because it all takes place within a single OS thread.

Hybrid. Models combining the benefits of the previous two are possible. For example, one might have a pool of OS "worker threads", onto which the Haskell threads are multiplexed in some way, and we will explore some such models in what follows.

All of these are reasonable implementations of Concurrent Haskell, each with different tradeoffs between performance and implementation complexity. We do not want to inadvertently rule any of these out by overspecifying the FFI extensions for concurrency.

Note that our focus is on *expressiveness*, and not on *increasing performance through parallelism*. In fact, all existing implementations of Concurrent Haskell serialise the Haskell threads, even on a multiprocessor. Whether a Haskell implementation can efficiently take advantage of a multiprocessor is an open research question, which we discuss further in Section 7.

3 The problem we are trying to solve

Concurrent Haskell and the Haskell FFI were developed independently, but they interact in subtle and sometimes unexpected ways. That interaction is the problem we are trying to solve.

Our design principle is this: *the system should behave as if it was implemented with one OS thread implementing each Haskell thread.* This behaviour is simple to explain, and avoids having to expose two "layers" of threads to the programmer.

However, if implemented naively, the one-to-one model is expensive to implement. The multiplexed model is much cheaper and, where no foreign calls (out or in) are involved, the one-to-one model and the multiplexed model cannot be distinguished by the Haskell program. When foreign interaction enters the picture, matters become more complicated. In the rest of this section we identify several implicit consequences of our design principle, and discuss how the multiplexed implementation technique can accommodate them.

3.1 Foreign calls that block

Some foreign calls, such as the C function `read()`, may *block* awaiting some event, or may simply take a long time to complete. In the absence of concurrency, the Haskell program making the call must also block or take a long time, but not so for Concurrent Haskell. Indeed, our design principle requires the opposite:

Requirement 1: a `safe` foreign call that blocks should block only the Haskell thread making the call. Other Haskell threads should proceed unimpeded.

Notice that we only require that a `safe` foreign call be non-blocking to the other Haskell threads. It would be difficult to make a high-performance `unsafe` foreign call non-blocking, because that would force the implementation to perform the same state-saving as for a `safe` call, since the Haskell system must continue running during the call.

Requirement 1 seems obvious enough, but the Haskell FFI specification is silent on this point, and indeed until recently GHC did not satisfy the requirement. This caused confusion to Haskell programmers, who were surprised when a foreign call blocked their entire Concurrent Haskell program.

Requirement 1 might seem to completely rule out the multiplexed implementation, because if the Haskell execution thread blocks, then execution of Haskell threads will halt. However a variants of the multiplexed model solves the problem:

- At a foreign call, arrange that the foreign function is executed by some other OS thread (freshly spawned, or drawn from a pool), while execution of other Haskell threads is continued by the single Haskell execution thread. This approach pays the cost of a OS thread switch at every (safe) foreign call.

A hybrid model can also be designed to satisfy this requirement:

- Have a pool of OS threads, each of which *can* play the role of the Haskell execution thread, but only one at a time *does*. At a safe foreign call, the Haskell execution thread leaves the Haskell world to execute the foreign call, allowing one (and only one) member of the pool to become the new Haskell execution thread. No OS thread switch is required on a call, but on the return some inter-OS-thread communication is required to obtain permission to become the Haskell execution thread again.

3.2 Fixing the OS thread for a foreign call

Some C libraries that one might wish to call from Haskell have an awkward property: *it matters which calls to the library are made from which OS thread.* For example, many OpenGL functions have an implicit "rendering context" parameter, which the library stores in OS-thread-local state. The (perfectly reasonable) idea is that OpenGL can be used from multiple threads, for example to render into independent windows simultaneously.

This in turn means that to use the OpenGL library from Concurrent Haskell, the FFI must satisfy:

Requirement 2: it must be possible for a programmer to specify that a related group of foreign calls are all made by the same OS thread.

Notice that there is no constraint on which OS thread executes any particular Haskell thread – we need only control which OS thread executes the foreign calls.

Requirement 2 is automatically satisfied by the one-to-one execution model, provided we are willing to say that the "related" calls are all carried out by a single Haskell thread. The multiplexed model (basic version) also automatically satisfies Requirement 2, because all foreign calls are executed by a single OS thread, but only at the cost of violating Requirement 1. Alas, satisfying Requirement 1 using the variant described in Section 3.1, seems to be incompatible with Requirement 2, because this variant deliberately use a pool of interchangeable OS threads. The hybrid model suffers from the same problem.

We are forced, therefore, to propose a small extension to Concurrent Haskell, in which we divide the Haskell threads into two groups:

- A *bound thread* has a fixed associated OS thread for making FFI calls.
- An *unbound thread* has no associated OS thread: FFI calls from this thread may be made in any OS thread.

The idea is that each bound Haskell thread has a dedicated associated OS thread. It is guaranteed that any FFI calls made by a bound Haskell thread are made by its associated OS thread, although pure-Haskell execution can, of course, be carried out by any OS thread. A group of foreign calls can thus be guaranteed to be carried out by the same OS thread if they are all performed in a single bound Haskell thread.

We do not specify that all Haskell threads are bound, because doing so would specify that Haskell threads and OS threads are in one-to-one correspondence, which leaves the one-to-one implementation model as the only contender.

Can several Haskell threads be bound to the same OS thread? No: this would prevent the one-to-one implementation model and cause difficulties for the others. For each OS thread, there is at most a single bound Haskell thread.

3.3 Multi-threaded clients

Suppose a C program wants is using a library written in Haskell, and it invokes a Haskell function (via `foreign export`). This Haskell function forks a Haskell thread, and then returns to the C program. Should the spawned Haskell thread continue to run? According to our design principle, it certainly should – as far as the programmer is concerned, there is not much difference between forking a Haskell thread and forking an OS thread.

Requirement 3a: Haskell threads spawned by an foreign in-call continue to run after the in-call returns.

A closely related issue is this. Suppose the C program using the Haskell library itself makes use of multiple OS threads. Then our design principle implies that if one invocation runs Haskell code that blocks (on an `MVar`, say, or in another foreign call) that should not impede the progress of the other call:

Requirement 3b: multiple OS threads may concurrently invoke multiple Haskell functions (via `foreign export`), and these invocations should run concurrently.

To support this behaviour in the multiplexed model is not difficult, but requires some specific mechanism. In both cases, the current Haskell execution OS thread must pay attention to the possibility of another OS thread wanting to make an in-call, lest the latter wait indefinitely while the former chunters away. In fact, the same mechanism is necessary to respond to an OS thread returning to Haskell from a safe foreign out-call.

3.4 Callbacks

A common idiom in GUI libraries is for the application to make a call to the event loop in the library, which in turn makes calls back into the application in the form of callbacks. Callbacks are registered prior to invoking the event loop.

Consider how this works for a Haskell application calling an external GUI library. The callbacks will be `foreign-export`-ed Haskell functions, so the event loop (in C) will call the callback (in Haskell), which may in turn make a foreign call to a GUI function (in C again). It is essential that this latter call is made using the OS thread as runs the event loop, since the two share thread-local state. Hence:

Requirement 4: it must be possible to ensure that a foreign out-call from Haskell is made by the same OS thread that made the foreign in-call.

With the notion of bound threads in hand, this is not hard to achieve. We simply specify that a foreign in-call creates a bound thread, associated with the OS thread that performed the in-call. Any foreign out-calls made by that (bound) Haskell thread will therefore be executed by the invoking OS thread.

Indeed, this is the *only* primitive mechanism for creating a bound thread:

- An unbound Haskell thread is created using Concurrent Haskell's existing `forkIO` combinator.
- A bound thread is created by a foreign invocation.

We provide a `forkOS` combinator, which allows a Haskell thread (rather than a foreign invocation) to create a new bound thread, but it works by making a foreign call with invokes a callback (see Section 4.2.1).

3.5 Summary

This concludes our description of the problems we address, and of the core of our design. There is no new syntax, and only an implicit distinction between two types of Haskell threads, depending on the way in which the thread was created. The next section describes the language extension in detail, including the small number of combinators that we provide as library functions to allow programmers to work with bound threads.

4 The Concurrent Haskell Foreign Function Interface

Thus motivated, we now summarise our proposed changes to the existing Concurrent Haskell design, and the Haskell FFI specification.

4.1 Specific extensions

We propose the following specific additions:

Bound threads. There are two types of Haskell threads, *bound* and *unbound*. A bound thread is permanently associated with a particular OS thread, and it is guaranteed that all foreign functions invoked from that bound thread will be run in the associated OS thread. In all other ways, bound and unbound threads behave identically.

An OS thread can be associated with at most one Haskell thread.

The new function `isCurrentTheadBound` provides a way for the Haskell programmer to find out whether the current thread is bound or not:

```
isCurrentThreadBound :: IO Bool
```

We define `isCurrentThreadBound` to always return `True` when the current Haskell thread is a bound thread. It may also return `True` when the current Haskell thread is indistinguishable from a bound thread by both Haskell code and foreign code called by it.

Therefore, an implementation using the one-to-one threading model (see Section 6.1) may return `True` for all threads, even for Haskell threads created using `forkIO`, because every Haskell thread has its associated OS thread and can safely access thread-local state.

Foreign import. When a Haskell thread invokes a `foreign import` annotated with `safe`, other Haskell threads in the program will continue to run unimpeded. This is not necessarily true if a Haskell thread invokes a `foreign import` annotated with `unsafe`.

Notice that `unsafe` calls are not *required* to block Haskell threads if the foreign call blocks; instead the behaviour is unspecified. In particular, it is legitimate for a simple, low-performance implementation to implement `unsafe` calls as `safe` calls.

Foreign export. Invoking a function declared with `foreign export` creates a new Haskell thread which is bound to the OS thread making the call.

The main thread. In a complete, standalone Haskell program, the system should run `Main.main` in a bound Haskell thread, whose associated OS thread is the main OS thread of the program. It is as if the program contained the declaration

```
foreign export ccall "haskellMain"
   Main.main :: IO ()
```

and the Haskell program was started from C by invoking `haskellMain()`.

4.2 Derived combinators

Given the basic functionality outlined above, we can define some useful combinators. These are provided to the programmer via the `Control.Concurrent` library.

4.2.1 `forkOS`

The `forkOS` function has the same type as `forkIO`:

```
forkOS :: IO () -> IO ThreadId
```

Like `forkIO`, it also creates a new Haskell thread, but additionally it creates a new OS thread and binds the new Haskell thread to it. This is accomplished by simply making a foreign call to an external function that (a) creates the new OS thread, and (b) in the new OS thread, invokes the requested action via a callback, thus creating a new bound Haskell thread.

We give the implementation of `forkOS` below for reference, although we have not introduced all of the concepts used in it. It assume the existence of an external function `createOSThread` to create the OS thread; its implementation is simple, but depends on the particular thread creation primitives used on the current operating system.

```
forkOS action = do
    mv <- newEmptyMVar
    entry <- wrapIO $ do
                t <- myThreadId
                putMVar mv t
                action
    createOSThread entry
    tid <- takeMVar mv
    freeHaskellFunPtr entry
    return tid

foreign import ccall "createOSThread"
  createOSThread :: FunPtr (IO ()) -> IO ()
foreign import ccall "wrapper"
  wrapIO :: IO () -> FunPtr (IO ())
```

4.2.2 *runInBoundThread*

The runInBoundThread combinator runs a computation in a bound thread. If the current thread is bound, then that is used; otherwise a new bound thread is created for the purpose. The combinator is useful when the program is about to make a group of related foreign calls that must all be made in the same OS thread.

The implementation is straightforward:

```
runInBoundThread :: IO a -> IO a
runInBoundThread action = do
    bound <- isCurrentThreadBound
    if bound
        then action
        else do
            mv <- newEmptyMVar
            forkOS (action >>= putMVar mv)
            takeMVar mv
```

Note that runInBoundThread does not return until the IO action completes.

5 Operational Semantics

In order to make the design for our language extension precise, we now give an operational semantics for Concurrent Haskell with the FFI and bound threads. The operational semantics is highly abstract: it does not model any details of actual computation at all. Instead, it models only the operations and interactions we are interested in:

- The running system consists of a pool of native (OS) threads and a pool of Haskell threads.
- Haskell threads may fork new Haskell threads (forkIO), make foreign calls, and perform unspecified IO operations.
- Native threads have an identifier and a stack. A native thread may be currently executing Haskell code or foreign code, depending on what is on top of the stack. Native threads executing foreign code may make a call to Haskell code, creating a new Haskell thread.
- The semantics models the relationship between native threads and Haskell threads, and the difference between bound and unbound Haskell threads.

Further relevant semantics for IO code in Haskell can be found in Peyton Jones' "Tackling the Awkward Squad"[9] and the original Concurrent Haskell paper[10].

The syntax of a native thread is given in Figure 1. A native thread of form $N[S]$ has thread identifier N, while S is an abstraction of its call stack. If H is on top of the stack, the thread is willing to execute a Haskell thread. If $F^{si}\ h$ is on top of the stack, the thread is in the process of dealing with a call to a foreign function, which will return its result to the Haskell thread h. The safety of the foreign call is given by si, which is either u meaning unsafe, or s meaning safe.

A native thread of the form $N[\bullet]$ is a thread which originates in foreign code; it does not have any Haskell calls anywhere on its stack.

A native thread of form $N[H]$ has a stack that exists only to serve Haskell threads, and so can safely block inside a foreign call without impeding other Haskell threads. We call these threads "worker threads".

The syntax of a Haskell thread is given in Figure 2. A Haskell thread h of form $(a)_N$ has action a. The indicator N identifies the native thread N to which the Haskell thread is *bound*.

An action a is a sequence of operations, finishing with a return of some kind. An operation is either an unspecified IO operation (such as performing some evaluation, or operating on an MVar), a call to the primitive forkIO, or a call to a foreign function f.

We do not model the data passed to, or returned from, a foreign call, nor any details of what the IO operations are.

Note that forkOS is not mentioned alongside forkIO here. While spawning a new unbound thread requires direct support by the runtime system, creating a new bound thread is done by making a foreign call to the operating system-provided thread creation primitive. Therefore, forkOS need not be considered when we discuss the semantics.

5.1 Evolution

The symbol $\mathcal{N}$ refers to a set of native threads, and $\mathcal{H}$ to a set of Haskell threads. An executing program consists of a combination of the two, written $\mathcal{N};\mathcal{H}$.

We describe how the system evolves in a very standard way, using transition rules, of form

$$\mathcal{N};\mathcal{H} \Rightarrow \mathcal{N}';\mathcal{H}'$$

The structural rules are these:

$$\frac{\mathcal{N};\mathcal{H} \Rightarrow \mathcal{N}';\mathcal{H}'}{\mathcal{N}\cup\{t\};\mathcal{H} \Rightarrow \mathcal{N}'\cup\{t\};\mathcal{H}'} \qquad \frac{\mathcal{N};\mathcal{H} \Rightarrow \mathcal{N}';\mathcal{H}'}{\mathcal{N};\mathcal{H}\cup\{h\} \Rightarrow \mathcal{N}';\mathcal{H}'\cup\{h\}}$$

These standard rules allow us to write the interesting transitions with less clutter. The transition rules for the system are given in Figure 3.

We informally describe each rule in the semantics below:

IO A Haskell thread may perform an arbitrary IO operation. Note that we only require that there is one native thread ready to

$$\begin{array}{llcll}
\text{Native thread} & t & ::= & N[S] & \\
\\
\text{Stack} & S & ::= & \varepsilon & \text{Empty} \\
 & & | & H : S & \text{Executing Haskell} \\
 & & | & F^{si}\ h : S & \text{Executing a foreign call} \\
 & & | & \bullet & \text{Executing foreign code only} \\
\\
\text{Call Safety} & si & ::= & u & \text{Unsafe} \\
 & & | & s & \text{Safe}
\end{array}$$

Figure 1: Syntax of Native Threads

$$\begin{array}{llcll}
\text{Haskell thread} & h & ::= & (a)_{bt} & \\
\\
\text{Bound thread id} & bt & ::= & \varepsilon & \text{Not bound} \\
 & & | & N & \text{Bound to native thread N} \\
\\
\text{Haskell action} & a & ::= & p >> a & \text{Sequence} \\
 & & | & RET & \text{Return from a call into Haskell} \\
\\
\text{Operation} & p & ::= & \tau & \texttt{IO}\text{ operation} \\
 & & | & \texttt{forkIO}\ a & \text{Fork a thread} \\
 & & | & F^{si}\ f & \text{Foreign call}
\end{array}$$

Figure 2: Syntax of Haskell Threads

$$\begin{array}{rcll}
N[H:S];(\tau >> a)_{bt} & \Rightarrow & N[H:S];(a)_{bt} & (IO) \\
N[H:S];(\texttt{forkIO}\ b >> a)_{bt} & \Rightarrow & N[H:S];(a)_{bt},(b)_{\varepsilon} & (FORKIO) \\
N[H:S];(F^{si}\ f >> a)_{N} & \Rightarrow & N[F^{si}\ (a)_{N} : H : S]; & (FCALL1) \\
N[H];(F^{si}\ f >> a)_{\varepsilon} & \Rightarrow & N[F^{si}\ (a)_{\varepsilon} : H]; & (FCALL2) \\
N[F^{si}\ a_{bt} : S]; & \Rightarrow & N[S];a_{bt} & (FRET) \\
N[\bullet]; & \Rightarrow & N[H:\bullet];(a >> RET\ )_{N} & (HCALL1) \\
N[F^{s}\ h : S]; & \Rightarrow & N[H : F^{s}\ h : S];\ (a >> RET\ )_{N} & (HCALL2) \\
N[H:S];(RET\ )_{N} & \Rightarrow & N[S]; & (HRET) \\
;(RET\ )_{\varepsilon} & \Rightarrow & ; & (HEND) \\
(nothing) & \Rightarrow & N[H]; & (WKR) \\
 & & \text{where } N \text{ is fresh} & \\
N[H]; & \Rightarrow & (nothing) & (WKREND) \\
(nothing) & \Rightarrow & N[\bullet]; & (EXT) \\
 & & \text{where } N \text{ is fresh} & \\
N[\bullet]; & \Rightarrow & (nothing) & (NEND)
\end{array}$$

Figure 3: Operational Semantics

execute Haskell code (with *H* on top of its stack). The native thread may or may not be the same as the native thread bound to this Haskell thread, if any.

FORKIO A Haskell thread invokes `forkIO`, giving rise to a new, unbound, Haskell thread.

FCALL1 A bound Haskell thread makes a foreign call. The call must be made in the native thread bound to this Haskell thread.

FCALL2 An unbound Haskell thread makes a foreign call. The call is made in a worker thread.

FRET A foreign call returns; the Haskell thread which made the call is reintroduced into the pool of Haskell threads.

HCALL1 A native thread running exclusively foreign code (no Haskell frames on the stack) makes a call to a Haskell function. A new bound Haskell thread is created.

HCALL2 A native thread currently executing a *safe* foreign call from Haskell invokes another Haskell function. A bound Haskell thread is created for the new call.

HRET A bound Haskell thread returns; the native thread which made the call continues from the call site.

HEND An unbound Haskell thread returns.

WKR This rule models the birth of new worker OS threads, in case they should all be blocked in a foreign call.

WKREND A worker thread exits.

EXT A new native thread is created by foreign code.

NEND A native thread, executing foreign code only, exits.

In a executing program, there may be multiple valid transitions according to the semantics at any given time. We do not specify which, if any, of the valid transitions must be performed by the implementation.

Note that in particular this admits an implementation that does nothing at all; legislating against such behaviour in the semantics is entirely non-trivial, so we do not attempt it. Rather, we informally state the behaviour we expect from an implementation:

- If a Haskell thread is performing an unsafe foreign call, then the implementation is allowed to refrain from making any further transitions until the call returns.
- If a Haskell thread is performing a safe call, then the implementation should continue to make valid transitions in respect of other Haskell threads in the system.
- The implementation should not otherwise *starve* any Haskell threads.

Transitions which are not part of this semantics are erroneous. An implementation may either detect an report the error, or it may behave in some other implementation-defined manner. An application which depends on implementation-defined behaviour is, of course, non-portable.

6 Implementation

In this section, we will outline three different implementations of the language extensions described in this paper. The third of these has been implemented in the Glasgow Haskell Compiler; it is the most complex of the three, but it provides the best performance.

We will also describe the issue of I/O multiplexing, i.e. how to transparently speed up I/O beyond the levels attainable using safe foreign calls.

6.1 One OS Thread for one Haskell Thread

A very simple and straightforward implementation would be to use the operating system supplied thread library to create exactly one OS thread for each thread in the system.

In such an implementation, there is no distinction between bound and unbound threads; every thread can be considered a bound thread. This is entirely in accordance with the operational semantics outlined in Section 5. Only three of the rules (FORKIO, FCALL2 and HEND) explicitly refer to unbound Haskell threads; it's easy to see that nothing in these rules prevents each Haskell thread from having its dedicated OS thread.

In Section 4.1, we defined that `isCurrentThreadBound` may return `True` whenever the calling Haskell thread is indistinguishable from a bound thread; we can therefore define `isCurrentThreadBound = return True`, and we do not need to keep track of how Haskell threads were created.

Concurrent Haskell's thread creation and synchronisation primitives are simply mapped to the corresponding operating system functions. The `forkIO` primitive can be implemented the same way as `forkOS`.

The only challenge is managing access to the heap; it is very hard to support truly simultaneous multithreaded Haskell execution (on SMP systems), so it will be necessary to have a global mutual exclusion lock that prevents more than one thread from executing Haskell code at the same time.

This global lock would have to be released periodically to allow other threads to run; it would also have to be released for safe foreign calls.

Incidentally, this is exactly the strategy used by the native-code O'Caml implementation (see Section 7.2).

6.2 All Haskell Threads in one OS Thread

The second approach is to extend the fully multiplexed scheme to include bound threads. This is a natural extension for an existing single-threaded Haskell implementation where performance of foreign calls is not critical.

A single OS thread (the Haskell execution thread) is allocated for the Haskell system, and is used exclusively to execute Haskell code. All Haskell threads are multiplexed using this OS thread.

Additionally, the Haskell execution thread must keep track of:

- Any OS threads which have made in-calls. Each of these has given rise to a bound Haskell thread.
- A pool of OS threads that can be used to make "safe" foreign calls.

When a Haskell thread makes an out-call, there are two cases to consider:

- The Haskell thread is bound. The Haskell execution thread must pass a message to the appropriate OS thread in order to make the call, and the OS thread must return the result via another message back to the Haskell execution thread.

- The Haskell thread is unbound. The situation is similar, except that the OS thread to make the call can be drawn from the pool.

The complicated part of this implementation is the passing of messages between OS threads to make foreign calls and return results: essentially this is a remote procedure call mechanism. However, if the Haskell system is an interpreter, it may already have support for making dynamic foreign calls in order to implement `foreign import`.

A compiled implementation is unlikely to want use this scheme, due to the extra overhead on foreign calls. For an interpreter, however, this implementation strategy may be less complicated than the hybrid scheme discussed in the next section.

6.3 GHC's Implementation

GHC's run-time system employs one OS thread for every bound thread; additionally, there is a variable number of so-called "worker" OS threads that are used to execute the unbound (lightweight) threads.

Only one of these threads can execute Haskell code at any one time; the global lock that ensures this is referred to as "the Capability". GHC's main scheduler loop is invoked in all threads; all but one of the scheduler loops are waiting for the Capability at any one time.

Having more than one Capability available would indicate that truly simultaneous multithreaded Haskell execution is available; our current implementation does not however support this, because it would require synchronised access to the heap and other shared state. Whether the implementation can be practically extended in this direction is an open question.

6.3.1 Passing around the Capability

A thread will relinquish its Capability (i.e. execution of Haskell code will continue in a different OS thread) under the following conditions:

1. A safe (i.e. non-blocking) foreign call is made (FCALL1/2).

   For an unsafe call, we just hold on to the Capability, thereby preventing any other threads from running.

2. Another OS thread is waiting to regain the Capability after returning from a foreign call.

3. Another OS thread is waiting for the Capability because that thread is handling a foreign call-in.

4. The scheduler loop determines that the next Haskell thread to run may not be run in the OS thread that holds the Capability.

   When a scheduler loop encounters a Haskell thread that is bound to a different OS thread, it has to pass the Capability to that OS thread. When a scheduler in a bound OS thread encounters an unbound thread, it has to pass the Capability to a worker OS thread.

5. The Haskell thread bound to the current OS thread terminates (HRET).

   If the current OS thread has a bound Haskell thread and this Haskell thread terminates by returning, the OS thread will release the Capability and the scheduler loop will exit, returning to the foreign code that called it.

Threads that are just returning from a foreign call and threads that are handling a call to Haskell from foreign code are given priority over other threads; whenever it enters the scheduler, the thread that holds the capability checks whether it should yield its capability to a higher-priority thread (items 2 and 3 in the above list).

After yielding the capability and after passing the capability to another thread (item 4 in the list), the thread will immediately try to reacquire the capability; the thread will be blocked until another thread passes a capability to it again (via item 4 above), or until the Capability becomes free without being explicitly passed anywhere (item 5).

6.4 I/O Multiplexing

Traditional "multiplexing" run time systems that do not support non-blocking foreign calls usually still provide support for non-blocking input and output.

The obvious way to do this on POSIX systems is to use the `select` or `poll` system calls together with non-blocking I/O. When a `read` or `write` request fails to return the requested amount of data, the Haskell thread in question will be suspended. The scheduler loop will periodically use `select` or `poll` to check whether any suspended Haskell threads need to be woken up; if there are no runnable Haskell threads, the entire run-time system will block in the `select` or `poll` system call.

The Concurrent FFI makes this machinery unnecessary; a "safe" foreign call to `read` or `write` will have the desired effect for a multi-threaded Haskell program. However, using `select` or `poll` it is possible to achieve much better performance than using safe foreign calls, because it does not require an extra OS thread for each potentially-blocking I/O operation.

At first, we tried extending GHC's existing (single-OS-thread) implementation of I/O multiplexing to work with the hybrid threading model described above. In this scheme, an OS thread that blocks inside `select` still held the Capability to prevent multiple OS threads from using `select` simultaneously. When foreign code called in to or returned to Haskell while the RTS was waiting for I/O, it was necessary to interrupt the `select` by sending a dummy byte across a pipe, which slowed down foreign calls (both incoming and outgoing) a lot.

Fortunately, it turned out that a more efficient solution can be implemented entirely in Haskell, with no special support from the run time system beyond the extensions described in this paper.

The Haskell I/O library spawns an unbound Haskell thread, called the "I/O Service Thread", which uses a foreign call to `select` or a similar system call to watch a set of file descriptors. One of these file descriptors is the read end of a dedicated "wakeup pipe" which will be used to notify the service thread when the set of file descriptors to be watched has changed.

When an unbound Haskell thread needs to block in order to wait for some I/O, it will do the following:

1. Store the file descriptor in question in a global mutable vari-

able (an MVar).

2. Wake up the service thread by writing a byte to the wakeup pipe.
3. Wait for the service thread to notify us via an MVar.

The I/O service thread will repeatedly do the following:

1. Grab the set of file descriptors to be watched from the global mutable variable.
2. Do a safe foreign call to `select` or a similar system call in order to block until the status of one of the file descriptors or of the wakeup pipe changes.
3. If a byte has arrived on the wakeup pipe, read it from there in order to reset the pipe's state to non-readable.
4. Notify all Haskell threads waiting for file descriptors that have become readable or writable via their MVars.
5. Repeat.

When a bound thread needs to wait for a file descriptor to become readable, it should just safe-call `select` for that file descriptor, because that will be more efficient than waking the I/O service thread.

This scheme manages to keep the number of separate OS threads used when *n* unbound threads are doing I/O at the same time down to just two as opposed to *n* when safe foreign calls are used. GHC's previous scheme (released in GHC 6.2) needed just one OS thread in the same situation, but at the cost of one call to select every time through the scheduler loop, a `write()` to a pipe for every (safe) foreign call, and a lot of additional complexity in the run time system.

The new scheme requires no help from the run time system, removes the periodic call to `select` and supports more efficient foreign calls, at the cost of some inter-OS-thread messaging for every read or write that actually needs to block. According to our measurements, this overhead can be neglected.

Note also that this scheme is not tied to GHC's hybrid threading model; While there would be no performance gain for a one-to-one implementation, it also makes sense to use this I/O multiplexing scheme on top of the all-in-one-OS-thread implementation outlined in Section 6.2.

7 Related Work

We believe there is nothing in the literature that bears directly on the particular issues addressed in this paper. However, there is a great deal of folklore and existing practice in the form of language implementations, which we review here.

To summarise the related work: there is a general trend amongst languages with concurrency support to move from lightweight threads to OS threads in one-to-one mapping with the language's own threads. The most commonly quoted reasons for the switch are for accessing foreign library functions that might block, and scaling to SMP machines.

In relation to this paper, all of these languages could support the bound/unbound thread concept, which would then give the implementation freedom to use cheaper lightweight threads for the unbound threads. To our knowledge, there are no other languages that actually do support this idea.

7.1 Java

Java[2] began with a lightweight threading implementation, with all Java threads managed by a single OS thread (Java calls this "green threads"). Later implementations of Java moved to a native threading model, where each Java thread is mapped to its own OS thread. The reasons for the switch seem to be primarily

- Non-scalability of green threads to SMP machines
- Inability to call functions in external libraries which may block, without blocking the entire system

And perhaps the motivation was partly due to the JNI, which works smoothly because native threads are one-to-one with Java threads.

In contrast to Java, scaling to SMP machines is not a goal for us. There is no efficient SMP-capable Concurrent Haskell implementation, because doing so is still an open research question; the main sticking point is how to synchronise access to the main shared resource (the heap) without killing performance of the system. Furthermore, scaling to multiprocessors can often be achieved in the same ways as scaling to a cluster, by using multiple processes with explicit communication.

7.2 O'Caml

O'Caml[4] supports a choice between user-level and native threads for its bytecode interpreter, but compiled code must use native threads. An O'Caml programmer using native threads may currently assume that each O'Caml thread is mapped to a single OS thread for the purposes of calling external libraries.

Native threads were chosen over user-level threads for compiled code for the following reasons:

- The difficulty of implementing multithreaded I/O in a user-level scheduler across multiple platforms is high. Using native threads allows this issue to be handed off to the operating system, which significantly reduces implementation complexity and improves portability.
- Compiled O'Caml threads use the machine stack. With user-level threads, the scheduler must therefore be able to manage multiple machine stacks, which is heavily platform-dependent.

In O'Caml, a non-blocking foreign call is made by defining a C function which wraps the foreign call between the special calls `enter_blocking_section()` and `leave_blocking_section()`; this may only be done when using the native implementation of threads. Similarly, calls to O'Caml functions from C must be wrapped between `leave_blocking_section()` and `enter_blocking_section()`. This is equivalent to, if slightly less convenient than, Haskell's `safe` foreign calls and callbacks.

O'Caml could straightforwardly be extended with the concept of bound threads, which would leave the implementation free to use user-level threads with a pool of native threads for foreign calls in the same way as GHC. This would of course entail more implementation complexity, which may be worse than for GHC due to the use of the machine stack by O'Caml native code as noted above (GHC uses separate thread stacks managed by the runtime).

7.3 C♯ and .NET

The .NET Common Language Runtime (CLR) uses a one-to-one mapping between CLR threads and native Windows threads. Hence, threads in C♯ are fairly heavyweight.

To mitigate this, the .NET base class libraries include the `ThreadPool` class, which manages a pool of worker threads and a queue of tasks to be performed, including asynchronous I/O and timers. The `ThreadPool` class multiplexes the waiting operations onto a single thread, which significantly reduces the cost of a blocking operation compared with using a new thread. Computation tasks can also be submitted to the `ThreadPool`, and will be performed whenever there is a free thread in the pool. Therefore, `ThreadPools` achieve cheaper concurrency by avoiding repeated thread creation/deletion, at the expense of possibly having to wait for a computation to be performed if the thread pool is empty.

When used for multiple I/O requests, the `ThreadPool` concept is basically equivalent to the I/O multiplexing scheme used in GHC (Section 6.4). The main difference is that GHC's scheme is hidden from the programmer, who automatically gets the benefit of optimised multiplexing for all I/O operations provided the underlying implementation supports it.

7.4 User-level vs. kernel threads

Why should we care about lightweight threads? Many other languages have ditched the concept in favour of a one-to-one mapping between the language's own threads and native OS threads.

The reason is that lightweight Haskell threads can still be significantly cheaper than using OS threads. For example, the fastest implementation of native threads on Linux, the Native POSIX Threads Library[6] claims 20μsec per thread creation/exit, whereas GHC's implementation of Concurrent Haskell can achieve an order of magnitude improvement over this: the `conc004` test from GHC's test suite performed 10^6 thread creation/exit operations in 1.3sec on a 1Gz PIII, giving a thread creation/exit time of 1.3μsec. The NPTL paper doesn't give details on what hardware was used for their measurements, but a 1GHz PIII would seem to be a reasonable guess, being a midrange system at the date of that publication.

Native threads on other OSs are even more expensive; Windows for example has notoriously expensive operating system threads.

These implementations of OS threads are mapping OS threads onto *kernel threads*, and the kernel is managing the scheduling of threads. This is the reason for much of the overhead: many thread operations require a trip into the kernel.

So can OS threads be implemented in user space? Certainly; there are many implementations of purely user-space threading libraries, and these are indeed often faster than kernel threads. One problem, however, is that this doesn't let the multithreaded application take advantage of a multiprocessor; for that you need at least one kernel thread for each processor, so to this end hybrid models have been developed[8, 3] which use a mixture between user-space and kernel threads (sometimes called an $M:N$ threading model, indicating that M OS threads are mapped to N kernel threads).

It remains to be seen whether an implementation of OS threads can approach the performance of lightweight Concurrent Haskell threads. If that were to happen, then there would be no reason not to use a one-to-one implementation for Concurrent Haskell, and the bound/unbound concept would be redundant. However, there are reasons to believe that this is unlikely to happen, at least in the near future:

- The Native POSIX Threads Library[6] is 1:1, and claims better performance than a competing N:M implementation[3]. The improvement is largely attributed to the complexity of implementing the N:M scheme.
- Each OS threads by definition needs its own machine stack. Machine stacks are immovable, so must be allocated a fixed portion of the address space with enough room for the stack to grow. Since the library doesn't know ahead of time how much stack space a thread will need, it must guess, and inevitably this will end up wasting a lot of address space, which on a 32-bit machine is a scarce resource. In contrast, Haskell threads have stacks that are fully garbage collectable, and can be moved and grown at will.

Anderson et. al.[5] proposed a way to effectively combine the benefits of user-level threads and kernel threads by having explicit communication between the kernel scheduler and the user-level thread scheduler. A derivative of this scheme is currently being implemented in the FreeBSD operating system; no performance measurements were available at the time of writing.

8 Conclusion

We have designed a simple extension to the Haskell Foreign Function Interface, for specifying precisely the interaction between the FFI and Concurrent Haskell. It allows for the following features:

- Non-blocking foreign calls
- Callbacks and Call-ins from multithreaded applications
- Interacting with multithreaded foreign libraries, and foreign libraries that make use of thread-local state

Furthermore, the extensions require no new syntax, and have a simple operational semantics. A few simple library functions are provided for the programmer to work with the extensions.

Moreover, we have done this without requiring any fundamental restructuring of existing Haskell implementations: there is no requirement that the Haskell runtime be multithreaded, or that particular OS threads are used to execute Haskell code. However, we do accommodate an efficient implementation based on lightweight Haskell threads and a pool of OS worker threads for execution.

There is an implementation of the efficient scheme in a production Haskell Compiler (GHC), and we are currently gathering experience in using it.

9 References

[1] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[2] The Java language. `http://java.sun.com/`.

[3] Next Generation POSIX Threading. `http://www-124.ibm.com/pthreads/`.

[4] The O'Caml language. `http://www.ocaml.org/`.

[5] 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 Transactions on Computer Systems*, 10(1):53–79, February 1992.

[6] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for linux. Technical report, Redhat, February 2003. `http://www.redhat.com/whitepapers/developer/POSIX_Linux_Threading.pdf`.

[7] Manuel Chakravarty (ed.). The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.

[8] Richard McDougall and Jim Mauro. *Solaris Internals*. Prentice Hall, 2000.

[9] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In CAR Hoare, M Broy, and R Steinbrueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, NATO ASI Series, pages 47–96. IOS Press, 2001.

[10] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.