PROGRAMMABLE CONCURRENCY IN A PURE AND LAZY LANGUAGE

Peng Li

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2008

Steve Zdancewic
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

Acknowledgements

First, I thank my dissertation advisor, Steve Zdancewic, who has always been supportive to me in the last five years. Steve taught me how to do research, co-authored many papers with me, gave me insightful feedbacks and practically line-by-line writing advices, encouraged me to look for research directions that I am most interested in and obtained funding for my dissertation research. As an advisor, he could not have been more helpful.

My other mentors on programming languages research also deserve my thanks. Benjamin Pierce enlightened me on functional programming and the theory of programming languages; Stephanie Weirich impressed me on what type systems can do; Simon Peyton Jones educated me on the principles and philosophies of language design; Simon Marlow showed me how far one can go to make things run faster. In particular, I thank Benjamin Pierce and Stephanie Weirich for serving in my thesis committee.

I thank all the PL Club members for sharing their knowledge in the weekly discussions. I particularly thank Geoffery Washburn for helping me out with the CIS-670 project and patiently explaining programming language concepts to me when I first started doing research in programming languages. I should also give special thanks to Stephen Tse who has been a great friend to talk about information-flow type systems and functional programming.

I also thank Milo Martin and his students Colin Blundell and Joe Devietti for sharing research ideas on transactional memory. Milo has offered me critical feedbacks on one of my papers and invaluable suggestions on shaping up my dissertation proposal. I am grateful that he agreed to serve in my committee.

I give special thanks to my friend Yun Mao, who co-authored a paper with me and also provided me helpful comments on the idea of using Haskell for systems programming. I also thank the systems group at Penn who supported me in the first year of my doctoral study.

I thank Microsoft Research Cambridge for generously offering me the opportunity to intern with Simon Peyton Jones on 2007. I thank Andrew Tolmach for collaborating on the Haskell Workshop paper; I also thank him for not only agreeing to serve in my thesis committee but also attending my defense from far away.

This dissertation is funded by NSF 0541040-CCF: *Unifying Events and Threads: Language Support for*

Network Services.

Last, but also the most, I thank my family members for their enduring love and support. My parents, Yong Li and Lihua Zu, dedicated all their love and effort to the education of their only child. During my highschool years, they saved hard to buy me computers and sponsored me to participate in various programming competitions. Their wise investments paved the road for this dissertation. My wife, Hui Sun, provided me infinite amount of spiritual, logistical and technical support since we have known each other in the past three years. This dissertation is dedicated for her love and companionship.

ABSTRACT

PROGRAMMABLE CONCURRENCY IN A PURE AND LAZY LANGUAGE

Peng Li

Steve Zdancewic

This dissertation presents a number of methods to build massively concurrent network applications in Haskell, a pure, lazy and functional programming language. The key challenge is to combine *threads* and *events*, two seemingly opposing programming models, into one single application, so the programmer can get both the ease of multithreaded programming and the performance of event-driven systems.

This dissertation first shows a purely application-level solution using monadic, continuation-passing style threads and evaluates its performance using benchmarking programs. It then shows a more generic solution obtained by re-designing the Haskell runtime system with lightweight concurrency primitives that can be used to implement high-level concurrency features in Haskell libraries. The new runtime system allows massive concurrency to be implemented using any of the following configurations: OS threads, user-level threads, or application-level, monadic threads. This dissertations presents the prototype implementation of the new runtime system and two concurrency libraries. It also compares the I/O performance of different concurrency configurations using benchmark programs and discusses their trade-offs.

The conclusion of this dissertation is summarized from three different perspectives. For systems programmers, it compares the two solutions to combine events and threads in Haskell. For developers of the Haskell runtime system, it discusses the trade-offs involved to adopt the new runtime system design. For language designers, it summarizes the lessons learned in the programming experience of using the Haskell language.

Contents

List of Tables

List of Figures

Preface

This dissertation was developed during the past several years of my doctoral research through a central theme: building massively concurrent network services in Haskell, a pure, lazy and functional programming language. It is not a coincidence that I chose this research direction; there is a long story on how I went onto it.

I started my doctoral study in the systems group at the University of Pennsylvania on 2002. In the first year of graduate school, I took Benjamin Pierce's CIS-500 class and learned about functional programming as well as some basic knowledge about the theory of programming languages, such as program semantics and type systems. Although I had been programming for 9 years (since 1993), I had never found programming languages so interesting until I took the course! I then worked on a language-related project with Yun Mao and Steve Zdancewic, leading to the publication of my first paper on integrity information-flow policies [35]. In 2003, I switched to the programming languages group and took the honor to be Steve's very first doctoral student.

Being fresh as a "programming language" student, I explored for quite a while on what research topics to work on. After working with Steve on a Java Card project [37] in 2003, our focus has shifted to language-based information-flow security. In 2005, our papers on *downgrading policies* were published in POPL and CSFW [38, 39].

By 2005, I felt that I was finally on the right track of doing a doctoral dissertation in programming languages and I should keep my research focused on one topic. At the same time, I felt that, deep in my blood, I am more of a "practical" person than of a "theoretical" person — I would very much like to take ideas from theoretical research and experiment them in the real world by designing and implementing programming languages and systems. Perhaps, I like programming languages mostly because I love *programming* so much in the first place.

Thinking this way, I became interested in the idea of *domain-specific languages* and tried to design one such language for information-flow security. I then discovered that functional programming languages, especially Haskell, are ideal for my purpose, because one can use higher-order functions to build control constructs and use them as an embedded language for domain-specific needs. In 2006, Steve and I published

a paper in CSFW [40], showing that one can encode information-flow in an arrow-based, embedded language and pack it into a Haskell library, without building a new language compiler like Jif [48].

The arrows paper is a crucial turning point of my research. I liked the Haskell language very much because of its powerful expressiveness, elegance and many other benefits, so I had been thinking, how would Haskell work in the real world? While I was browsing the literature, I went through a great paper by Koen Claessen on encoding lightweight processes using continuation-passing monads [18] and, due to my systems background, I instantly realized that this technique could potentially be the killer solution to tackle a tricky problem that I have had before in server programming, which is the dilemma of choosing between "threads" and "events", two seemingly opposing programming models for building network applications.

I generalized Claessen's technique and built a Haskell software package for building massively-concurrent network applications and tested its I/O performance, which looked promising. To demonstrate the ease of programming, I wrote a TCP stack entirely in Haskell and glued it into my software package by using only a few lines of code. Indeed, we were able to get the best of both worlds — the expressiveness of threads and the efficiency of events. Steve and I wrote up this discovery and submitted to OSDI, where our paper was rejected. Fortunately, our paper was later accepted in PLDI 2007 [41].

Shortly after we announced our paper on the newsgroups, we heard from the implementors of Glasglow Haskell Compiler and started collaboration on developing more generic support for lightweight concurrency in Haskell. In 2007, I spent three months at Microsoft Research Cambridge working with Simon Peyton Jones and Simon Marlow, on a lightweight concurrency design of the GHC runtime system. This work [36] was later published on Haskell Workshop 2007 with collaboration from Andrew Tolmach.

With the help of my dissertation committee, my dissertation has converged onto one single research direction: building massively concurrent network applications in Haskell. The key challenge is to make the concurrency implementation (low-level I/O mechanisms, scheduling algorithms, etc) *programmable* or *configurable* by the application programmer, so the optimal I/O performance can be achieved. Our PLDI paper presented a convenient, purely application-level solution; the new GHC runtime system design is a more generic solution and it allows more possible ways to build such systems. By early 2008, I have implemented a prototype of the new GHC runtime system as well as some simple concurrency libraries. To conclude, this dissertation presents a number of possible ways to implement massive concurrency in Haskell and compares these configurations using I/O performance benchmarks.

Chapter 1

Introduction

Modern network services present software engineers with a number of design challenges. Web servers, chat rooms, multiplayer games, and peer-to-peer data storage applications all require the ability to accommodate thousands to hundreds of thousands of simultaneous client connections. Such massively concurrent programs adds design complexity, especially when other requirements, such as high performance or fault tolerance, must also be met [64].

This dissertation deals with one particular challenge: combining *threads* and *events*, two seemingly opposing programming models for building such systems. Section 1.1 introduces the challenge; Section 1.2 overviews the technical development of the dissertation.

1.1 Challenge: programming models for massively concurrent network services

Massively concurrent network services are challenging to implement because the programmer needs to handle a large number of concurrent connections, for which I/O operations and computation must be properly interleaved to achieve high performance.

Historically, two implementation strategies for building such massively concurrent systems have been successful. In the *multithreaded* approach, each connection is serviced by a different thread of control, and these threads are implemented using the standard code structuring mechanisms, like procedures, found in sequential programs. Multithreaded systems usually provide synchronous I/O mechanisms. In the *event-driven* approach, each connection is decomposed into a series of asynchronous I/O requests. A single thread of control waits for completion of such asynchronous I/O requests and dispatches each completion event to an appropriate handler, which performs a (typically) small amount of processing; the flow of control in the

event-driven approach is encapsulated in the events themselves. Each of these approaches has benefits and drawbacks with respect to implementation, scalability, and ease of use, and there has been a long-standing debate as to which approach is best [33, 51, 69].

The rest of this section briefly reviews the basic ideas behind the multithreaded and event-driven models, and describes prior approaches to bringing these two models together.

1.1.1 The multithreaded programming model

There are dozens (if not hundreds) of different incarnations of the multithreaded programming model, ranging from the POSIX Threads API [17] for Unix C programming to Java's libraries and language support for synchronization [49], to more exotic concurrent programming languages like CML [60] or Polyphonic C# [9]. Rather than surveying them all here, this dissertation emphasizes their common features [12].

From the programmer's perspective, a *thread* can be thought of as a sequential program, and the control-flow of a thread is described using the standard constructs (function calls and returns, loops, exceptions and exception handlers, etc.) as for sequential programs.

Most multithreaded system provide synchronous interfaces for performing I/O, which is similar to the way I/O is performed in sequential programs. When a thread makes a synchronous I/O request, its execution is blocked until the I/O request has been completed. The interleaving of I/O and computation is managed by the thread scheduler, which is usually part of the language runtime system or the operating system.

When switching control from one thread to another, the scheduler must save all of the transient state of the first thread and restore the state of the second. To achieve good performance, network services like web servers may choose to forego *kernel-level threads* (threads that are natively supported by the operating system) in favor of *user-level threads*. User-level threads can deliver better performance because they eliminate the overhead of context-switching between user-mode and kernel-mode.

Threads can exchange information via named *channels* (which might buffer the communication) or through shared memory. Access to these shared resources is typically protected via *locking mechanisms* such as monitors, semaphores, or mutexes [65] to prevent concurrent accesses from leaving the system in an inconsistent state. These synchronization mechanisms can be used to establish *critical sections*, which are pieces of code that the scheduler effectively treats as atomic—only one thread may be in a particular critical section at a time. Other synchronization primitives allow threads to suspend execution and wait for a signal from another thread.

The primary advantage of the thread model is that the programmer can reason about the series of actions taken by a thread in the familiar way, just as for a sequential program. This leads to a familiar programming style in which the control flow for a single thread is made apparent by the program text, using ordinary constructs like **`if-then-else`** statements, loops, exceptions, function calls, and most importantly, synchronous

I/O.

For network services, a major drawback of the thread approach is its memory usage. In many multithreaded systems, not only the programming style is similar to that of sequential programs, but the runtime memory model is also similar to that in single-threaded systems. One such example is the runtime stack for languages like C. A stack is typically a continuous block of memory that is reserved for fast memory allocation and recycling in the last-in, first-out order. On single-threaded systems, a large amount of memory, such as a few megabytes, can be reserved for the stacks because there are not many of them. On multithreaded systems, however, stack space can be a limiting factor for scalability when tens of thousands of threads are available. Nevertheless, this problem can be solved by an alternative runtime model that does not use continuous stack spaces—a recent example in this vein is Capriccio [70], a threads package implemented by von Behren et al. specifically for use in building high-performance network services.

1.1.2 The event-driven programming model

Compared with threads, the event-driven approach inverts the roles of control and data in the model. The central component of the event-driven model is the *event loop*, which is a tight loop that accepts asynchronous I/O completion events and processes them in turn. An *event* (sometimes called a *message*) is registered when an asynchronous I/O request is initiated. The event itself is usually an object that describes which connection it belongs to and what operations needs to be done with it when the I/O operation is completed. The event loop examines each event received and dispatches it to its corresponding *handler*; and the handler performs the appropriate operations on the connection, possibly submitting more asynchronous I/O requests that are later fed back in to the main event loop.

In contrast to the multithreaded model, the event-driven approach allows the programmer to control how computation is represented and how computation and I/O are interleaved, thereby permitting application-specific optimizations that can significantly improve performance. In event-driven programming, each connection usually need only small amounts of local storage. Depending on the application, the amount of memory used by each connection can be as small as a few bytes. Furthermore, by grouping similar events together, they can be batch-processed to improve code and data locality [32].

The primary drawback of event-driven programming is that the control flow of a service becomes difficult to reason about. Rather than the familiar sequential code used in the threads model to express the steps needed to process a request, the equivalent program in the event-driven approach typically consists of several events that are linked together via their continuations. The control flow is no longer apparent from the program text, a problem that is made worse when complex behavior (such as loops or exceptions) must be encoded using events. Another problem is that long-running event handlers make the system as a whole unresponsive, because one long running handler prevents other events from being processed—handlers should

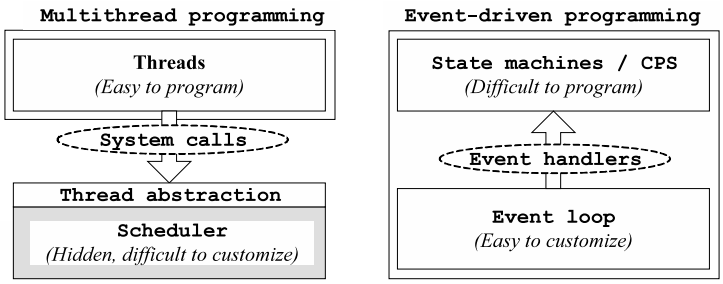

Figure 1.1: Threads vs. events

therefore cooperate to prevent any one of them from tying up the processor for too long. Finally, it can also be difficult to maintain local state across invocations of the same event handler—the state must somehow be encapsulated in the events themselves.

It has been argued that the performance benefits of the event-driven model outweigh the somewhat unnatural style of programming that they require [51]. At least two high-performance scalable web servers have been built using this approach [52, 73], both of which outperform Apache [5], which is implemented using threads. There are also libraries, such as Python's Twisted package [68] and C++'s Adaptive Communication Environment (ACE) [3] designed to make event-driven programs easier to develop. However, without explicit language support, writing software with these libraries is still cumbersome and the control flow issues are still present.

1.1.3 The duality between multithreaded and event-driven models

Both the multithreaded and event-driven approaches have their proponents and detractors. Recently, Ousterhout [51] has argued that "threads are a bad idea (for most purposes)," citing the difficulties of ensuring proper synchronization and debugging with thread-based approaches. A counter argument, put forth by von Behren, Condit, and Brewer [69], argues that "events are a bad idea (for high-concurrency servers)," essentially because reasoning about control flow in event-based systems is difficult and the apparent performance wins of the event-driven approach can be completely recouped by careful engineering [70]. However, they acknowledge that developing a sufficiently well-tuned, resource-aware thread scheduler can be a substantial challenge.

In 1978, Lauer and Needham tried to head off this debate by arguing that the multithreaded and event-driven models are dual to each other [33]. They describe a one-to-one mapping between the constructs of each paradigm and suggest that the two approaches should be equivalent in the sense that either model can be made as efficient as the other. The duality they present looks like this (using modern terminology):

Threads		Events
critical section	$\sim$	event handler
scheduler	$\sim$	event loop
exported function	$\sim$	event
procedure call	$\sim$	send event / await reply
waiting on condition variable	$\sim$	waiting for an event

The Lauer-Needham duality suggests that despite their large conceptual differences and the way people think about programming in them, the multithreaded and event-driven models are really the "same" underneath. Yet most existing approaches trade off threads for events or vice versa, choosing one model explicitly over the other. For instance, in their work on the Capriccio threads package, von Behren et al. acknowledge the validity of the duality but go on to conclude that the thread model is "better" because it makes it easier for programmers and tools like compilers to reason about control flow.

Another point of view shares the same sentiments as von Behren et al., namely that the Lauer-Needham duality is valid, but draw a different conclusion: rather than using the duality to justify choosing threads over events or vice versa (because either choice can be made as efficient and scalable as the other), one can see the duality as a strong indication that the programmer should be able to use *both* models of concurrency in the same system. The system should provide natural ways to support programming with both views so for each component of the system, the programmer can choose the appropriate view to fit the task at hand.

1.1.4 Combining threads and events: a hybrid model

Figure 1.1 summarizes the comparison of the two programming models. The figure shows the strengths and weaknesses of each model: multithreaded systems provide good abstractions but their underlying implementations are difficult to customize; event-driven systems are easy to customize but they have poor programming abstractions. Is it possible to combine these two models, keeping the good properties of both models while getting rid of their weaknesses? The result is Figure 1.2.

Figure 1.2 presents a *hybrid model* that can be seen either as a multithreaded system with a programmable scheduler, or as an event-driven system with a thread abstraction for representing control flow. The key is to implement both *thread abstractions* that represent control flow and *event abstractions* that provide scheduling primitives. These abstractions provide *control inversion* between the two worlds: the threads play the active role and make blocking calls into the I/O system, but the event-driven system also plays the active role and schedules the execution of threads as if calling event handlers. The hybrid model gives the best of two worlds: the expressiveness of threads and the customizability of events.

Implementing the hybrid model is challenging. Conventional thread abstractions are provided *outside* the application by means of OS/VM/compiler/library support. The scheduler code may live in a different

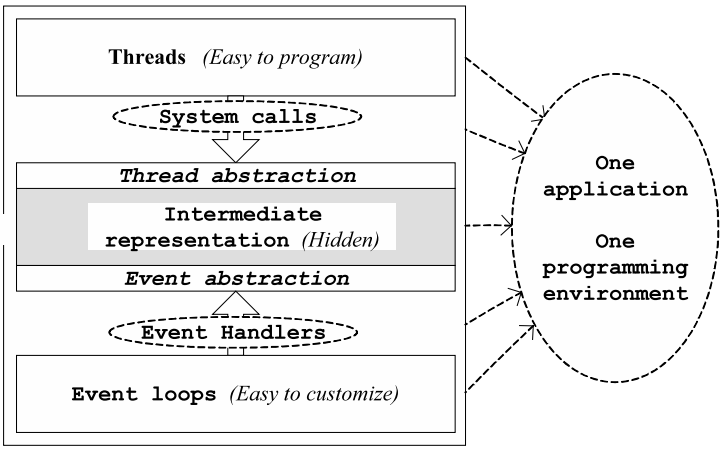

Figure 1.2: The hybrid model and application-level implementation

address space, be written in a different language, use low-level programming interfaces, be linked to different libraries, and be compiled separately from the multithreaded code. Such differences in the programming environments makes it inconvenient to combine the threads and the scheduler in the same application. Thus, a challenge is how to develop both components of the hybrid model (the boxes on the top and bottom of Figure 1.2) in the *same* programming environment, by which I mean:

- they are written in the same programming language;
- they can share the same software libraries;
- they live in the same address space;
- they can easily communicate using shared data structures;
- their source codes are compiled and optimized together to produce a single executable file.

In this dissertation, I present two solutions that implement the hybrid model in the Haskell programming language. In the first solution, the thread and event abstractions (the center box of Figure 1.2) are also developed in the exactly the same programming environment as the rest of the application; such concurrency abstractions are encoded using other languages features such as higher-order functions and monads and implemented as program modules. I call it an *application-level* implementation of the hybrid model. The second solution implements the thread and event abstractions as a hidden component in the Haskell runtime system and provides a set of *concurrency primitives* that can be used for programming with both threads and events.

Before introducing the solutions in Haskell, the next section first reviews some prior approaches to implementing the hybrid model.

1.1.5 Prior attempts to unify threads and events

Most prior approaches to reconciling threads and events have taken one model as primitive and tried to shoehorn the other model into the first using libraries or limited automation. The simplest way of doing this, exemplified by the Twisted programming library for Python [68], is to put the burden of converting between the two models almost entirely on the programmer. Twisted's library for event-driven programming, for example, leads to verbose coding style that amounts to performing a translation to continuation-passing style [7] by hand. ACE [3] provides similar functionality for C and C++.

To see what this kind of programming style looks like, consider the **`login`** function at the top left side of Figure 1.3. Written in C, this standard implementation uses a simple **`for`** loop to allow a user to attempt to authenticate up to three times. In the multithreaded approach, this same code could be used without change. In the event-driven approach, if the programmer wished to allow other processing to take place concurrently with the **`authenticate`** function call, the programmer would have to break up the **`login`** function into several pieces as shown in the remainder of Figure 1.3. Because there is some state (the loop variable **`i`**) that must be preserved across calls to **`authenticate`**, the programmer has to explicitly create closures [46], which consist of a function pointer and an environment. The top level functions **`login`** and **`authenticate`** take a continuation closure as an additional parameter. The auxiliary functions **`login_loop`** and **`auth_ret`** correspond to the loop entry point and the return continuation of the **`authenticate`** function, respectively—both of those program points are targets of jumps in the original program. In continuation-passing style, functions never return; instead they apply their continuation closure to the return value—for example the line "**`apply(k, true);`**" in **`auth_ret`** corresponds to the line "**`return true`**" in the standard version of **`login`**. These closures are the events that are processed by the event loop (which in this example would be hidden in the call to **`apply`**).

Although the details of this C code are not important, there are two key points this example shows: First, the event-driven programming model and continuation style are closely related. Second, programming in the continuation-passing style is less intuitive and less straightforward than sequential programming.

Adya et al. [4] improved the situation for C programming by providing the benefits of structured code for stack management in a cooperative, event-driven setting. They implemented a hybrid approach that automates the memory management for saving state across multiple event handlers (which becomes much more complex than the example shown in Figure 1.3 for more realistic settings, especially when freeing memory is taken into account). Their focus is on programming in C and C++, both of which do not have much support for creating and manipulating function closures. The automatic translation they propose is

```
/* A simple login function */
bool login(user u) {
  int i;
  for (i=0; i<3; i++) {
   if authenticate(u)
      return true;
  }
  return false;
}
```

```
void login_loop(env e, user u) {
  int i = (int)e[0];
  closure *k = (closure *)e[2];
  if (i < 3) {
    closure *k1 = close(&auth_ret,
                        3, i, u, k)
    authenticate(u, k1);
  } else {
    apply(k, false);
  }
}
```

```
/* The same function after    */
/* CPS and closure conversion */
typedef struct {
  void f(env, int);
  env  e;
} closure;

void login(user u, closure *k) {
  login_loop(mk_env(2, 0, k), u);
}
```

```
void auth_ret(env e, bool ans) {
  int  i = (int) e[0];
  user u = (user)e[1];
  closure *k = (closure *)e[2];
  if (ans) {
    apply(k, true);
  } else {
    login_loop(mk_env(2,i+1,k), u);
  }
}
```

Top Left: A simple C login function that allows a user to try to authenticate up to three times. Beneath it and on the right are the three functions resulting from CPS and closure conversion.

Figure 1.3: Continuation-passing style and closure conversion

essentially a form of closure conversion, although they do not provide any way to avoid programming in the explicit continuation-passing style.

One way of improving over these "do it (mostly) by hand" approaches is to augment the compiler with support for CPS and closure conversion. In some sense, this is what the Capriccio threads implementation does. I have also found this approach beneficial in the setting of Java card programming [37], where the underlying communication model of the smart card is event-driven, but the natural programming model for Java applications is to use threads.

All of these attempts highlight the benefits of using better language abstractions to make it easier to express the desired program behaviors. Functional languages like Scheme [31], ML [45], and Haskell [27, 66] all provide support for lightweight closures. Some dialects of these languages also support first class continuations (as in Scheme's `call/cc` operator), which can be used to build user-level threads implementations.

1.2 Proposed solution: lightweight concurrency in Haskell

First, this dissertation presents a Haskell solution based on *concurrency monads*. Unlike most previous work in the field, this approach provides clean interfaces to both multithreaded programming and event-driven programming in the same application, but it also does not require native support of continuations from compilers or runtime systems. Then, this dissertation investigates for a generic solution to support lightweight concurrency in Haskell, compares several possible concurrency configurations and summarizes the lessons learned.

1.2.1 An entirely application-level solution in Haskell

This dissertation tackles the threads-versus-events problem in the scope of the Haskell programming language. At the very beginning, a natural question is: why did I choose Haskell, instead of other languages?

The reason is that Haskell makes this task easy — in fact, a simple and elegant solution has already been invented prior to this dissertation: it works with the standard Glasgow Haskell Compiler [23] right out of the box; it requires no changes to the language compiler, runtime system or even standard libraries. The principle design comes from a functional pearl [18] published by Koen Claessen in 1999. Using a continuation-passing monad and lazy data structures, an application can have *both* lightweight, cooperative threads and an event-driven-style thread scheduler, all written in the Haskell language itself.

Chapter 3 shows how Claessen's design can be used to build massively concurrent network applications, with support for multiprocessors and various I/O mechanisms. It also evaluates the performance overheads of this design, showing that it indeed combines the best of both threads and events — the ease of multithreaded programming and the performance of event-driven systems.

Nevertheless, there are some limitations with Claessen's approach. Most importantly, the top-level of the application program has to be written in a user-defined monad, which not only introduces performance overheads, but also makes code highly specialized on a per-application basis and makes code sharing difficult among different applications. My implementation in Chapter 3 runs on top of the existing GHC runtime system, which itself is a sophisticated user-level thread system. This setup makes the overall system potentially redundant, inefficient and difficult to reason about. To overcome these limitations, some changes needs to be made to the existing Haskell runtime system design.

1.2.2 Generic support for lightweight concurrency in Haskell

Chapter 4 designs a new concurrency interface of the Haskell runtime system in order to provide a generic and more efficient solution for lightweight concurrency. Instead of implementing most concurrency features directly in the runtime system, the new design takes a modular approach: the runtime system becomes thinner and only implements a minimal set of *concurrency primitives*; the rest of the concurrency features are implemented as Haskell libraries.

With this new runtime system design, the programmer has the freedom to choose the concurrency implementation from a number of configurations, by using different concurrency libraries (or, by building concurrency features from scratch). The new runtime system design is completely orthogonal to Claessen's monad-based technique — the CPS monadic threads still works fine (if not better) in the new runtime system, yet the programmer can now implement similarly lightweight concurrency *without* using CPS monads, by using the *concurrency primitives* instead. Furthermore, the new runtime systems gives programmer direct control over OS threads: the programmer has the option to use one OS thread as exactly one Haskell thread; this setup could work well on systems that support efficient OS threads such as Linux.

Chapter 5 presents the details of my prototype implementation of the new runtime system design. Based on the prototype, I developed several concurrency library examples and tested the I/O performance of various concurrency configurations using two benchmark programs.

1.2.3 Comparisons and conclusions

At the end, Chapter 6 summarizes the conclusions of this dissertation from several perspectives. For the purpose of combining events and threads, it compares the benefits and drawbacks of the two approaches I presented in this dissertation. For the language implementors of Haskell, I discuss the trade-offs between the existing GHC runtime system and the new runtime system design. Finally, I discuss my lessons learned about the language design of Haskell from my programming experiences.

Chapter 2

Background: concurrent programming in Haskell

Before diving into the details of implementing massive concurrency in Haskell, it is helpful to provide a brief overview to the Haskell language, especially the features related to concurrency — this chapter serves the purpose.

Haskell is a *pure* and *lazy* functional programming language: standard computation has no side-effects; evaluation is non-strict. The combination of purity and laziness makes Haskell a kind of its own: it represents a radically different programming paradigm from all imperative languages like C or Java, and even many functional ones like ML. Because of its unique programming paradigm, Haskell has its own features and problems with respect to concurrency support.

The paper *"A History of Haskell: Being Lazy with Class"* [28] mentioned that Haskell is designed to be a non-strict or *lazy* language, and *purity* is a natural consequence of this design, mainly because the evaluation order is demand-driven and it is difficult for the programmer to reliably perform input/output as side-effects. Thus, Haskell is *pure*, and all the side-effects are performed through a novel mechanism called *monadic I/O*. However, on the other hand, one can also argue that *laziness* is a natural consequence of *purity* as well: the efficiency of many purely functional data structures heavily depends on the implementation technique called *lazy evaluation* [50].

Thus, purity and laziness are often tied together by their nature. Haskell is not the only language with this design — the Clean language [15] is also both pure and lazy.[1] However, what makes Haskell really unique, is its *monadic* approach to handle I/O and side-effects. The concurrency features in Haskell also heavily depends on monadic I/O — after all, concurrency is a side-effect.

[1] Unlike Haskell, Clean uses *uniqueness types* (or, linear types) to perform side-effects reliably.

2.1 Monadic I/O

Haskell supports a mechanism called *monads* [54, 71] that provides a generic interface for constructing computations with side effects. Although it is impossible to fully explain what a monad is and how it works in this dissertation, the following briefly introduces the concept of a state monad and discuss how *state threads* are used to implement monadic I/O in Haskell.

2.1.1 The state monad

One simple example of a monad is to construct computations with a mutable *state*: a value that can be read and modified during the execution of the computation. In a purely functional programming language, there is no direct support for mutable state, so the programmer has to simulate the *programming style* of mutable states in a special way: the programmer could add an extra parameter to every function in the program as if that parameter is the mutable state. To modify the state, a function can simply pass the new value of the state as the extra parameter to other functions.

Although this approach of using extra parameters could work, programming in this style can be verbose because the programmer has to do a lot of extra work to maintain the state. A better idea is to *encapsulate* the work of state-passing and to provide a composible interface to construct stateful computations: each stateful computation is represented by an abstract data type; computations can be composed using special operators, which maintain the passing of states.

The concept of a *monad* is a generalization of this idea. A data type can be thought of as a monad if it can be meaningfully composed with two operators. In Haskell's convention, these operators are called **`return`** and **`>>=`** (pronounced "bind"), respectively.

As an example, to encapsulate a mutable state, the monad can be defined as a data type **`StateMonad`** that stores a function that, when given a input state of type **`s`**, produces the output state (also of type **`s`**) and the result of the actual computation, which has type **`a`**:

```
data StateMonad s a = SM (s ->(s,a))

return :: a -> StateMonad s a
(>>=)  :: StateMonad s a -> (a -> StateMonad s b) -> StateMonad s b
read   :: StateMonad s s
write  :: s -> StateMonad s ()
```

The **`read`** and **`write`** operators read and write the encapsulated state, respectively. The **`return`** operator for the state monad "lifts" an ordinary expression of type **`a`** into the **`StateMonad`** monad by returning the

trivial action that performs no side effects (i.e. reading or writing the state) and yields the input as the final answer.

The infix operator **>>=** sequences the actions in its arguments: **e >>= f** returns a new action that first performs the stateful computation in **e** and passes the result computed by **e** to the function **f**, which contains another stateful computation. In addition, the states of **e** and **f** are threaded together by the binding operation.

With the state monad, the programmer can use **return**, **>>=**, **read** and **write** as the basic building blocks to construct stateful computations, without worrying how the state is maintained and passed around internally. However, programming with these operators, especially **return** and **>>=**, requires a special, "monadic" programming style that uses many nested anonymous functions, and the code can get quite verbose. For example, a function that reads the state twice and then updates it can be written as:

```
double :: StateMonad Int ()
double = read >>= \x ->
         read >>= \y ->
         return (x+y) >>= \z ->
         write z
```

Thankfully, Haskell has several features that make programming easier with monads:

- *Type classes* and *overloading*: because every monad has its own **return** and **>>=** operators, the general concept of monad can be defined as a *type class* **Monad** with these operators as its methods, so all these standard operators in different monads can have the same names. At compile time, the overloading of these operators is resolved using static typing information. This feature makes the monad a *generic* interface, just like arithmetic operators such as **+**, **-** and ***** for all numerical types.

- *Syntactic sugar*: Haskell provides the "do-syntax" to make programming easier with monads. Programs written in this syntax look much like those in conventional legacy programming languages like C or Java, but the sequencing of operations are automatically translated to the monadic programming style using **return** and **>>=**. For example, the **double** function mentioned above can also be written as:

  ```
  double :: StateMonad Int ()
  double = do { x <- read;
                y <- read;
                let z = x+y;
                write z
              }
  ```

Because Haskell supports automatic overloading of **`return`** and **`>>=`**, this syntactic sugar can be used with all monads that implements the **`Monad`** type class.

2.1.2 State threads and monadic I/O

The state monad introduced above has many limitations. To use it, the programmer has to find out *all* the states needed in a program and instantiate the state monad all at once. It only simulates a specific kind of side-effect, namely, mutable states, and it cannot be used for in-place memory updates or input/output, let alone concurrency.

Fortunately, with a small hack, the state monad can be turned into a very powerful mechanism called *state threads* [34] to support arbitrary side-effects. The idea is to use a virtual type called **`RealWorld`** which represents the entire state of the machine and use this virtual type as the state parameter in the state monad. Conceptually, any effectful operation has a type like **`RealWorld->RealWorld`**: in a purely functional view of the world, each such operation modifies the world by consuming a **`RealWorld`** and then creating a new one. The entire program can be thought of as a function that takes an initial **`RealWorld`** and returns the final **`RealWorld`**.

Of course, it would be impractical to store all the states in a value of **`RealWorld`**, which could be several gigabytes in size. If each value of **`RealWorld`** is used exactly *once* in the program, we can guarantee a linear use of these virtual states, and safely update them in-place. A *state thread* is a state monad, in which the actual implementation of the virtual state **`RealWorld`** is just an empty value, and the sole purpose of the virtual state is to represent the ordering of effectful operations.

In Haskell, the top-level program is written in the **`IO`** monad which is a state thread monad. The virtual state used in the **`IO`** monad is hidden in its implementation, so the user program cannot access the virtual state and violate its linear use. Each effectful primitive operation, such as allocating, reading and writing mutable memory references and input/output operations, consumes a **`RealWorld`** and produces a new **`RealWorld`**, and it is wrapped into an operation in the **`IO`** monad in the standard library.

Thus, the **`IO`** monad provides a way to reliably sequence operations with arbitrary side-effects in a pure and lazy language. The type of each function specifies what side-effects it may perform: a function of type **`Int->Int`** is guaranteed to be purely functional, and a function of type **`IO Int`** can perform arbitrary side-effects. **`IO`** is not the only state thread monad in Haskell; there are other state thread monads like **`STM`** [26] that encapsulate more fine-grained effects.

2.2 Threads and concurrent thunk evaluation

Having introduced monadic I/O, we are now ready to introduce concurrent, multithreaded programming in Haskell. The concept of a thread in Haskell is similar to that in imperative languages such as C or Java. Each thread performs an effectful computation task of type **`IO t`**. All threads in an application share the same memory space and execute concurrently. Threads can be created using the **`forkIO`** primitive, which accepts a computation task of type **`IO t`** and runs it in a new thread:

```
forkIO :: IO a -> IO ThreadId
```

Multithreaded programming in Haskell looks quite transparent to the programmer, but the runtime system has a tricky problem with concurrent evaluation: the interaction between *concurrency* and *lazy evaluation*. A good implementation of lazy evaluation (called a *fully lazy* one) requires that:

1. a term is not evaluated until its result is needed; and
2. a term is evaluated *at most once*: its result needs to be memoized to avoid duplicated evaluation.

A common implementation technique is to use a data structure called a *thunk* to store a such a term. A thunk is simply a memory cell providing an indirection to its content. It works like this:

1. before a thunk is evaluated, it stores a pointer to the code that generates its content;
2. when the thunk is first evaluated, the runtime system overwrites the thunk with a *blackhole* and evaluates the code that generates its content;
3. if a thunk marked with a blackhole is entered again, this means there is an infinite loop in the program;
4. when the content of a thunk has finished evaluation, the thunk is again overwritten to point to the result; and
5. subsequent accesses to the thunk will be lead to the memoized result.

This technique works beautifully in a single-threaded runtime system without concurrency support. However, when multithreading is added, the story about thunks becomes much more complicated, because two threads may attempt to evaluate the same thunk simultaneously. There are two problems to consider:

- When entering a thunk (i.e. starting to access a thunk), a thread finds out that the content of a thunk is a blackhole. In a single-threaded implementation this suggests that there is a dead loop in the program, but it may not be so in a concurrent setting. In the latter case, the current thread is *blocked*, and it should wait until the thunk has been updated by another thread and its result is available.

One possible implementation is to attach a waiting queue to the blackhole, so when a thunk is updated, all threads blocked on this thunk can be located and woken up. Another possibility is to let the runtime system periodically poll the blocked threads and resume them as appropriate.

- When two threads attempts to enter the same thunk, there is a race condition: some synchronization is required to make sure that only one thread starts to evaluate the thunk, while the other one sees a blackhole and waits. The same kind of synchronization needs to be considered when updating the thunk with its final result, so that threads waiting on the thunk can all be woken up. Therefore, the thunk must be protected by a synchronization mechanism, such as a lock.

  Because thunks are used frequently in the runtime system, the synchronization overhead on thunks can become a performance bottleneck of the system. For this reason, GHC has developed sophisticated non-blocking algorithms to optimize thunk synchronization [25].

  Another problem is that all such synchronization on thunks is implicit — the details are hidden in the runtime system and completely transparent to the programmer. It is well-known that lazy evaluation can make it difficult to reason about performance and resource usage in a program, and thunk synchronization makes it even more difficult.

The following example shows two threads competing to evaluate a thunk that contains some heavy computation. The variable **e** is bound to a term that could take a long time to compute. The program forks two threads to print the result of evaluating **e**. Although there is no explicit synchronization actions specified by the programmer, only one thread will evaluate **e** and the other thread will block on **e** and wait until evaluation of **e** has finished.

```
let e = ... (expensive computation) ...;
forkIO (print e);
forkIO (print e);
```

This dissertation presents a new runtime system design in Chapter 4 that implements threads and schedulers using concurrency libraries written in Haskell. Concurrency thunk evaluation is challenging for the new design because thunk evaluation is implemented in the runtime system, but the scheduling (and blocking) or threads are implemented in the concurrency library instead. A special interface is need between the runtime system and the concurrency library to handle thunk blocking.

2.3 Thread synchronization

Haskell provides several synchronization mechanisms for concurrent programming. The most commonly used one is *MVar* [53], pronounced “em-var”. An MVar is a blocking synchronization variable. It can be

thought of as a queue that holds at most one element; reading an empty MVar or writing to a full MVar will block until the operation can proceed. Of course, MVar operations are effectful —they live in the **`IO`** monad:

```
data MVar a

newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

Using MVar as a primitive, many other blocking synchronization mechanisms can be encoded, such as arbitrary-sized blocking queues and traditional locks.

Recently, software transactional memory (STM) emerged as a more expressive synchronization mechanism in Haskell [26]. The programmer uses atomic memory transactions to communicate among threads. A memory transaction is a computation that contains reading and writing operations on *transactional variables*. The runtime system schedules the execution of transactions as if they are executed in a serial ordering.

```
data TVar a

newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
atomically :: STM a -> IO a
retry      :: STM ()
orElse     :: STM a -> STM a -> STM a
```

The STM design in Haskell has a strong transactional semantics. Besides accessing transactional variables, a memory transaction cannot have other side-effects such as deleting a file or launching a missile. This is achieved by requiring that all the transactional code must be written in the **`STM`** monad, which is a state thread monad that only permits operations on transactional variables. At the outermost level, a **`STM`** transaction is executed using the **`atomically`** operation, which yields a **`IO`** action.

Another nice property of Haskell's STM is that transactional code can be *composed*: two transactional computations can either be composed sequentially using the monadic binding operator, or be composed "in parallel" as nondeterministic choices using a special operator called **`orElse`**. In the latter case, the programmer can use the **`retry`** operation (or exceptions) to partially roll back a transaction and perform backtracking on other nondeterministic choices.

Note that the Haskell STM can perform *blocking* synchronization of threads: if a transaction cannot proceed because all of its nondeterministic choices are rolled back using **`retry`**, it will *block* until some updates have been made to the transactional variables it has read. For this reason, STM can be used to

encode many blocking synchronization mechanisms, such as the MVar. Interestingly, the MVar encoded in STM (called TMVar — "transactional" MVar in the STM library) is often more convenient to use than the original version of MVar, because multiple TMVar operations can be easily combined to a single atomic action in STM.

This dissertation presents the design of Primitive Transactional Memory (PTM) in Chapter 4 as a lightweight concurrency primitive for developing concurrency libraries in Haskell. The design of PTM is mostly derived from the current STM interface: PTM and STM are very similar, except that the blocking synchronization mechanisms are different.

2.4 Concurrency in the Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is a highly optimized, open source compiler for Haskell. In particular, GHC has sophisticated support for concurrent programming:

- Lightweight, preemptive user-level threads. Thread creation and MVar synchronization are highly efficient [63]. The GHC runtime system handles the concurrent thunk evaluation problem (as mentioned in Section 2.2) using a sophisticated lock-free algorithm called *lazy blackholing* [25].

- Multiprocessor support. The GHC runtime system multiplexes a large number of user-level threads onto a few OS threads in order to fully utilize multiprocessor systems [25]. This feature is often called the $M : N$ mapping in the literature on user-level thread implementations. In the GHC runtime system, every OS worker thread is used as a "virtual" processor in the user-level thread system; it has its own allocation area and run-queue for executing lightweight Haskell threads. Threads stick to one processor by default (to get good locality), but idle processors can steal threads from busy ones.

- Support for the Haskell *Foreign Function Interface* [25, 43], *software transactional memory* [26] and other concurrency features, such as asynchronous exceptions [44], parallel sparks [67], etc.

Most of these features are implemented in GHC's runtime system (RTS). The GHC RTS is written in C and assembly. It manages the execution of a Haskell program, including setting up the STG machine (GHC's Haskell runtime model), memory allocation, garbage collection, I/O, foreign calls and most of the concurrency features.

Although the current GHC provides lightweight Haskell threads, it is more biased to computation-intensive applications and it is not yet suitable for building massively concurrent network applications. The problem is on the current I/O implementation of the GHC RTS.

The existing GHC RTS implements non-blocking I/O using the "select" interface on Unix-like systems. Although this interface is highly portable and implements non-blocking I/O for sockets and pipes, it has

strong limitations on the number of concurrent connections it can handle. Furthermore, its execution overhead scales up almost linearly with the number of concurrent I/O requests. Using this interface, it is difficult to handle more than few hundreds of concurrent connections in a network service, and the interface itself can be a performance bottleneck.

Another significant drawback of the I/O implementation in the GHC RTS is that it does not properly implement disk I/O. It treats I/O requests on all types of file descriptors uniformly, monitors them using the "select" interface and uses non-blocking I/O on them. Unfortunately, the "select" interface cannot be used to implement asynchronous disk I/O: even when it shows a file is readable or writable, the subsequent reading or writing operation may be blocked indefinitely. Therefore, the current disk I/O implementation in the GHC RTS is equivalent to synchronous I/O. This could lead to poor performance when the disk I/O requests of many lightweight Haskell threads are multiplexed onto a few OS worker threads, because the OS worker thread may be blocked by the synchronous I/O request and cannot do more useful work until the I/O request is completed.

Although GHC has poor I/O performance in its current implementation, the Haskell language is particularly suitable for implementing massively concurrent network services, as I will show in the next Chapter. This dissertation presents two solutions to develop high-performance, massively concurrent networking applications in Haskell; both solutions surpasses the current GHC implementation on Linux I/O performance.

Chapter 3

A poor man's lightweight concurrency

This chapter presents a lightweight, *application-level* solution to build massively concurrent network applications in Haskell: the concurrency features, such as threads and events, are entirely encoded in the standard Haskell language and implemented as part of the application program.

The core idea comes from the *concurrency monad* invented by Koen Claessen in 1999 in the functional programming community [18]. The rest of this chapter shows how Claessen's design is extended to support various I/O mechanisms and multiprocessor environments and it evaluates the performance of this approach using I/O benchmarks.

3.1 The concurrency monad

The continuation-passing style (CPS) programming technique is a promising solution to combine the benefits of threads and events: CPS has expressive control flow and CPS can be readily represented in many programming languages. The challenge, however, is to build appropriate abstractions to hide the details of continuation passing — such details are mostly mechanical plumbing work, yet they can be *too much* for the application programmer.

The simplest idea to hide continuation passing is to perform a source-to-source CPS translation [7] in the compiler. This approach is inconvenient because it often requires nonstandard compiler extensions. Furthermore, because the hybrid programming model requires that the multithreaded code have the same programming environment as the scheduler code, the compiler has to translate the *same* language to itself. Such translations can be verbose, inefficient and not type-safe.

Is it possible to achieve the same goal without writing compiler extensions? One solution, developed in the functional programming community and supported in Haskell, is to use a monad to hide the details of

continuation passing. The solution adapted here is to design the thread control primitives (such as **`fork`**) as monadic combinators, and use them as a domain-specific language directly embedded in the program. Such primitives hide the "internal plumbing" of continuation-passing in their implementation and gives an abstraction for multithreaded programming.

In principle, this monad-based approach can be used in any language that supports the functional programming style. However, programming in the monadic style is often not easy, because it requires frequent use of binding operators and anonymous functions, making the program look quite verbose. As shown earlier in Chapter 2, Haskell has many features to facilitate programming with monads.

In 1999, Koen Claessen showed that cooperative multithreading can be represented using a monad [18]. His design extends to an application-level implementation technique for the hybrid model, where the monad interface provides the thread abstraction and a lazy data structure provides the event abstraction. This section revisits this design, and the next section shows how to scale up this technique to multiplex I/O in network server applications.

3.1.1 Traces and system calls

This chapter uses the phrase "*system calls*" to refer to the following thread operations at run time:

- Thread control primitives, such as **`fork`** and **`yield`**.
- I/O operations and other effectful **`IO`** computations in Haskell.

A central concept of Claessen's implementation is the *trace*, a structure describing the sequence of system calls made by a thread. A trace may have branches because the corresponding thread can use **`fork`** to spawn new threads. For example, executing the (recursive) **`server`** function shown on the left in Figure 3.1 generates the infinite trace of system calls on the right.

```
server = do {
  sys_call_1;
  fork client;
  server;
}

client = do {
  sys_call_2;
}
```

```
     SYS_CALL_1
        |
      SYS_FORK
     /        \
SYS_CALL_2   SYS_CALL_1
            /
      SYS_FORK
     /        \
SYS_CALL_2   SYS_CALL_1
              ... ...
```

Figure 3.1: Some threaded code (left) and its trace (right)

A run-time representation of a trace can be defined as a tree using algebraic data types in Haskell. The definition of the trace is essentially an enumeration of system calls, as shown in Figure 3.2. Each system call

in the multithreaded programming interface corresponds to exactly one type of tree node. For example, the **SYS_FORK** node has two sub-traces, one for the continuation of the parent thread and one for the continuation of the child. Note that Haskell's type system distinguishes code that may perform side effects as shown in the type of a **SYS_NBIO** node, which contains an **IO** computation that returns a trace.

```
-- A list of system calls used in the multithreaded programming style:
sys_nbio c                  -- Perform a non-blocking IO function c
sys_fork c                  -- Create a new thread running function c
sys_yield                   -- Switch to another thread
sys_ret                     -- Terminate the current thread
sys_epoll_wait fd event     -- Block and wait for an epoll event on a file descriptor
... ...
```

```
data Trace =                -- Haskell data type for traces
   SYS_NBIO (IO Trace)
 | SYS_FORK Trace Trace
 | SYS_YIELD Trace
 | SYS_RET
 | SYS_EPOLL_WAIT FD EPOLL_EVENT Trace
 | ... ...
```

Figure 3.2: System calls and their corresponding traces

Lazy evaluation of traces and thread control: A trace can be thought of as the output of a thread execution: as the thread runs and makes system calls, the nodes in the trace are generated. What makes the trace interesting is that computation is *lazy* in Haskell: a computation is not performed until its result is used. Using lazy evaluation, the consumer of a trace can control the execution of its producer, which is the thread: whenever a node in the trace is examined (or, forced to be evaluated), the thread runs to the system call that generate the corresponding node, and the execution of that thread is suspended until the next node in the trace is examined. In other words, the execution of threads can be controlled by traversing their traces.

Figure 3.3 shows how traces are used to control the thread execution. It shows a run-time snapshot of the system: the scheduler decides to resume the execution of a thread, which is blocked on a system call **sys_epoll_wait** in the **sock_send** function. The following happens in a sequence:

1. The scheduler forces the current node in the trace to be evaluated, by using the **case** expression to examine its value.

2. Because of lazy evaluation, the current node of the trace is not known yet, so the continuation of the thread is called in order to compute the value of the node.

3. The thread continuation runs to the point where the next system call **sys_nbio** is performed.

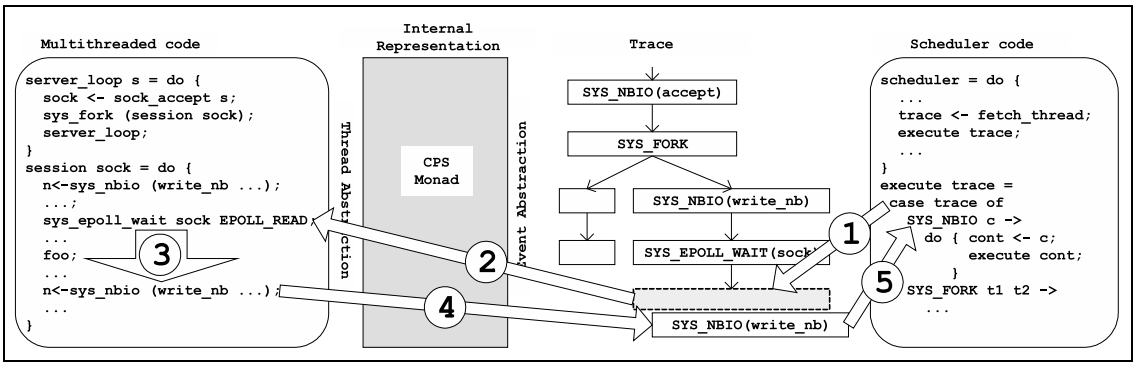

Figure 3.3: Thread execution through lazy evaluation (the steps are described in the text)

4. The new node in the trace is generated, pointing to the new continuation of the thread.

5. The value of the new node, **`SYS_NBIO`** is available in the scheduler. The scheduler then handles the system call by performing the I/O operation and runs the thread continuation, from Step 1 again.

The lazy trace gives the *event abstraction* just as needed. It is an abstract interface that allows the scheduler to play the "active" role and the threads to play the "passive" role: the scheduler can use traces to actively "push" the thread continuations to execute. Each node in the trace essentially represents part of the thread execution.

The remaining problem is to find a mechanism that transforms multithreaded code into traces.

3.1.2 The CPS monad

The *thread abstraction* is provided by using a monad. System calls can be implemented as monadic operations, and the *do-syntax* of Haskell offers an imperative programming style. The implementation of this monad is quite tricky, but the details are hidden from the programmer (in the box between the thread abstraction and the event abstraction in Figure 1.2).

The monad to be designed should encapsulate the side-effects of a multithreaded computation, that is, generating a trace. The tricky part is that if one simply represents a computation with a data type that carries its result value and its trace; such a data type cannot be used as a monad, because the monads require that computations be sequentially composable. Given two complete (possibly infinite) traces, there is no meaningful way to compose them sequentially.

The solution is to represent computations in *continuation-passing style* (CPS), where the final result of the computation is the trace. A computation of type **`a`** is thus represented as a function of type **`(a->Trace)->Trace`** that expects a continuation, itself a function of type **`(a->Trace)`**, and produces a

`Trace`. This representation can be used to construct a monad `M`. The standard monadic operations, `return` and `>>=`, are defined in Figure 3.4.

```
newtype M a = M ((a->Trace)->Trace)
instance Monad M where
  return x  = M (\c -> c x)
  (M g)>>=f = M (\c -> g (\a -> let M h = f a in h c))
```

Figure 3.4: The CPS monad

Intuitively, `return` takes a value `x` of type `a` and, when given a continuation `c`, simply invokes the continuation `c` on `x`. The implementation of `(>>=)` threads the continuations through the operations to provide sequential composition: given the "final" continuation `c`, `(M g) >>= f` first calls `g`, giving it a continuation (the expression beginning `(\a -> ...`) that expects the result `a` computed by `g`. This result is then passed to `f`, the results of which are encapsulated in the closure `h`. Finally, because the final continuation of the whole sequential composition is `c`, that continuation is passed to `h`. Altogether, this can be thought of as (roughly) doing `g;f;c`.

Given a computation `M a` in the above CPS monad, one can access its trace by adding a "final continuation" to it, that is, adding a leaf node `SYS_RET` to the trace. The function `build_trace` in Figure 3.5 converts a monadic computation into a trace.

```
build_trace :: M a -> Trace
build_trace (M f) = f (\c-> SYS_RET)
```

Figure 3.5: Converting monadic computation to a trace

In Figure 3.6, each system call is implemented as a monadic operation that creates a new node in the trace. The arguments of system calls are filled in to corresponding fields in the trace node. Because the code is internally organized in CPS, one can fill the trace pointers (fields of type "`Trace`") with the continuation of the current computation (bound to the variable `c` in the code).

```
sys_nbio f = M(\c->SYS_NBIO (do x<-f;return (c x)))
sys_fork f = M(\c->SYS_FORK (build_trace f) (c ())
sys_yield  = M(\c->SYS_YIELD (c ()))
sys_ret    = M(\c->SYS_RET)
sys_epoll_wait fd ev = M(\c->SYS_EPOLL_WAIT fd ev (c ()))
```

Figure 3.6: Implementing some system calls

The implementation of the monad encapsulates *all* of the "twisted" parts of the internal plumbing in the continuation-passing style. The implementation of the monad can be put in a library, and the programmer

only needs to understand its interface. To write multithreaded code, the programmer simply uses the do-syntax and the system calls; to access the trace of a thread in the event loop, one just applies the function **`build_trace`** to it to get the lazy trace.

3.2 Building network services

With Claessen's technique introduced in the previous section, events and threads can be combined to build scalable network services:: the code for each client is written in a lightweight, monad-based thread, while the entire application is an event-driven program that uses asynchronous I/O mechanisms.

I generalized Claessen's technique to combine multithreaded and event-driven programming and developed a framework for building massively concurrent network services in Haskell. My implementation uses version 6.6 or above of the Glasgow Haskell Compiler (GHC) [23]. Improving on Claessen's original, proof-of-concept design, my implementation offers the following:

- *True parallelism:* Application-level threads are mapped to multiple OS threads and take advantage of multiprocessor systems.

- *Modularity and flexibility:* The scheduler is a customizable event-driven system that uses asynchronous I/O mechanisms. I implemented support for Linux epoll and AIO; I also implemented a proof-of-concept TCP stack using userspace packet queues on Linux.

- *Exceptions:* Multithreaded code can use exceptions to handle failure, which is common in network programming.

- *Thread synchronization*: Non-blocking synchronization comes almost for free—software transactional memory (STM) in GHC can be transparently used in application-level threads. I also implemented blocking synchronization mechanisms such as mutexes.

It is worth noting that GHC already supports efficient, lightweight user-level threads, but the implementation does not use one GHC thread for each client directly, because the default GHC library uses the portable (yet less scalable) **`select`** interface to multiplex I/O and it does not support non-blocking disk I/O. Instead, the implementation employs only a few GHC threads, each mapped to an OS thread in the GHC run-time system. The rest of this chapter uses the name *monadic thread* for the application-level, "cheap" threads written in the do-syntax, and does not distinguish GHC threads from OS threads.

3.2.1 Programming with monadic threads

In the do-syntax, the programmer can write code for each client session in the familiar, multithreaded programming style, just like in C or Java. All the I/O and effectful operations are performed through *system calls*. For example, the **sys_nbio** system call takes a non-blocking Haskell **IO** computation as its argument. The **sys_epoll_wait** system call has a blocking semantics: it waits until the supplied event happens on the file descriptor.

The multithreaded programming style makes it easy to hide the non-blocking I/O semantics and provide higher level abstractions by encapsulating low-level operations in functions. For example, a blocking **sock_accept** can be implemented using non-blocking **accept** as shown in Figure 3.7: it tries to accept a connection by calling the non-blocking **accept** function. If it succeeds, the accepted connection is returned, otherwise it waits for an **EPOLL_READ** event on the server socket, indicating that more connections can be accepted.

```
sock_accept server_fd = do {
  new_fd <- sys_nbio (accept server_fd);
  if new_fd > 0
     then return new_fd
     else do { sys_epoll_wait fd EPOLL_READ;
               sock_accept server_fd;
             }
}
```

Figure 3.7: Wrapping non-blocking I/O calls to blocking calls

3.2.2 Programming the scheduler

Traces provide an abstract interface for writing thread schedulers: a scheduler is just a tree traversal function. To make the technical presentation simpler, suppose there are only three system calls: **SYS_NBIO**, **SYS_FORK** and **SYS_RET**. Figure 3.8 shows code that implements a naive round-robin scheduler; it uses a task queue called **ready_queue** and an event loop called **worker_main**. The scheduler does the following in each loop: (1) fetch a trace from the queue, (2) force the corresponding monadic thread to execute (using **case**) until a system call is made, (3) perform the requested system call, and (4) write the child nodes (the continuations) of the trace to the queue.

In the actual scheduler implementation, more system calls are supported. Also, a thread is executed for a large number of steps before switching to another thread to improve locality.

```
worker_main ready_queue = do {
  -- fetch a trace from the queue
  trace <- readChan ready_queue;
  case trace of
    -- Non-blocking I/O operation: c has type IO Trace
    SYS_NBIO c ->
      do { -- Perform the I/O operation in c
           -- The result is cont, which has type Trace
           cont <- c;
           -- Add the continuation to the end of the ready queue
           writeChan ready_queue cont;
         }
    -- Fork: write both continuations to the end of the ready queue
    SYS_FORK c1 c2 ->
      do { writeChan ready_queue c1;
           writeChan ready_queue c2;
         }
    SYS_RET -> return (); -- thread terminated, forget it
  worker_main ready_queue; -- recursion
}
```

Figure 3.8: A round-robin scheduler for three sys. calls

3.2.3 Exceptions

Exceptions are useful especially in network programming, where failures are common. Because the threaded code is internally structured in CPS, exceptions can be directly implemented as system calls (monad operations) shown in Figure 3.9. The code in Figure 3.10 illustrates how exceptions are used by a monadic thread.

The scheduler code in Figure 3.8 needs to be extended to support these system calls. When it sees a **SYS_CATCH** node, it pushes the node onto a stack of exception handlers maintained for each thread. When it sees a **SYS_RET** or **SYS_THROW** node, it pops one frame from the stack and continues with either the normal trace or exception handler, as appropriate.

3.2.4 Multiple event loops and multiprocessor support

Figure 3.11 shows the event-driven system in the full implementation. It consists of several event loops, each running in a separate OS thread, repeatedly fetching a task from an input queue or waiting for an OS event and then processing the task/event before putting the continuation of the task in the appropriate output queue. The **worker_main** event loops are simply thread schedulers: they execute the monadic threads and generate events that can be consumed by other event loops.

To take advantage of multiprocessor machines, the system runs multiple **worker_main** event loops in parallel so that multiple monadic threads can make progress simultaneously. This setup is based on the

```
sys_throw e -- raise an exception e
sys_catch f g -- execute computation f using the exception handler g

data Trace =
 ... -- The corresponding nodes in the trace:
 | SYS_THROW Exception
 | SYS_CATCH Trace (Exception->Trace) Trace
```

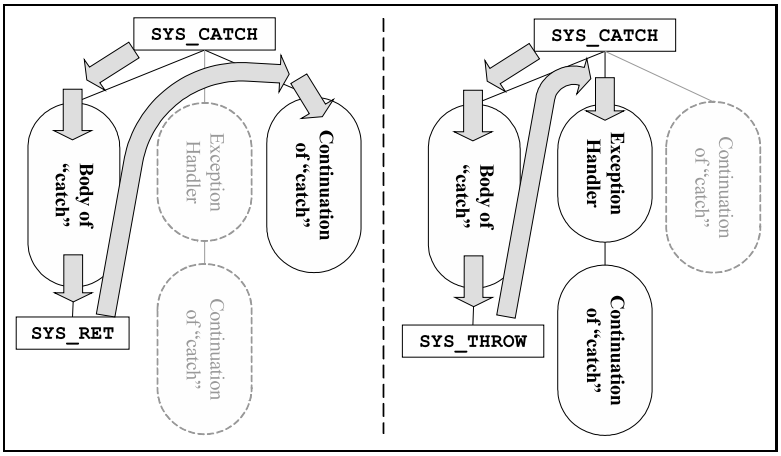

Figure 3.9: System calls for exceptions

assumption that all the I/O operations submitted by **SYS_NBIO** are non-blocking and thread-safe. Blocking or thread-unsafe I/O operations are handled in this framework either by using a separate queue and event loop to serialize such operations, or by using synchronization primitives (described below) in the monadic thread.

This event-driven architecture is similar to that in SEDA [73], but the events here are finer-grained: instead of requiring the programmer to *manually* decompose a computation into stages and specify what stages can be performed in parallel, this event-driven scheduler *automatically* decomposes a threaded computation into fine-grained segments separated by system calls. Haskell's type system ensures that each segment is a purely functional computation without I/O, so such segments can be safely executed in parallel.

Most user-level thread libraries do not take advantage of multiple processors, primarily because synchronization is difficult due to shared state in their implementations. The event abstraction here makes this task easier, because it uses a strongly typed interface in which pure computations and I/O operations are completely separated. The current design can be further improved by implementing a separate task queue for each scheduler and using work stealing to balance the loads.

```
-- send a file over a socket
send_file sock filename =
  do { fd  <- file_open filename;
       buf <- alloc_aligned_memory buffer_size;
       sys_catch (
         copy_data fd sock buf 0
       ) \exception -> do {
            file_close fd;
            sys_throw exception;
          } -- so the caller can catch it again
       file_close fd;
     }
-- copy data from a file descriptor to a socket until EOF
copy_data fd sock buf offset =
  do { num_read <- file_read fd offset buf;
       if num_read==0 then return () else
         do { sock_send sock buf num_read;
              copy_data fd sock buf (offset+num_read);
            }
     }
```

Figure 3.10: Multithreaded code with exception handling

3.2.5 Asynchronous I/O in Linux

The implementation supports two high-performance, event-driven I/O interfaces in Linux: epoll and AIO. Epoll provides readiness notification of file descriptors; AIO allows disk accesses to proceed in the background. A set of system calls for epoll and AIO are defined in Figure 3.12.

These system calls are interpreted in the scheduler **`worker_main`**. For each **`sys_epoll_wait`** call, the scheduler uses a library function to register an event with the OS epoll device. The registered event contains a reference to **`c`**, the child node that is the continuation of the application thread.

When the registered event is triggered, such events are harvested by a separate event loop **`worker_epoll`** shown in Figure 3.13. It uses a library function to wait for events and retrieve the traces associated with these events. Then, it put the traces into the ready queue so that their corresponding monadic threads can be resumed.

Under the hood, the library functions such as **`epoll_wait`** are just wrappers for their corresponding C library functions implemented through the Haskell Foreign Function Interface (FFI).

AIO is similarly implemented using a separate event loop. The programmer can easily add other asynchronous I/O mechanisms in the same way.

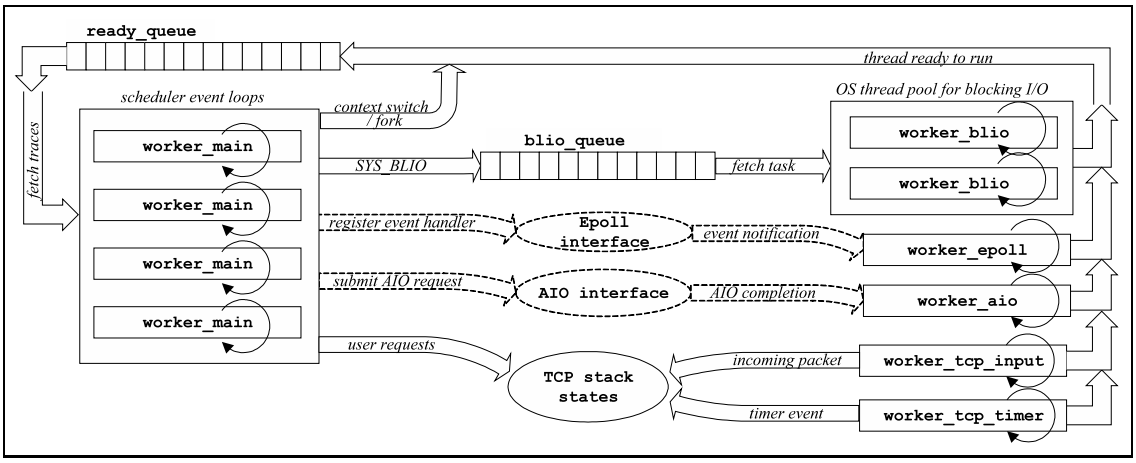

Figure 3.11: The event-driven system: event loops and task queues

```
-- Block and wait for an epoll event on a file descriptor
sys_epoll_wait fd event
-- Submit AIO read requests, returning the number of bytes read
sys_aio_read fd offset buffer

data Trace =
  ... -- The corresponding nodes in the trace:
  | SYS_EPOLL_WAIT FD EPOLL_EVENT Trace
  | SYS_AIO_READ  FD Integer Buffer (Int -> Trace)
```

Figure 3.12: System calls for epoll and (read-only) AIO

3.2.6 Supporting blocking I/O

For some I/O operations, the OS may provide only synchronous, blocking interfaces. Examples are: opening a file, getting the attributes of a file, resolving a network address, etc. If such operations are submitted using the **`SYS_NBIO`** system call, the scheduler event loops will be blocked.

The solution is to use a separate OS thread pool for such operations, as shown in Figure 3.11. Similar to **`sys_nbio`**, there is another system call **`sys_blio`** with a trace node **`SYS_BLIO`** (where BLIO means blocking IO). Whenever the scheduler sees **`SYS_BLIO`**, it sends the trace to a queue dedicated for blocking I/O requests. On the other side of the queue, a pool of OS threads are used, each running an event loop that repeatedly fetches and processes the blocking I/O requests.

3.2.7 Thread synchronization

The execution of a monadic thread is interleaved with purely functional computations and effectful computations (submitted through system calls). On multiprocessor machines, the GHC runtime system transparently manages the synchronization of purely functional computations (e.g. concurrent thunk evaluation) in

```
worker_epoll sched =
  do { -- wait for some epoll events
       results <- epoll_wait;
       -- for each thread object in the results,
       -- write it to the ready queue of the scheduler
       mapM (writeChan (ready_queue sched)) results;
       worker_epoll sched;
     }
```

Figure 3.13: Dedicated event loop for epoll

multiple OS threads. For the synchronization of effectful computations, this implementation offers several options.

For *non-blocking* synchronization, such as concurrent accesses to shared data structures, the *software transactional memory* [26] (STM) provided by GHC can be used directly. Monadic threads can simply use **`sys_nbio`** to submit STM computations as **`IO`** operations. As long as the STM transactions do not block, the scheduler event loops can run them smoothly.

For *blocking* synchronization, like the producer-consumer model, which affects thread scheduling, the programmer can define his own synchronization primitives as system calls and implement them as scheduler extensions. For example, *mutexes* can be implemented using a system call **`sys_mutex`**. A mutex is represented as a memory reference that points to a pair **`(l,q)`** where **`l`** indicates whether the mutex is locked, and **`q`** is a linked list of thread traces blocking on this mutex. Locking a locked mutex adds the trace to the waiting queue inside the mutex; unlocking a mutex with a non-empty waiting queue dispatches the next available trace to the scheduler's ready queue. Other synchronization primitives such as MVars in Concurrent Haskell [53] can also be similarly implemented.

Finally, because the implementation supports the highly-scalable epoll interface in Linux, monadic threads can also communicate efficiently using pipes provided by the OS.

3.2.8 Application-level network stack

The implementation includes an optional application-level TCP stack. The end-to-end design philosophy of TCP suggests that the protocol can be implemented inside the application, but it is often difficult due to the event-driven nature of TCP. In the hybrid programming model, the ability to combine events and threads makes it practical to implement transport protocols like TCP at the application-level in an type-safe fashion.

I implemented a generic TCP stack in Haskell, in which the details of thread scheduling and packet I/O are all made abstract. The TCP implementation is systematically derived from a HOL specification [13]. Although the development is a manual process, the purely functional programming style makes the translation from the HOL specification to Haskell straightforward.

I then glued the generic TCP code into my event-driven system in a modular fashion. In Figure 3.11, there are two event loops for TCP processing: **`worker_tcp_input`** receives packets from a kernel event queue and process them; **`worker_tcp_timer`** processes TCP timer events. The system call **`sys_tcp`** implements the user interface of TCP. A library (written in the monadic thread language) hides the **`sys_tcp`** call and provides the same high-level programming interfaces as standard socket operations.

3.3 Experiments

The main goal of the experiments was to determine whether the Haskell implementation of the hybrid concurrency model could achieve acceptable performance for massively-concurrent network applications like web servers, peer-to-peer overlays and multiplayer games. Such applications are typically bound by network or disk I/O and often have many idle connections. A second goal is to investigate the memory overheads of using Haskell and the concurrency monad. The concurrency primitives are implemented using higher-order functions and lazy data structures so I was concerned would impose too many levels of indirection and lead to inefficient memory usage. Such programs allocate memory frequently and garbage collection plays an important role.

A number of benchmarks were designed to assess the performance of the Haskell implementation. The I/O benchmarks use comparable C programs with the Native POSIX Thread Library (NPTL) as references. Here NPTL is an efficient implementation of Linux kernel threads; it was chosen because it is readily available in today's Linux distributions, and many user-level thread implementations and event-driven systems also use NPTL as a de-facto reference of performance.

Software setup: The experiments used Linux (kernel version 2.6.15) and GHC 6.5, which supports multiprocessor and software transactional memory. The C versions of benchmarks configured NPTL so that the stack size of each thread is limited to 32KB. This limitation allows NPTL to scale up to 16K threads in the tests.

3.3.1 Benchmarks

Memory consumption: This test uses a machine with dual Xeon processors and 2GB RAM. GHC's suggested heap size for garbage collection was set to be 1GB.

In the application-level scheduler, each thread is represented using a trace and an exception stack. The trace is an unevaluated thunk implemented as function closures, and the exception stack is a linked list. All the thread-local state is encapsulated in these memory objects.

To measure the minimal amount of memory needed to represent such a thread, I wrote a test program

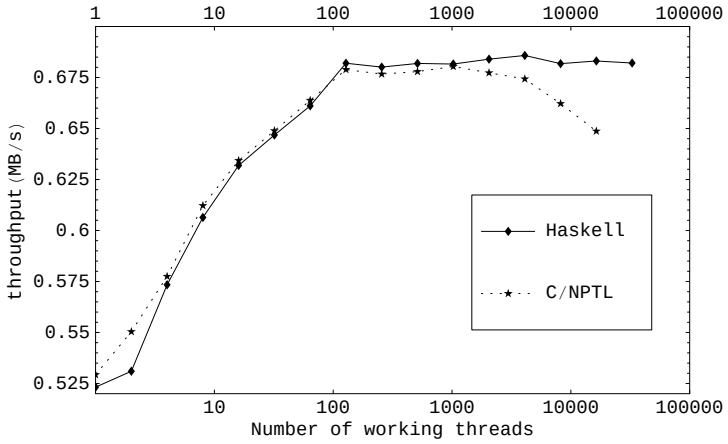

Figure 3.14: Disk head scheduling test

that launches ten million threads that just loop calling **`sys_yield`**. Using the profiling information from the garbage collector, I found that the live set of memory objects is as small as 480MB after major garbage collections—each thread costs only 48 bytes in this test.

Of course, ten million threads are impractical in a real system. The point of this benchmark is that the representation of a monadic thread is so lightweight it is never a bottleneck of the system. The memory scalability of my system is like most event-driven systems; it is only limited by raw resources such as I/O buffers, file descriptors, sockets and application-specific, per-client states.

Disk performance: The disk and FIFO I/O benchmarks were run on a single-processor Celeron 1.2GHz machine with a 32KB L1 cache, a 256KB L2 cache, 512MB RAM and a 7200RPM, 80GB EIDE disk with 8MB buffer. GHC's suggested heap size for garbage collection was set to be 100MB.

The event-driven system uses the Linux asynchronous I/O library (libaio), so it benefits from the kernel disk head scheduling algorithm just as the kernel threads and other event-driven systems do. I ran the benchmark used to assess Capriccio [70]: each thread randomly reads a 4KB block from a 1GB file opened using **`O_DIRECT`** without caching. Each test reads a total of 512MB data and the overall throughput is measured, averaged over 5 runs. Figure 3.14 compares the performance of the Haskell-based, application-level thread library with NPTL. This test is disk-bound: the CPU utilization is 1% for both programs when 16K threads are used. The Haskell system slightly outperforms NPTL when more than 100 threads are used. The throughput of the Haskell-based thread library remains steady up to 64K threads—it performs just like the ideal event-driven system.

FIFO pipe performance—mostly idle threads (IO): The event-driven scheduler uses the Linux epoll interface for network I/O. To test its scalability, I wrote a multithreaded program to simulate network server applications where most connections are idle. The program uses 128 pairs of active threads to send and

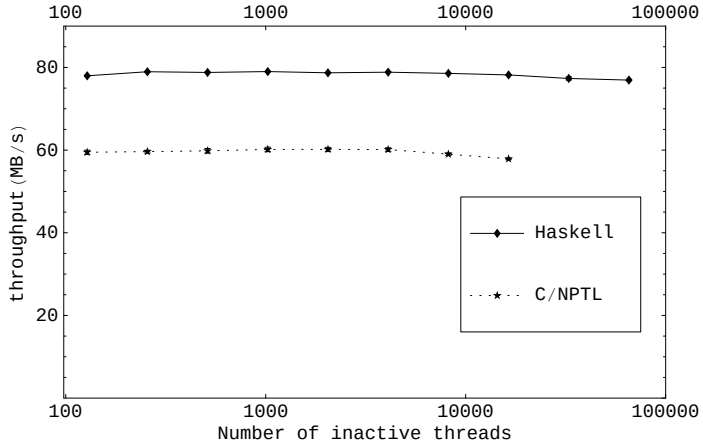

Figure 3.15: FIFO pipe scalability test (simulating idle network connections)

receive data over FIFO pipes. In each pair, one thread sends 32KB data to the other thread, receives 32KB data from the other thread and repeats this conversation. The buffer size of each FIFO pipe is 4KB. In addition to these 256 working threads, there are many idle threads in the program waiting for epoll events on idle FIFO pipes.

Each run transfers a total amount of 64GB data. The average throughput of 5 runs is used. Figure 3.15 shows the overall FIFO pipe throughput as the number of idle threads changes. This test is bound by CPU and memory performance. Both NPTL and the Haskell application-level threads demonstrated good scalability in this test, but the throughput of Haskell is 30% higher than NPTL. The performance difference is caused by the different I/O mechanisms used.

In all the I/O tests in Figures 3.14 and 3.15, garbage collection takes less than 0.2% of the total program execution time.

Although the Haskell-based system slightly outperforms NPTL in the above benchmarks, this dissertation is not trying to prove that the Haskell-based system has absolutely better performance. The goal is to demonstrate that the hybrid programming model implemented in Haskell can deliver *practical* performance that is comparable to other systems: a good programming interface can be more important than a few percent of performance.

3.3.2 Case study: a simple web server

To test this approach on a more realistic application, I implemented a simple web server for static web pages using the application-level thread library. I reused some HTTP parsing and manipulation modules from the Haskell Web Server project [42], so the main server consists of only 370 lines of code using monadic threads. To take advantage of Linux AIO, the web server implements its own caching. I/O errors are handled

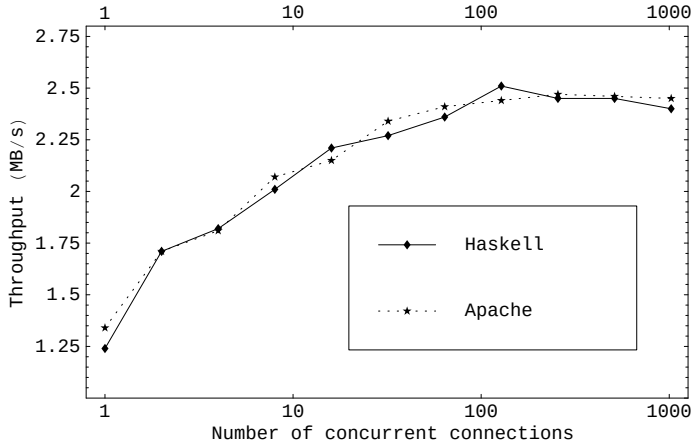

Figure 3.16: Web server under disk-intensive load

gracefully using exceptions. Not only is the multithreaded programming style familiar, but the event-driven architecture also makes the scheduler explicit and modular. The scheduler, including the CPS monad, system call implementations, event loops and queues for AIO, epoll, mutexes, blocking I/O and exception handling (but not counting the wrapper interfaces for C library functions), is only 220 lines of well-structured code. The scheduler is designed to be customized and tuned: the programmer can easily add more system I/O interfaces or implement application-specific scheduling algorithms to improve performance. The web server and the I/O scheduler are completely type-safe: debugging is made much easier because many low-level programming errors are rejected at compile-time.

Figure 3.16 compares my simple web server to Apache 2.0.55 for a disk-intensive load. I used the default Apache configuration on Debian Linux except that I increased the limit for concurrent connections. Using the same Haskell library, I implemented a multithreaded client load generator in which each client thread repeatedly requests a file chosen at random from among 128K possible files available on the server; each file is 16KB in size. The server ran on the same machine used for the IO benchmarks, and the client machine communicated with the server using a 100Mbps Ethernet connection. My web server used a fixed cache size of 100MB. Before each trial run I flushed the Linux kernel disk cache entirely and pre-loaded the directory cache into memory. The figure plots the overall throughput as a function of the number of client connections. On both servers, CPU utilization fluctuates between 70% and 85% (which is mostly system time) when 1,024 concurrent connections are used. My simple web server compares favorably to Apache on this disk-bound workload. For mostly-cached workloads (not shown in the figure), the performance of my web server is also similar to Apache.

The web server also works with my application-level TCP stack implementation. By editing one line of code in the web server, the programmer can choose between the standard socket library and the customized

TCP library. Because the protocol stack is now part of the application, we can tailor and optimize the TCP stack to the server's specific requirements. For example, urgent pointers and active connection setup are not needed. We can implement server-specific algorithms directly in the TCP stack to fight against DDoS attacks. Furthermore, my TCP stack is a zero-copy implementation[1]; it uses IO vectors to represent data buffers indirectly. The combination of packet-driven I/O and disk AIO provides a framework for application-level implementation of high-performance servers.

3.4 Limitations

The experimental results suggest that the concurrency monad can be used to write scalable, massively concurrent systems software with good I/O performance. Nevertheless, this approach has several inherent limitations.

First, the programmer is required to write the top-level program using a continuation-passing monad. This leads to a few problems:

- The continuation-passing monad limits the overall program design: the code is highly specialized on a per-application basis; such code looks different from regular Haskell programs. As a result, it can be difficult to share code among multiple applications.

- The continuation-passing monad introduces additional overheads: at run time, the state of each monadic thread is represented by nested heap closures; such closures are allocated and updated on every step of thread execution; they also create more garbage to be collected in the runtime system. Although such overhead is insignificant in networking applications when I/O (rather than computation) is the bottleneck, this overhead may be significant in other application scenarios. Specifically, I will show later in Chapter 5 that this overhead becomes significant on multiprocessor machines, where garbage collection becomes a bottleneck of the system.

Secondly, the overall concurrency implementation has a significant amount of redundancy: the application-level threads are first mapped onto standard Haskell threads, which is actually an implementation of user-level threads in GHC; the user-level threads are then mapped onto real OS threads by the GHC runtime system; the OS then maps the OS threads onto real processors. The application-level concurrency mechanisms implemented in this way cannot easily bypass the lower-level thread layers and cannot have fine-grained control on the underlying system resources. For example, there is no way to "pin" a computation onto a particular physical processor. Another consequence is that, even if the underlying operating system provides

[1] My current implementation uses iptables queues to read packets, so there is still unnecessary copying of incoming packets. However, this is not a fundamental limit—an asynchronous packet I/O interface would allow us to implement true zero-copying.

efficient OS threads (as the case of Linux), the programmer cannot directly take advantage of such features in the existing Haskell programming environment, because the user program has to be overlayed on top of the user-level threads implemented in the Haskell RTS.

Finally, the concurrency implementation is *cooperative*, rather than *preemptive*: it is difficult to break up a potentially long computation in one thread and yield control to another thread — doing so would require that the potentially long computation be written in the top-level concurrency monad, which may significantly hurt performance and make programming cubersome.

3.5 Related work and discussion

3.5.1 Language-based concurrency

This dissertation is not the first to address concurrency problems by using language-based techniques. There are languages specifically designed for concurrent programming, such as Concurrent ML (CML) [59] and Erlang [8], or for event-driven programming such as Esterel [10]. Java and C# also provide some support for threads and synchronization. Most of these approaches pick either the multithreaded or event model. Of the ones mentioned above, CML is closest to the work in this dissertation because it provides lightweight threads and event primitives for constructing new synchronization mechanisms, but its thread scheduler is still hidden from the programmer. There are also domain-specific languages, such as Flux [16], intended for building network services by composing existing C or Java libraries.

Rather than supporting lightweight threads directly in the language, there is also work on using language-level continuations to implement concurrency features [19, 62]. My work uses a similar approach, except that I do not use language-level continuations — the CPS monad and lazy data structures are sufficient for my purpose.

It is worth noting that this chapter only focuses on the domain of massively-concurrent network programming. Similar problems have also been studied in the domain of programming graphical user interfaces some time ago, and prior work showed different approaches to combine threads and events by using explicit message passing with local event handlers [22, 55].

3.5.2 Event-driven systems and user-level threads

The application-level thread library are closely related to two projects: SEDA [73] and Capriccio [70]. It can be seen as a combination of both projects in Haskell: the event-driven architecture of SEDA and the multithreaded programming style of Capriccio. Capriccio uses compiler transformations to implement linked stack frames; my application-level threads uses first-class closures to achieve the same effect.

Besides SEDA [73], there are other high-performance, event-driven web servers, such as Flash [52]. Larus and Parkes showed that event-driven systems can benefit from batching similar operations in different requests to improve data and code locality [32]. However, for complex applications, the problem of representing control flow with events becomes challenging. There are libraries and tools designed to make event-driven programs easier by structuring code in CPS, such as Python's Twisted package [68] and C++'s Adaptive Communication Environment (ACE) [3]. Adya et al. [4] present a hybrid approach to automate stack management in C and C++ programming.

Multiprocessor support for user-level threads is a challenging problem. Event-driven systems, in contrast, can more readily take advantage of multiple processors by processing independent events concurrently [74]. A key challenge is how to determine whether two pieces of code might interfere: my thread scheduler benefits from the strong type system of Haskell and the use of software transactional memory.

3.6 Summary

This chapter presented a technique to combine events and threads into a hybrid programming model in Haskell. The concurrency features are encoded in the standard Haskell language; they provide convenient programming abstractions yet they are lightweight. The experiments demonstrate that this approach can be used to write massively concurrent systems software with good performance.

That said, this approach also has its own limitations: the threads are not preemptive; the top-level program must be written in a user-defined monad, and the overall system design is redundant and opaque. The next chapter shows how to overcome these limitations with a more generic solution — a new runtime system design for Haskell.

Chapter 4

Generic support for lightweight concurrency in Haskell

This chapter presents a generic solution to support massively concurrent network services in Haskell: a new runtime system design that provides a set of low-level *concurrency primitives* that can be used by the programmer to build up high-level concurrency features as software libraries. The new runtime system design is not limited to networking applications — it is a general-purpose design that can be used to develop many kinds of lightweight concurrency mechanisms in Haskell.

The work presented here is the result of collaboration with Simon Peyton Jones, Simon Marlow and Andrew Tolmach. Much of the technical contents in his chapter has been published in the paper *"Lightweight Concurrency Primitives in GHC"* [36], but some design details have been changed since then. Compared to the published paper, the major changes are: (i) stack continuations are completely dropped and replaced by stack identifiers and stack switching, (ii) **`waitCond`** is replaced by a lower-level primitive **`sleepHEC`**, and (iii) minor extensions in state local states and preemption, foreign outcalls, etc.

4.1 A new runtime system design

The concurrency monad provides a lightweight solution to develop network services in Haskell but it has many limitations. The concurrency monad is a user-defined interface rather than a standard interface in Haskell. It is generally difficult to write reusable software libraries in the concurrency monad because the library author and the user need to agree on a common definition of the monad interface, which is likely to be application-specific. The concurrency monad does not support *preemption*, which can be useful in certain applications that involves potentially long computation.

One way to overcome these limitations is to use a more heavyweight solution — providing efficient I/O support directly from the Haskell runtime system. GHC already has sophisticated support for lightweight threads that both uses the standard IO monad interface and supports preemption.

Unfortunately, GHC's I/O implementation is quite rigid and it cannot be easily modified except by its language implementors. GHC's runtime system has approximately 90,000 lines of C and assembly code; it is large and difficult to debug. Furthermore, high-performance, non-blocking or asynchronous I/O mechanisms are often platform-specific, so the I/O implementation needs to be specialized for each platform in order to support such I/O mechanisms. This means a lot of work for the runtime system developers.

An attractive alternative to a monolithic runtime system written by the language implementators is to support concurrency using a *library* written in the language itself. The runtime system only needs to be re-engineered *once*, and the rest of the changes can all be made by developing or modifying the libraries that implements concurrency. This chapter explores such a modular design.

The first challenge is to design the detailed interface between the *concurrency library* written in Haskell, and the underlying *substrate*, or runtime system (RTS), written in C. Whilst the basic idea is quite conventional, the devil is in the details, especially since we want to support a rich collection of features, including: foreign calls that may block, bound threads [43], asynchronous exceptions [44], transactional memory [26], parallel sparks [67] and multiprocessors [25]. In particular, the concurrency library should be able to implement an I/O system that does not require user-defined monads, yet has similar performance with that in Chapter 3 and supports *preemption*.

The second challenge is to describe precisely how the concurrency primitives should behave. Concurrent programming is a notoriously slippery topic, so this chapter provides a precise operational semantics as the specification.

The remaining question is about choosing the exact mechanisms to be implemented as concurrency primitives:

- A key decision is what synchronization primitives are provided by the substrate. This chapter proposes a simplified transactional memory as this interface in Section 4.3.2, a choice that fits particularly well with a lazy language.

- The substrate does not follow the common practice of using continuations as a mechanism from which concurrency can be built. Instead, it uses a more primitive mechanism that simply switches execution from one thread to another. Furthermore, context switching and memory transactions are combined in a single **`switch`** primitive introduced in Section 4.3.4.

- The whole issue of thread-local state becomes pressing in user-level threads library, because a computation must be able to ask "what is my thread identifier?". This chapter proposes a robust interface

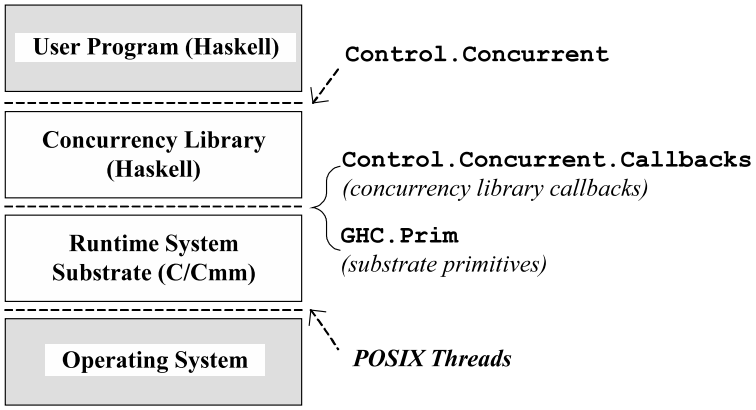

Figure 4.1: Components of the new RTS design

that supports local state in Section 4.3.5.

- Interfacing Haskell code to foreign functions and OS threads, especially if the foreign functions may themselves block, is particularly tricky. This chapter build on earlier work to solve this problem by introducing the concept of a Haskell Execution Context (HEC) as a low-level, coarse-grained concurrency mechanism.

4.2 Setting the scene

The goal is to design a *substrate interface*, on top of which a variety of *concurrency libraries*, written in Haskell, can be built (Figure 4.1). The substrate is implemented by the RTS developers themselves and hence, so far as possible, it should implement *mechanism*, leaving *policy* to the library. In general, one should strive to put as little as possible in the substrate, and as much as possible in the concurrency libraries.

The *substrate interface* consists of two parts:

1. A set of *substrate primitives* in Haskell, including primitive *data types* and *operations* over these types (Section 4.3).

2. A set of *concurrency library callbacks*, specifying interfaces that the concurrency library must implement (Section 4.4).

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

A single fixed substrate should support a variety of concurrency libraries. Haskell's existing concurrency

interface (**forkIO**, **MVar**s, **STM**) is one possibility. Another intriguing one is a compositional (or "virtualizable") concurrency interface [57], in which a scheduler may run a thread that itself is a scheduler, and so on. Another example might be a scheduler for a Haskell-based OS [24] or virtual machine (e.g. HALVM) that needs to give preferential treatment to threads handling urgent interrupts.

In addition to multiple clients, one should have in mind multiple implementations of the concurrency substrate. The primary implementation will be based on OS-threads and there can be several ways to do it: (i) each Haskell thread is executed on exactly one OS thread, and (ii) many lightweight Haskell threads are multiplexed onto one OS thread. Another possibility is that the RTS runs directly on the hardware, or as a virtualized machine on top of a hypervisor, and manages access to multiple CPUs.

Although written in Haskell, the concurrency library code may require the developer to undertake some crucial proof obligations that Haskell will not check statically; for example, "you cannot make a context switch to a thread that is already running, and a checked runtime error will result if you do so". This is still (much) better than writing it in C because safety is not violated.

The following design choices are the starting point :

- It must be possible to write a concurrency library that supports *preemptive* concurrency of *light-weight threads*, perhaps tens of thousands of them. It may be too expensive to use a whole CPU, or a whole OS thread, for each Haskell thread. Instead, a scheduler must be able to multiplex many Haskell fine-grain threads onto a much smaller number of coarse-grain computational resources provided by the substrate.

- Scheduling threads — indeed the very notion of a "thread" — is the business of the concurrency library. The substrate knows nothing of threads, instead supporting a lower-level primitive that switches from one computation to another.

- Since the substrate does not know about Haskell threads, it cannot deal with *blocking* of threads. Hence, any communication mechanisms that involve *blocking*, such as **MVars** and Software Transactional Memory (STM), are also the business of the concurrency library.

- Garbage collection is the business of the substrate, and requires no involvement from the concurrency library.

- The system should run on a shared-memory multi-processor, in which each processor can independently run Haskell computations against a shared heap.

- Because we are working in a lazy language, two processors may attempt to evaluate the same suspended computation (thunk) at the same time, and something sensible should happen.

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

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

data StackId
newStack   :: IO () -> IO StackId
getStackId :: PTM StackId
switch     :: PTM StackId -> IO ()

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

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

Figure 4.2: The substrate primitives

- The design must be able to accommodate a scheduler that implements the current FFI design [43], including making an out-call that blocks (on I/O, say) without blocking the other Haskell threads, out-calls that re-enter Haskell, and asynchronous in-calls.

4.3 Substrate primitives

It is now ready to embark on the main story, beginning with the substrate primitives. The type signatures of these primitives are shown in Figure 4.2, and the rest of this section explains them in detail.

The details of concurrency primitives are notoriously difficult to describe in English, so this chapter also give an operational semantics that precisely specifies their behavior. The syntax of the system is shown in Figure 4.3, while the semantic rules appear in Figures 4.4, 4.5, 4.6, 4.7, 4.8 and 4.10. These figures may look intimidating, but they will be explained as we go.

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

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

Terms

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

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

Figure 4.3: Syntax of terms, states, contexts, and heaps

4.3.1 Haskell Execution Context (HEC)

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

Informally, a HEC has the following behavior:

- A HEC is always in one of three states: *running* on a CPU or OS thread, *sleeping*, or making an *out-call*.
- A Haskell program initially begins executing on a single OS thread running a single HEC.
- When an OS thread enters the execution of Haskell code by making an in-call through the FFI, a fresh HEC is created in the *running* state, and the Haskell code is executed on this HEC. Note that this is the *only* way to create a new HEC.
- When the Haskell code being run by the HEC returns to its (foreign) caller, the HEC is deallocated, and its resources are returned to the operating system.
- When a running HEC makes a foreign out-call, it is put into the *outcall* state. When the out-call returns, the HEC becomes *running*, and the Haskell code continues to run on the same HEC.
- A HEC can enter the *sleeping* state voluntarily by executing **`sleepHEC`**. A sleeping HEC can be woken up by another HEC executing **`wakeupHEC`**. These two primitives are explained in Section 4.3.3.

Figure 4.3 shows the syntax of program states. The program state, P, is a "soup" S of HECs, and a heap Θ. A soup of HECs is simply an un-ordered collection of HECs $(H_1 \mid \ldots \mid H_n)$. Each HEC is a tuple (M, D, h, s) where h is the unique identifier of the HEC, s is the unique identifier of the current computation, and M is the term that it is currently evaluating. The D component is the stack-local state, whose description is deferred to Section 4.3.5. A sleeping HEC has a subscript "*sleeping*"; a HEC making a blocking foreign out-call has a subscript "*outcall*". The heap is a finite map from names to terms, plus a (disjoint) finite map from names to paused computations represented by pairs (M, D).

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

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

whose basic rules are shown in Figure 4.4. (More rules will be introduced in subsequent Figures.)

The $(IOAdmin)$ rule says that if any HEC in the soup has a term of form $\mathbb{E}[M]$, and M can make a purely-functional transition to N, then the HEC moves to a state with term $\mathbb{E}[N]$ without affecting any other components of the state. Here, $\mathbb{E}$ is an *evaluation context*, whose syntax is shown in Figure 4.3, that describes

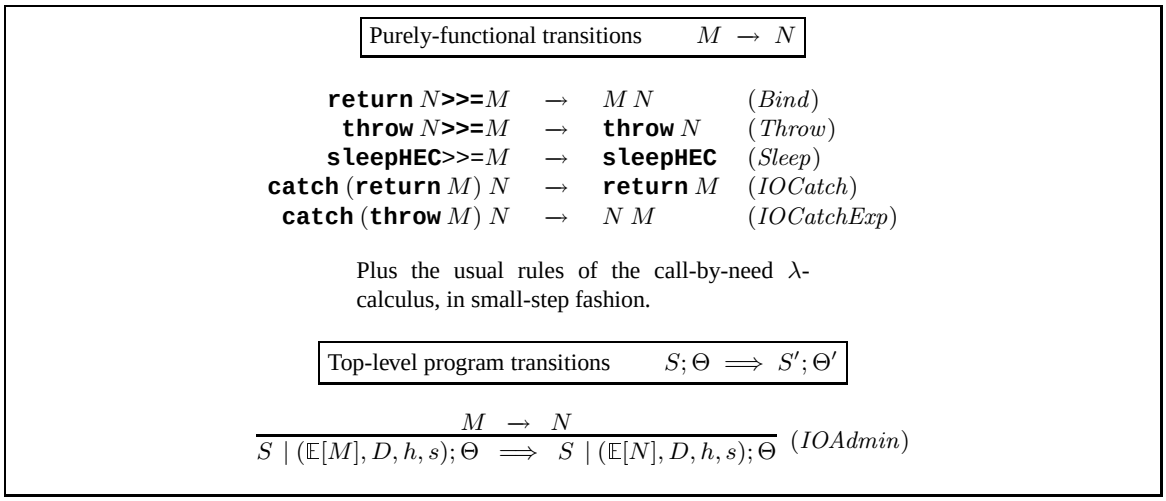

Figure 4.4: Operational semantics (basic transitions)

where in the term the next reduction must take place. A *purely-functional transition* includes β-reduction, arithmetic, **case** expressions and so on, which are not shown in Figure 4.4. However, the figure does show the purely-functional rules that involve the monadic operators **return**, **(>>=)**, **catch**, and **throw**. Notice also that a HEC in the *sleeping* state or the *outcall* state never takes a $(IOAdmin)$ transition.

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

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

- The identifier of the OS thread responsible for the HEC.
- An OS condition variable, used to allow the HEC to go sleep and be woken up later.
- Registers of the STG machine (the Haskell runtime model used in GHC).
- The current Haskell execution stack.
- The current heap allocation area; each HEC allocates in a separate area to avoid bottlenecking on the allocator.

- A "remembered set" for the garbage collector. It is important for performance reasons that the generational garbage-collector's write barrier is lock-free, so a per-HEC remembered set is used. It is benign for an object to be in multiple remembered sets.

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

4.3.2 Primitive transactional memory (PTM)

Since a program has multiple HECs, each perhaps executing on a different CPU, the substrate must provide a safe way for the HECs to communicate and synchronize with each other. The standard way to do so, and the one directly supported by most operating systems, is to use locks and other forms of low-level synchronization such as condition variables. However, while locks provide good performance, they are notoriously difficult to use. In particular, program modules written using locks are difficult to *compose* easily and correctly [26].

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

Here is how one might realize this pattern in Haskell:

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

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

These observations motivated us to seek an alternative synchronization mechanism. One such alternative is *transactional memory* (TM), which is known to offer a more robust and modular basis for concurrency [26]. There is a dilemma, however: the fully-featured *software transactional memory* (STM) implemented in the existing GHC system supports blocking, and cannot therefore be part of the substrate because thread blocking is now supposed to be implemented in the concurrency library itself, rather than in the runtime system.

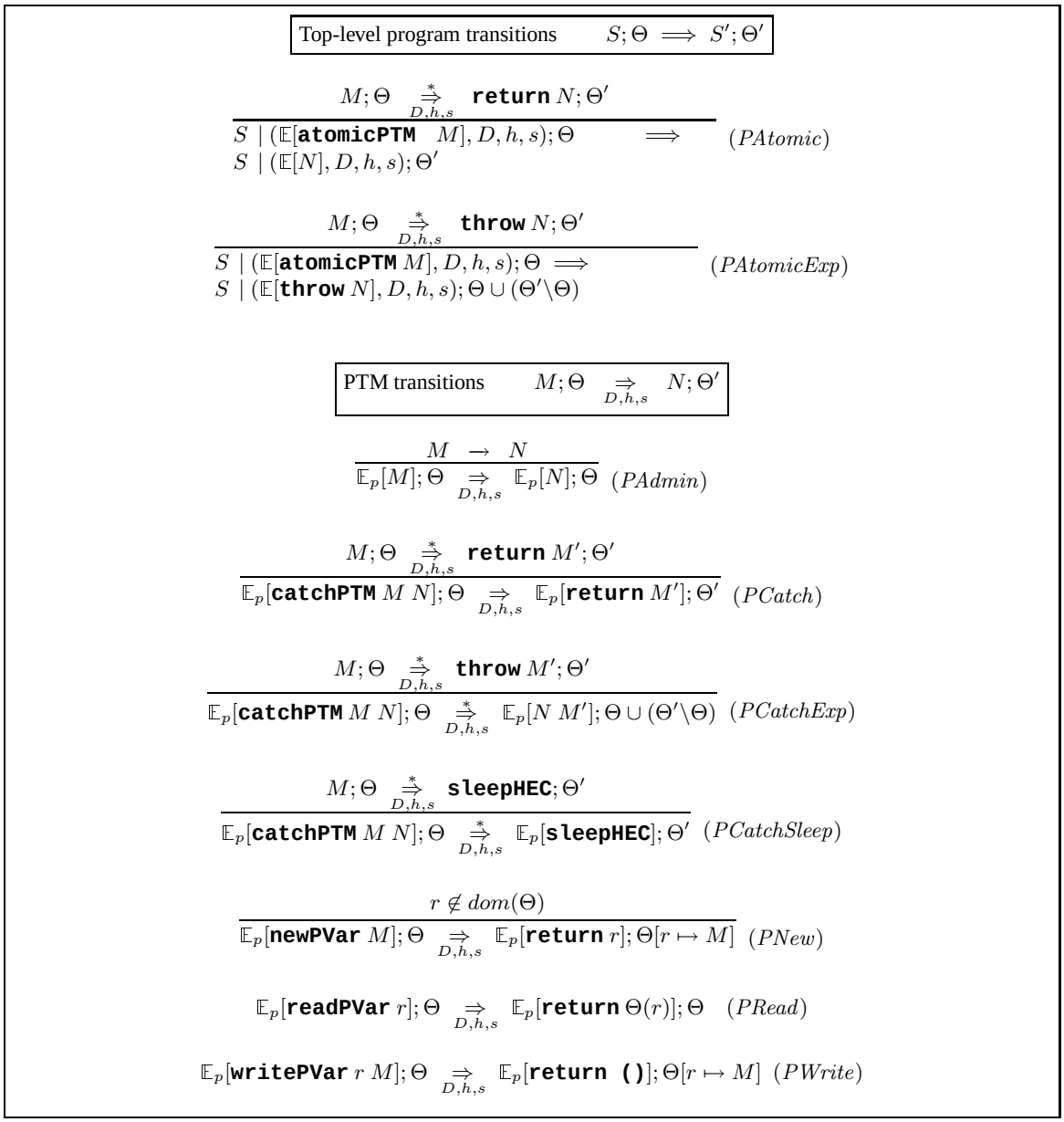

Figure 4.5: Operational semantics (PTM transitions)

Fortunately, all we require in terms of low-level synchronization is the ability to perform atomic transactions; the composable blocking and choice operators provided by STM can be safely omitted. Therefore, the substrate offers an interface called *primitive transactional memory* (PTM) [1], whose type signature is shown in Figure 4.2. Like STM, PTM is a monad, and its computations are fully compositional. Unlike STM, however, the basic PTM design does not have the **`retry`** primitive, so the question of blocking threads does not arise.

As Figure 4.2 shows, a PTM transaction may allocate, read, and write transactional variables of type **`PVar a`**. And that is about all, exceptions aside. Thus, a PTM transaction amounts to little more than an atomic multi-word read/modify/write operation. In operational terms, **`atomicPTM`** runs a PTM computation while buffering the reads and writes in a transaction log, and then commits the log all at once. If read-write conflicts are detected at the time of commit, the transaction is re-executed immediately. A good implementation of PTM should guarantee forward progress.

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

The semantics of PTM

Figure 4.5 presents the semantics of PTM. A PTM transition takes the form

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

The term M is, as usual, the current monadic term under evaluation. The heap Θ gives the mapping from **`PVar`** locations r to values M (Figure 4.3). The subscript D, h, s on the arrow says that these transitions are carried out by the HEC h, with stack-local state D and stack identifier s. Stack-local state will be discussed in Section 4.3.5, stack identifiers will be discussed in Section 4.3.4, and D, s can be ignored until then.

The PTM transitions in Figure 4.5 are quite conventional. Rule $(PAdmin)$ is just like $(IOAdmin)$ in Figure 4.4. The three rules for **`PVar`**s — $(PNew), (PRead)$, and $(PWrite)$ — allow one to allocate, read, and write a **`PVar`**.

The semantics of exceptions is a little more interesting. In particular, $(PCatchExp)$ explains that if M

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

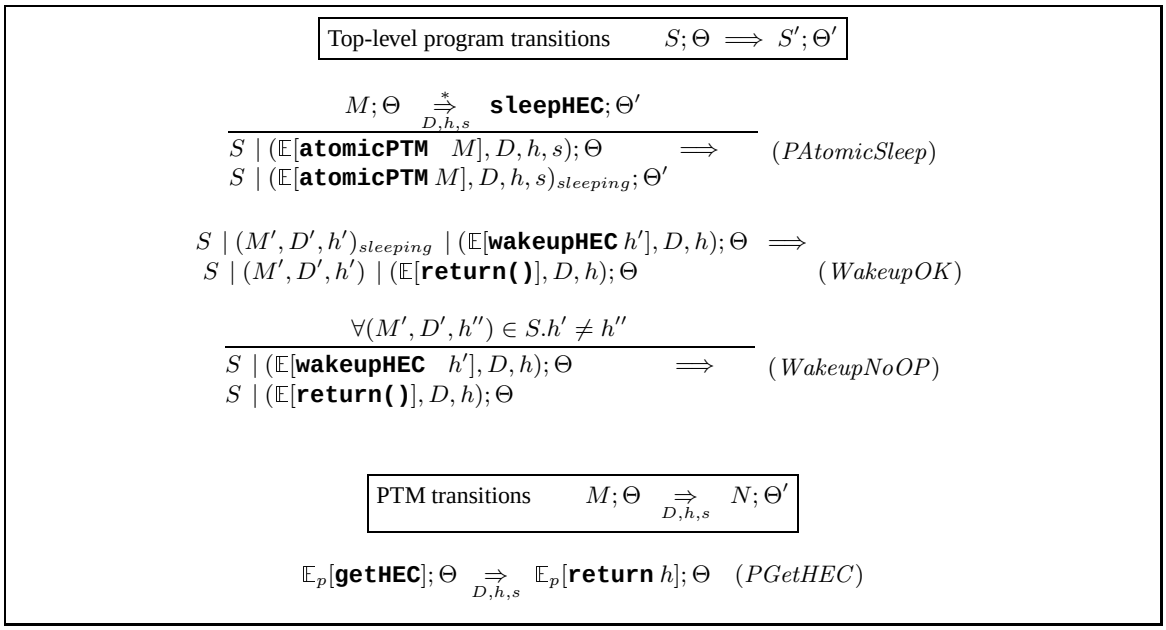

Figure 4.6: Operational semantics (HEC blocking)

throws and exception, then *the effects of M are undone*. To a first approximation that means simply that we abandon the modified Θ', reverting to Θ, but with one wrinkle: any **PVar**s allocated by M must be retained, for reasons discussed by [26]. The heap $\Theta' \backslash \Theta$ is that part of Θ' whose domain is not in Θ.

The rules for **atomicPTM** in Figure 4.5 link the PTM transitions to the top-level IO transitions. The $(PAtomic)$ rule embodies the key idea, that *multiple* PTM transitions are combined into a *single* program transition. In this way, no HEC can observe another HEC half-way though a PTM operation.

4.3.3 HEC blocking

A PTM transaction allows a HEC safe access to mutable shared states between HECs. But what if a HEC wants to block? For example, suppose there are four HECs running, but the Haskell program has only one thread, so that there is nothing for the other three HECs to do. They could busy-wait, but that would be a poor choice if a HEC was mapped to an operating system thread in a multi-user machine, or in a power-conscious setting. Instead, we want some way for a HEC to *block*.

The common requirement is that we want to block a HEC until some conditions are met, for example, when tasks become available. Traditionally, such code is often implemented using *condition variables*, which themselves need to be protected using locks. Since PTM is now used instead of locks, a *transactional* interface, **sleepHEC**, is designed to perform blocking based on condition testing. The semantics is shown in Figure 4.6.

```
sleepHEC   :: PTM a
```

```
wakeupHEC  :: HEC -> IO ()
```

The **sleepHEC** operation commits the transaction half-way and puts the HEC to sleep *at the same time*.

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

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

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

Note that this design of **sleepHEC** makes the PTM semantics less composable: when combining two PTM computations into a larger one, the programmer should be aware that a **sleepHEC** between their execution could commit the computation half-way and lose the all-or-nothing semantics. The programmer should always reason about transactions containing **sleepHEC** from the top-level. A good way to wrap up **sleepHEC** is to use the following **waitCond** function, which runs a transaction that contains no **sleepHEC**, and only calls **sleepHEC** at the top level:

```
waitCond :: PTM (Maybe a) -> IO a
waitCond action = atomicPTM $ do
  result <- action
  case result of
    Nothing -> sleepHEC
    Just x -> return x
```

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

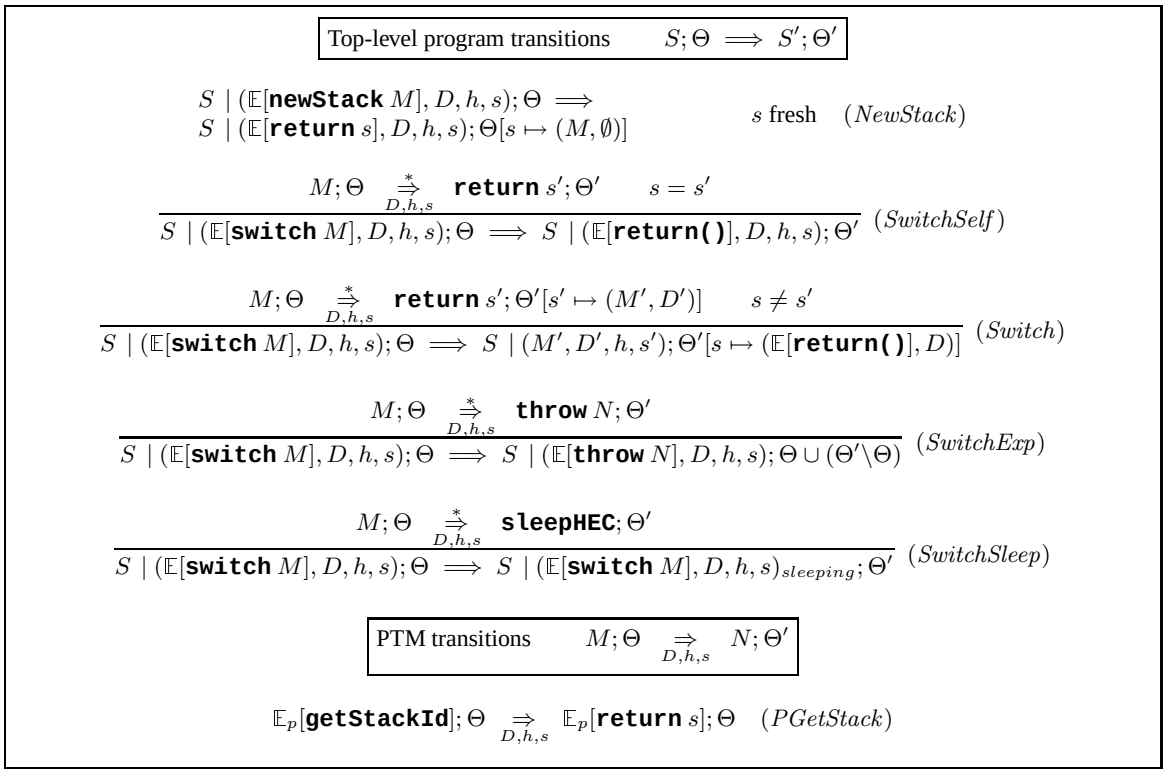

Figure 4.7: Operational semantics (stack continuations and context switching)

4.3.4 Stack switching

A HEC is an abstraction of a virtual processor; in a given system it is expected to have a handful of HECs running, roughly one for each physical CPU. To model fine-grain Haskell threads, we need an abstraction of a Haskell computation, together with a way to allow a HEC to multiplex its resources over such computations. Here we simply gives a unique identifier to one such computation. Becasue the run-time representation of such a computation is mostly encapsulated in a stack, the identifier is called a *stack identifier*.

One new data type and several new primitive operations are provided (Figure 4.2):

```
data StackId
newStack   :: IO () -> IO StackId
getStackId :: PTM StackId
switch     :: PTM StackId -> IO ()
```

An **StackId**, or *stack identifier*, should be thought of as a pointer to an execution stack, representing an I/O-performing Haskell computation that is either executing or suspended in mid-execution. The call **(newStack io)** makes a new **StackId** that, when scheduled, will perform the action **io**. The current stack identifier can be obtained by calling **getStackId**. The primitive **switch** is the interesting part. The call (**switch** M) does the following:

- It runs the primitive transaction M. This transaction may read and write some **`PVar`**s — for example, it may write **`s`** into a ready-queue — before returning an **`StackId`**, say s'. We call s' the *switch target*.

- Then, the transaction is committed. If the commit is unsuccessful, the transaction is restarted. Otherwise, the following context switching happens *at the same time* of commit: **`switch`** makes s' into the computation that the current HEC executes; The old computation on the current HEC is suspended.

These steps are made precise by the rules of Figure 4.7. An **`StackId`** is represented by a *stack identifier*, s. The heap Θ maps a stack identifier to a pair (M, D) where M is the term representing the suspended computation, and D is its stack-local state. Again, the discussion of the stack-local state is deferred until Section 4.3.5. Rule $(NewStack)$ simply allocates a new **`StackId`** in the heap, returning its identifier s.

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

Figure 4.7 describes precisely how **`switch`** behaves if its argument throws an exception: the **`switch`** is abandoned with no effect (allocation aside). It also describes how **`switch`** works with **`sleepHEC`**: the transaction is put to sleep and re-executed when waken up later.

Using stack switching

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

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

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

Implementing stack switching

By design, **StackId**s have a particularly cheap representation. In GHC, a Haskell computation runs on a stack, which itself is held in a *stack object* allocated in the run-time heap. Initially the stack object is small, but it can grow by being copied into a larger area if it overflows. An **StackId** is represented simply by a pointer to the stack object for its stack.

This work is similar to the idea of identifying stacks with second-class continuations [19]. However, the **switch** primitive deals rather neatly with a tiresome and non-obvious problem. Consider the call

```
switch (do {s<-getStackId; stuff})
```

The computation **stuff** must run on *some* stack, and it's convenient and conventional for it to run on the current stack. But suppose **stuff** writes **s** into a mutable variable (the ready queue, say) and then, while **stuff** is still running, another HEC picks up **s** and tries to run it. Disaster. Two HECs are running two different computations on the same stack. Fisher and Reppy recognized this problem and solved it by putting a flag on **s** saying "I can't run yet", and arranging that any HEC that picks up **s** would busy-wait until the flag is reset, which is done by **switch** when **stuff** finishes [19]. The current GHC runtime deals with this by ensuring that there is always a lock that prevents the thread from being rescheduled until the switch has finished, and arranging to release the lock as the very last operation before switching - again this is fragile, and has been a rich source of bugs in the current GHC RTS implementation.

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

The other error one must be careful of is when a stack is the target of more than one **switch** — no two HECs should ever run on the same stack at the same time. To check for this error one need a status bit in the runtime representation of a stack object to indicate whether a stack is under execution or not. Such bits must be checked and flipped *at the same time* a context switch is being made.

4.3.5 Global and stack-local states

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

- What is my thread identifier?

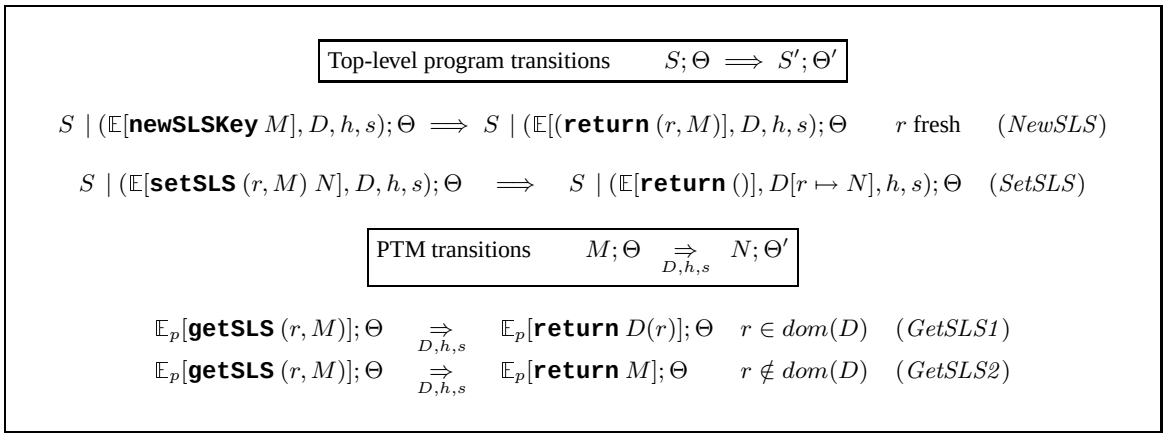

Figure 4.8: Operational semantics (stack-local state transitions)

- Where is my scheduler's data structure (for example, the ready queue)?

The code in Section 4.3.4 gives a more concrete example, in which the scheduler's data structure needs to be located. In principle there is nothing to prevent one adding a **`ThreadId`** parameter to every function that needs to know the thread identifier; and similarly for the other cases like the scheduler's task queues. However, doing so is extremely inconvenient and non-modular. In Haskell, one is already, in effect, passing the state of the world to every (effectful) function via the monad, and one would like all other state-passing to be implicit.

Global state

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

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

This is quite fragile due to the unsafe nature of the code, and the whole issue of accommodating effectful but benign top-level computations in Haskell has been frequently and heatedly discussed on the Haskell mailing list.[2] For the purposes of this dissertation it is simply assumed that *some* decent solution is available, so that one can write something like this:

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

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

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

Stack-local states

If thread identifiers are the only local states that is needed for each stack, perhaps the **`StackId`** itself is sufficient for this use. However, there can be more places where the concurrency library needs to maintain *thread-local* states. Such examples include: the seed for a random number generator (sharing a global one is a concurrency bottleneck); the **`stdin`** and **`stdout`** handles; and so on.

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

Thus motivated, the substrate supports *stack-local states* (SLS) directly, using the following design shown in Figure 4.2:

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

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

The detailed semantics are given in Figure 4.8. Several points are worth noticing:

- A stack is represented by a pair (M, D) of a term M to be evaluated and a *dictionary* D that maps SLS keys to values (Figure 4.3).
- A running HEC (M, D, h, s) includes the dictionary of the running computation. When **`switch`** switches to a new computation, it loads its dictionary into the HEC (rule $(Switch)$ in Figure 4.7).

- The **`newStack`** primitive makes a new stack whose dictionary is empty (rule $(NewStack)$).

- The **`newSLSKey`** primitive takes an *initial value* as its first argument, and a SLS Key is represented by a pair (r, M) of a unique identifier r and the the initial value M. Typically there will be a handful of SLS keys (**`stdin`**, the current scheduler, the random-number seed), but many stack continuations each with a potentially-different set of bindings for the keys. The SLS keys would usually be globally allocated; for example:

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

- If **`getSLS`** is given a key (r, M) whose identifier r is not present in the dictionary for the current computation, it returns the initial value M. This eliminates the necessity to initialize the dictionary with a binding for every SLS that could possibly be used.

- Stack-local state is manipulated only by the computation that owns it, and hence does not need to be transacted. Hence **`getSLS`** is a PTM operation, because it is convenient to be able to read it during a PTM transaction, while **`setSLS`** is an IO operation because one do not want the complication of having to undo **`setSLS`** operations if the transaction aborts. Note that **`setSLS`** operations are expected to be fairly rare.

- If the programmer wants to manipulate *shared* state accessed via the SLS mechanism, or to treat SLS state transactionally, the right thing to do is to make the SLS value a **`PVar`** and access it using PTM transactions.

In implementation terms, the identifier r of a SLS key (r, M) can be just a small integer, and the dictionary can be an array of slots in the stack object. Some overflow mechanism is needed for when there are more than a handful of SLS keys in use. Although not shown in the formal semantics, it is worth noting that the runtime system should automatically garbage-collect unused stack-local states: a stack and its local state are deallocated at the same time. An implementation is not required to reclaim unused SLS key values because such values are supposed to be globally-shared constants, and we do not expect there to be many of them.

HEC-local states?

One might naively expect the substrate to support *HEC-local* states as well. A HEC could use local state to maintain its own scheduling data structures, such as task queues. But, in reality, such structures are almost always globally shared by all HECs so that load can be balanced using work stealing algorithms. In such

cases global states are often more suitable. Also, HEC-local states are not useful outside the concurrency library because HEC is a low-level concept only available in the concurrency library. In contrast, stack-local states have broader applications: end-users can use them as *thread-local* states without much change.

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

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

4.4 Preemption, foreign calls, and asynchrony

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

Figure 4.9: The concurrency library callbacks

In addition to the substrate primitives shown in the previous section, the substrate interface also includes some callback functions, shown in Figure 4.9. These are functions supplied by the concurrency library, that are invoked by the RTS[3].

An external IO transition, $S; \Theta \stackrel{a}{\Longrightarrow} S'; \Theta'$, is an IO transition tied to an action a; see Figure 4.10. The actions include FFI in-calls, timer events and blocking events.

4.4.1 Preemption

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

The RTS substrate maintains a timer that ticks every 50ms by default. When a timer event is detected, the RTS substrate calls a timer handler function **`timerHandler`** exported by the concurrency library.[4] (This

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

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

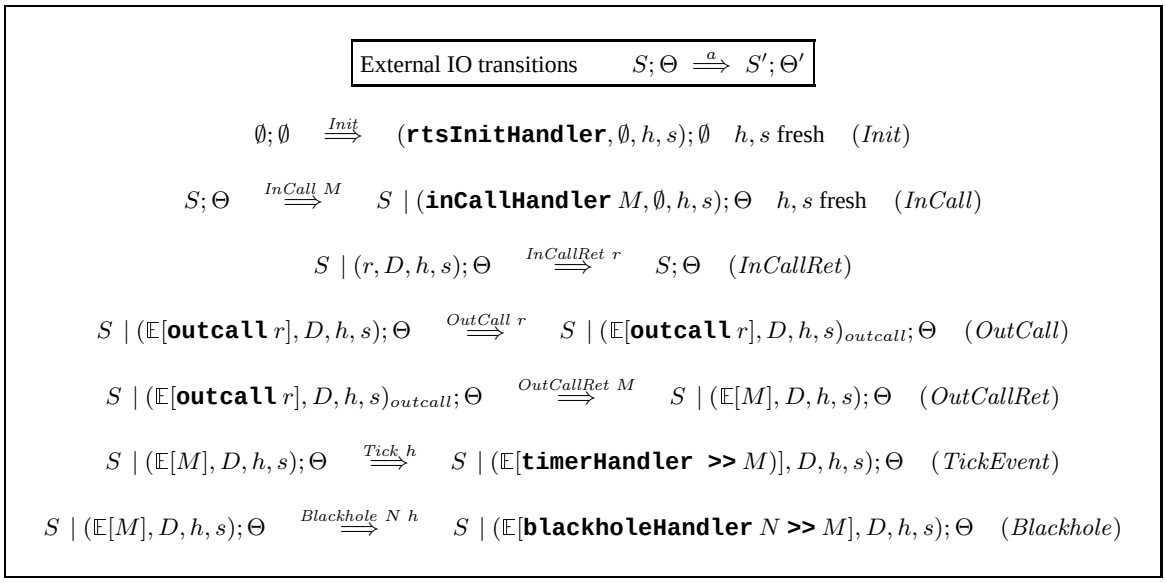

Figure 4.10: Operational semantics (external interactions)

is the first time that the RTS calls the concurrency library; most of the calls work the other way around. Figure 4.9 summarizes all the call-backs that will be discuss later.)

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

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

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

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

An alternative is to provide the following additional operations to read/write another stack's local states.

Clearly, these operations are quite dangerous and they should only be used in a principled manner, for example, only when the other stack is guaranteed to be *not* running, or only when stacks are initialized.

```
getHisSLS :: SLSKey a -> StackId -> PTM a
setHisSLS :: SLSKey a -> StackId -> a -> IO ()
```

4.4.2 Interrupting execution at thunks

In principle, any attempt to evaluate a thunk may see a *blackhole* because the thunk is already being evaluated by another thread [25]. If a blackhole is found, the best general policy is to pause the current thread until evaluation the thunk has completed (or at any rate until there is reason to believe that it *may* have completed). This implies that thunk evaluation sometimes needs to interact with the scheduler. In the old RTS design, the scheduler is built into the RTS, so it is easy to implement this policy. In the new design, however, implementing this policy requires a delicate communication between the substrate (which alone can detect when a thread evaluates a thunk that is already under evaluation) and the library (which alone can perform context switching and blocking of threads).

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

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

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

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

4.4.3 Asynchronous exceptions

One would like to implement asynchronous exceptions [44] in the concurrency library. Asynchronous exceptions are introduced by the **`throwTo`** operation:

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

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

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

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

2. the target thread actually raises the exception.

Fortunately, only step (2) absolutely requires specialized substrate support, namely a single operation, given earlier in Figure 4.2:

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

The **`raiseAsync`** function raises an exception in the context of the current thread, but in a special way: any thunk evaluations currently under way will be suspended [58] rather than simply terminated as they would be by a normal, synchronous exception. If the suspended thunk is ever forced later, evaluation can be restarted without loss of work.

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

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

4.4.4 Foreign calls

Foreign calls and concurrency interact in delightfully subtle ways [43]. It boils down to the following requirements:

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

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

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

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

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

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

The in-call handler is the entry point of the concurrency library: the schedulers accept jobs from the in-call handler. In a standalone Haskell program, the RTS makes an in-call to **`Main.main`** after the concurrency library is initialized (Section 4.4.5).

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

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

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

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

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

For **unsafe** calls, the compiler implementation can choose to bypass the out-call handler to improve performance.

This design leaves one corner case to consider: what if the concurrency library wants to make a blocking foreign call *inside* the **outCallHandler**? Clearly, this could easily enter an infinite loop. A solution is to add one more tag such as "**blocking**" in addition to "**safe**" and "**unsafe**", so that the compiler will generate the appropriate code for performing a blocking call, but **outCallHandler** will not be triggered in such cases.

4.4.5 Initialization handler

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

4.5 Developing concurrency libraries

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

The substrate design suggests that the concurrency library should be written in a cooperative fashion (although preemption is supported). A stack represents the runtime state of a Haskell thread. Threads can be created using **newStack** and make context switches to each other. Thread-local information, such as thread identifiers, can be implemented straightforwardly using stack-local states.

The interesting question is how to design the scheduler. Naively, the simplest scheduler can consist of a globally shared data structure with some common procedures, such as adding a new thread, switching to the next thread, blocking and unblocking, etc. However, the scheduler can be quite complicated when many concurrency features are implemented. The rest of this section presents the design of a simplest scheduler and leave more sophisticated designs for later chapters of this dissertation.

4.5.1 A simple concurrency library

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

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

To make an context switch by voluntarily yielding control, one uses the **`switch`** primitive together with a PTM transaction:

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

To support preemptive scheduling, one can simply set the timer handler to be **`yield`**:

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

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

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

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

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

Figure 4.11: Implementation of takeMVar

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

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

Figure 4.11 shows how to implement **`takeMVar`**; the **`putMVar`** operation is the dual case. A pending read request is implemented using a temporary **`PVar`**. If the **`MVar`** is empty, the current thread will be blocked, but a function closure is created to unblock the current thread later. If the **`MVar`** is full and there are additional threads waiting to write to the **`MVar`**, one of them is unblocked by executing its corresponding closure.

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

4.6 Related work

The stack switching design is similar to the idea of implementing concurrency using lightweight continuations [61, 62, 72]. There are two different strategies for implementing lightweight continuations in a language: first-class continuations can be made cheap for CPS-based runtime implementations [6] such as SML/NJ, and one-shot continuations [14] are more suitable for stack-based implementations. Our design is similar to the latter because the GHC runtime model is stack-based. However, we completely abandoned the concept of "continuation" here and chose a low-level, **`StackId`**-based design for several reasons:

- The continuations do not need to consume return values: they can be implemented using mutable references as shown in the **`MVar`** example in Figure 4.11.
- The type system cannot guarantee a linear use of continuations: calling the same continuation twice would result in disaster; the runtime system must check these errors dynamically. In terms of safety, a continuation-based design is no safer than the **`StackId`**-based design.

In short, a **`StackId`** can be thought of as a continuation: it is just that such continuations does not accept return values and the continuations captured on the same stack will all have the same runtime representation.

There are many related work in the field of continuation-based concurrency. Morrisett and Tolmach[47] extended continuation-based concurrency for SML/NJ to multiprocessors, by adding primitives types and operations for virtual processors and synchronization. The Sting language [30, 29] is a variant of Scheme that supports multiple parallel-programming constructs in a unified framework, which includes threads and virtual processors as primitive types.

Fisher and Reppy designed BOL as a compiler intermediate language to implement concurrency mechanisms [19]. They observed the problem as mentioned in Section 4.3.4, that a continuation shall not be used until the current thread has yielded control. BOL solves this problem by locking the current thread before publishing the continuation; the design here of the **`switch`** primitive solves this problem by combining context switching with a memory transaction.

The recent Manticore project [20, 21, 56] is similar to this work. The Manticore language is specifically designed to develop low-level runtime frameworks that support heterogeneous parallelism and complex scheduling policies. Manticore is based on a strict, ML-like language design; the design here uses the Haskell language itself, which is pure and lazy, and also deals with special problems in Haskell such as thunk blackholing. The HEC abstraction is similar to Manticore's notion of a *vproc* (virtual processor). Manticore uses the *compare-and-swap* operation and concurrent queues as synchronization primitives. In contrast, our substrate supports the higher-level notion of *transactional memory*. On the other hand, the Manticore substrate supports load-balancing and migration across vprocs, whereas our approach handles these entirely within the library.

Lastly, Berthold et. al. designed a run-time environment for implicitly parallel programs, using Concurrent Haskell and the existing GHC runtime system as a substrate [11].

4.7 Summary

This chapter presented a generic Haskell runtime system design to support lightweight concurrency. The new design uses transactional memory as the synchronization primitive and simple stack switching to implement lightweight concurrency.

The new design is aimed for many goals. For the purpose of this dissertation, it provides a flexible foundation for building high-performance network applications, but it also simplifies the GHC runtime system and improves the safety and customizability of Haskell concurrency implementation.

The next chapter shows the implementation details of the new runtime system prototype and compares the performance of various concurrency configurations based on the implementation.

Chapter 5

Implementation and evaluation of the new runtime system design

This chapter shows the prototype implementation of the new GHC runtime system design as well as two concurrency libraries. Then, six different concurrency configurations are presented and compared using I/O benchmark programs.

5.1 Prototype implementation of the new runtime system

This section describes a prototype implementation of the new runtime system design. The prototype is developed by modifying the current GHC 6.8 runtime system. The current GHC RTS has approximately 90,000 lines of code; the new RTS prototype implementation has approximately 80,000 lines of code; the differential patch size between the two RTS is approximately 20,000 lines of code.[1]

The following sections describes the implemented features individually.

5.1.1 Haskell execution context (HEC)

In contrast with the existing GHC RTS, the most significant change of the new RTS design is the concept of a *Haskell execution context* (HEC), which is the abstraction for an OS thread. First, I review how the existing GHC RTS uses OS threads.

The existing GHC RTS is essentially an user-level thread system. Internally, the RTS uses a small number of OS threads, called *tasks* in the GHC RTS jargon, to execute Haskell programs. There is a fixed number of tokens floating around at run time; such tokens are called *capabilities* and each represents a virtual

[1] The statistics includes blank lines and comments.

processor that can be used to run user-level threads. The total number of capabilities can be specified using a RTS flag "**-N**". At run time, some tasks are used as virtual processors; each such task holds a capability and runs Haskell computations. The rest of the tasks do not hold capabilities; they are used as a pool of OS threads to execute blocking foreign out-calls.

The new RTS design pulls most of the above details out of the RTS: a HEC is essentially a *task* in the existing RTS, but it is now accessible by the Haskell programmer. The concept of a *capability* is now entirely outside the RTS — it belongs to the concurrency library instead. From the RTS's point of view, all HECs execute Haskell computations. Each HEC in the new RTS is basically a task holding a capability in the existing RTS.

To implement the HEC, I removed the definitions of tasks and capabilities as well as most code related to scheduling. I then added a data structure **HEC** (defined in **HEC.h**) that can be thought of as the combination of a **Task** and a **Capability**. Each **HEC** has the following fields:

- machine states for the running Haskell computation: registers, stack, memory allocation area, mutable lists;
- locks, flags and condition variables for synchronizing the HECs;
- miscellaneous flags and pointers to link all the HECs together on a list;
- fields used by primitive transactional memory (explained in later sections).

All the HECs are chained onto a linked list called **all_hecs**. Dead HECs are marked by a special flag field and are recycled during garbage collection.

The main control flow of the RTS is redesigned to support programming with HECs. HECs are created by foreign in-calls made from the C world to the Haskell world. At initialization time, the RTS makes an in-call to the **rtsInitHandler** Haskell function. After the RTS is initialized, the entry code of the RTS makes another in-call to the **main** Haskell function. All other HECs are created in the Haskell world by the concurrency library.

The life cycle of a HEC and a foreign in-call is specified in the **process_in_call** method in **HEC.c**. It allocates a new **HEC** data structure, initializes its fields, inserts it into the **all_hecs** list, runs a Haskell computation and finally marks the **HEC** as dead. All these steps are executed in the same OS thread.

The HEC implementation has plenty of details such as synchronization, memory allocation and garbage collection. The following sections discuss these details.

5.1.2 Primitive transactional memory

The new RTS design uses *primitive transactional memory* (PTM) as the primary mechanism to synchronize among HECs. Because PTM is essentially a stripped-down version of the existing *software transactional memory* (STM) in GHC, I reused (yet simplified) most of the STM code to implement PTM. Each HEC has its own PTM transaction records. Such records are part of the **`HEC`** data structure.

Compared to STM in the existing RTS, the major change in the low-level PTM implementation is the use of locks. In the existing GHC RTS, there is only a fixed number (usually equal to the number of physical processors in the system) of OS threads executing Haskell computations, so the STM implementation uses *spin locks* to get the best performance. In the new RTS, however, there can be thousands of HECs running PTM transactions at the same time, so spin locks is an bad idea and standard pthread locks should be used instead. During development, my experiment shows that, when the number of concurrent transactions exceeds the number of physical processors in the systems, the performance of spin-lock-based PTM is two orders of magnitudes lower than that of pthread-lock-based PTM. When the number of concurrent transactions is smaller than the number of physical processors, however, spin-lock-based PTM is about 30 percent faster than pthread-lock-based PTM. Overall, the pthread lock seems to be the right choice.

The STM has two implementations in the existing GHC RTS: coarse-grained locking and fine-grained locking. Because I used pthread locks as the low-level locking mechanism, I only implemented the coarse-grained version for simplicity because the fine-grained locking version requires more low-level control over locks that is not straightforward to implement with pthread locks.

Compared to an ideal PTM implementation, this prototype has a few limitations. First, it does not efficiently support re-entrance of PTM transactions. If a PTM transaction is running, a timer handler is triggered and the timer handler starts another PTM transaction, the outer PTM transaction will be invalidated. A better implementation would save the old transaction record upon invoking the new transaction and restore the old transaction record when the new transaction is finished. Another limitation is that it currently does not properly support exception catching and partial roll-back inside PTM transactions. Finally, each HEC's transaction records have a fixed size hard-coded in the RTS. In a better implementation, the transaction records should be made resizable at run time.

5.1.3 Stacks, context switching and local states

The GHC runtime system uses a *thread stack object* (TSO) to encapsulate most running states of a Haskell thread. The new RTS design uses exactly the same design, except that stacks are now made explicit to the programmer — each stack is represented as a **`StackId`** in the RTS concurrency primitives. This design brings some changes to the RTS implementation.

First, stacks are now managed in the concurrency library, not in the RTS itself. The existing GHC RTS uses a number of linked lists to chain the thread stack objects together in the scheduler. For the purpose of memory management, such links are treated as "weak pointers" during garbage collection — the collector must traverse through these links carefully and update them as needed. In the new design, most of such lists are eliminated because schedulers are no longer part of the RTS. Instead, each **`StackId`** is implemented as a pointer to the TSO, so the garbage collection code becomes cleaner because there are no special lists to traverse — TSOs can now be garbage collected just like other memory objects.

Second, the new RTS must implement the **`switch`** primitive to perform context switching of stacks. The implementation of **`switch`** combines two operations together: committing a PTM transaction and performing a context switch. The following sequence is used at time of commit:

1. The target stack is located. Because stack objects can be resized and relocated in the GHC RTS, finding the target stack object requires traversing through a link until the live TSO is reached.

2. On the current stack, rollback to the frame that starts the transaction.

3. Save the state (relevant machine registers) of the current Haskell computation into the stack object. The current stack object is now detached from the current HEC and it is ready to be fetched by another HEC.

4. Try to commit the transaction. If the commit is successful, load the machine state from the target stack and keep running. Otherwise, restart the transaction on the current stack.

The new design also supports *stack local states*, which is readily built into the TSO. Currently, the RTS prototype only supports a finite number of local states, so the local states is simply implemented as an fix-sized array in each TSO. In the future, a better implementation would use a resizable array instead. A global array is used to store the default values for local states.

5.1.4 HEC blocking and garbage collection

Garbage collection (GC) is single-threaded in the GHC runtime system. To perform GC, all the OS threads used in the RTS must work in a cooperative fashion so that they all stop running Haskell code before GC is entered. Once GC is finished, they all resume to continue their work. The most tricky question is how these OS threads collaborate to initiate the GC process.

In the existing GHC RTS, whichever *task* that runs out of memory initiates the GC by fetching all the *capability* tokens in the system. Once all the tokens are available, it is guaranteed that no Haskell computation is running and GC can proceed.

The new RTS design completely changes the story because there is no capabilities floating around in the RTS. Each HEC runs a Haskell computation and there can be a large number of HECs. The story is further complicated because a HEC can go to sleep or make foreign out-calls that can potentially block the HEC indefinitely. During the development of the new RTS, I developed two algorithms for GC collaboration.

GC collaboration using a global counter

In early stages of the prototype development, I designed a GC collaboration algorithm that uses a global counter and two condition variables. The global counter represents the number of HECs that are actively running in the RTS and it is protected by a main lock in the RTS. The two condition variables are called **`gc_cond_master`** and **`gc_cond_slaves`**, all protected by the main lock. The HEC that initializes GC is the GC master — it sets a global GC flag **`gc_is_running`** and waits on the condition variable **`gc_cond_master`** until the active HEC counter goes to zero. All other HECs detect the global GC flag **`gc_is_running`** by periodical pooling at preemption points, decrement the global HEC counter and wait on the condition variable **`gc_cond_slaves`**, signaling the GC master when the HEC counter reaches zero. After the GC master completes GC, it signals all the slaves to go back to work.

HEC blocking is handled as following:

1. When a HEC wants to block itself indefinitely by calling **`sleepHEC`** or performing a blocking foreign out-call, the HEC puts its own data structure in a state that is ready to be garbage-collected and decrements the global HEC counter before the blocking operation. If a GC is in process, it also signals the GC master if the counter reaches zero.

2. While the HEC is being blocked, it is not counted in the global HEC counter, so other HECs can perform garbage collections freely.

3. After returning from the blocking operation, the HEC increments the global HEC counter, waiting on **`gc_cond_slave`** if a GC is still in process.

I used this design until I discovered that it leads to poor performance for foreign out-calls on multiprocessor machines. The reason is that each time a potentially blocking foreign call is being made, the global RTS lock must be fetched and released twice, before and after the foreign out-call. This global RTS lock then becomes the bottleneck for making foreign out-calls on a multi-processor system.

Because the goal of this RTS prototype is to test its performance in network applications, the performance of foreign out-calls is critical given that most of the I/O operations are accessed using out-calls. To overcome this performance bottleneck, I designed another algorithm for GC collaboration, shown in the next subsection.

GC collaboration using fine-grained locking

To optimize the performance for foreign out-calls on multiprocessor machines, the out-call operation must not use any global lock — doing so will create a performance bottleneck. Nevertheless, a HEC has to synchronize with other HECs in *some* way to support collaborative GC.

My solution uses fine-grained locks for GC collaboration. Each HEC has a lock called **`running`**. The HEC must hold this lock when running Haskell computations. There is also a global lock called **`gc_lock`**. When a HEC wants to do GC, it releases its own **`running`** lock and grabs the global **`gc_lock`**. Once the **`gc_lock`** is held, the current HEC becomes the GC master. The GC master HEC then traverses through the **`all_hecs`** list and grabs the **`running`** lock on *all* HECs in the RTS. After GC, the GC master releases the **`running`** lock on all HECs, grabs its own **`running`** lock and resumes execution.

This algorithm assumes that other running HECs (called GC slaves) have a way to detect when GC is happening. This can be done either by periodic polling at preemption points, or by using the memory allocation mechanism I will describe later in Section 5.1.5. Each GC slave enters the GC routine in exactly the same way that the GC master does, except that when they successfully acquire the **`gc_lock`** (immediately after the GC master releases the **`gc_lock`**), the GC has already finished. Of course, it would be a waste for each GC slave to repeat the GC again, so I added a flag **`need_gc`** in each HEC. Before GC, the **`need_gc`** flag is set to **`true`**. The main GC routine is called only when **`need_gc`** is true, and when the main GC routine finishes, the GC master clears the **`need_gc`** flag on all HECs. Thus, the GC slaves will see that **`need_gc`** is **`false`** and skip the GC.

A blocking operation such as an out-call or sleeping is now cheap to implement: the HEC only needs to release its own **`running`** lock before blocking and acquire this lock after blocking. In fact, this is exactly how foreign out-calls are implemented in the new RTS. HEC sleeping is more complex and I show it in the next subsection.

HEC sleeping and waking up

The new RTS design has two primitives for blocking HEC synchronization: **`sleepHEC`** and **`wakeupHEC`**. These operations are implemented using a lock **`sleep_lock`** and a condition variable **`sleep_cond`** on each HEC. When a HEC goes to sleep, it waits on the condition variable **`sleep_cond`**, whose access is protected by the lock **`sleep_lock`**.

The tricky part is that **`sleepHEC`** combines two actions together: committing the transaction and blocking the HEC. A correct implementation should not miss the wake-ups as in the following scenario:

1. When calling **`sleepHEC`**, the PTM transaction is committed and the current HEC identifier h is written in some data structure, say, a queue.

2. Another HEC sees h in the queue, retrieves it from the queue and wake it up using **`wakeupHEC`**.

3. The HEC h goes to sleep. It will sleep forever because h has already been removed in the queue and no one else can wake it up.

What makes the story even more complex is the **`running`** lock I described in the previous subsection: each HEC has its own **`running`** lock that must be released before sleeping and acquired after sleeping. The interaction of all these locks must be arranged carefully, or deadlock will happen.

Perhaps the easiest way to describe **`sleepHEC`** is to use the following pseudo code:

```
ACQUIRE_LOCK(&(hec->sleep_lock));
if (stmCommitTransaction(hec)) {
  RELEASE_LOCK(&hec->running_lock);
  waitCondition(&(hec->sleep_cond), &(hec->sleep_lock));
  ACQUIRE_LOCK(&hec->running_lock);
  dirtyTSO(hec->r.rCurrentTSO);
}
RELEASE_LOCK(&(hec->sleep_lock));
```

The **`wakeupHEC`** operation is implemented by adding a buffer to each transaction record that stores a list of HECs to be woken up. Then, after the transaction is committed, all the HECs in the list are woken up using the following pseudo code:

```
for h in wakeup_queue do
   ACQUIRE_LOCK( &(h->sleep_lock));
   signalCondition( &(h->sleep_cond));
   RELEASE_LOCK( &(h->sleep_lock));
```

5.1.5 Optimizing memory management

One design goal of the new RTS is to support a large number (e.g. 10,000) of OS threads on network applications, so the concurrency library has the option to implement one Haskell thread as exactly one OS thread and still support massively concurrency on systems like Linux, which has efficient OS thread support. This poses the biggest implementation challenge for the new RTS because the existing GHC RTS was not designed with such a requirement in mind, particularly on memory management.

Block-based memory allocation

Figure 5.1 shows the layout of memory allocation area in the existing GHC RTS. Historically, the GHC RTS was first a single-threaded system and the main memory allocation area is called the *nursery* in the RTS

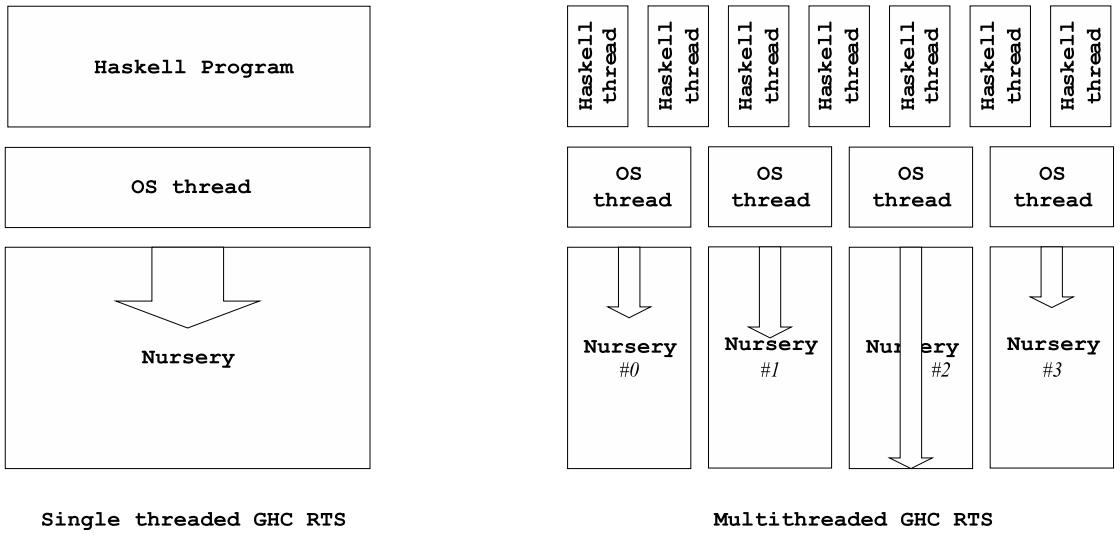

Figure 5.1: Nursery management in the existing GHC RTS

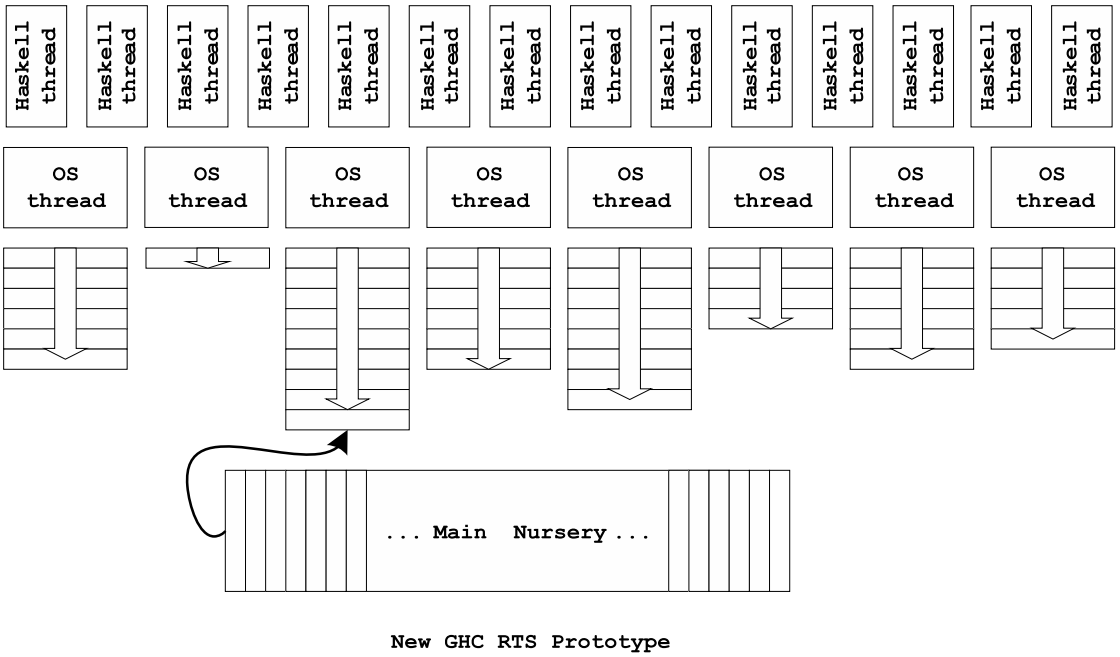

Figure 5.2: Nursery management in the new GHC RTS

jargon. When the nursery is full, garbage is collected, the nursery is then evacuated and reset to the empty state.

This design became more complex when the recent GHC RTS was extended with multiprocessor support: the RTS now maintains N nurseries of identical sizes where N is the number of processors. The memory layout is shown on the right side of Figure 5.1. Each capability, or virtual processor in the existing GHC RTS, runs on one of these nurseries as its private memory allocation area. As soon as one of the nurseries is full, the entire RTS is garbage collected; all the nurseries are then evacuated and reset to the same size again.

Therefore, this design does not make the best use of all nurseries and there can be a sigificant waste of memory when the allocation speed is unequal in all allocation areas. However, it makes it easy to adapt the existing GHC RTS code into a multiprocessor setting. On today's systems, N is not a large number — maybe four to eight, so it is still not too bad. However, when there are tens of thousands of HECs in the new RTS and each HEC requires its own memory allocation area, the existing nursery management scheme is clearly not usable.

The new RTS uses a different memory allocation strategy, as shown in Figure 5.2. All the HECs share a global allocation area which is also called the main nursery. The main nursery consists of a list of small memory blocks; each small block has a few kilobytes. Each HEC also has its own private allocation area that initially has only one block. Whenever a HEC's private allocation area is used up, it tries to fetch a new block from the main nursery and extend the current HEC's private allocation area. This fetching operation is protected by a lock. When there is nothing to fetch, the HEC enters garbage collection. The GC routine then recycles each HEC's private allocation area and return them to the main nursery list.

The advantage of this strategy is that there is no waste of memory — all blocks are used up before GC is entered. Furthermore, there is no need for the HECs to perform periodic polling for collaborative GC, because when the nursery is used up, the running HECs will all enter GC at nearly the same time[2].

With this new memory management strategy, the new RTS efficiently supports up to tens of thousands of HECs — the practical limitation is now on the scalability of the underlying OS threads, rather than on the RTS itself.

Finally, it is worth noting that this implementation technique is mostly independent of the new RTS design — it can be applied to the existing GHC RTS as well.

[2]Unlike in many languages, most Haskell programs makes frequent allocations, so when the heap allocation area is full, the execution of a Haskell program will usually not go very far — it will stop quickly and ask for GC.

Shrinking the C stack size

When tens of thousands of OS threads are used, virtual memory space becomes a scarce resource especially on 32-bit systems, because each OS thread has a reserved virtual memory space for the C calling stack that is usually several megabytes long. The GHC runtime system does not make extensive use of the C stack in most places when executing Haskell code, so the default C stack size can be set to be as small as a few kilobytes to save the virtual memory space. Nevertheless, garbage collection is an exception: the GC algorithm can recurse deeply into nested data structures and it must use a deep stack. Because any HEC can become the GC master, one cannot predict how much C stack space a HEC will use when a HEC is created.

I solved this problem by using a dedicated OS thread as the "GC worker" thread in the new RTS prototype. This GC worker thread has a deep C stack, is mostly idle and only becomes active when GC is entered. The GC master HEC communicates with the GC worker thread and let it perform the GC using its own C stack. Thus, all HECs can use a small C stack space (the current setting used is 64KB for safety and ease of debugging) by default, without worrying about C stack overflows in GC.

5.1.6 Preemption and thunk blocking

Preemption is implemented in a way similar to the existing GHC RTS. Each HEC has a boolean flag called **`ctx_switch`**. A timer routine in the RTS is triggered every once a while and it sets the **`ctx_switch`** flag to **`true`** in all HECs. This flag is checked whenever a allocation block is used up in the HEC's private allocation area. If the flag is detected, it is reset to **`false`** and the timer handler is triggered.

The timer handler is launched by delicately setting up a Haskell thread's stack in a way as if the **`timerHandler`** function is called from the current computation. When **`timerHandler`** returns, the original Haskell computation on the current stack continues as if nothing has happened. This is a low-level hack but it works reliably in the GHC RTS.

I also implemented the **`blackholeHandler`** that works similarly when a blackhole is entered due to concurrent thunk evaluation. The existing GHC RTS uses a sophisticated lock-free algorithm called "lazy blackholing" to evaluate and update thunks. In the unlikely event that a thunk is entered by two concurrent HECs at the same time and runs for a long time, one of the HEC shall have its computation suspended (by converting its stack frames into heap closures) and rolled back to the point before entering the thunk. All these mechanisms still work in the new RTS.

The only caveat in the current RTS prototype is that the **`blackholeHandler`** is not implemented in exactly the same way as described in Section 4.4.2. It has the type **`IO()`** and it does not take an argument function that can be used to test the status of the thunk.

```
blackholeHandler :: IO ()
```

Nevertheless, I will show that it is sufficient to build usable concurrency libraries using this simple blackhole handler. The next section shows two concurrency library designs based on the new GHC RTS prototype implementation.

5.2 Developing concurrency libraries

The concurrency library is written as a component of the **`base`** GHC library package, which serves as the root in the GHC library hierarchy. In my current prototype implementation, the modules related to concurrency in the **`base`** package are organized as following:

- **`Substrate.hs`**: This module wraps the lower-level concurrency primitives in the module **`GHC.Prim`** into the higher-level substrate primitives shown in Figure 4.2.

- **`ConcLib.hs`**: This module implements most of the concurrency library features, such as **`forkIO`**, **`MVar`**, thread schedulers and various callback functions. The location of this module is hard-coded into the runtime system, so the runtime system knows where to find the callback functions such as the timer handler, in-call handler, etc.

- **`Control.Concurrent`**: In the existing GHC library, these modules are implemented using the concurrency primitives in **`GHC.Prim`**. In my new prototype implementation, these modules use **`ConcLib.hs`** instead.

- **`System.IO`**: A significant portion of the IO modules needs to be rewritten for each concurrency library implementation. This would be a lot of work, so I did not change most of them. Instead, each application program that uses the concurrency library prototype comes with its own IO module that only implements the features that are necessary, such as opening, reading, writing and closing files and sockets. This setup suffices for the purpose of performance benchmarking.

To switch from one concurrency library to another, the programmer needs to link the application program with a different **`base`** library package and to use a different IO module.

I developed two concurrency libraries. The first library **`ConcLib_OS`** is a thin wrapper over OS threads: each Haskell thread is implemented using one unique OS thread; Haskell threads and the underlying OS threads have a one-to-one mapping. The second library **`ConcLib_MP`** implements $M : N$ user-level threads; Haskell threads are multiplexed onto a small, fixed number of OS threads.

5.2.1 The OS thread library

The **ConcLib_OS** concurrency library uses one HEC as one Haskell thread. The implementation is fairly straightforward — it does not use many lightweight concurrency features in the new RTS, such as stacks, context switching and local states. The RTS initialization handler, timer handler, in-call handler and out-call handler all do nothing. A thread identifier is implemented using the **HEC** type. The **forkIO** operation creates a new HEC. The **yield** operation makes a foreign call to **sched_yield()**, the OS thread primitive for yielding.

A non-trivial question is how to implement the blackhole handler. In my current implementation, this handler simply makes a OS thread yield. The resulting effect is that threads blocked on blackholes will periodically attempt to re-evaluate the thunk and yield repeatedly until the thunk has been updated. This strategy is similar to the one used in the existing GHC RTS, except that the concurrency library developer has no control on the frequency of retries — it is determined by the underlying OS thread implementation. However, because OS thread yielding takes little time to execute, the retry frequency can hardly make a difference for most programs.

The MVar is implemented using **sleepHEC** and **wakeupHEC**. The type **MVar** is defined as following — it is a mutable PVar cell that contains the state of the MVar (the **Maybe** part in the definition) and two queues. The first queue contains the threads blocked on the reading end of the MVar; the other is for threads blocked on the writing end.

```
newtype MVar a = MVar (PVar (Maybe a, Queue HEC, Queue HEC)) deriving (Eq)
```

The implementation of **takeMVar**, shown in Figure 5.3, demonstrates the intended use of **sleepHEC** and **wakeupHEC**. The **putMVar** operation is the dual case and thus omitted.

I/O operations are trivial to implement: they work directly using the Foreign Function Interface (FFI) with no special handling in the concurrency library. All the details of performing such foreign calls are handled in the RTS. The programming model is the same as using POSIX threads in C.

5.2.2 The user-level thread library

The **ConcLib_MP** concurrency library multiplexes lightweight Haskell threads onto a fixed number of OS threads. Overall, this concurrency library prototype is based on a cooperative design (however it does support preemption); its runtime configuration is exactly the same as shown in Figure 3.11 earlier in Chapter 3, excluding the TCP stack in the figure. Each event loop in Figure 3.11 is implemented using a HEC. Lightweight Haskell threads are represented using their stack identifiers (of type **StackId**). Task queues are implemented using globally shared PVars. The HECs synchronize with each other using PTM transactions, **sleepHEC** and **wakeupHEC**, similar to the MVar implementation in Figure 5.3.

```
newMVar :: a -> IO (MVar a)
newMVar x = do
  ref <- newPVarIO ((Just x), emptyQueue, emptyQueue)
  return $ MVar ref

takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = atomically $ do
    (st, takeQ, putQ) <- readPVar ref
    case st of
     Just x -> do
       putQ' <- case dequeue putQ of
                  Nothing -> return emptyQueue
                  Just (hd, tl) -> do
                     wakeupHEC hd
                     return tl
       writePVar ref $ (Nothing, takeQ, putQ')
       return x
     Nothing -> do
       h <- getHEC
       writePVar ref $ (Nothing, enqueue takeQ h, putQ)
       sleepHEC
```

Figure 5.3: Implementation of MVar operations in the OS thread library

The RTS initialization handler allocates the task queues and creates all the HECs in the user-level thread system. A fixed number of HECs are used as virtual processors to execute Haskell computations. A pool of HECs are allocated to execute blocking foreign out-calls. Dedicated HECs are created to monitor asynchronous/nonblocking IO mechanisms such as epoll and AIO in Linux. After initialization, all the worker HECs are idle and waiting for tasks.

Then, the RTS makes an in-call to the **`main`** function in the Haskell world. The in-call handler creates a new Haskell thread, feeds it into the task queue and waits for the thread to terminate.

The HECs used as virtual processors are the engines of the user-level thread system. They work in the same way as an event loop: fetch a task, run the task for some time and then loop for the next task. The virtual processor HEC (and all other HECs in the concurrency library) has its own initial stack called the *scheduler stack*; each Haskell thread also has its own stack called the *task stack*. To run a task, the virtual processor HEC makes a context switch to the task stack. Control is transferred back from the task stack to the scheduler stack when the thread yields control, makes a blocking foreign call, gets blocked or preempted.

The concurrency library uses stack local states to store some of the scheduling information. One state is used to distinguish scheduler stacks and task stacks; another state is used by task stacks to store information about its virtual processor HEC, in particular the identifier of the scheduler stack so that it knows where to yield control when a context switch needs to be made.

The **`forkIO`** operation creates a Haskell thread. It calls the **`newStack`** primitive to create the new stack,

initializes its local states and puts it on the run queue. The action of adding a task stack to the run queue, however, is not trivial, because it involves synchronization of HECs. The run queue has a list of idle HECs in it — whenever a task is added to the run queue and the idle list is not empty, a HEC is woken up. On the other hand, when a virtual processor HEC runs out of tasks, it adds itself to the idle list and goes to sleep using **`sleepHEC`**.

The timer handler first examines the stack local state to see where it is running. Nothing is done on a scheduler stack, but if it is running on a task stack, it preempts the current Haskell thread by calling **`yield`**, which puts the current stack identifier into the run queue and transfers control back to the virtual processor HEC's scheduler stack.

The blackhole handler does the similar operation as the timer handler: when a thread is blocked on a thunk, it yields. However, in the unlikely case that the current stack is a scheduler stack, it cannot simply do nothing. In this case, the OS thread yield is called instead just like in the OS thread library.

The MVar is implemented similarly as in Figure 4.11. The only difference is that, in order to perform a context switch, control is transferred to the virtual processor's stack directly without fetching the next task.

The out-call handler puts the current thread into the task queue used by the HEC pool for performing blocking foreign calls. When the call is finished, the thread is put back to the main run queue.

The concurrency library also supports asynchronous I/O mechanisms such as epoll and AIO in Linux. For epoll, it provides the following interface:

```
epoll_waitfd :: SockFD -> EPOLL_EVENT -> IO ()
```

When a Haskell thread calls **`epoll_waitfd`**, it is blocked until the anticipated **`EPOLL_EVENT`** is detected on the file descriptor. The epoll event is usually one of the following: (i) the file descriptor becomes readable, and (ii) the file descriptor becomes writable. The implementation of **`epoll_waitfd`** registers the epoll event with the stack identifier, then yields control to the scheduler stack. When the epoll event is detected by the epoll monitoring HEC, the stack identifier is put back into the main run queue. AIO is implemented similarly.

All the above features are implemented in **`ConcLib.hs`** as part of the **`base`** Haskell library package. Besides, each application is also attached with an I/O module that provides higher-level, blocking I/O interfaces that encapsulate the non-blocking I/O primitives such as **`epoll_waitfd`**. This setup is for doing experiments quickly without a major overhaul of **`System.IO`**. Ideally, the modules under **`System.IO`** should be modified instead.

5.3 Concurrency configurations

The new runtime system prototype makes it possible to develop network applications in many different concurrency configurations. I developed benchmark programs to evaluate and compare I/O performance under six configurations. Each configuration is explained below.

GHC threads (existing RTS)

The first configuration is the standard GHC 6.8.2 without any modification. Haskell threads are created using **`forkIO`**. Programs use the standard Haskell I/O library. Disk files are accessed using **`openBinaryFile`** and **`hSeek`**. Sockets are set to binary mode with no buffering. Input and output are performed using **`hGetBuf`** and **`hPutBuf`** on file handles.

The performance experiments in Chapter 3 did not include this configuration only because it has much worse I/O performance. Nevertheless, it is worth including it in a more comprehensive test — at least, it justifies the motivation of this dissertation.

C threads

The second configuration uses POSIX threads and C rather than Haskell. It is the same configuration used in the performance tests in Chapter 3. Each concurrent network connection is handled by an OS thread. Disk files are accessed using **`open`** and **`lseek`**. Input and output are performed using **`read`** and **`write`** on file descriptors directly using blocking I/O. This configuration is used as a frame of reference for all other Haskell configurations.

The the stack size of each thread is limited to 32KB. This limitation allows Linux threads to scale to a large number of threads.

Concurrency monads (existing RTS)

This configuration is exactly the same as used in Chapter 3. Each network connection is handled by an application-level monadic thread. The event scheduler runs on top of standard Haskell threads. The GHC RTS then internally maps these Haskell threads to OS threads. The program runs on the standard GHC 6.8.2 with unmodified runtime system.

The event-driven scheduling system uses epoll-based non-blocking I/O for sockets and pipes and uses Linux AIO for asynchronous disk access.

The next three configurations are based on the new RTS prototype.

Concurrency monads (new RTS)

This configuration is exactly the same as the previous one, except that it uses the new RTS prototype, together with the **`ConcLib_OS`** concurrency library. Each network connection is handled by an application-level monadic thread. The event scheduler runs on top of standard Haskell threads, each mapped to an OS thread by the **`ConcLib_OS`** concurrency library.

Compared to the existing RTS, the new RTS is less redundant and gives the application programmer more control on OS threads. For example, the programmer can specify which processors an OS thread can use, by using the **`pthread_setaffinity_np`** interface on Linux.

OS threads (new RTS)

This configuration uses the **`ConcLib_OS`** concurrency library directly. Each network connection is handled by one Haskell thread created using **`forkIO`**. The application program is attached with a small I/O module that uses the foreign function interface to access the blocking I/O functions in the C library, such as file and socket access. The I/O mechanisms are exactly the same as programming in C with POSIX threads using blocking I/O: disk files are accessed using **`open`** and **`lseek`**; input and output are performed using **`read`** and **`write`** on file descriptors directly using blocking I/O.

User-level threads (new RTS)

This configuration uses the **`ConcLib_MP`** concurrency library. Each network connection is still handled by one Haskell thread created using **`forkIO`**. The application program is attached with a small I/O module that uses the foreign function interface to access the I/O functions in the C library. Most I/O mechanisms such as socket operations are nonblocking or asynchronous, but a few others such as file opening are blocking and synchronous. Nonblocking and asynchronous operations are synchronized by the **`ConcLib_MP`** concurrency library, using primitives such as **`epoll_waitfd`**.

From the operating system's point of view, this configuration works in the same way as the concurrency monads do. However, the programmer now uses standard Haskell threads instead of user-defined, monadic threads.

5.4 Performance experiments

All the performance experiments are done on the same computer. The hardware setup is: Intel Core 2 Quad Q6000 processor 2.40GHz 1066MHz FSB with 8MB L2 cache, 4GB DDR2 667MHz non-ECC SDRAM and 250GB SATA 3.0Gb/s harddrive with 8MB DataBurst cache. The software setup is the default installation of Ubuntu Linux 7.04 x86_64 with kernel version 2.6.20-16-server SMP. This Linux distribution uses

the Native POSIX Thread Library (NPTL) by default. Swapping is disabled. All unnecessary services are shut down. The GHC version is 6.8.2.

The following two sections compare the I/O performance of the six concurrency configurations, using similar benchmarks to the ones used in Chapter 3.

5.4.1 Disk head scheduling

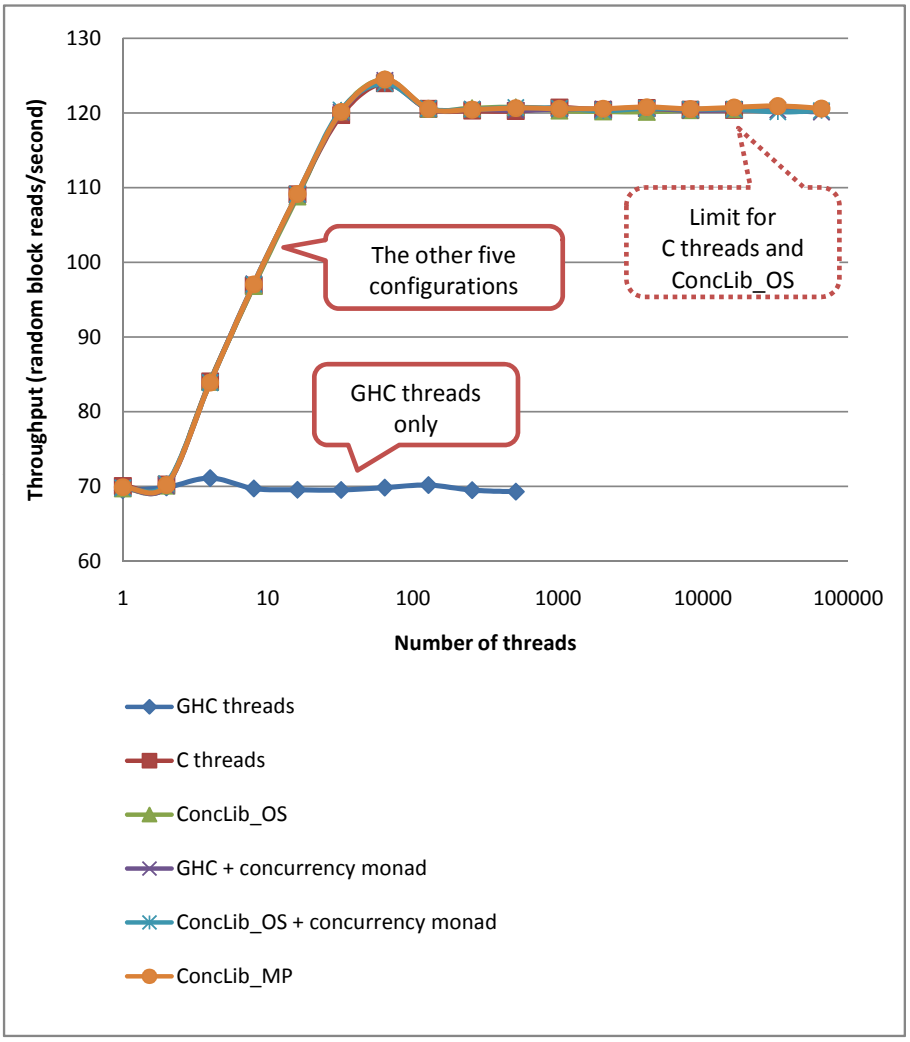

Figure 5.4: Disk head scheduling test

I repeated the disk head scheduling test used in Section 3.3 on the new test machine for all six configurations. The benchmark program launches a large number of threads. Each thread randomly reads 4KB blocks from a 128GB file opened using **`O_DIRECT`** without caching. Each test reads a total of 128MB data and the overall throughput is measured, using the median of 5 runs. The result is shown in Figure 5.4. To make the

results easy to understand, I converted the unit of throughput to the number of block reads per second.

This benchmark tests whether the multithreading system can take advantage of the disk head scheduling algorithms provided by the operating system. The overall throughout should increase with the number of concurrent disk accesses because less disk head movement is needed to fulfill an access request when more requests are pending. The throughput saturates at a certain point, which is roughly 64 to 128 threads in this test.

The result is similar to that obtained earlier in Chapter 3. All configurations behave similarly except the original GHC threads, which does not seem to benefit from concurrent disk accesses at all. The reason is that the GHC RTS uses nonblocking I/O operations to manage the I/O requests of user-level threads, yet it does not use any asynchronous disk I/O mechanism. In contrast, the concurrency monad configurations and **`ConcLib_MP`** all uses Linux AIO for disk accesses and do not have this problem. In addition, the existing GHC RTS cannot scale to 1024 or more threads. This is quite unfortunate, as the GHC RTS implements an sophisticated user-level thread system solely to improve performance.

Although the other five configurations are indistinguishable in the figure, it is worth noting that they have different limits: all the OS-thread-based configurations (C threads and **`ConcLib_OS`**) are tested up to 16K threads and the lightweight concurrency configurations (concurrency monads and **`ConcLib_MP`**) are tested up to 64K threads. These tests are mostly bound by memory; I will discuss these limits in more details in Section 5.4.3.

Compared to the earlier results in Figure 3.14, the OS threads perform better in this test, where a 64-bit operating system is used. In Figure 3.14, the C and NPTL line dropped significantly at 16K threads when the virtual memory is near exhaustion on a 32-bit system — I suspect that the increased memory footprint is the reason that made OS threads slow. In fact, the OS thread performance does start to drop in the test in Figure 5.4 when more than 16K threads are used, but the system I used does not support up to 32K threads.

The lightweight concurrency configurations perform best in this test. In fact, 64K threads is not the upper limit they can reach. I did not test further only because it would have taken much more time — each run could take several days to finish.

To summarize:

- The existing GHC RTS has poor disk head scheduling performance and does not scale to a large number of threads in I/O-intensive applications.
- The OS-thread-based configurations perform well on 64-bit Linux systems, but they still have a large memory footprint and do not support up to 32K threads.
- The lightweight concurrency configurations perform best in this test.

5.4.2 FIFO pipe with idle connections

I then repeated the FIFO pipe test with idle connections used in Chapter 3 on the new test machine for all six configurations. This test simulates network server applications where most connections are idle. The benchmark program uses M pairs of active threads to send and receive data over FIFO pipes. In each pair, one thread sends a data token to the other thread, receives a data token from the other thread and repeats the conversation. In addition to these actively working threads, there are many idle threads in the program waiting for data on idle FIFO pipes.

Unlike in the previous test used in Chapter 3, the token size is set to 1 byte instead of 4KB. The 4KB token size better simulates a practical network service, but the 1-byte token size could make the benchmark more stressful and reveal more minute differences on the results.

I ran this test using three different sets of parameters shown below.

GHC threads, one pair of active threads

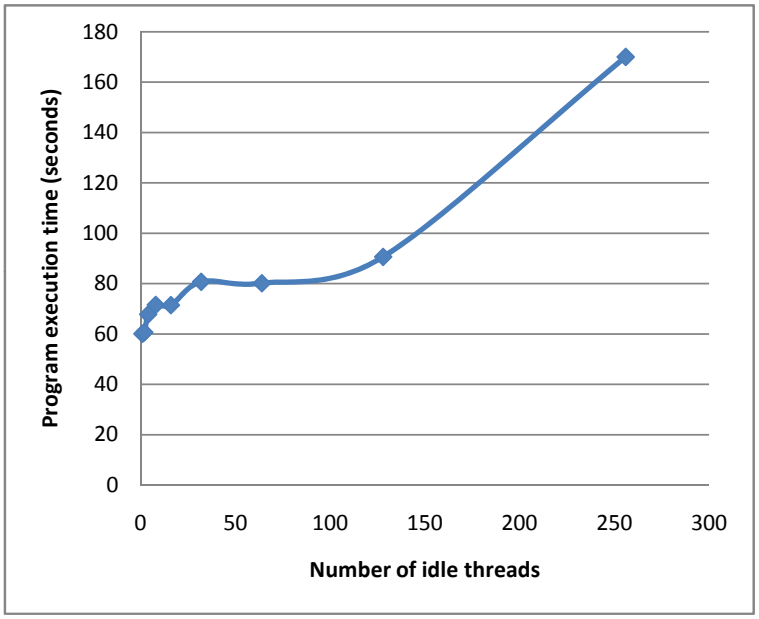

Figure 5.5: FIFO pipe with idle connections, one sender and one receiver, GHC threads only

The first test uses only one concurrency configuration, GHC threads, because its performance is much worse than all other configurations. In this test, there is only one pair of active threads sending and receiving data tokens, all other threads are idle. A total of 4M tokens are ping-ponged, and the total program execution time is recorded in Figure 5.5.

Although most threads are idle, the program execution time increases significantly as the number of idle threads increases. The problem is that the standard I/O library in GHC uses the portable "select" interface,

whose performance scales almost linearly with the number of file descriptors used. Furthermore, it does not support more than 1K file descriptors.

All configurations, one pair of active threads

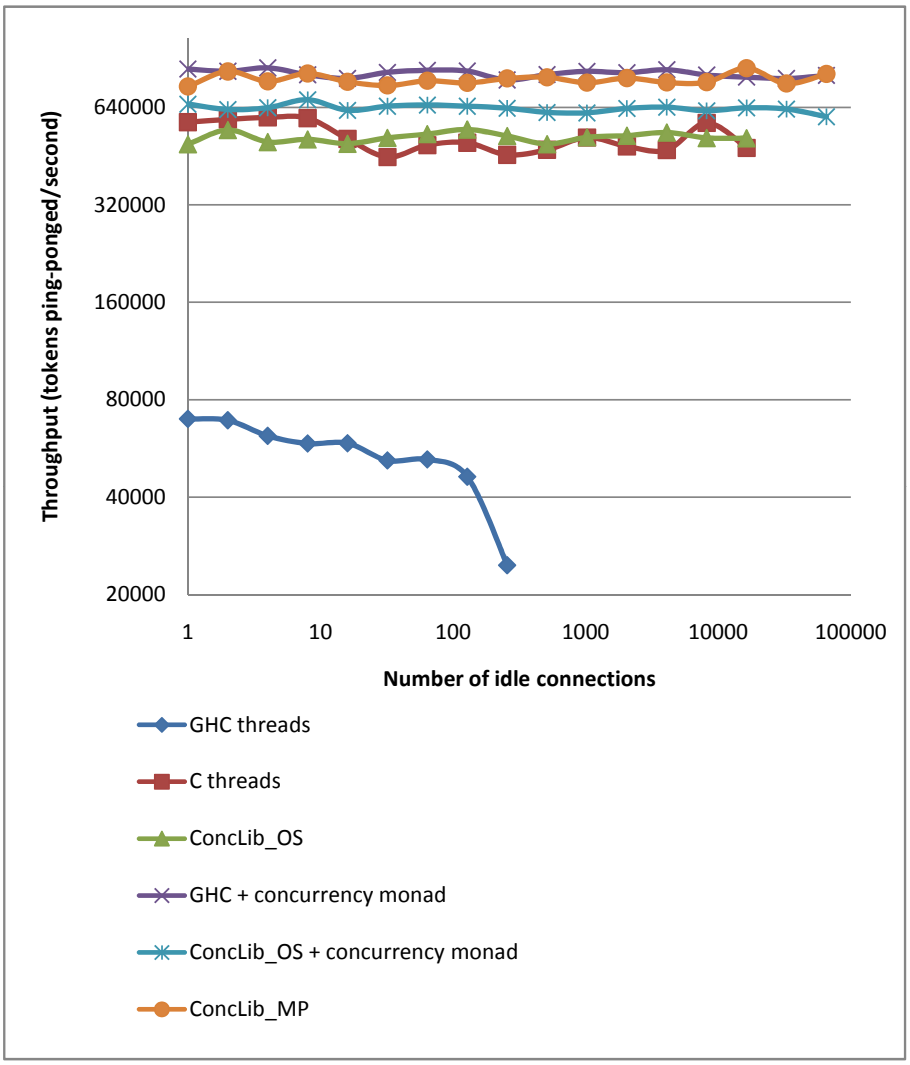

Figure 5.6: FIFO pipe with idle connections, one sender and one receiver

The second test puts all six configurations together and measures the *throughput*, the numbers of tokens ping-ponged per second in the system. Again, there is only one pair of active threads sending and receiving data tokens, all other threads are idle. A total of 32M tokens are ping-ponged, the total program execution time is recorded, and the throughput is calculated and shown in Figure 5.5. Each data point is ran 5 times and the median result is used.

Similar to the disk head scheduling test, all the OS-thread-based configurations are tested up to 16K

threads and all the lightweight concurrency configurations are tested up to 64K threads. All these configurations scales quite well with respect to idle connections; the differences among them are not significant.

The purpose of this test is to show a "easy" case before showing the next one. Because there is only one pair of threads, only *one* physical processor is kept busy during this test. The next test utilizes *all* processors in the system and shows different results.

All configurations, 128 pair of active threads

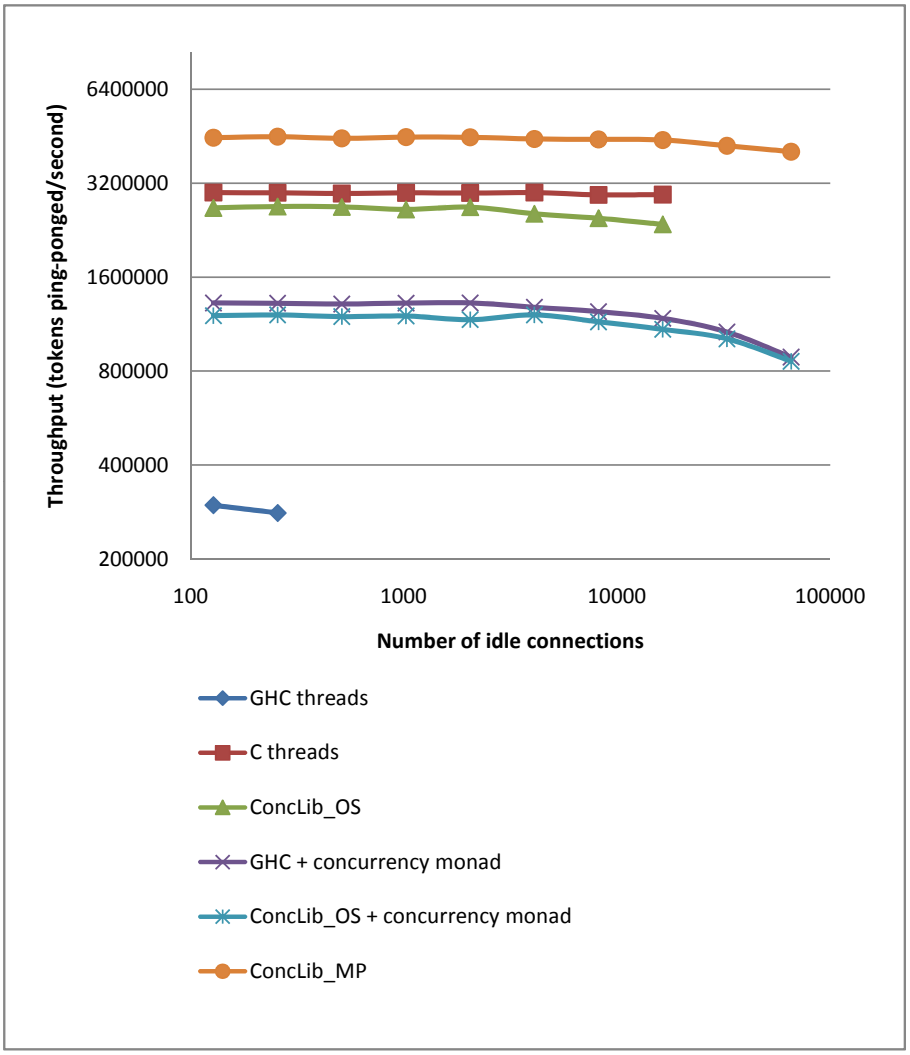

Figure 5.7: FIFO pipe with idle connections, 128 senders and 128 receivers

The third test uses exactly the same parameters as the previous one, except that there are 128 pairs of active threads. This test setup is similar to the test used in Figure 3.15. The result shown in Figure 5.7, however, is quite different.

The six configurations are clearly clustered in four groups, ordered from lower performance to higher performance:

1. The GHC threads has the lowest performance and scalability.

2. The two concurrency monad configurations are much lower than OS threads and **`ConcLib_MP`**. This is perhaps the biggest surprise in my benchmark programs — they worked better than OS threads in other tests.

3. The two configurations that use OS threads give nearly the same performance, but still significantly lower than the **`ConcLib_MP`**.

4. The **`ConcLib_MP`** gives the best performance.

Compared to the previous test (all configurations, one pair of active threads), the main difference is that the performance of concurrency monad configurations dropped significantly. I discovered the reason by profiling the benchmark programs. In the two concurrency monad configurations, garbage collection takes up to 40 percent of total execution time. In other configurations, however, garbage collection takes less than one percent of time and is hardly noticeable. This suggests that garbage collection is a bottleneck for concurrency monads on a multi-processor machine.

This observation is not surprising because GHC's garbage collector is single-threaded, both in the existing RTS and in the new RTS. A program that makes more allocations will have poorer performance scalability on multiprocessor GHC. Unfortunately, programs written in the concurrency monad do make frequent allocations — the concurrency monad essentially uses linked heap closures as an execution stack, so memory is allocated on every single step of a program written in the concurrency monad.

This performance difference is exaggerated by the fact that the new benchmark program in this chapter uses a 1-byte token size, in contrast to the 4KB token size used in the test in Chapter 3. The 1-byte token size ensures that the benchmark program spend as much time as possible on control flow and I/O scheduling rather than copying data, so the overheads of the concurrency monad are made most visible.

Conclusions:

- GHC threads again has poor scalability on this I/O benchmark.

- Garbage collection becomes a performance bottleneck for concurrency monads on multiprocessor machines, although it is a general problem for all Haskell programs that make frequent allocations.

- OS threads give near-best performance, although not the best.

- The lightweight concurrency library, **`ConcLib_MP`**, gives the best performance on this I/O benchmark.

5.4.3 Memory consumption

The following table shows the resource consumption and limits of different configurations, excluding the existing GHC. These numbers are obtained by running test programs that fork a large number of threads. The test program is a modified version of the disk head scheduling benchmark program used in Section 5.4.1 that loops without exiting. Then, the memory usage of the entire program is monitored using the **`top`** utility in Linux. When the memory usage becomes stable, the usage of virtual memory and resident memory is recorded, and the average memory consumption per thread is calculated.

configuration	max. number of threads	virtual memory per thread	resident memory per thread	buffer size per thread
C threads	less than 32K	49KB	16KB	4KB
`ConcLib_OS`	less than 32K	100KB	25KB	4KB
GHC + CPS monad	at least 256K	6KB	5KB	4KB
`ConcLib_OS` + CPS monad	at least 256K	6KB	5KB	4KB
`ConcLib_MP`	at least 256K	10KB	9KB	4KB

Table 5.1: Resource usage and limits of concurrency configurations

The table also shows the maximum number of threads each configuration can support on the test machine. Note that for the last three configurations, there is no hard bounds on the number of threads. I have been able to test them up to 256K threads. In real applications, the number of threads are bound by system resources such as memory, file descriptors, etc.

The rightmost column shows the amount of memory the application explicitly allocates in each thread: a 4KB buffer used to read disk files. One can see that the the concurrency monad is the most efficient: the per-thread memory consumption is not much more than the 4KB buffer itself. The **`ConcLib_MP`** concurrency library is slightly less efficient, because each thread has its own execution stack, which takes a few kilobytes when initialized and resized as needed.

The configurations based on OS threads are much less efficient. One difference between the two is that C threads use a 32KB stack size limit in the benchmark program, where as the new GHC RTS uses a 64KB stack size limit by default. It is possible to make the GHC RTS use a smaller stack when the debugging functions are disabled.

Chapter 6

Conclusion

This chapter summarizes the dissertation from several perspectives because the project spans over two research fields in computer science: it solves a *systems* problem using a *language-based* approach. Systems programmers, Haskell implementors and programming language designers may each find their own interests in this dissertation.

First, I will start from a *systems* perspective and discuss my conclusions for network programming: I have presented two approaches to combine threads and events in Haskell. How does the two approaches compare with each other?

Then, I will take the viewpoint of *language implementation* because the bulk of my work has been on designing and implementing a new runtime system for Haskell: How does the new runtime system compare with the existing one?

Finally, I will discuss from a *language design* perspective and summarize my experience with the Haskell language: What are the benefits and drawbacks of using Haskell for concurrent programming?

6.1 For system programmers

This dissertation tackles the problem of combining *threads* and *events*: instead of choosing one and giving up the other, my solutions provide interfaces to write programs in the multithreaded style, but also an interface (the "event" view) to control how threads are implemented and scheduled. As the result, the ease-of-use of threads and the performance of events are both obtained.

I have presented two approaches to develop such high-performance network applications in Haskell: the concurrency monad and the new runtime system design with lightweight concurrency primitives. Each of these approaches has its own benefits and drawbacks, as I compare in the following table. The first three

rows shows benefits of the concurrency monad; the last three rows shows the benefits of the new runtime system design.

	concurrency monad	new runtime system
works with current GHC?	yes	no
level of customization	application	standard library
per-thread memory cost	less than 100 bytes	a few kilobytes
computation speed	slow	fast
thread model	cooperative	preemptive
programming interface	CPS monad (non-standard)	IO monad (standard)

Table 6.1: Comparison of the concurrency monad and the new runtime system

Benefits of the concurrency monad Perhaps the best reason to use the concurrency monad is its *convenience*: it works with the current GHC right out of the box. It is a completely *application-level* solution, so no modification is required for the compiler, runtime system or the standard libraries. In contrast, the new runtime system design not only requires a major overhaul of the runtime system, but each concurrency configuration also requires a different standard base library to be developed. While such standard libraries can be made modular and reusable, the concurrency monad approach is more agile and lightweight to adapt to specific requirements of an application.

The other significant advantage of the concurrency monad is its *compactness* on thread representation. At run time, the state of each monadic thread is entirely represented as nested heap closures; it does not use a reserved stack area to store local states. As a result, each thread consumes a relatively small amount of memory (typically less than 100 bytes) other than the necessary thread-local states. This level of compactness makes the concurrency monad a good choice for network applications that deal with a massive amount of mostly-idle connections. In contrast, the new runtime system uses a reserved, resizable stack for each thread, which takes a few kilobytes at least.

Benefits of the new runtime system Although the standard threads implemented in the new runtime system is not as compact as the concurrency monad in terms of memory usage, it trades off memory compactness for *speed* on multiprocessor systems. The concurrency monad works like a small interpreter that monitors the program execution by manipulating its own data structures on ever step. In particular, such work involves extra memory allocations, which lead to poor performance on multiprocessor systems because the current GHC garbage collector is single-threaded. If GHC can have a better garbage collector that runs efficiently on multiprocessors in the future, perhaps the performance difference between the concurrency

monad and the new runtime system could be significantly reduced. Besides the problem on multiprocessors, the performance overheads of the concurrency monad and possible compiler optimizations for it still deserves further investigation.

Another benefit of the new runtime system is that it provides *preemptive* multithreading, where as the concurrency monad only provides cooperative multithreading. The latter can be particularly difficult to use in Haskell, because the programming interface is strongly typed and effectful operations are only restricted to the monads that permit them. The programmer cannot easily insert yielding operations at arbitrary places in the program; they can only do so at the top-level of the concurrency monad. If a long Haskell computation needs to be preempted in the middle of execution, it has to be written in the concurrency monad, which can be not only inconvenient, but also expensive due to the performance overheads of the concurrency monad.

With the new runtime system, the programmer can write multithreaded code in the the *standard* IO monad, rather than in a user-defined concurrency monad. Many Haskell libraries uses the IO monad as the default interface for writing effectful programs, so the new runtime system poses no obstacle for the programmer to write and use software libraries. With the concurrency monad, however, the library writer and the user have to agree on the definition of the concurrency monad, which is likely to be application-specific.

Summary The concurrency monad is good for *convenience, agility* and *compactness*. The threads provided by the new runtime system is *fast, preemptive* and has a *standard* interface.

It is worth noting that although each of the two approaches has its benefits and drawbacks, they are in some sense *orthogonal* to each other — the concurrency monad can be used seamlessly with the new runtime system and the OS thread library. The new runtime system does not eliminate the need for the concurrency monad but it only provides more choices. With the existing runtime system in GHC, the concurrency monad is the only option. With the new runtime system, however, the programmer can choose the configuration that best fits the task at hand.

6.2 For Haskell developers

This dissertation presents a new design of the Haskell runtime system (RTS) and its prototype implementation, so an important question from the Haskell community is how it compares with the existing RTS in GHC for general use. The question becomes more complex when different possible concurrency configurations for the new RTS are considered. The rest of this section compares the existing RTS with the new RTS and various concurrency configurations.

Should the GHC developers adopt the new runtime system design? I do not have a simple, cut-and-dried

	existing RTS (user-level threads)	new RTS + user-level thread lib.	new RTS + OS thread lib.
modular concurrency library	no	yes	yes
implementation and maintenance complexity	highest	moderate	lowest
operating system I/O interface	non-blocking I/O	asynchronous / non-blocking I/O	synchronous / blocking I/O (portable API)
Linux I/O performance	worst	best	near best
Linux I/O thread limit	1K	>256K	32K
MVar sync performance	fast	slow	slow
STM support	full	partial (PTM)	partial (PTM)

Table 6.2: Comparison of runtime systems and concurrency configurations

answer because the existing RTS and the new RTS each have their own advantages. The decision depends on how different factors are weighted. This section compares the existing RTS and the new RTS on a few aspects and discuss their trade-offs. For ease of presentation, the comparison is summarized in Table 6.2.

Implementation and maintenance complexity An important goal of the new RTS design is the *modularity* of concurrency implementation. The details of threads and schedulers are detached from the RTS and they are wrapped into a concurrency library written in Haskell. The concurrency library writer only needs to know a small, well-typed interface which is the lightweight concurrency primitives. No knowledge is required on the internal implementation details of the RTS. In contrast, modifying the concurrency implementation in the current GHC RTS requires expert knowledge on most RTS internal details, such as memory management, garbage collection, the execution model of Haskell programs and so on.

Because of modularity, the new RTS and the concurrency library together has less overall implementation complexity than the existing GHC RTS. The concurrency library is easy to access and maintain — developers who do not know the internal details of the RTS can still get hands on it.

Operating system I/O interface The existing GHC RTS implements a user-level thread system. It uses non-blocking I/O mechanisms to multiplex the I/O requests of lightweight Haskell threads onto a few OS threads. Most non-blocking I/O operations are implemented in the standard library written in Haskell. The RTS uses a dedicated OS thread, called the I/O manager, to synchronize I/O operations. The I/O manager is written in C and it is part of the RTS.

One problem with such asynchronous/non-blocking I/O interfaces is that they are usually platform-dependent: each operating system typically has its own interface for performing asynchronous/non-blocking I/O. For example: Linux has epoll for socket I/O and AIO for disk I/O; Sun Solaris uses a different AIO

interface than Linux does; Windows uses a mechanism called I/O completion ports. With all such differences, it is difficult to find an asynchronous I/O programming interface that is available on all platforms — multiple versions of the I/O manager have to be written to support asynchronous/non-blocking I/O on different platforms.

The other problem is that the implementation of asynchronous disk I/O in some operating systems is currently not as mature as synchronous I/O. For example, the existing Linux and Windows AIO implementations may fall back to synchronous disk I/O when used in buffered mode — to guarantee that disk I/O is fully *asynchronous*, the file needs to be accessed in unbuffered, direct I/O mode, and the programmer may have to implement caching inside the application [1, 2]. As I have tested, even in the direct I/O mode, the current Linux AIO implementation may still exhibit blocking behavior occasionally. To avoid these problems, a user-level thread system can choose not to use asynchronous I/O for disk accesses, but to use a large pool of OS threads to perform synchronous disk I/O instead. However, the OS thread pool also introduces additional performance overhead and implementation complexity.

The existing GHC RTS addresses the API portability problem by using a portable non-blocking I/O interface called "select" for all Unix-based systems and a separate interface for Windows. Unfortunately, the "select" interface has poor performance for socket and pipe I/O. The existing GHC RTS also has *no* support for asynchronous disk I/O at all: it simply uses synchronous I/O for disk file accesses. Such a design has advantages on *portability* of programming interface and *simplicity* of implementation, but it has significant penalties on *I/O performance* as I have tested using performance benchmarks.

With the new GHC RTS design, the user can also implement a user-level thread system that does the same work of the old GHC RTS, but the implementation is independent of the RTS — the concurrency library is written in Haskell, so the developer does not need to know the RTS internals in order to implement I/O libraries for each platform.

However, a more important benefit of the new RTS is that the user now has the option to use *synchronous* and *blocking* I/O with the OS thread library. Such I/O operations does not require special handling in the RTS or concurrency library; an I/O operation can simply be implemented using a foreign function call to its corresponding C library function. The programming interfaces for such I/O operations are also mostly platform-independent, so it is easy to write an I/O library that runs on multiple platforms.

To summarize, the new RTS design makes it easier to implement I/O with optimal performance on multiple platforms: a user-level thread implementation would benefit from its modularity; an OS thread implementation could take advantage from the portability of synchronous and blocking I/O programming interfaces.

I/O performance and limits on Linux As shown by the performance experiments in Section 5.4, the existing GHC RTS has poor I/O performance and scalability on Linux because its I/O implementation uses the portable "select" interface and it does not properly implement disk I/O. The two concurrency library prototypes I implemented with the new RTS both significantly outperform the existing GHC on I/O benchmarks: they scale to larger numbers of threads and higher I/O throughput.

It is also possible to overcome the I/O limitation with the current GHC RTS by improving its the I/O implementation. The following is a list of steps that need to be taken:

- The developer needs to be familiar with both the internal details of the existing RTS and the asynchronous I/O mechanisms on the platform to be supported.
- Platform-specific I/O manager modules, such as Linux epoll support, needs to be added to the existing RTS. A consequence is that the overall complexity of the RTS will be increased, and maintenance will become more difficult given the monolithic RTS design.
- Supporting asynchronous disk I/O requires caching to be implemented in the RTS; such a design may not be suitable for general-purposed applications. A better design would be to use an OS thread pool for synchronous disk I/O. Fortunately, the existing GHC RTS already has such an OS thread pool for performing blocking foreign out-calls, the developer only needs to modify the standard I/O library so that disk I/O will be performed using the OS thread pool. The actual performance of this implementation strategy, however, still needs to be tested.

MVar performance The MVar synchronization operation in the new RTS is significantly slower than in the existing RTS. The user-level thread library written in Haskell is roughly 10 times slower than the existing RTS on MVar operations; the OS thread library is roughly 20 times slower on Linux. This performance difference is explained by the following reasons:

- The existing RTS implements MVar operations using highly optimized C and assembly code, while the new RTS implements them in Haskell libraries using the concurrency primitives.
- The MVar operation in Haskell concurrency libraries is built using the PTM interface, which is a pure software implementation of transactional memory. The existing RTS implements MVar synchronization using spin locks, which is much more efficient.
- The MVar operations in the OS thread library involve OS context switches.

Because of the advantage on MVar performance, the existing GHC RTS is better for concurrent programs in which MVar synchronization is the performance bottleneck. However, it is not clear whether such programs represent a significant class of real-world applications.

On the other hand, the recent software transactional memory (STM) implementation in the existing GHC RTS is also significantly slower than the MVar, yet STM is a more appealing interface to use than the MVar in many applications because of its compositional properties. If STM becomes the new standard in Haskell concurrency instead of the MVar, do we still care about the MVar performance? Of course, the new RTS would have had another challenging problem to deal with, which is to implement STM itself. I will discuss this problem in the next paragraph.

STM and other concurrency features The new RTS design provides users the freedom to develop new concurrency features in Haskell libraries. One such example is a priority scheduler, in which threads can be assigned with priorities and scheduled using certain algorithms. However, when efficiency and performance is considered, the new RTS design is certainly not ready for *all* concurrency features. The software transactional memory (STM) interface is one such example.

In the existing RTS, STM is implemented directly using C. The PTM in the new RTS shares a large amount of code with the STM implementation in the existing RTS; their performance is similar to each other. The problem is that PTM is a lower-level interface for writing the concurrency library, rather than for writing regular Haskell applications, so the application programmer does not have access to transactional memory by default.

One option to provide transactional memory to the application programmer is to simply wrap up the PTM interface in the concurrency library and export it to the applications. The problem is that only the basic reading, writing and committing operations can be exported, but the HEC waiting and sleeping cannot be exported because it would reveal details of the concurrency library implementation and the substrate interface. The result would be a simpler STM implementation without the `retry` primitive.

Another option is to encode the STM using lightweight concurrency primitives, like implementing the MVar. One apparent problem with this approach is efficiency — STM or PTM is already much slower than locks or MVar; implementing STM on top of PTM would make it two orders of magnitude slower. The other problem is typing restrictions: implementing TM in Haskell would require the values carried by the transactional variables to be comparable, i.e. members of the `Eq` typeclass.

A final solution is to extend the substrate interface with more support for TM, so that STM can be efficiently built using PTM.

The only difference between PTM and STM is automatic retrying versus manual wake-up: In STM, when a transaction is rolled back using `retry`, it adds the current thread identifier into the waiting queues of all the transactional variables in its read-set; when a transaction commits, it traverses though the waiting queues of all the transaction variables in its write-set and wake up the threads blocked on them.

Given these differences, a feasible way to implement STM over PTM is to use a PTM transaction as a

STM transaction and to add a few concurrency primitives that (1) allow each transactional variable to be attached with an object, and (2) allow the programmer to traverse the objects attached to the transactional variables in the read-set or the write-set of a nested PTM transaction. Thus, the concurrency library can implement STM as a wrapper over the PTM, while adding the details of how threads are manipulated at commit or rollback time. The `retry` primitives can be implemented using exceptions.

Conclusion Currently, the new RTS and the existing RTS each have their own advantages. The new RTS is better for its modularity, ease of maintenance, portability and good Linux I/O performance. The existing RTS is better on MVar synchronization performance and full STM support.

However, this comparison may change in the future in several ways:

- With a moderate amount of engineering, it is possible that the I/O performance on the existing RTS can be improved.
- If STM becomes the new standard of Haskell concurrency, the performance of MVar operations may be no longer important.
- My speculation is that new RTS design can be extended to provide efficient STM support for the application programmer.

If all the above become true in the future, the existing RTS and the new RTS may not have much difference on performance and STM support, but modularity, ease of maintenance and portability may finally be the reason to adopt the new RTS design.

Discussion Table 6.2 shows that the new RTS with the OS thread library competes favorably to the other two configurations in most aspects: it has the lowest implementation and maintenance complexity; it uses synchronous and blocking I/O interfaces that are easily portable; its I/O performance on Linux can be good enough for most applications. In addition, the OS thread concurrency library does not use most of the lightweight concurrency primitives implemented in the new RTS, such as context switching and local states, so the overall implementation complexity could be further reduced if the OS thread library was the only configuration to be supported.

The above suggests a possibility to further simplify the RTS design to a potentially optimal point on some operating systems. Instead of providing lightweight threads or concurrency primitives, the RTS could simply use a monolithic design that eliminates the need for concurrency libraries and only supports direct OS threads and synchronous I/O. This *hypothetical* RTS design is different from both the existing RTS and the new RTS I have presented earlier in the dissertation; such a thin, monolithic RTS design would be easy

to port to multiple platforms and easy to maintain. The caveat is that its performance will be dependent on the operating system used, but it can at least work well on Linux.

To avoid confusion, it is worth clarifying that this dissertation has mentioned three RTS designs up to now:

1. The existing, monolithic GHC RTS, with the possibility to improve its I/O performance in the future.
2. The new, modular RTS design that provides lightweight concurrency primitives and uses concurrency libraries.
3. The hypothetical, thin, monolithic RTS that only supports OS threads.

The new RTS I described in Chapter 5 plus the OS thread library together can be seen as a running prototype for the hypothetical, thin, monolithic RTS. The actual implementation can be derived from my new RTS implementation by removing most of the lightweight concurrency primitives and building the MVar operations using locks and condition variables instead.

As a final note, the block-based nursery management scheme introduced in Section 5.1.5 can be adopted to the existing RTS implementation, regardless of the new RTS design.

6.3 For language designers

In the past several years, I have learned Haskell, used it to develop network services and worked on its runtime system implementation. At the end of the dissertation, I would like to comment on the design of the Haskell language based on my experience.

I use the title *"Programmable Concurrency in a Pure and Lazy Language"* for this dissertation because I think *"Programmable Concurrency"* represents the general idea for the concurrency monad and the new runtime system design and the phrase *"Pure and Lazy"* captures the essence of the Haskell language. The following sections discuss my experience with purity and laziness in Haskell, respectively.

6.3.1 The benefits of purity

In Haskell, computation is purely functional by default. Side-effects, such as input and output or even manipulation on shared memory states, are implemented using monads. The use of such side-effects is restricted by the type system so that pure computation and impure computation are clearly distinguished everywhere in the program. This language design approach is drastically different from most imperative programming languages and many functional programming languages, where computation is impure and has unrestricted side-effects by default.

For real-world applications like network services, programming in Haskell seems quite inconvenient at the beginning, because there are strong restrictions on the use of side-effects: after all, I/O and shared memory states are almost all that matter for such applications. Some tasks that seem simple in languages like C might be quite challenging in Haskell, for example, using a globally shared state in all threads, or printing a debugging message in a deeply nested function. The Haskell language forces the programmer to precisely specify the side-effects of a function in its type, and such type specifications are enforced everywhere in the program with no mercy.

However, the purity of Haskell shows great benefits in a concurrent world, where program complexity often arises from the interaction of multiple threads. A pure computation is guaranteed to have no side-effects with other parts of the program in a concurrent setting. For example, when traversing a purely functional tree structure, the programmer never needs to worry about locks and conflicts with other threads, even if the structure is shared with other threads at run time. As the application program scales up, the initial inconvenience of being restrictive and verbose on the use of side-effects is soon compensated by the benefit of being spartan and explicit on them.

The purity of Haskell also makes it possible to control the side-effects of program fragments in a fine-grained fashion using type systems and guarantee certain isolation properties that are important for concurrency. A convincing example is the software transactional memory (STM) design in Haskell, where the side-effects of transactional computations are limited to operations on transactional variables. This design makes the STM interface highly compositional while providing the strong, all-or-nothing semantics.

The design of the new runtime system, the lightweight concurrency primitives and the concurrency prototypes in this dissertation can be seen as a field test of the STM design. My experience with the TM-based concurrency primitives has been so far positive: the concurrency library built with PTM is more compact and easier to reason about than its corresponding modules built in the existing GHC RTS, which is implemented using locks and conditional variables and intermingled with other components of the RTS.

My observation is that a *pure* language design is particularly beneficial for *concurrent* programming.

6.3.2 The cost of laziness

My experience with the *laziness* of Haskell has been a two-sided story.

On the positive side, laziness plays an important role in the implementation of the concurrency monad. Laziness, together with other features such as higher-order functions and syntactic sugar for monads, make Haskell so expressive such that the poor man's threads can be built entirely inside Haskell, without additional compiler or runtime system support.

On the negative side, however, most of my systems programming experiences in Haskell do not benefit much from laziness, except the concurrency monad. What could be worse, laziness has been an obstacle in

several places of my dissertation project.

The first problem is concurrent thunk evaluation. In a multithreaded world, an unevaluated thunk may be simultaneously accessed from multiple threads. To avoid duplication of a potentially expensive computation, such threads need to synchronize with each other. This creates a challenge for the new runtime system design, because thunk evaluation is the task of the runtime system, but thread synchronization is implemented in the concurrency library and the runtime system does not know the details about it. The problem is solved by using a callback interface from the runtime system to the concurrency library.

Because thunk evaluation is common in the execution of Haskell programs, putting thread synchronization operations on every thunk access could be expensive. The GHC RTS solves this problem by using a complex algorithm called *lazy blackholing* that starts thunk evaluation optimistically, check for duplicated computation periodically and roll back duplicated computation when necessary. Although I did not change most parts of this algorithm, it took me a long time to understand its implementation and make it work with the new RTS prototype.

Besides the complexity of the runtime system, laziness is a generally challenging aspect of Haskell programming, because laziness makes it difficult for the programmer to reason about evaluation order and memory usage. In my experience of developing the concurrency library, my program occasionally run into stack overflow errors and it often takes a long time to find out that the problem is on the order of evaluation for large, purely functional data structures. When the concurrency library manages more than 100,000 threads, its internal data structures, such as task queues, need to be carefully engineered to make sure that operations on such large structures do not cause memory leaks. When the program complexity scales up, it is generally difficult to get everything right even for an experienced Haskell programmer — the strictness annotations and order of evaluation must be all correct, or there *will* be memory leaks.

The difficulty of programming with laziness is also the reason that the new RTS design uses PTM instead of locks. As explained in Section 4.3.2, it is often a bad thing when a thread runs into a long computation while holding a lock, yet in a lazy language it is generally difficult to guarantee that such things never happen.

My summary is that laziness improves the expressiveness of a purely functional language, yet it is challenging to program with and it leads to difficulties on language implementation, particularly on a concurrent system. Fortunately, most of these implementation challenges can be conquered using delicate system designs and careful engineering.

Bibliography

[1] Asynchronous Disk I/O Appears as Synchronous on Windows NT, Windows 2000, and Windows XP. Windows Knowledge Base, Article 156932, Revision 4.5. http://support.microsoft.com/kb/156932.

[2] Kernel Asynchronous I/O (AIO) Support for Linux. http://lse.sourceforge.net/io/aio.html.

[3] The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications. 11th and 12th Sun Users Group Conference, December 1993 and June 1994.

[4] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management without Manual Stack Management. In *Proceedings of the 2002 Usenix Annual Technical Conference*, 2002.

[5] The Apache Web Server. http://www.apache.org.

[6] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[7] Andrew W. Appel and Zhong Shao. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5:189–219, 1992.

[8] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[9] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.

[10] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[11] Jost Berthold, Abyd Al-Zain, and Hans-Wolfgang Loidl. Adaptive High-Level Scheduling in a Generic Parallel Runtime Environment. In *Symposium on Trends in Functional Programming (TFP)*, New York, USA, April 2007.

[12] Andrew D. Birrell. An Introduction to Programming with Threads. Technical Report 35, DEC SRC, January 1989.

[13] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 265–276, New York, NY, USA, 2005. ACM Press.

[14] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*, pages 99–107, Philadelphia, May 1996. ACM.

[15] Tom H. Brus, Marko C. J. D. van Eckelen, Maarten. O. van Leer, and Marinus J. Plasmeijer. Clean — a language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, pages 364–384. LNCS 274, Springer Verlag, September 1987.

[16] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: A language for programming high-performance servers. In *2006 USENIX Annual Technical Conference*, pages 129–142, June 2006.

[17] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

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

[19] Kathleen Fisher and John Reppy. Compiler support for lightweight concurrency. Technical memorandum, Bell Labs, March 2002.

[20] Matthew Fluet, Mike Rainey, and John H. Reppy. A scheduling framework for general-purpose parallel languages. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2008.

[21] Matthew Fluet, Mike Rainey, John H. Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007)*, pages 37–44, January 2007.

[22] Emden R. Gansner and John H. Reppy. A Multi-threaded Higher-order User Interface Toolkit. In *User Interface Software, Bass and Dewan (Eds.)*, volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

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

[24] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, New York, NY, USA, 2005. ACM Press.

[25] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, September 2005.

[26] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, June 2005.

[27] Paul Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, New York, NY, USA, 2000.

[28] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, 2007.

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

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

[31] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[32] James R. Larus and Michael Parkes. Using Cohort-Scheduling to Enhance Server Performance. In *USENIX Annual Technical Conference, General Track*, pages 103–114, 2002.

[33] Hugh C. Lauer and Roger M. Needham. On the Duality of Operating Systems Structures. In *Proceedings Second International Symposium on Operating Systems*. IRIA, October 1978.

[34] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 24–35, New York, NY, USA, 1994. ACM.

[35] Peng Li, Yun Mao, and Steve Zdancewic. Information Integrity Policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*, September 2003.

[36] Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew Tolmach. Lightweight Concurrency Primitives for GHC. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.

[37] Peng Li and Steve Zdancewic. Advanced control flow in Java Card programming. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 165–174, June 2004.

[38] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[39] Peng Li and Steve Zdancewic. Practical Information-flow Control in Web-based Information Systems. In *Proc. of 18th IEEE Computer Security Foundations Workshop*, 2005.

[40] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *"Proc. of 19th IEEE Computer Security Foundations Workshop"*, 2006.

[41] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services — implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[42] Simon Marlow. Developing a High-performance Web Server in Concurrent Haskell. *Journal of Functional Programming*, 12, 2002.

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

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

[45] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[46] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed Closure Conversion. In *Symposium on Principles of Programming Languages*, pages 271–283, 1996.

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

[48] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[49] Scott Oaks and Henry Wong. *Java Threads, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[50] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[51] John K. Outsterhout. Why Threads Are A Bad Idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, 1996.

[52] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[53] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 1996.

[54] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[55] Rob Pike. A Concurrent Window System. *Computing Systems*, 2(2):133–153, Spring 1989.

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

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

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

[59] John H. Reppy. CML: A Higher Concurrent Language. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 293–305, New York, NY, USA, 1991. ACM Press.

[60] John H. Reppy. CML: a higher-order concurrent language. In *ACM Conference on Programming Languages Design and Implementation (PLDI'91)*. ACM, June 1991.

[61] John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.

[62] Olin Shivers. Continuations and Threads: Expressing Machine Concurrency Directly in Advanced Languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, 1997.

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

[64] William Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[65] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[66] Simon Thompson. *Haskell: the craft of functional programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[67] Phil W. Trinder, Kevin Hammond, James S. Mattson, Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable parallel implementation of haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*. ACM Press, Philadelphia, May 1996.

[68] The Twisted Project. http://twistedmatrix.com/.

[69] Rob von Behren, Jeremy Condit, and Eric Brewer. Why Events Are A Bad Idea (for high-concurrency servers). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[70] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Ninteenth Symposium on Operating System Principles (SOSP)*, October 2003.

[71] Philip Wadler. Monads for Functional Programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

[72] Mitchell Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28, August 1980.

[73] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, 2001.

[74] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.