A Language-based Approach to Unifying Events and Threads

Peng Li
University of Pennsylvania

Steve Zdancewic
University of Pennsylvania

Abstract

This paper presents a language-based technique to unify two seemingly opposite programming models for building massively concurrent network services: the event-driven model and the multithreaded model. The result is a *unified concurrency model* providing both thread abstractions and event abstractions. Using this model, each component in an application can be implemented using the appropriate abstraction, simplifying the design of complex, multithreaded systems software.

This paper shows how to implement the unified concurrency model in Haskell, a pure, lazy, functional programming language. It also demonstrates how to use these techniques to build an *application-level* thread library with support for multiprocessing and asynchronous I/O mechanisms in Linux. The thread library is type-safe, is relatively simple to implement, and has good performance. Application-level threads are extremely lightweight (scaling to ten million threads) and our scheduler, which is implemented as a modular and extensible event-driven system, outperforms NPTL in I/O benchmarks.

1 Introduction

Modern network services present software engineers with a number of design challenges. Web servers, multiplayer games, and Internet-scale data storage applications must accommodate thousands of simultaneous client connections. Such massively-concurrent programs are difficult to implement, especially when other requirements, such as high performance and strong security, must also be met.

Events vs. threads: Two implementation strategies for building such inherently concurrent systems have been successful. Both the multithreaded and event-driven approaches have their proponents and detractors. The debate over which model is "better" has waged for many years, with little resolution. Ousterhout [17] 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, by von Behren, Condit, and Brewer [23], 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 [22].

From the programmer's perspective, both models are attractive. Sometimes it is desirable to think about the system in terms of threads (for example, to describe the sequence of events that occur when processing a client's requests), and sometimes it is desirable to think about the system in terms of events and event handlers (for example, to simplify reasoning about asynchronous communications and resource scheduling).

A unified concurrency model: This paper shows that events and threads can be unified in a single concurrency model, allowing the programmer to design parts of the application as if she were using threads, where threads are the appropriate abstraction, and parts of the system using events, where they are more suitable. Section 2 gives some additional background about the multithreaded and event-driven models and motivates the design of our unified concurrency model.

In our model, for higher-level application code, the programmer can use use a multithreaded programming style similar to C and Java, with familiar control-flow elements such as sequencing, functions calls, conditionals and exceptions, and with user-defined system calls for I/O and thread control. Figure 1 shows two sample functions written in the multithreaded style.

For lower-level I/O code, the programmer can conveniently use asynchronous OS interfaces such as `epoll` and AIO in Linux. The thread scheduler has an extensible, modular event-driven architecture in which the application-level threads can be seen as event handlers.

A language-based approach: Languages like C and C++ have historically been used to implement high-performance, concurrent software. However, they suffer from well known security and reliability problems that have prompted a move toward type-safe languages like Java and C#. Their general-purpose threads packages are typically quite heavyweight though, and none of these languages provide appropriate abstractions to simplify event-driving programming. Implementing scalable thread or event systems is feasible using these languages, but the results can be cumbersome to use. Techniques such as compiler transformations can address these problems to some extent [22], but even then the engineering challenges typically force the programmer to choose between threads or events—they don't get both.

Our case-study implementation of the unified concurrency model, described in detail in Section 3, is written in the programming language Haskell [12]. Haskell is a pure, lazy, strongly-typed, functional language with many advanced language features, such as *type classes*, that make it very convenient to implement the unified concurrency model. Our implementation is based on techniques developed some time ago by Koen Claessen [8] in the programming languages research community.

Application-level threads: Using this language support, we have built an *application-level* thread library, in which the threaded code and thread scheduler are written *inside* the application. Section 4 describes the thread library, which uses both event-driven and multithreaded programming models, and shows how it can flexibly support several synchronization mechanisms for interthread communication and shared state.

Compared to traditional approaches (both multithreaded and event-driven), our application-level thread library has many advantages for building scalable network services:

- *Flexibility:* The programmer can choose to use the appropriate programming models for different parts of the system.
- *Scalability:* The implementation of application-level threads are extremely lightweight: it scales up to 10,000,000 threads on a modest test system.
- *Parallelism:* The application-level threads can execute on multiple processors concurrently.
- *Performance:* The thread scheduler behaves like a high-performance event-driven system. It outperforms equivalent C programs using NPTL in our I/O benchmarks.
- *Safety:* The implementation of the thread library and the application programming interface are both type-safe.

```
-- 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 1: Example of multithreaded code in Haskell

Section 5 gives the results of our performance experiments and describes a simple web server we implemented as a case study. Our experience shows that the Haskell is a reasonable language for building scalable systems software: it is expressive, succinct, efficient and type-safe; it also interacts well with C libraries and APIs.

Summary of contributions:

1. A unified programming model that combines event-driven and multithreaded programming. This model is suitable for building highly scalable, concurrent systems software.
2. A Haskell implementation of the interfaces for the unified concurrency model, based on techniques from the programming languages community.
3. An application-level thread library and accompanying experiments that demonstrate the feasibility and benefits of this approach.

2 Unifying events and threads

This section gives some background on the multithreaded and event-driven approaches and motivates the design of the unified concurrency model.

2.1 The thread–event duality

In 1978, Lauer and Needham [15] argued that the multithreaded and event-driven models are dual to each other. 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 presented looks like this:

Threads		Events
thread continuation	$\sim$	event handler
scheduler	$\sim$	event loop
exported function	$\sim$	event
procedure call	$\sim$	send event / await reply

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. Most existing approaches trade off threads for events or vice versa, choosing one model over the other. We propose a different route: rather than using the duality to justify choosing threads over events or vice versa (since either choice can be made as efficient and scalable as the other), we see the duality as a strong indication that the programmer should be able to use *both* models of concurrency in the same system. The duality thus suggests that we should look for natural ways to support switching between the views as appropriate to the task at hand.

2.2 A comparison of events vs. threads

Programming: 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 approach leads to a natural programming style in which the control flow for a single thread is made apparent by the program text, using ordinary language constructs like conditional statements, loops, exceptions, and function calls.

Event-driven programming, in contrast, is hard. Most general-purpose programming languages do not provide appropriate abstractions for programming with events. The control flow graph of an event-driven program has to be decomposed to multiple event handlers and represented as some form of state machines with explicit message passing or in continuation-passing style (CPS). Both representations are difficult to program with and reason about, as indicated by the name of Python's popular, event-driven networking framework, "Twisted" [21].

Performance: The multithreaded programming style does not come for free: In most operating systems, a thread uses a reserved segment of stack address space, and the virtual memory space exhausts quickly on 32-bit systems. Thread scheduling and context switching also have significant performance overheads. However, such performance problems can be reduced by well engineered thread libraries and/or careful use of cooperative multitasking—a recent example in this vein is Capriccio [22], a user-level threads package specifically for use in building highly scalable network services.

The event-driven approach exposes the scheduling of interleaved computations explicitly to the programmer, thereby permitting application-specific optimizations that significantly improve performance. The event handlers typically perform only small amounts of work and usually need only small amounts of local storage. Compared to thread-based systems, event-driven systems can have the minimal per-thread memory overheads and

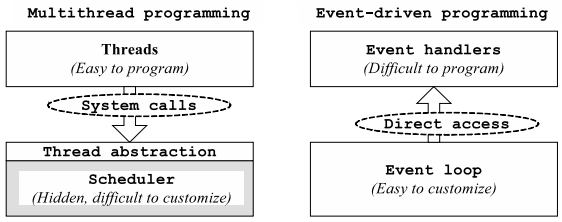

Figure 2: Threads vs. events

context switching costs. Furthermore, by grouping similar events together, they can be batch-processed to improve code and data locality [14].

Flexibility and customizability: Most thread systems provide an abstract yet rigid, synchronous programming interface and the implementation of the scheduler is mostly hidden from the programming interface. Hiding the scheduler makes it inconvenient when the program requires the use of asynchronous I/O interfaces not supported by the thread library, especially those affecting the scheduling behavior. For example, if the I/O multiplexing of a user-level thread library is implemented using the portable **`select`** interface, it is difficult to use an alternative high-performance interface like **`epoll`** without modifying the scheduler.

Event-driven systems are usually more flexible and customizable because the programmer has direct control of resource management and direct access to asynchronous OS interfaces. Many high-performance I/O interfaces (such as asynchronous I/O, **`epoll`** and kernel event queues) provided by popular OSes are asynchronous or event-driven, because this programming model corresponds more closely to the hardware interrupts. An event-driven system can directly take advantage of such asynchronous, non-blocking interfaces, while using thread pools to perform synchronous, blocking operations.

Another concern is that most user-level cooperative thread systems do not take advantages of multiple processors, and adding such support is often difficult. Event-driven systems can easily utilize multiple processors by processing independent events concurrently [26].

2.3 The unified concurrency model

One important reason that the user-level threads in systems like Capriccio achieve good performance is that the thread scheduler is essentially an event-driven application that uses asynchronous I/O interfaces to make scheduling decisions. Although the performance of user-level threads packages can rival their event-driven counterparts, they are less flexible than event-driven systems, because the scheduler is mostly hidden from the programmer and new event-based interfaces sources cannot be easily added. An event-driven application such as the

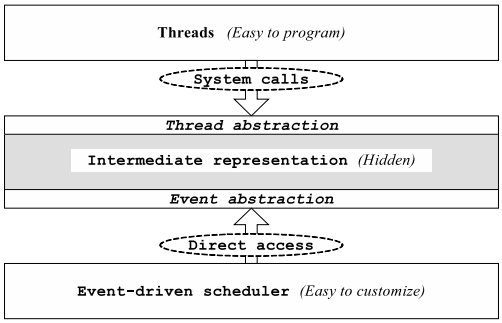

Figure 3: The unified concurrency model

Flash web server [18], on the other hand, has flexibility when choosing I/O interfaces, but there is no appropriate abstraction for generic multithreaded programming. This situation is illustrated in Figure 2.

Ideally, we would like to use an *application-level* thread system, where the thread abstraction is provided *inside* the application and the scheduler is a programmable part of the application. We use the term *application-level* threads in contrast with the more general concept of *user-level* threads, which are typically implemented in a library that hides the thread scheduler. The problem is how to provide appropriate *abstraction* when the scheduler is part of the application: the scheduler is a complex piece of code, it heavily uses low-level, unsafe operations and internal data structures. Without appropriate abstraction, writing such an application requires almost as much work as implementing an user-level thread scheduler from scratch.

The key idea of this paper is to separate the low-level, internal representation of threads from the scheduler implementation. As illustrated in Figure 3, the goal is to design a software library (as the box in the center) that provides two different abstractions for application-level threads: the *thread view*, which allows per-thread code be written in the natural, imperative, multithreaded style as shown previously in Figure 1, and the *event view*, which allows the threads be passively manipulated by the underlying scheduler in an abstract, type-safe way.

The scheduler is part of the application as in most event-driven systems. The programmer needs to implement only high-level algorithms and data structures for dispatching events and scheduling threads. Figure 17 (described later) shows three example event loops programmed in this style; such code is concise and easily customizable for a given application. Reusable libraries wrap the low-level operating system calls, which helps keep the scheduler code clean.

This dualized model gives the best of two worlds: the expressiveness of threads and the customizability of events. The next section shows how this design is implemented in Haskell.

3 Implementing the unified model

In the duality of threads and events, an atomic block of instructions in a thread continuation (also called a *delimited continuation*) corresponds to an event handler, and the thread scheduler corresponds to the main event loop. The relationship between threads and events can be made explicit using a *continuation-passing style* (CPS) translation [3]. However, CPS translation is painful to deal with in practice—programming purely in an event-driven (or message passing) style essentially amounts to doing CPS translation manually. The goal of the unified programming model is to hide the details of CPS translation from the programmer. As shown in Figure 3, in order to hide the implementation details of the middle box in a software library, the language mechanism should provide adequate abstraction for both the *thread view* and the *event view* interfaces.

Thread view: Code for each thread is written in a natural, sequential style with support for most common control-flow primitives: branches, function calls, loops, exception handling, as shown in Figure 1. Thread control and I/O can be implemented using a set of system calls configurable for each application. The key point of this abstraction is that the internal representation of the threads should be completely hidden at this level and the programmer does not need to manage the details such as continuation passing.

Event view: The internal representation of threads can be conveniently accessed from the scheduler. The event loop (scheduler) can (1) examine a system call request (such as I/O) submitted from a thread, and (2) execute a delimited continuation of a thread in the same way as invoking an event handler function, passing appropriate arguments as responses to the system call requests. This allows the main scheduler to play the "active" role and the threads to play the "passive" role: the scheduler actively "pushes" the thread continuations to make progress. This active programming model makes it easy to use control-flow primitives provided by the programming language to express the scheduling algorithms.

Here we use the techniques developed by Koen Claessen [8] to present an simple, elegant and lightweight abstraction mechanism that uses the following language features (which will be further explained below):

- *Monads* provide the *thread abstraction* by defining an imperative sub-language of Haskell with system calls and thread control primitives.
- *Higher-order functions* provide the internal representation of threads in continuation-passing style.
- *Lazy data structures* provide the *event abstraction*,

which is a lazy tree that represents the trace of system calls generated by threads.

3.1 Traces

Traces and system calls: A central concept in the unified concurrency model is a *trace*, which is a tree 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 5 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 5: 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 a set of system calls, as shown in Figure 6. Each system call in the multithreaded programming interface corresponds to exactly one type of tree node. For example, the **`SYS_FORK`** node has two subtraces, 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 nonblocking 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 6: System calls and their corresponding traces

Lazy traces and thread control: We can think of the trace as an output of the thread execution: as the thread runs, the nodes in the trace are generated. What makes the trace interesting is that the computation is *lazy*: a computation is not performed until its result is used. Using lazy evaluation, the consumer of a trace can control the execution of a thread: whenever a node in the trace is accessed, 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 accessed. In other words, the execution of threads can be controlled by traversing their traces.

Figure 4 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 decides to run the thread until it reaches the next system call. It simply 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 has not been created yet, so the continuation of the thread is launched in order to compute the value of the node.
3. The thread runs until it performs the next system call, **`sys_nbio`**.
4. The thread is suspended again, because it has performed the necessary computation to create the new node in the trace.
5. The value of the new node, **`SYS_NBIO`** is available in the scheduler. The scheduler then handles this system call by performing the non-blocking I/O operation and running the continuation of the thread.

Therefore, the lazy trace provides the *event abstraction* we need: it is an abstract interface that allows the main 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 a trace is essentially a delimited continuation that represents part of the thread execution.

The remaining problem is how to provide a mechanism that transforms multithreaded code into traces—how do we design a software module that provides both the *thread abstraction* and the *event abstraction* in Figure 4? The answer is the CPS monad. The next two sections introduces the concept of *monads* and shows how the CPS monad solves these problems.

3.2 Background: monads

Haskell has support for a mechanism called *monads* [24, 11] that provide an abstract type of computations with side effects. **`Monad`** is a standard interface for programming with functional combinators, which can be used as a domain-specific language. By designing thread control primitives as monadic combinators, the monad interface can be used as an abstractions for multithreaded

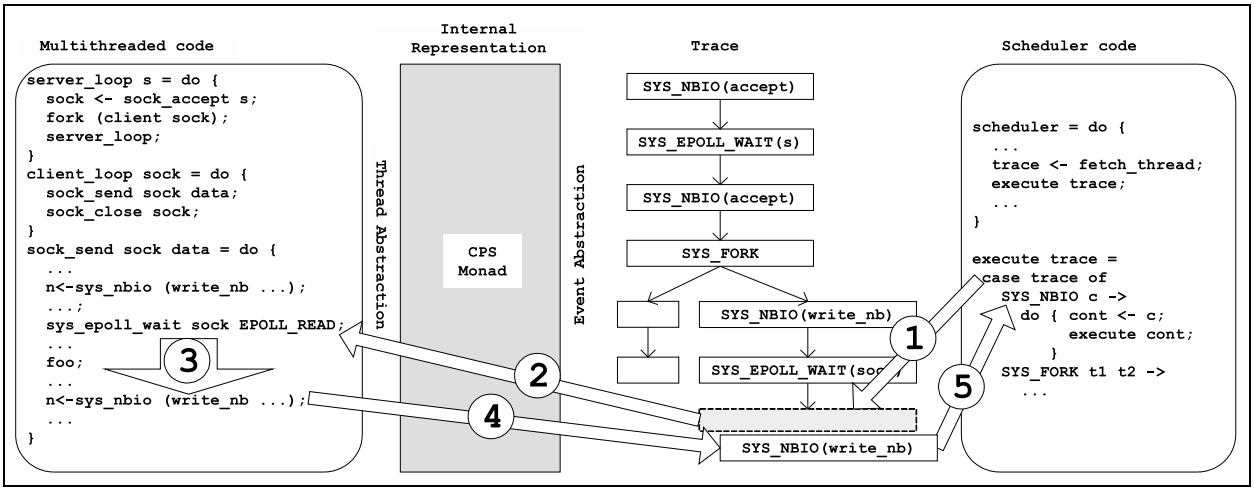

Figure 4: Thread execution through lazy evaluation (the steps are described in the text)

programming, because it provide a way of hiding the "internal plumbing" needed to write programs in CPS style.

The basic idea behind monads can be explained by looking at Haskell's **IO** monad, which is provided as part of its standard libraries. A value of type **IO** α is an effectful computation that may perform some actions as side-effects before yielding a value of type α. **IO** side effects include input and output, reading and writing shared memory references and accessing many system utilities. For example, the function **hGetChar** takes a handle to an I/O stream and returns an action that, when executed, reads a character from the stream. The function **hPutChar** is similar, but it takes a handle and a character to output and produces no result (Haskell's type **()** is similar to **void** in C). These functions have the following types:

```
hGetChar :: Handle -> IO Char
hPutChar :: Handle -> Char -> IO ()
```

There are also two standard operations called **return** and **bind** that all monads must implement. For the **IO** monad, these operations have types:

```
return :: α -> IO α
(>>=)  :: IO α -> (α -> IO β) -> IO β
```

The **return** combinator "lifts" an ordinary expression of type α into the **IO** monad by returning the trivial action that performs no side effects and yields the input as the final answer. The infix combinator **>>=**, pronounced "bind", sequences the actions in its arguments: **e >>= f** returns a new action that first performs the I/O operations in **e** and passes the result computed by **e** to the function **f**, which may then produce subsequent actions. In Haskell there is support for the so called "**do**" notation, which provides more familiar looking syntactic sugar for binding multiple actions in series. For example, the two implementations of **double** in Figure 7 are equivalent—both read in a character and output it twice (here and elsewhere we use Haskell's syntax **x->e** for an anonymous function with argument **x** and body **e**):

```
double :: Handle -> IO ()
double h =
 hGetChar h >>= ( \x ->
 hPutChar x >>= ( \_ ->
 hPutChar x ))
```

```
double h =
 do { x <- hGetChar h;
      hPutChar x;
      hPutChar x;
    }
```

Figure 7: The "**do**" syntax of Haskell

The **IO** monad is but one example of a wide variety of well-known monads, most of which are useful for encapsulating side effects. Haskell has a mechanism called type classes that allows the "**do**" notation to be overloaded for programmer-defined monads, and we exploit this feature to give natural syntax to the threads implemented on top of continuation-passing mechanisms we describe next.

3.3 The CPS monad

The goal is to design a monad that provides a *thread abstraction*, so the programmer can write multithreaded code using the overloaded "**do**"-syntax with a set of system calls. The implementation of this monad is tricky, but the details are hidden from the programmer (in the box between the thread abstraction and the event abstraction in Figure 3).

The monad encapsulates the side effect of a multithreaded computation, that is, generating a trace. The tricky part is that, if we simply represent 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 in a meaningful way. 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 (lifting and sequential composition) are defined in Figure 8.

```
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 8: The CPS monad **M**

Given a computation **M a** wrapped in the above CPS monad, we 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 9 converts a monadic computation into a trace:

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

Figure 9: Converting monadic computation to a trace

In Figure 10, 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. Since the code is internally organized in continuation-passing style, we are able to 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 mx) (c ())
sys_yield  = M(\c->SYS_YIELD (c ()))
sys_ret    = M(\c->SYS_RET)
sys_epoll_wait fd event =
    M (\c -> SYS_EPOLL_WAIT fd event (c ()))
```

Figure 10: Implementing system calls

For readers unfamiliar with monads in Haskell, it may be difficult to follow the above code. Fortunately, these bits of code encapsulate *all* of the "twisted" parts of the internal plumbing in the CPS. The implementation of the monad can be put in a library, and the programmer needs to use only its interface. To write multithreaded code, the programmer simply uses the "**do**"-syntax and the system calls in Figure 10; 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.

3.4 Programming with threads

Using the monad abstraction and the system calls, the programmer can write code for each thread in a natural, multithreaded programming style. The system calls provide the basic non-blocking I/O primitives. The **sys_nbio** system call works with any non-blocking I/O function. The threaded programming style makes it easy to hide the non-blocking semantics and provide higher level abstractions by using nested function calls. For example, a blocking **sock_accept** can be implemented using non-blocking **accept** as shown in Figure 11.

```
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 11: Wrapping non-blocking operations

The **sock_accept** function 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. Figure 1 also shows similar examples of multithreaded code.

3.5 Programming with events

A simple 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 13 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. Examine the current node of the trace. This step causes the corresponding user thread to execute until a system call is generated.

```
worker_main ready_queue = do {
  -- fetch a trace from the queue
  trace <- readChan ready_queue;
  case trace of
    -- Nonblocking 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 13: A round-robin scheduler for three sys. calls

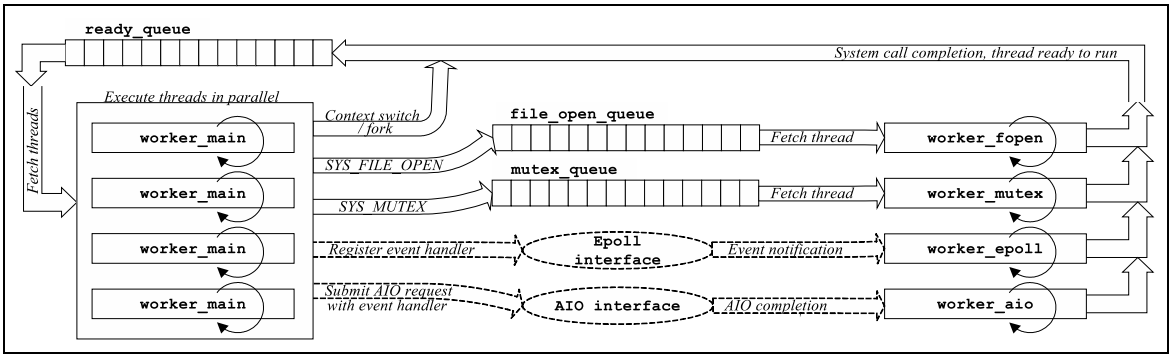

Figure 12: The event-driven thread scheduler: event loops and task queues

```
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 14: System calls for exceptions

3. Perform the requested system call
4. Write the child nodes (the continuations) of the trace to the queue.

This scheduler can already run some simple user threads, but it only executes one operation at a time and it does not support any I/O or synchronization primitives.

Advanced control flow: Because the threaded code is internally structured in CPS, it is easy to implement control-flow operations. For example, a scheduler can support exceptions by adding two new system calls as shown in Figure 14. The code in Figure 1 illustrates how exceptions are used by an application-level thread.

The tree traversal algorithm in **worker_main** must be modified so that when it sees a **SYS_CATCH** node it pushes the node onto a stack of exception handlers maintained for each thread. The scheduler then continues execution until it reaches either a **SYS_RET** node, indicating normal termination, or a **SYS_THROW** node, indicating exceptional termination. The scheduler then pops back to the **SYS_CATCH** node and continues with either the normal trace or exception handler, as appropriate.

Note that if an application does not need exceptions, the programmer can remove them to simplify the scheduler code. Conversely, if more complex per-thread state or fancier scheduling algorithms are required, this model accommodates them too. It is easy to customize the scheduler to the needs of the application.

4 An application-level thread library

This section shows how to scale up the event-driven design described above to build a real thread library that takes advantage of asynchronous I/O interfaces in Linux. The development version of the Glasgow Haskell Compiler (GHC) [9] already supports efficient, lightweight user-level threads, but the default GHC run-time library uses the portable (yet less scalable) **select** interface to multiplex I/O and it does not directly support non-blocking disk I/O. Our application-level thread library implements its own I/O primitives using the Foreign Function Interface (FFI); our thread scheduler uses only several native GHC threads internally, each mapped to a seperate OS thread by the GHC run-time library.

The rest of this section shows the detailed design. First we discuss how to accommodate multiple kernel-level threads to make better use of OS-level resources. Next, we consider asynchronous I/O and low-level event-driven interfaces. We then describe several synchronization mechanisms that can be used to provide interthread communication and shared state.

4.1 Using multiple event loops

Figure 12 shows the architecture of the event-driven scheduler in our full implementation. It consists of several event loops (like **worker_file_open**) that process events generated by a **worker_main** loop. Each such loop runs in its own kernel 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 queues are implemented using standard thread synchronization primitives supported by GHC.

To boost performance on multiprocessor machines, the scheduler runs multiple **worker_main** event loops in parallel so that multiple application-level threads can make progress simultaneously. This setup is based on the assumption that all the non-blocking I/O operations submitted by **SYS_NBIO** are thread safe. 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 mutexes in the application-level thread (such synchronization primitives are described below).

```
-- 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
-- Open a file, returning the file descriptor
sys_file_open filename mode

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

Figure 15: System calls for **epoll** and AIO

```
case trace of
 ...
 SYS_EPOLL_WAIT fd event c  -> epoll_add fd event c
 SYS_AIO_READ  fd off buf f -> aio_read  fd off buf f
 SYS_FILE_OPEN filename mode f ->
        writeChan (file_open_queue sched) trace
```

Figure 16: Handling system calls for **epoll** and AIO

```
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;
     } -- recursively calls itself and loop
worker_aio sched =
  do { -- wait for some AIO events
       results <- aio_wait;
       mapM (writeChan (ready_queue sched) results;
       worker_aio sched;
     }
worker_file_opener sched =
  do { -- fetch a task from corresponding scheduler queue
       -- each task has type (String, FD → Trace)
       (SYS_FILE_OPEN filename mode c) <-
                    readChan (file_open_queue sched);
       -- perform a blocking call
       fd <- native_file_open filename mode;
       -- apply c to fd, and write it to the ready queue
       writeChan (ready_queue sched) (c fd);
       worker_file_opener sched;
     }
```

Figure 17: Event loops for **epoll**, AIO and file opening

This event-driven architecture is similar to that in SEDA [25], but our events are finer-grained: instead of requiring the programmer *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. Our event abstraction makes this task easier, because it uses a strongly typed interface in which pure computations and I/O operations are completely separated. In Figure 4, the five steps of lazy thread execution are purely functional: they process an "effect-free" segment of the user thread execution. All I/O operations (including updates to shared memory) are submitted via system calls and are explicitly performed in the scheduler; the programmer has complete control on what I/O operations to run in parallel.

4.2 Epoll and Asynchronous IO (AIO)

The trace representation of threads makes it easy to take advantage of event-driven I/O interfaces. To support **epoll** and AIO in Linux, the thread library provides a set of system calls and trace nodes shown in Figure 15.

A special system call for opening files is needed because it is a blocking operation. Our scheduler uses a separate OS thread to perform such operations. Here, we omit AIO writing to save space as it is similar to reading. To execute these system calls in the scheduler, the **worker_main** event loop in Figure 13 is extended with a few branches in Figure 16.

For an **epoll** request, the scheduler calls a library function **epoll_add** to register an event handler with the system **epoll** device. The registered event contains a reference to **c**, the child node that is the continuation of the application thread, so the **epoll** event loop can send it to **ready_queue** when the event arrives. AIO requests are handled similarly. For a file-open request, the scheduler moves the entire trace to a queue called **file_open_queue**, which is used to synchronize with another event loop.

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

Finally, three event loops process these events, as shown in Figure 17. Each event loop runs in a separate OS thread. They work in the same way: the loop waits for some events from its event source, processes the events and then puts the resulting traces (corresponding to application threads) back into the **ready_queue**.

An application programmer can easily add other event sources to the thread library using the same procedure:

1. Add definitions for system calls and trace nodes.
2. Interpret the system calls in **worker_main** and register the event handlers.
3. Add event loops to process the events.

4.3 Thread synchronization

Of course, application-level threads must be able to communicate with each other. The thread library implementation provides several synchronization options.

To begin, application-level threads use cooperative multitasking: context switches may happen only at the system calls defined by the thread library interface. A standard call **`sys_yield`** allows a thread to explicitly yield control. Also recall that the **`return`** operation causes the scheduler to treat a pure (side-effect free) block of code as an atomic operation. For single processor machines, this means that the programmer has complete control over the atomicity of blocks of code.

On multiprocessor machines, it is necessary to prevent conflicts due to multiple kernel threads. For non-blocking operations, the programmer can use software transactional memory (STM) [10], which is supported directly by GHC. Application threads use **`SYS_NBIO`** to submit STM transactions as non-blocking **`IO`** operations. This approach supports communication via shared memory data structures with thread-safety provided by the transaction semantics.

For blocking operations, the GHC implementation of STM cannot be used because the it may block the execution of worker threads. Nevertheless, the developer can add application-specific system calls to provide synchronization primitives. As an example, the thread library provides mutexes via two system calls: **`sys_mutex_lock`** and **`sys_mutex_unlock`**. A mutex is implemented as a memory reference that points to a pair **`(locked, queue)`** where **`locked`** indicates whether the mutex is locked, and **`queue`** is a linked list of thread traces blocking on this mutex. Just as for other event sources (like **`sys_file_open`**), an event loop and a task queue are added[1] to process mutex system calls: 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 **`ready_queue`**. This setup of queues and event loops guarantees the thread safety of mutex operations. Other synchronization primitives such as the **`MVar`** in Concurrent Haskell [13] can also be similarly implemented.

Finally, because the thread library supports the **`epoll`** interface, threads can also communicate efficiently using pipes. This mechanism is suitable for writing programs in message-passing style on either single processor or multiprocessor machines.

5 Experiments

The main goal of our tests was to determine whether our Haskell implementation of the unified concurrency model could achieve acceptable performance for massively concurrent applications like web servers and multiplayer games. Such applications are typically bound by network or disk I/O and often have many idle connections. A second goal of our tests was to determine how our implementation performs on a multiprocessor, since one of the benefits of our approach is the ease with which our thread library can take advantage of software transactional memory. Finally, we wanted to investigate possible overheads of using Haskell itself: Haskell programs allocate memory frequently and the runtime heavily depends on garbage collection. Also, our application-level threads are implemented using higher-order functions and lazy datastructures that we were concerned would impose too many levels of indirection and lead to inefficient code.

We implemented a number of benchmarks designed to assess the performance of our thread library and the overheads of using Haskell. For I/O benchmarks we tested against comparable C programs using the Native POSIX Thread Library (NPTL), which is an efficient implementation of Linux kernel threads. We also built a simple web server as a case study in using application-level threads and compared it against Apache.

Software setup: The experiments used Linux (kernel version 2.6.15) and the development snapshot of GHC 6.5, which supports multiple processors and software transactional memory. Except for the memory consumption test, we set the suggested size for GHC's garbage collected heap to be 100MB. The C versions of our 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 our tests.

Machine setup: We used different machine configurations to run different benchmarks: Those involving disk and networking I/O 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. The multiprocessor experiments (which did not need disk or network I/O) used an SMP machine with four Pentium-III 550MHz processors, a 512KB L2 cache, and 512MB RAM. Finally, the memory consumption benchmark used a dual Xeon 3.20GHz uniprocessor with 2GB RAM and 1024KB L2 Cache. Each experiment described below is labeled with IO, MP, or Mem to indicate which of the three machine configurations was used, respectively.

5.1 Benchmarks

Memory consumption (Mem): To measure the minimal run-time state needed for each application-level thread, we wrote a test program that launches ten million threads that just loop calling **`sys_yield`**. For this benchmark we set the GHC's suggested heap size for garbage collection to be 1GB. Using the profiling information from the garbage collector, the live set of mem-

[1] As pointed out by Simon P. Jones, this is an overweighted design. The mutex can be implemented completely inside the main worker threads, without using additional queues and event loops.

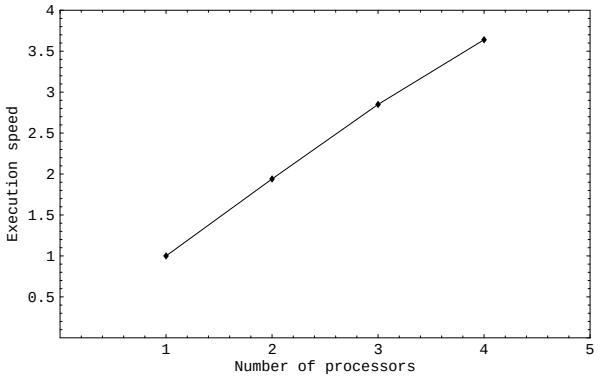

Figure 18: Speedup using multiple processors

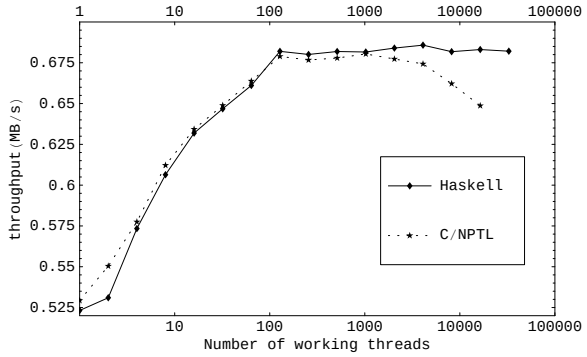

Figure 19: Disk head scheduling test

ory objects is as small as 480MB after major garbage collections—each thread costs only 48 bytes in this test.

Multiprocessor computation (MP): We tested 1024 application-level threads, each performing a CPU-intensive operation before yielding control. Figure 18 shows the speedup as the number of kernel processors increases. In this test, each processor runs its own copy of **`worker_main`** as a kernel thread and STM is used for synchronization. A 3.65 times speed up is achieved when 4 CPUs are used.

Disk performance (IO): Our test scheduler uses the Linux asynchronous I/O library, so it benefits from the kernel disk head scheduling algorithm just as the kernel threads and other event-driven systems do. We ran the benchmark used to assess Capriccio [22]: 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 19 compares the performance of our thread library with NPTL. This test is disk-bound: the CPU utilization is 1% for both programs when 16K threads are used. Our thread library outperforms NPTL when more than 100 threads are used. The throughput of our thread library remains steady up to 64K threads—it performs just like the ideal event-driven system.

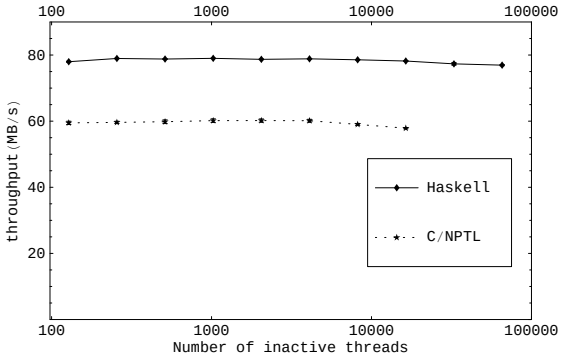

Figure 20: FIFO pipe scalability (simulating idle network connections)

FIFO pipe performance—mostly idle threads (IO): Our event-driven scheduler uses the Linux **`epoll`** interface for network I/O. To test its scalability, we 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 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 are used. Figure 20 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 our Haskell threads demonstrated good scalability in this test, but the throughput of Haskell is 30% higher than NPTL. To further investigate this performance difference, we designed the next benchmark.

FIFO pipe performance—no idle threads (IO): This benchmark is like the previous one, except that all threads are actively sending/receiving data over FIFO pipes. Figure 21 plots the total throughput as the number of active threads increases. The overall shape of two curves appears to be determined by cache performance: each thread allocates a 32KB buffer, so the memory footprint of the program quickly exceeds the CPU cache size as the number of threads increases.

The comparison between two curves is interesting: NPTL performs better when there are fewer than 32 threads, but our thread library becomes faster as more threads are used. We conjecture that this performance difference is caused because our thread library needs fewer kernel context switches. Because the FIFO pipe has a buffer size of 4KB, the C program performs a context switch after each 4KB data is transfered.

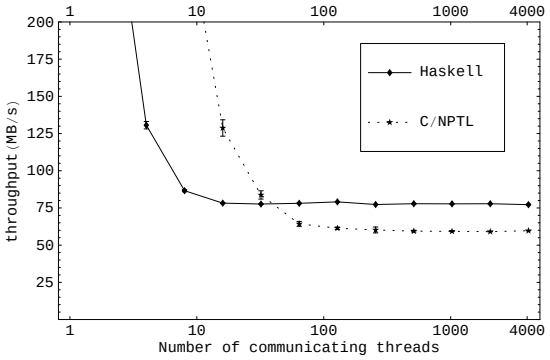

Figure 21: FIFO pipe performance (no idle threads)

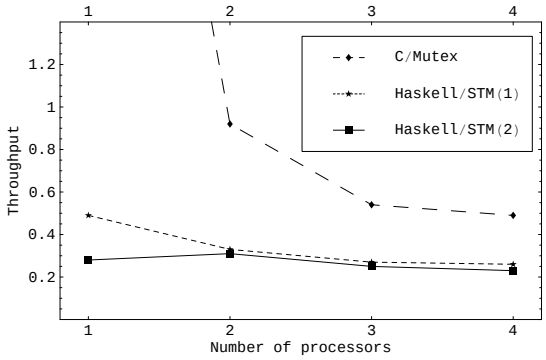

Figure 22: STM synchronization overheads

The **`worker_main`** event loop in our thread scheduler groups the data transfers from all application threads and batches them. When there are plenty of application-level threads waiting for execution, the kernel thread running **`worker_main`** does not yield control until it is preempted by other kernel threads. Therefore, our thread library causes many fewer kernel context switches when the concurrency level is high. Intuitively, our scheduler works like a pipeline: it delivers the best performance when it is fully loaded.

STM and Haskell overhead (MP): This benchmark tests the overhead of using STM transactions in Haskell. In the C program, there is one kernel thread per processor; each thread is a loop that locks a mutex, increments a shared integer, and then unlocks the mutex. Program 1 is written in Haskell using one standard kernel thread per processor. Program 2 is written in Haskell using our application-level thread library, again with one kernel thread per processor and one application-level thread for each kernel thread. In Programs 1 and 2, each thread increments the shared integer within an STM transaction. Figure 22 compares the throughput of these programs in millions of increment operations per second.

The results show that when there is no concurrency, the combination of Haskell and its STM implementation is significantly slower than C. However, as the amount of contention for the shared state increases, the difference diminishes significantly. With four processors, the C code is less than twice as fast as the Haskell code. Note that this is a worst-case scenario for software transactional memory: it assumes that, in the common case, writes from two concurrent transactions won't conflict (causing one of the transactions to roll back), but in this test every write to the shared memory is guaranteed to conflict. Also note that the difference in performance between Program 1 and Program 2 in the case of one processor is caused by the overhead of using the CPS monad in our thread library. With multiple processors, the overhead can be shared among the CPUs and the difference becomes negligible.

Memory allocation and garbage collection: The implementation of our CPS monad requires memory allocation on almost every line of code. Fortunately, GHC implements garbage collection efficiently. In the I/O tests in Figures 19, 20 and 21, garbage collection takes less than 0.2% of the total program execution time.

5.2 Case study: A simple web server

To test our approach on a more realistic application, we implemented a simple web server for static web pages using our thread library. We reused some HTTP parsing and manipulation modules from the Haskell Web Server project [16], so the main server consists of only 370 lines of multithreaded glue code. To take advantage of Linux AIO, the web server application implements its own caching. I/O errors are handled gracefully using exceptions. Not only is the multithreaded programming style natural and elegant, but the event-driven architecture also makes the scheduler clean. The scheduler, including the CPS monad, system call implementations, event loops and queues for AIO, **`epoll`**, mutexes, file-opening 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 thread scheduler are completely type-safe: debugging is made much easier because most low-level programming errors are rejected at compile-time.

Figure 23 compares our simple web server to Apache 2.0.55 for a disk-intensive load. We used the default Apache configuration on Debian Linux except that we increased the limit for concurrent connections. Using our thread library, we implemented a multithreaded client load generator in which each client thread repeatedly requests a file chosen at random from among 128K pos-

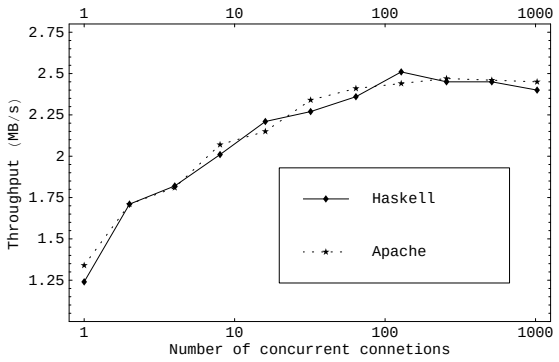

Figure 23: Web server under disk-intensive load

sible 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. Our web server used a fixed cache size of 100MB. Before each trial run we 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. Our simple web server compares favorably to Apache on this disk-bound workload.

For mostly-cached workloads (not shown in the figure), the performance of our web server is also comparable to Apache. A future work is to test our web server on more realistic workloads and implement more advanced scheduling algorithms, such as *resource aware scheduling* used in Capriccio [22].

6 Evaluation and discussion

We find the results of our experiments and our experience with implementing a web server using the unified concurrency model encouraging. Although our thread library implementation in Haskell is slower than C in terms of raw speed, it performs quite well in our tests and scales well in terms of the number of concurrent threads it can handle. In this section we discuss some of the non-quantifiable aspects of programming with this concurrency model, and describe our experience with using Haskell.

Programming experience: The primary advantage of our approach is the simplified programming model it provides: threads can be written in a natural sequential style, yet custom, event-driven schedulers can easily be defined by using the trace abstraction. As one example, our simple web server uses a file cache whose state is shared across all threads handling client connections. Implementing the caching code, modifying the server to use the cache, and debugging the implementation took under two hours. The cache implantation itself takes only 80 lines of code.

To test the extensibility of the scheduler (and as part of an ongoing project), we built an application-level TCP stack in Haskell and plugged it into our thread library. The TCP stack adds only one more event loop to Figure 12; this loop is driven by packet I/O events, timer events, and user thread requests. These scheduler changes can be made cleanly, without requiring an complete rewrite of the code. Having the TCP stack, the thread scheduler and web server in the same user level application, the programmer has complete control of the networking code.

Using Haskell: Because Haskell is a pure, lazy, functional language, we were initially concerned about performance. However, in our experience Haskell programs, while slower than C programs, are not orders of magnitude slower. The Computer Language Shootout Benchmarks [20] give other anecdotal evidence corroborating this assessment: Haskell (GHC) performs well on a wide range of tasks, and many Haskell programs perform better than their corresponding C or C++ programs. When performance or OS libraries are needed, GHC provides good interoperability with C code via its Foreign Function Interface.

In exchange for performance, Haskell delivers many features that simplify program development, including a very expressive static type system, type inference, lightweight closures, garbage collection, and convenient syntax overloading. We heavily use these features in our thread library implementation; it might be possible to implement the unified concurrency model in a general-purpose language lacking some of these features, but the results would likely be cumbersome to use. Nevertheless, it is worth investigating how to apply our approach in more mainstream languages like Java.

7 Related work

We are 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)[19] and Erlang [4], or for event-driven programming such as Esterel [6]. Java and C# also provide some support for threads and synchronization. There are also domain-specific languages, such as Flux [5], intended for building network services out of existing C libraries. Most of these approaches pick either the multithreaded or event model. Of the ones mentioned above, CML is closest to our work because it provides very lightweight threads and an event primitive for constructing new synchronization mechanisms, but its thread scheduler is still part of the language runtime.

The application-level thread library is motivated by two projects: SEDA [25] and Capriccio [22]. Our goal is to get the best parts from both projects: the event-driven architecture of SEDA and the multithreaded programming style of Capriccio. Capriccio uses compiler transformations to implement linked stack frames; our application-level threads uses first-class closures to achieve the same effect.

Besides SEDA [25], there are other high-performance, event-driven web servers, such as Flash [18]. Laurus and Parkes showed that event-driven systems can benefit from batching similar operations in different requests to improve data and code locality [14]. 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 by structuring code in CPS, such as Python's Twisted package [21] and C++'s Adaptive Communication Environment (ACE) [1]. Adya et al. [2] present a hybrid approach to automate stack management in C and C++ programming.

Multiprocessor support for user-level threads is a challenging problem. Cilk [7] uses a work-stealing algorithm to map user-level threads to kernel threads. Event-driven systems, in contrast, can more readily take advantage of multiple processors by processing independent events concurrently [26]. A key challenge is how to determine whether two pieces of code might interfere: our thread scheduler benefits from the strong type system of Haskell and the use of software transactional memory.

Our thread abstraction is inspired by Claessen's lightweight concurrency model, which also uses CPS monads and lazy data structures [8]. This paper extends Claessen's work with more practical features such as exception handling, inter-thread communication and I/O interfaces.

8 Conclusion

Events and threads should be combined into an unified programming model in general-purpose programming languages. With proper language support, application-level threads can be made extremely lightweight and easy to use. Our experiments demonstrate that this approach is practical and our programming experience suggests that this is a very appealing way of writing scalable, massively concurrent software.

References

[1] 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.

[2] 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.

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

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

[5] Emery Berger, Brendan Burns, Kevin Grimaldi, Alex Kostadinov, and Mark Corner. Flux: A Language for Programming High-Performance Servers. In *Proceedings of the 2006 Usenix Annual Technical Conference*, 2006.

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

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Parallel and Distributed Computing*, 37(1), August 1996.

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

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

[10] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable Memory Transactions. In *Proceedings of the ACM Symposium on Principles andPractice of Parallel Programming, to appear*, Jun 2005.

[11] 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.

[12] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[13] Simon 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, 21–24 1996.

[14] James R. Larus and Michael Parkes. Using cohort-scheduling to enhance server performance. In *USENIX Annual Technical Conference, General Track*, pages 103–114, 2002.

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

[16] Simon Marlow. Developing a high-performance web server in Concurrent Haskell. *Journal of Functional Programming*, 12, 2002.

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

[18] 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.

[19] J. H. Reppy. Concurrent ML: Design, Application and Semantics. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, Berlin, Heidelberg, 1993.

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

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

[22] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Ninteenth Symposium on Operating System Principles (SOSP)*, October 2003.

[23] 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.

[24] P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

[25] 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.

[26] 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.