Combining Events And Threads For Scalable Network Services

Implementation And Evaluation Of Monadic, Application-level Concurrency Primitives

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

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

This paper proposes to combine two seemingly opposed programming models for building massively concurrent network services: the event-driven model and the multithreaded model. The result is a hybrid design that offers the best of both worlds—the ease of use and expressiveness of threads and the flexibility and performance of events.

This paper shows how the hybrid model can be implemented entirely at the application level using *concurrency monads* in Haskell, which provides type-safe abstractions for both events and threads. This approach simplifies the development of massively concurrent software in a way that scales to real-world network services. The Haskell implementation supports exceptions, symmetrical multiprocessing, software transactional memory, asynchronous I/O mechanisms and application-level network protocol stacks. Experimental results demonstrate that this monad-based approach has good performance: the threads are extremely lightweight (scaling to ten million threads), and the I/O performance compares favorably to that of Linux NPTL.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Concurrent Programming; D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures, Control structures, Frameworks; D.4.1 [*Operating Systems*]: Process Management—Concurrency, Multiprocessing / multiprogramming / multitasking, Scheduling, Threads.

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

Keywords Event, Thread, Concurrency, Networking, Programming, Scalability, Implementation, Monad, Haskell.

1. Introduction

Modern network services present software engineers with a number of design challenges. Peer-to-peer systems, multiplayer games, and Internet-scale data storage applications must accommodate tens of thousands of simultaneous, mostly-idle 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 [19] 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 [25], 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 [26].

The debate over threads and events never seems to end because a programmer often has to choose one model and give up the other. For example, if a Linux C programmer uses POSIX threads to write a web server, it is difficult to use asynchronous I/O. On the other hand, if the programmer uses epoll and AIO in Linux to write a web server, it will be inconvenient to represent control flow for each client. The reason for this situation is that conventional thread abstraction mechanisms are *too rigid*: threads are implemented in the OS and runtime libraries and the user cannot easily customize these components and integrate them with the application. Although threads can be made lightweight and efficient, an event-driven system still has the advantage on *flexibility and customizability*: it can always be tailored and optimized to the application's specific needs.

The hybrid model: Ideally, the programmer would design parts of the application using threads, where threads are the appropriate abstraction (for per-client code), and parts of the system using events, where they are more suitable (for asynchronous I/O interfaces). To make this *hybrid model* feasible, not only should the system provide threads, but the thread scheduler interface must also provide a certain level of *abstraction*, in the form of *event handlers*, which hides the implementation details of threads and can be used by the programmer to construct a modular event-driven system.

Many existing systems implement the hybrid model to various degrees; most of them have a bias either toward threads or toward events. For example, Capriccio [26] is a user-level, cooperative thread library with a thread scheduler that looks very much like an event-driven application. However, it provides no abstraction on the event-driven side: the scheduler uses a low-level, unsafe programming interface that is completely hidden from the programmer.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

On the other hand, many event-driven systems uses continuation-passing style (CPS) programming to represent the control flow for each client; the problem is that CPS programs are often difficult to write and understand. Although CPS is a scalable design, it is not as intuitive as conventional multithreaded programming styles (for most programmers).

Application-level implementation: The hybrid model adopted here encourages both the multithreaded components and the event-driven components of the application be developed in a uniform programming environment. To do so, it is most convenient to implement the concurrency abstractions (both thread abstractions and event abstractions) entirely *inside* the application, using standard programming language idioms. We call this idea *application-level* implementation.

Programming concurrency abstractions entirely inside the application is a significant challenge on legacy programming language tools: for languages like C, implementing user-level threads and schedulers involves a lot of low-level, unsafe programming interfaces. Nevertheless, it is indeed feasible to implement the hybrid model directly at application-level using modern functional programming languages such as Haskell. In 1999, Koen Claessen [8] showed that (cooperative) threads can be implemented using only a monad, without any change to the programming language itself.

A hybrid framework for network services: This paper uses Claessen's technique to implement the *hybrid model* entirely inside the application and develop a framework for building massively concurrent network services. Our implementation is based on Concurrent Haskell [14], supported by the latest version of GHC [11]. Improving on Claessen's original, proof-of-concept design, our implementation offers the following:

- *True Parallelism:* Application-level threads are mapped to multiple OS threads and take advantage of SMP systems.
- *Modularity and Flexibility:* The scheduler is a customizable event-driven system that uses high-performance, asynchronous I/O mechanisms. We implemented support for Linux epoll and AIO; we even plugged a TCP stack to our system.
- *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. We also implemented blocking synchronization mechanisms such as mutexes.

Evaluation: Our hybrid concurrency framework in Haskell competes favorably against most existing thread-based or event-based systems for building network services. It provides elegant interfaces for both multithreaded and event-driven programming, and it is all type-safe! Experiments suggest that, although everything is written in Haskell, a pure, lazy, functional programming language, the performance is acceptable in practice:

- *I/O performance:* Our implementation delivers performance comparable to Linux NPTL in disk and FIFO pipe performance tests, even when tens of thousands of threads are used.
- *Scalability:* Besides bounds on memory size and other system resources, there is no limit on the number of concurrent clients that our implementation can handle. In the I/O tests, our implementation scaled to far more threads than Linux NPTL did. From the OS point of view, it is just as scalable as an event-driven system.
- *Memory utilization:* Our implementation has extremely efficient memory utilization. All the thread-local state is explicitly controlled by the programmer. As we have tested, each monadic thread consumes as little as 48 bytes at run time, and our system is capable of running 10,000,000 such threads on a real machine.

Summary of contributions: The idea of the concurrency monad is not new at all—we are building on work done by functional programming researchers. Our contribution is to experiment with this elegant design in real-world systems programming and evaluate this technique, both qualitatively and quantitatively:

1. We scaled up the design of the concurrency monad to a real-world implementation, providing elegant and flexible interfaces for building massively concurrent network services using efficient asynchronous I/O.
2. We proved that the monad-based design has good performance: it delivers optimal I/O performance; it has efficient memory utilization and it scales as well as event-driven systems.
3. We demonstrated the feasibility of the hybrid programming model in high-performance network servers, providing future directions for both systems and programming language research.

Our experience also suggests that Haskell is a reasonable language for building scalable systems software: it is expressive, succinct, efficient and type-safe; it interacts well with C libraries and APIs.

2. The hybrid programming model

This section gives some background on the multithreaded and event-driven approaches for building massively concurrent network services, and motivates the design of the hybrid model.

2.1 A comparison of events vs. threads

Programming: The primary advantage of the multithreaded 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 into multiple event handlers and represented as some form of state machine 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" [24].

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 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 [26], a user-level thread library 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

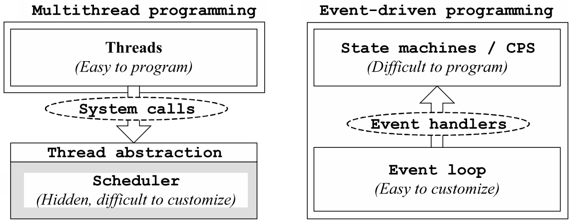

Figure 1. Threads vs. events

storage. Compared to multithreaded systems, event-driven systems can have the minimal per-thread memory overheads and context switching costs. Furthermore, by grouping similar events together, they can be batch-processed to improve code and data locality [15].

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 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 AIO, epoll and kernel event queues) provided by popular OSes are asynchronous or event-driven, because this programming model corresponds more closely to the model of 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 at the same time.

Another concern is that most user-level cooperative thread systems do not take advantages of multiple processors, and adding such support is often difficult [26]. Event-driven systems can easily utilize multiple processors by processing independent events concurrently [29].

2.2 The hybrid programming model

In 1978, Lauer and Needham [16] argued that the multithreaded and event-driven models are dual to each other. They described an 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	∼	event handler
scheduler	∼	event loop
exported function	∼	event
blocking call	∼	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. The current situation is shown in Figure 1: multithreaded systems are difficult to customize, and event-driven systems have poor programming abstractions.

We adopt a different route: rather than using the duality to justify choosing threads over events or vice versa (since either

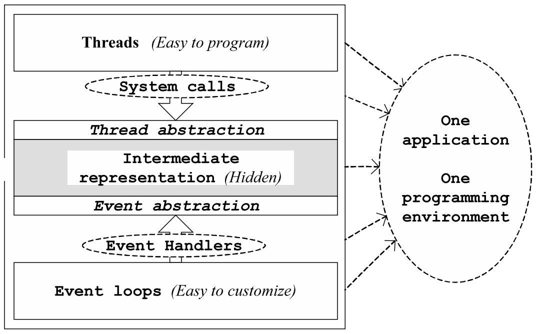

Figure 2. The hybrid model and application-level implementation

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.

Figure 2 presents the *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 underlying 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.

2.3 Application-level implementation

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 address space, be written in a different language, use very 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 2) in the *same* programming environment, by which we 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.

This problem can be solved if the thread/event abstractions (the center box of Figure 2) are also developed in the same programming environment, entirely *inside* the application as program modules. We refer to this approach as *application-level* implementation. A good approximation of this idea is event-driven systems with continuation-passing-style (CPS) event handlers [24]. In many languages, CPS can be implemented entirely inside the application, using standard programming idioms such as function closures or objects. Furthermore, the CPS approach also solves the problem of *control inversion*: CPS represents multithreaded computation, but such computation can be manipulated as values and called as event handlers. The only caveat is that CPS is not as intuitive and easy to understand as C-like, imperative threads—not every programmer can think in CPS!

The next section shows that, with a little more support from programming languages, good *abstractions* can be provided to hide the details of CPS from the programmer, and the result is an elegant implementation of the *hybrid model*.

3. Implementing the hybrid model in Haskell

The CPS programming technique provides a great foundation for application-level implementation of the hybrid model: CPS has expressive control flow, and CPS can be readily represented in many programming languages. The challenge is to build the *thread abstraction* and hide the details of continuation passing.

The simplest idea is to perform a source-to-source CPS translation [3] in the compiler. This approach is not entirely *application-level* because it requires nonstandard compiler extensions. Furthermore, because we want the multithreaded code to have the same programming environment as the scheduler code, the compiler has to translate the *same* language to itself. Such translations are often verbose, inefficient and not type-safe.

Can we achieve the same goal without writing compiler extensions? One solution, developed in the functional programming community and supported in Haskell, is to use *monads* [18, 27]. The Haskell libraries provide a *Monad* interface that allows generic programming with functional combinators. The solution we adopt 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 CPS 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. Haskell has two features that significantly simplify this programming style:

- *Operator Overloading*: Using *type classes*, the standard monad operators can be made generic, because the overloading of such operators can be resolved by static typing.
- *Syntactic Sugar:* Haskell has a special *do-syntax* for programming with monads. Using this syntax, the programmer can write monadic code in a C-like imperative programming style, and the compiler automatically translates this syntax to generic monadic operators and anonymous functions.

In 1999, Koen Claessen showed that cooperative multithreading can be represented using a monad [8]. His design extends to an elegant, 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 use this technique to multiplex I/O in network server applications.

3.1 Traces and system calls

In this paper, we use 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 4 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 4. 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 list of system calls, as shown in Figure 5. 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 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 5. System calls and their corresponding traces

Lazy evaluation of traces and thread control: We can think of a trace 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 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.

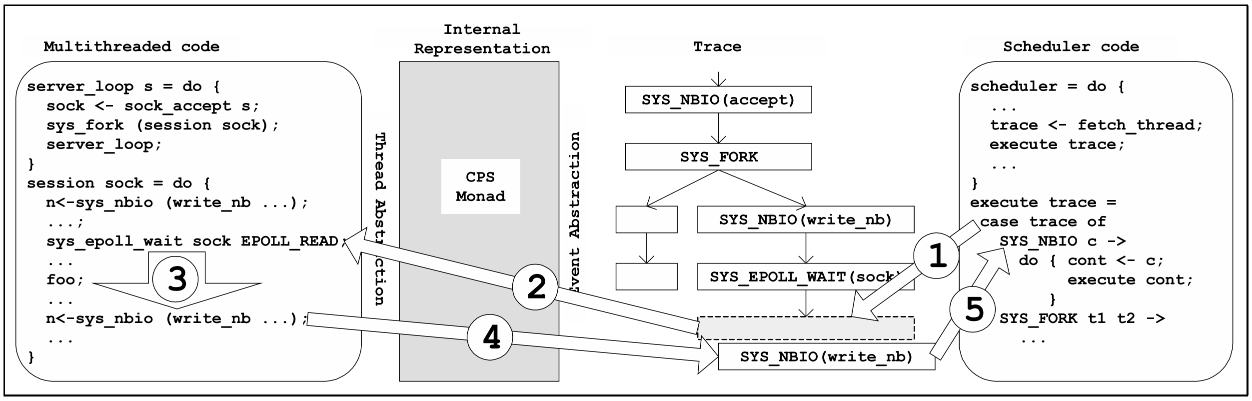

Figure 3. Thread execution through lazy evaluation (the steps are described in the text)

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.
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* we need. 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.

3.2 The CPS monad

A monad can provide the *thread abstraction* we need. System calls can be implemented as monad 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 2).

Haskell's monad typeclass, shown in Figure 6, requires a monad implemented using a parameterized abstract datatype **`m`** to support two key operations. The first operation, **`return`**, takes a value of type **`a`** and "lifts" it into the monad. The second infix operation, **`(>>=)`** (pronounced "bind"), provides a sequential composition of computations in the monad.

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Figure 6. Haskell's Monad interface (excerpt)

The monad we need encapsulates the side effects 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. 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 sequential composition) for this monad are defined in Figure 7.

```
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 7. The CPS monad **`M`**

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, since the final continuation of the whole sequential composition is **`c`**, that continuation is passed to **`h`**. Altogether, we can think of this as (roughly) doing **`g;f;c`**.

Given a computation **`M a`** 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 8 converts a monadic computation into a trace.

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

Figure 8. Converting monadic computation to a trace

In Figure 9, 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 CPS, 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).

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.

```
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 9. Implementing some system calls

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.

4. Building network services

With the concurrency primitives introduced in the previous section, we can combine events and threads to build scalable network services in an elegant way: the code for each client is written in a "cheap", monad-based thread, while the entire application is an event-driven program that uses asynchronous I/O mechanisms.

Our current implementation uses the Glasgow Haskell Compiler (GHC) [11]. It is worth noting that GHC already supports efficient, lightweight user-level threads, but we do 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, we employ only a few GHC threads, each mapped to an OS thread in the GHC runtime system. In the rest of the paper, we use the name *monadic thread* for the application-level, "cheap" threads written in the do-syntax, and we do not distinguish GHC threads from OS threads.

4.1 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 10: 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 10. Wrapping non-blocking I/O calls to blocking calls

4.2 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 11 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.

```
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 11. A round-robin scheduler for three sys. calls

In our 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.

```
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 12. System calls for exceptions

4.3 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 12. The code in Figure 13 illustrates how exceptions are used by a monadic thread.

The scheduler code in Figure 11 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.

4.4 Multiple event loops and SMP support

Figure 14 shows the event-driven system in our 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

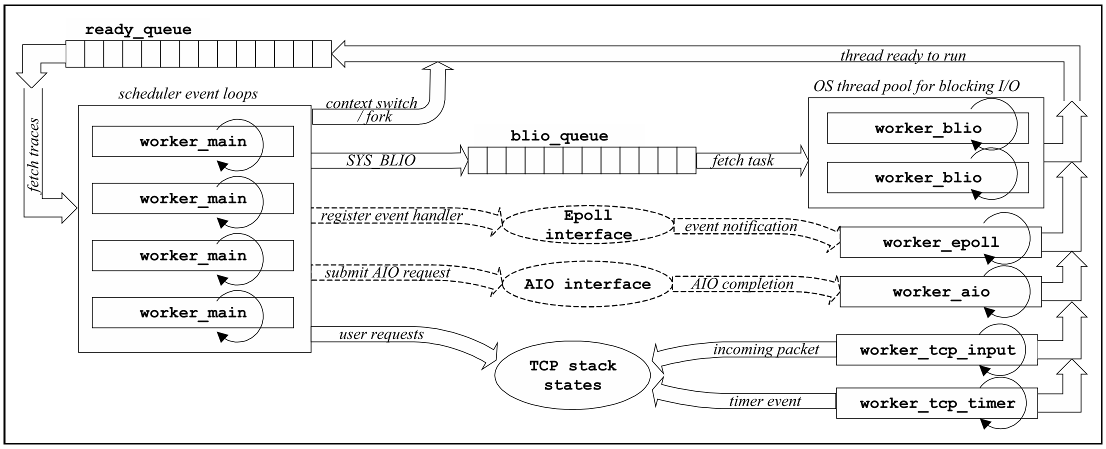

Figure 14. The event-driven system: event loops and task queues

```
-- 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 13. Multithreaded code with exception handling

execute the monadic threads and generate events that can be consumed by other event loops.

To take advantage of SMP 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 assumption that all the I/O operations submitted by **SYS_NBIO** are nonblocking 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 [28], 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. Our current design can be further improved by implementing a separate task queue for each scheduler and using work stealing to balance the loads.

4.5 Asynchronous I/O in Linux

Our 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 15.

```
-- 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 15. System calls for epoll and (read-only) 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
```

Figure 16. Dedicated event loop for epoll

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 16. 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. In our system, the programmer can easily add other asynchronous I/O mechanisms in the same way.

4.6 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 14. Similar to `sys_nbio`, we define 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.

4.7 Thread synchronization

The execution of a monadic thread is interleaved with purely functional computations and effectful computations (submitted through system calls). On SMP machines, the GHC runtime system transparently manages the synchronization of purely functional computations (e.g. concurrent thunk evaluation) in multiple OS threads. For the synchronization of effectful computations, our implementation offers several options.

For *nonblocking* synchronization, such as concurrent accesses to shared data structures, the *software transactional memory* [12] (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 computations are nonblocking (i.e. without `retry` operations), the scheduler event loops can run them smoothly without being blocked.

For *blocking* synchronization, like the producer-consumer model, that affects thread scheduling, we can define our 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 [14] can also be similarly implemented.

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

4.8 Application-level network stack

Our 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 our 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 elegant and type-safe way.

We 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 [6]. Although the development is a manual process, the purely functional programming style makes the translation from the HOL specification to Haskell straightforward.

We then glued the generic TCP code into our event-driven system in a modular fashion. In Figure 14, 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.

5. Experiments

The main goal of our tests 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. In addition, we wanted 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 that we were concerned would impose too many levels of indirection and lead to inefficient memory usage. Such programs allocate memory very frequently and garbage collection plays an important role.

We used a number of benchmarks designed to assess the performance of our Haskell implementation. 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 chose NPTL 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 defacto reference of performance.

Software setup: The experiments used Linux (kernel version 2.6.15) and GHC 6.5, which supports SMP and software transactional memory. 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.

5.1 Benchmarks

Memory consumption:[1] In our 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, we wrote a test program that launches ten million threads that just loop calling `sys_yield`. Using the profiling information from the garbage collector, we 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 our 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:[2] Our event-driven system uses the Linux asynchronous I/O library, so it benefits from the kernel disk head

[1] The memory consumption test used a machine with dual Xeon processors and 2GB RAM. We set the GHC's suggested heap size for garbage collection to be 1GB.

[2] 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. We set GHC's suggested heap size for garbage collection to be 100MB.

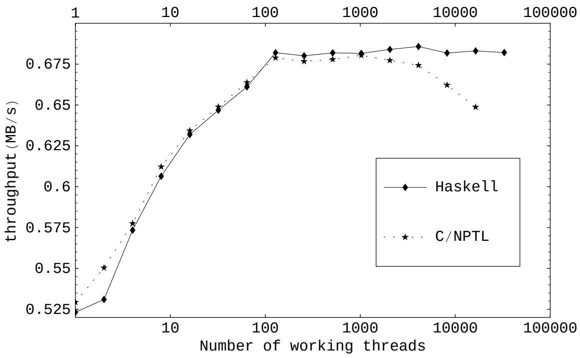

Figure 17. Disk head scheduling test

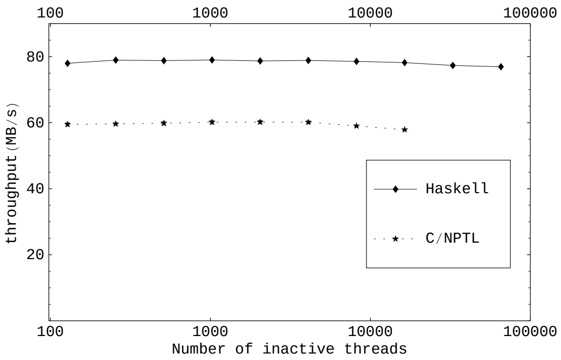

Figure 18. FIFO pipe scalability test (simulating idle network connections)

scheduling algorithm just as the kernel threads and other event-driven systems do. We ran the benchmark used to assess Capriccio [26]: 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 17 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 slightly 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.

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 is used. Figure 18 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.

In all the I/O tests in Figures 17 and 18, garbage collection takes less than 0.2% of the total program execution time.

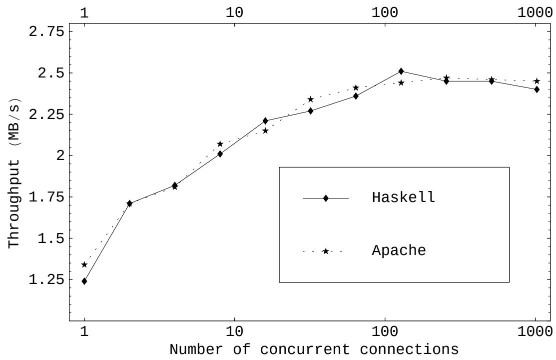

Figure 19. Web server under disk-intensive load

Although our system slightly outperforms NPTL in the above benchmarks, we are not trying to prove that our 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!

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 [17], 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 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, 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 19 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 system, we 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. Our web server used a fixed cache size of 100MB. Before each trial run we flushed the Linux kernel disk cache entirely and preloaded 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 similar to Apache. A future work is to test our web server using more realistic workloads and implement more advanced scheduling algorithms, such as *resource aware scheduling* used in Capriccio [26].

The web server also works with our 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 are able to 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, our TCP stack is a zero-copy implementation[3]; 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.

6. Related work and discussion

6.1 Language-based concurrency

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)[22] and Erlang [4], or for event-driven programming such as Esterel [5]. 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 our work because it provides very 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 [7], intended for building network services by composing existing C or Java libraries.

Rather than supporting lightweight threads directly in the language, there is also a large body of work on using language-level continuations to implement concurrency features [23, 9]. Our work uses a similar approach, except that we do not use language-level continuations — the CPS monad and lazy data structures are sufficient for our purpose.

It is worth noting that this paper 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 [10, 21] showed different approaches to combine threads and events by using explicit message passing with local event handlers.

6.2 Event-driven systems and user-level threads

The application-level thread library is motivated by two projects: SEDA [28] and Capriccio [26]. 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 [28], there are other high-performance, event-driven web servers, such as Flash [20]. Larus and Parkes showed that event-driven systems can benefit from batching similar operations in different requests to improve data and code locality [15]. 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 [24] 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. Event-driven systems, in contrast, can more readily take advantage of multiple processors by processing independent events concurrently [29]. 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.

6.3 Type-safe construction of the software stack

Our work is orthogonal, but in some sense similar to the Singularity [13] project, which constructs an operating system using type-safe languages. Singularity uses language-based abstractions to isolate processes and eliminate overheads of hardware-enforced protection domains; we use language-based abstractions to isolate software components and eliminate overheads of OS abstractions. Singularity reconstructs the traditional software stack from bottom up; our approach attempts to do so from top down.

In our hybrid programming model, many software components that are traditionally implemented in the OS, such as threads, IO libraries and network stacks, can be modularly lifted into the application and share the same programming environment. In fact, the event-driven system described in Figure 14 looks quite like a small operating system itself. This application-level approach has many advantages for building high-performance network services: the interfaces among these components are completely type-safe; each component can be customized to the application's needs; compiling them together also opens opportunities for optimizations across components.

7. Conclusion

Events and threads should be combined into a hybrid programming model in general-purpose programming languages. With proper language support, application-level concurrency primitives 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 systems software.

Acknowledgements

We would like to thank all the members of the PLClub at the University of Pennsylvania, Milo M.K. Martin, Yun Mao, Simon Peyton Jones and Simon Marlow for their help and feedbacks on this project. In addition, we would like to thank the PLDI reviewers for their valuable comments and extensive proofreading of the original draft. This work is supported by NSF grant CCF-0541040.

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.

[3] Our 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.

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

[6] 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.

[7] 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.

[8] Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3):313–323, 1999.

[9] Kathleen Fisher and John Reppy. Compiler Support for Lightweight Concurrency. Technical memorandum, Bell Labs, March 2002.

[10] 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.

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

[12] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable Memory Transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, June 2005.

[13] Galen C. Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, USA, October 2005.

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

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

[16] 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.

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

[18] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.

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

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.