Mio: A High-Performance Multicore IO Manager for GHC

Andreas Voellmy Junchang Wang
Paul Hudak
Yale University, Department of Computer Science
andreas.voellmy@yale.edu
junchang.wang@yale.edu
paul.hudak@yale.edu

Kazuhiko Yamamoto
IIJ Innovation Institute Inc.
kazu@iij.ad.jp

Abstract

Haskell threads provide a key, lightweight concurrency abstraction to simplify the programming of important network applications such as web servers and software-defined network (SDN) controllers. The flagship Glasgow Haskell Compiler (GHC) introduces a run-time system (RTS) to achieve a high-performance multicore implementation of Haskell threads, by introducing effective components such as a multicore scheduler, a parallel garbage collector, an IO manager, and efficient multicore memory allocation. Evaluations of the GHC RTS, however, show that it does not scale well on multicore processors, leading to poor performance of many network applications that try to use lightweight Haskell threads. In this paper, we show that the GHC *IO manager*, which is a crucial component of the GHC RTS, is the scaling bottleneck. Through a series of experiments, we identify key data structure, scheduling, and dispatching bottlenecks of the GHC IO manager. We then design a new multicore IO manager named *Mio* that eliminates all these bottlenecks. Our evaluations show that the new Mio manager improves realistic web server throughput by 6.5x and reduces expected web server response time by 5.7x. We also show that with Mio, McNettle (an SDN controller written in Haskell) can scale effectively to 40+ cores, reach a throughput of over 20 million new requests per second on a single machine, and hence become the fastest of all existing SDN controllers.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Run-time environments, Glasgow Haskell Compiler; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages, Concurrent, distributed, and parallel languages; D.4.1 [*OPERATING SYSTEMS*]: Process Management—Concurrency, Scheduling, Synchronization

General Terms Languages, Performance, Design

Keywords Haskell, GHC, lightweight threads, network programs, IO manager, event notification, multicore processors, concurrency, scalability

Haskell '13, September 23–24, 2013, Boston, MA, USA.

ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/2503778.2503790

1. Introduction

Haskell threads (also known as green threads or user threads) provide a key abstraction for writing high-performance, concurrent programs [16] in Haskell [12]. For example, consider a network server such as a web server. Serving a web request may involve disk IOs to fetch the requested web page, which can be slow. To achieve high performance, web servers process other requests, if any are available, while waiting for slow disk operations to complete. During high load, a large number of requests may arrive during the IOs, which gives rise to many requests in progress concurrently. A naive implementation, using one *native thread* (i.e. OS thread) per request would lead to the use of a large number of native threads, which would substantially degrade performance due to the relatively high cost of OS context switches [22]. In contrast, Haskell threads are *lightweight* threads, which can be context switched without incurring an OS context switch and with much lower overhead. Hence, lightweight Haskell threads are particularly well-suited for implementing high performance servers.

Haskell's lightweight threads reduce the incentive for programmers to abandon the simple, threaded model in favor of more complex, event-driven programming to achieve acceptable performance, as is often done in practice [22]. Event-driven programs require that a programmer reorganize a sequential program into a harder-to-understand, more complex structure: pieces of *non-blocking* code segments and a state machine that determines the operations that will be performed upon each event. In contrast, the processing logic at a network server to handle a client request is typically much easier to understand when programmed as a single thread of execution rather than as collection of event handlers [21].

Given the importance of Haskell threads, the Glasgow Haskell Compiler (GHC) [9], the flagship Haskell compiler and runtime system (RTS), provides substantial support to implement Haskell threads. For example, realizing that CPU may become a bottleneck, GHC RTS introduces a load-balancing multicore scheduler to try to leverage the current and future trend of multicore processors. To avoid memory bottlenecks, GHC RTS introduces a parallel garbage collector and efficient multicore memory allocation methods. To avoid using one native thread for each blocking I/O operation, GHC RTS introduces an *IO manager* [10, 15] to support a large number of Haskell threads over a small number of native threads. The objective of these components is to provide a highly scalable Haskell thread implementation.

Unfortunately, despite introducing many important components, GHC RTS did not scale on multicores, leading to poor performance of many network applications that try to use lightweight Haskell threads. In particular, our experiments demonstrate that even embarrassingly concurrent network servers are typically un-

able to make effective use of more than one or two cores using current GHC RTS.

Our first contribution of this paper is that we diagnose the causes for the poor multicore performance of Haskell network servers compiled with GHC. We identify that the GHC IO manager (also known as the "new IO manager" [15]) as the scaling bottleneck. Through a series of experiments, we identify key data structure, scheduling, and dispatching bottlenecks of the GHC IO manager.

Our next contribution is that we redesign the GHC IO manager to overcome multicore bottlenecks. Our new design, called *Mio*, introduces several new techniques: (1) *concurrent callback tables* to allow concurrent use of the IO manager facilities, (2) *per-core dispatchers* to parallelize the work of dispatching work to waiting threads, improve locality, and reduce cross-core interactions and (3) *scalable OS event registration* to reduce the use of non-scalable and expensive system calls. The new techniques are all "under-the-hood" and will apply transparently to all Haskell programs without modification. Mio is simple to implement, with only 874 new lines of code added to the GHC code base and 359 old lines of code deleted. Mio will be released as part of GHC 7.8.1.

We comprehensively evaluate Mio using two network applications. With Mio, realistic HTTP servers in Haskell scale to 20 CPU cores, achieving peak performance up to factor of 6.5x compared to the same servers using previous versions of GHC. The latency of Haskell servers is also improved: using the threaded RTS with Mio manager, when compared with *non-threaded* RTS, adds just 6 microseconds to the expected latency of the server under light load, and under a moderate load, reduces expected response time by 5.7x when compared with previous versions of GHC. We also show that with Mio, McNettle (an SDN controller written in Haskell), can scale effectively to 40+ cores, reach a throughput of over 20 million new requests per second on a single machine, and hence become the fastest of all existing SDN controllers.

The rest of the paper is organized as follows. Section 2 presents background on the design and API of the GHC IO manager. Section 3 shows problems of the GHC IO manager and our approaches to overcome them. Section 4 and 5 discuss implementation details and OS bottlenecks and bugs, respectively. Section 6 presents an evaluation of our techniques. Sections 7 and 8 present related work and conclusion, respectively.

2. Background: GHC Threaded RTS

A GHC user can choose to link a Haskell program with either threaded RTS or non-threaded RTS. In this section, we briefly review the operation of the threaded RTS and the GHC IO manager, which are presented in more detail in [10, 11, 15], respectively.

2.1 Threaded RTS Organization

A concurrent Haskell program is written using Haskell threads typically created as a result of invocations of *forkIO* by the user program. These Haskell threads are multiplexed over a much smaller number of native threads. Figure 1 shows the structure of GHC's threaded RTS. A Haskell program running with the threaded RTS makes use of N cores (or CPUs), where N is typically specified as a command-line argument to the program. The RTS maintains an array of *capability* (also called a *Haskell Execution Context (HEC)*) data structures with each capability maintaining the state needed to run the RTS on one core. This state includes a run queue of runnable Haskell threads and a message queue for interactions between capabilities. At any given time, a single native thread is executing on behalf of the capability and holds a lock on many of its data structures, such as the run queue. The native thread running the capability repeatedly runs the thread scheduler, and each iteration of the scheduler loop processes the message queue, balances its run queue with other idle capabilities (i.e. performs load balancing) and executes the next Haskell thread on its run queue (among other things).

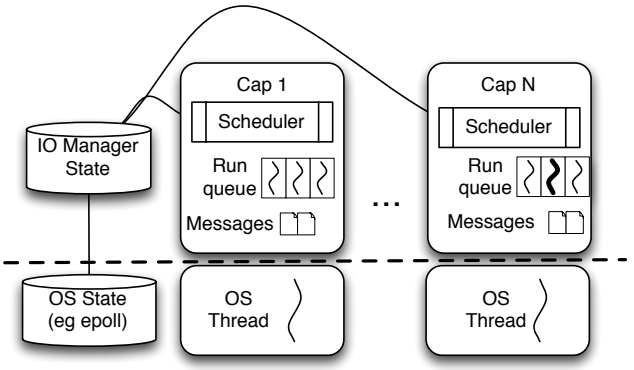

Figure 1. Components of the threaded RTS, consisting of N capabilities (caps), each running a scheduler which manages its capability's run queue and services its messages from other capabilities. At any given time, a single native thread is executing the capability. The system uses a single GHC IO manager component, shared among all capabilities.

In order to support blocking foreign calls, the RTS may in fact make use of more than N native threads. When a Haskell thread performs a foreign call that may block, the native thread running the capability releases the capability and creates a new native thread to begin running the capability, and then performs the blocking OS call. This allows the RTS to continue running other Haskell threads and to participate in garbage collections, which requires synchronization across all capabilities. In fact, a pool of native threads is maintained per capability so as to avoid creating a new native thread for each blocking call.

2.2 GHC IO Manager

Although the GHC IO manager is an ordinary Haskell library, distributed as part of the Haskell base libraries, it is in fact tightly coupled with the threaded RTS, which is primarily written in C. During the RTS initialization sequence, the RTS calls a function implemented in the GHC IO manager library (i.e. in Haskell) to initialize the state of the GHC IO manager. This function initializes a value of type $EventManager$ and writes this to a global variable. This value is indicated as the state labelled "IO Manager State" in Figure 1. The $EventManager$ structure contains several pieces of state, including a *callback table*, which keeps track of the events that are currently being waited on by other Haskell threads. The callback table is a search tree, keyed on file descriptor, and has type $IntMap\ [FdData]$. Each value stored in the table is a list of $FdData$ values, where an $FdData$ value includes the file descriptor, a unique key, the type of operation (i.e. read or write) being waited on, and a callback function to invoke when the operation is ready. Specifically, the GHC IO manager provides the following API to retrieve the GHC IO manager and to register or unregister interest in file-related events:

$$
\begin{aligned}
&getSystemEventManager :: IO\ (Maybe\ EventManager)\\
&registerFd :: EventManager\\
&\qquad \rightarrow IOCallback \rightarrow Fd \rightarrow Event \rightarrow IO\ FdKey\\
&unregisterFd :: EventManager \rightarrow FdKey \rightarrow IO\ ()
\end{aligned}
$$

The Fd parameter is the file descriptor of an open file and the $Event$ parameter is either $Read$ or $Write$. The $FdKey$ value returned by a call to $registerFd$ includes both the file descriptor and the unique key, uniquely identifying the subscription. The

unique key is automatically generated by the GHC IO manager in $registerFd$ and is used to allow subscriptions on the same file descriptor by multiple threads, with each thread able to cancel its subscription independently.

The callback table is stored in an $MVar$ [16] in a field of the $EventManager$ value. An $MVar$ is essentially a mutable variable protected by a mutex. In addition, an $MVar$ maintains a FIFO queue of threads waiting to access the variable, ensuring that threads are granted fair access to the variable. The $registerFd$ function takes the callback table lock, and then inserts (or modifies) an entry in the callback table and registers the event subscription using the underlying OS event notification mechanism, for example `epoll` [8] on Linux, `kqueue` [7] on BSD variants, and `poll` on other OSes, and finally restores the callback table lock. Similarly, $unregisterFd$ takes the callback table lock, removes the appropriate entry, removes the event subscription from the underlying OS event queue, and then restores the lock. Note that the callback table is stored in an $MVar$ in order to control concurrent access to both the callback table and the OS event queue, since Haskell threads executing on different capabilities may attempt to register events concurrently, possibly on the same file descriptor.

The GHC IO manager initialization function also forks a Haskell thread, called the *dispatcher thread*. The dispatcher thread executes a poll loop, repeating the following steps. It performs a blocking call to the appropriate OS function to wait on all registered events (e.g. `epoll_wait` on Linux). When it returns from the call, it retrieves the ready events, and for each one, it takes the callback table lock, retrieves the relevant callback, restores the lock and then invokes the provided callback. Having dispatched all the callbacks, the dispatcher thread repeats the process indefinitely. The dispatcher thread is shown in Figure 1 as the bold Haskell thread on capability N's run queue. With this approach, the RTS uses a single extra native thread to perform a blocking system call, instead of using one native thread per Haskell thread performing a blocking system call.

2.3 OS Event Notification

As alluded to above, each OS provides one or more event notification facilities. Some mechanisms, such as `select` and `poll`, offer a single system call which accepts an argument listing all the events of interest to the caller and blocks until at least one of those events is ready; upon returning the system call indicates which of the input events have occurred. Other mechanisms, such as `epoll` on Linux and `kqueue` on BSD allow the program to register or unregister interest in events, and then separately wait on all registered events. This second method is more efficient when a large number of files is being monitored, since it avoids passing the full list of events of interest to the OS each time the program checks for event readiness. For concreteness, Figure 2 shows the API provided by `epoll`.

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *event);
int epoll_wait(int epfd,
               struct epoll_event *events,
               int maxevents, int timeout);
```

Figure 2. `Epoll` API.

Specifically, `epoll_create` creates a new `epoll` instance and returns a file descriptor for the instance. `epoll_ctl` registers or unregisters (depending on the value of `op`) interest in an event on file descriptor `fd` for events indicated by the `event` pointer with the `epoll` instance `epfd`. In particular, `struct epoll_event` includes a bitset (also known as an *interest mask*) indicating which event types, such as `EPOLLIN` and `EPOLLOUT` (readable and writable, respectively) the caller is interested in. Finally, `epoll_wait` waits for events registered with `epoll` instance `epfd` and receives up to `maxevents` events into the `events` array, or returns with no events if `timeout` time has elapsed.

2.4 Thread API

The file registration API presented in the previous section is used to implement more convenient functions used by Haskell threads. In particular, the GHC IO manager provides the following two Haskell functions:

$$threadWaitRead, threadWaitWrite :: Fd \to IO\ ()$$

These functions allow a Haskell thread to wait until the OS indicates that a file can be read from or written to without blocking. They are logically blocking, i.e. the calling Haskell thread will appear to block until the requested operation can be performed by the OS.

These two functions are typically used in the following way. Haskell programs mark any files (e.g. sockets) they use as non-blocking. Then, when performing a read or write on a file, the OS will either perform the operation without blocking, or will return an error code indicating that the operation could not be performed without blocking. The Haskell thread then typically handles this condition by executing the $threadWaitRead$ or $threadWaitWrite$ as appropriate. This call will only return when the file is readable or writable, at which point the thread will attempt the preceding sequence again. More concretely, a thread may perform the following command to send a number of bytes on a given socket:

```
send :: Fd → Ptr Word8 → Int → IO Int
send sock ptr bytes = do
  result ← c_send sock ptr (fromIntegral bytes) 0
  if result ≡ −1 then do
      err ← getErrno
      if err ≡ eAGAIN then do
          threadWaitWrite sock
          send sock ptr bytes
        else
          error "unexpected error"
    else
      return (fromIntegral result)
```

In this code, $sock$ is the file descriptor of an open socket, ptr is a pointer to a byte array, and $bytes$ is the number of bytes to send from the beginning of ptr. We assume that we have a function c_send that calls the `send()` system call of the underlying OS. We also assume that we have a function $getErrno$ that provides the error number returned and a constant $eAGAIN$ that is used by the OS to indicate when the operation would block. Such code is typically implemented in a library that provides more convenient functions. For example, the $Network.Socket.Bytestring$ library provides a function $sendAll$ whose implementation is similar to that given for $send$ above.

$threadWaitRead$ and $threadWaitWrite$ are both implemented using $threadWait$, shown in Figure 3. $threadWait$ first creates a new, empty $MVar$ (distinct from the $MVar$ holding the callback table), which we call the invoking *thread's wait variable*, and uses $getSystemEventManager$ to get the manager value. It then registers a callback using $registerFd$ for the current file descriptor and appropriate $Event$ (i.e. read or write event), and then proceeds to wait on the initially empty $MVar$. Since it encounters an empty $MVar$ (typically), the scheduler for the capability running this thread will remove the current thread from its run queue. Later, when the OS has indicated that the event is ready, the dispatcher

thread will invoke the registered callback, which first unregisters the event (using $unregisterFd\_$) with the IO manager and then writes to the thread's wait variable, causing the scheduler to make the original thread runnable again. As a result, the original thread will be placed on the run queue of the capability on which it was previously running. At this point, the $threadWait$ function returns and the original thread continues.

```
threadWait :: Event → Fd → IO ()
threadWait evt fd = mask_ $ do
  m ← newEmptyMVar
  Just mgr ← getSystemEventManager
  let callback reg e = unregisterFd_ mgr reg ≫ putMVar m e
  reg ← registerFd mgr callback fd evt
  evt' ← takeMVar m `onException` unregisterFd_ mgr reg
  when (evt' `eventIs` evtClose) $ ioError $
    errnoToIOError "threadWait" eBADF Nothing Nothing
```

Figure 3. Method for a thread to wait on an event.

3. Analysis & Multicore IO Manager Design

In this section, we demonstrate the bottlenecks in the GHC IO manager and the techniques we use to overcome these. We first introduce a simple HTTP server written in Haskell, which we will use throughout this section. We then diagnose and solve bottlenecks one at a time, with each solution revealing a subsequent bottleneck at a higher number of cores. In particular, we explain our three main techniques to improve performance: concurrent callback tables, per-core dispatchers and scalable OS event registration.

3.1 The Simple Server

We illustrate the problems with the GHC IO manager by using a concurrent network server written in Haskell, called `SimpleServer`[1], that *should* be perfectly scalable: it serves each client independently, using no shared state and no synchronization between clients (e.g. locks) anywhere in the Haskell program. With a sufficient number of concurrent clients, this is a plentiful, perfectly parallelizable workload. As a result, we would expect that as we utilize more cores, `SimpleServer` should be able to serve more requests per second, provided there are a sufficient number of clients. Unfortunately, as we will show, with the GHC IO manager, this is not the case.

`SimpleServer` is a drastically simplified HTTP server. Its key parts are listed in Figure 4. The server consists of a main thread that starts a listening socket and then repeatedly accepts incoming connections on this socket, making use of the `network` package for basic datatypes and functions like $accept$. The main thread then forks a new worker (Haskell) thread for each newly accepted connection. A worker thread then repeatedly serves requests. To handle a single request, it receives $ByteString$ values (using the $Network.Socket.ByteString.recv$ function) until a single request of length $requestLen$ has been received, and then sends a single response (using the $Network.Socket.ByteString.sendAll$ function). The connection is closed when the worker thread receives a zero length $ByteString$. The $sendAll$ and $recv$ functions internally call $threadWaitWrite$ and $threadWaitRead$ whenever the respective operation would block, much as the $send$ function shown earlier does.

This highly simplified server performs no disk access, performs no HTTP header parsing, and allocates only one $ByteString$ for responses. Using this simplified server allows us to avoid application-level bottlenecks that may arise from functions such as HTTP parsing and protection against various denial of service attacks that are required in real HTTP servers. In Section 6, we evaluate more realistic web servers.

[1] `https://github.com/AndreasVoellmy/SimpleServer`

```
main :: IO ()
main = do
    listenSock ← startListenSock
    forever $ do
      (sock, _) ← accept listenSock
      forkIO $ worker sock

worker :: Socket → IO ()
worker sock = loop requestLen
  where
    loop left
      | left ≡ 0 = do
        sendAll sock reply
        loop requestLen
      | otherwise = do
        bs ← recv sock left
        let len = B.length bs
        when (len ≢ 0) $ loop (left − len)
```

Figure 4. Main parts of `SimpleServer`.

We evaluate `SimpleServer` using a closed system, connecting multiple clients to the server where each client repeatedly sends a request, waits for the response, and then repeats. The clients generate requests with the fixed length expected by `SimpleServer`. Throughout this section, we evaluate the server's throughput, in requests/second, using 400 concurrent clients. We run `SimpleServer` and the clients on different Linux servers using a setup described in detail in Section 6.1.

3.2 Concurrent Callback Tables

Unfortunately, as the curve labelled "Current" in Figure 5 shows, the GHC IO manager scales poorly, reaching a maximum performance at 2 cores serving 37,000 requests per second and then declining, with per-core performance rapidly deteriorating. Despite our application being perfectly parallelizable, the RTS essentially forces a completely sequential execution.

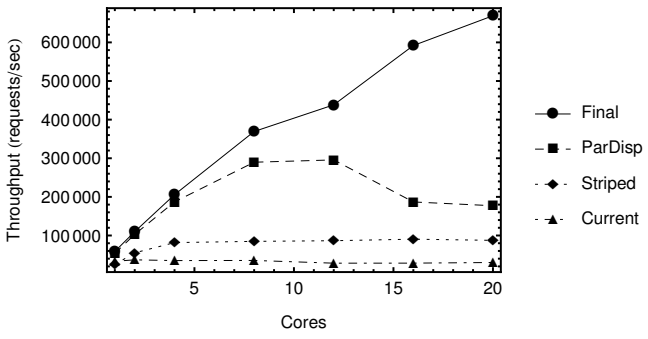

Figure 5. Throughput of `SimpleServer` shown as number of requests served per second. "Current" means it runs with the GHC IO manager. "Striped", "ParDisp" and "Final" are described in Sections 3.2, 3.3 and 3.4, respectively.

The most severe problem in the GHC IO manager is that a single $MVar$ is used to manage the callback table. The problem becomes evident by carefully examining an *event log* [6] recorded during an execution of `SimpleServer`. By compiling and running `SimpleServer` with event logging turned on, an execution of `SimpleServer` generates a log file that records the timing of various events, such as thread start and stop times, thread wakeup messages, thread migrations, and user-defined events. We then visual-

ize the event logs offline, using the Threadscope program[2], which produces graphical timelines for an event log. Threadscope produces one timeline for each capability (labelled "HEC") and also produces an overall utilization timeline (labelled "Activity"). The capability timelines are white when no Haskell thread is being executed and are light gray when executing a Haskell thread. If there is sufficient space, the thread ID is indicated in the light gray region indicating a running Haskell thread. In the event logs shown in this paper, we use solid, black bars to indicate when one thread sends a thread wakeup message for another Haskell thread, and we use dashed, gray bars to indicate when a `SimpleServer` worker thread calls $threadWaitRead$ on the thread's socket.

Figure 6 shows a detailed fragment of an event log recorded when running `SimpleServer` using the GHC IO manager and using 4 capabilities (i.e. $N = 4$) and which demonstrates contention on the callback table variable. In particular, we see that thread 18 on HEC 1 attempts to wait on its socket (the dashed, gray bar), but fails due to the callback table variable already being taken, in this case by thread 2, which is the dispatcher thread of the GHC IO manager. Therefore, thread 18 enqueues itself on the callback table variable wait queue and is removed from the run queue, after which HEC 1 becomes idle. Eventually, when the dispatcher thread (thread 2) runs again on HEC 0 (after migrating from HEC 3 just moments earlier), it releases the callback table variable and HEC 1 is messaged to indicate that thread 18 is next in line to take the callback table variable (the solid line on HEC 0 following thread 18's wait). The dispatcher thread then immediately attempts to take the callback table variable again, causing it to be queued and descheduled. Just after this, thread 50 on HEC 2 also attempts to wait on its socket (dashed gray line), but also finds the callback table variable taken, and therefore also enqueues itself on the variable's waiter queue. HEC 1 then runs thread 18, which finishes its registration, returns the lock, and notifies HEC 0 to run thread 2 again (second solid bar for thread 18). When thread 2 runs again, it finishes its work, in this case dispatching a new event to thread 13, and notifies HEC 2 for thread 50.

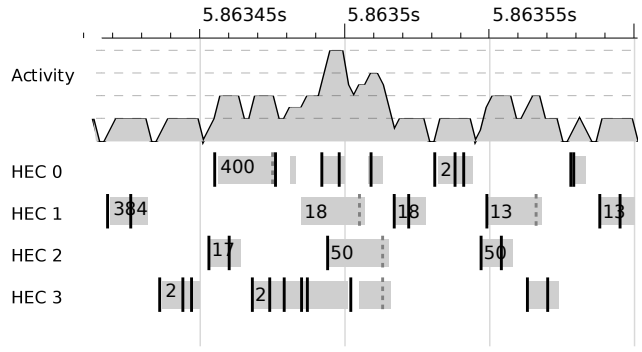

Figure 6. Event log fragment from an execution of `SimpleServer` with the GHC IO manager and 4 capabilities and illustrating contention for the callback table variable.

This sequence shows in detail that not only are workers interfering with each other in order to register interest in events, but workers may interfere with the dispatcher, preventing the dispatcher from placing runnable threads on the run queues of idle cores. As the wait queues on the callback table variable build up, the dispatcher thread may ultimately have to wait its turn in a long line to get access to the callback table. This results in the episodic behavior seen in Figure 7. When the dispatcher gains access to the callback table, most activity has died down, and it is able to dispatch a large number of threads. Subsequently, these threads serve their connections and then as they attempt to wait, large queues form, and the dispatcher thread is unable to run long enough to dispatch any more work. Thus, system activity begins to decline, until the dispatcher gains access and starts the cycle again.

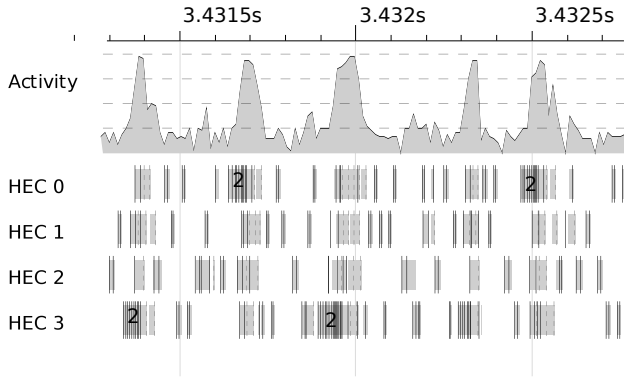

Figure 7. Event log fragment from an execution of `SimpleServer` with the GHC IO manager and 4 capabilities: lock contention leads to HECs mostly idling.

We can resolve this severe contention by using a data structure that allows for more concurrency. We apply a simple solution, which we call *lock striping*: we use an array of callback tables and hash file descriptors to array locations. Specifically, we change the type of the callback table field of the $EventManager$ data type from $MVar\ (IntMap\ [FdData])$ to $Array\ Int\ (MVar\ (IntMap\ [FdData]))$ and use a fixed array size of 2^5. With this data structure, registration or notification on a file descriptor simply performs the original registration or notification procedure to the callback table at the location that the file descriptor hashes to. Hence, registrations or notifications on different file descriptors do not interfere. We store the callback table in each array location in an $MVar$ so that we can perform both the callback table update and the operation on the OS event subsystem in one atomic step for any file descriptor. This prevents a race condition in which two updates on the same file descriptor are applied to the callback table in one order and to the OS event subsystem in the reverse order.

To illustrate the effectiveness of this technique, we measure performance applying only lock striping to the GHC IO manager. The curve labelled "Striped" in Figure 5 shows that the resulting server scales better through 4 cores, reaching a performance of over 80,000 requests per second, more than doubling peak performance of the program using the GHC IO manager.

Figure 8 shows a segment of an event log taken with 4 cores that indicates in more detail how the severe contention is relieved and how work is dispatched more smoothly. The activity time-graph is close to maximum throughout this segment, indicating that all four cores are now effectively utilized. In more detail, we can see the dispatcher thread (thread 2) on HEC 0 is able to dispatch work to threads (indicated by solid black bars on HEC 0) at the same time that the worker threads serve client requests and register callbacks to wait on again (dashed bars on HECs 1, 2 and 3).

3.3 Per-Core Dispatchers

Unfortunately, concurrent callback tables are not sufficient to allow `SimpleServer` to continue scaling beyond 4 cores. The event log fragment shown in Figure 9 is taken from a run with 8 cores and provides insight into the problem. In particular, we see that the dispatcher thread (on HEC 1) is very busy, while other HECs are mostly idle. We see that as the dispatcher thread notifies each

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

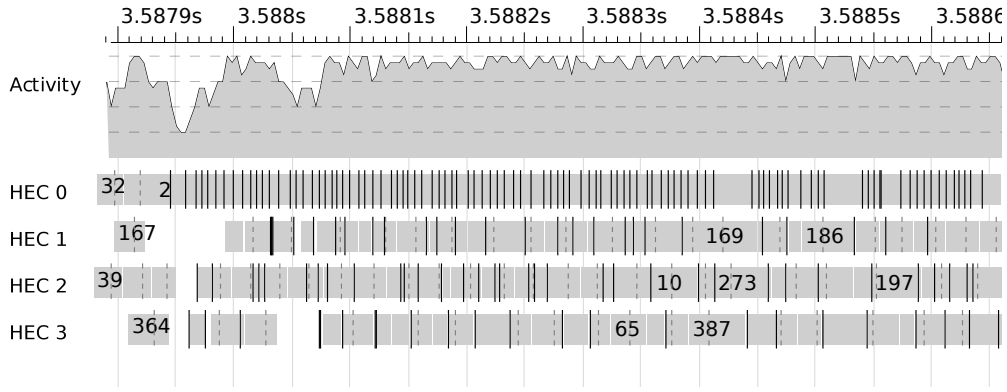

Figure 8. Event log fragment from an execution of `SimpleServer` using the concurrent callback table and 4 capabilities and illustrating concurrent registrations and dispatching.

thread, the notified thread quickly executes and finishes. The dispatcher is simply unable to create work fast enough to keep 7 other cores busy. The underlying problem is that GHC IO manager design essentially limits the notification work to consume no more than 1 CPU core, and this workload (`SimpleServer`) results in full utilization of the dispatcher component, creating a bottleneck in the system.

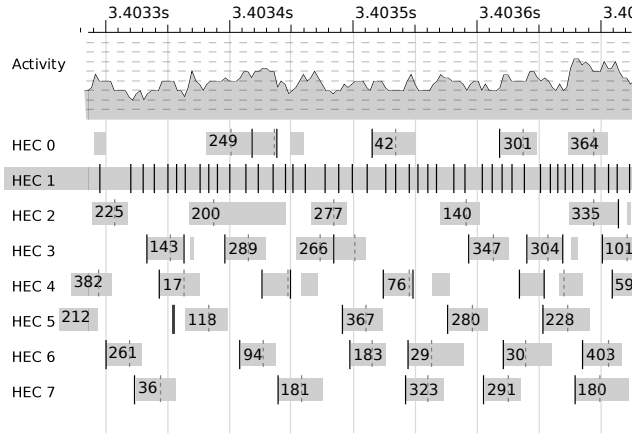

Figure 9. Timeline for an execution of `SimpleServer` using concurrent callback tables and 8 capabilities: The dispatcher thread on HEC 1 is fully utilized and has become a bottleneck.

We solve this problem by introducing per-core dispatcher threads, effectively eliminating the restriction of 1 CPU core for dispatching work. In particular, the intialization sequence of the RTS creates an array of N $EventManager$ values, where N is the number of capabilities used by the program. We also fork N dispatcher threads, each pinned to run only on a distinct capability, and each monitoring the files registered with its respective $EventManager$. We modify the $threadWait$ functions so that they register the callback with the manager for the capability that the calling thread is currently running on. This design allows each capability to perform the dispatching for Haskell threads running on its capability. Since the RTS scheduler balances threads across capabilities, this often leads to balancing the dispatching work as well. Furthermore, since a newly awoken thread is scheduled on the same capability that it previously ran on, the dispatcher's invocation of callbacks does not cause any cross-capability thread wakeup messages to be sent, improving core locality.

Unfortunately, this design introduces a problem that threatens to negate its benefits: a dispatcher thread often makes blocking OS calls, and therefore relinquishes its HEC and causes a context switch to another native thread which begins executing the HEC's work [11]. Using one dispatcher per HEC increases the frequency with which this expensive operation occurs. For example, running `SimpleServer` using per-core dispatchers with 8 capabilities, we incur 35,000 context switches per second.

To alleviate this problem, we modify the dispatcher loop, to first perform a non-blocking poll for ready events (this can be accomplished with `epoll` by calling `epoll_wait` with a `timeout` of 0). If no events are ready, the dispatcher thread yields (by calling the $Control.Concurrent.yield$ function), causing the scheduler to move it to the end of its capability's run queue. When the dispatcher thread returns to the front of the run queue, it again performs a non-blocking poll, and only if it again finds no ready events, then it performs a blocking OS call, which removes it from the capability's run queue. This optimization is effective when events become ready frequently, as the dispatcher collects events without incurring context switches. In the case that events become ready infrequently, the loop falls back to blocking calls. With this revision, the context switch rate for `SimpleServer` drops to 6,000 per second when using 8 capabilities.

The curve labelled "ParDisp" in Figure 5 shows the improved scaling resulting from applying per-core dispatchers with the aforementioned optimized dispatcher loop. `SimpleServer` now serves nearly 300,000 requests per second using 8 cores, achieving over triple the throughput of "Striped". The event log shown in Figure 10 shows how the 8 capabilities are now fully utilized and that the work of dispatching is now distributed over many capabilities, with HECs 0, 4, 5, and 7 visibly dispatching threads in this event log fragment (the closely spaced solid bars indicate dispatching of threads).

3.4 Scalable OS Event Registration

Even after adding per-core dispatchers, the server stops scaling after about 8 cores, despite all cores being fully utilized. The event log fragment of Figure 10 provides insight into the problem. Although many worker threads finish quickly, returning to wait on their socket, some workers, for example thread 380 on HEC 6 and thread 126 on HEC 3 take 2-4 times as much time to service a request as other threads. Most interestingly, a large fraction of the time for these anomalous threads occurs *after* they request to wait on their socket (the dashed gray bar). For example thread 126 requires more than 100 microseconds to register its callback with the IO manager and block on its wait $MVar$, whereas thread 24 on

HEC 1 finishes in around 20 microseconds. While at 8 cores, these slow-down occurrences are a small fraction of the overall thread executions, the frequency of this behavior becomes larger as more cores are added. This clearly suggests shared memory contention for some data structures shared among the capabilities.

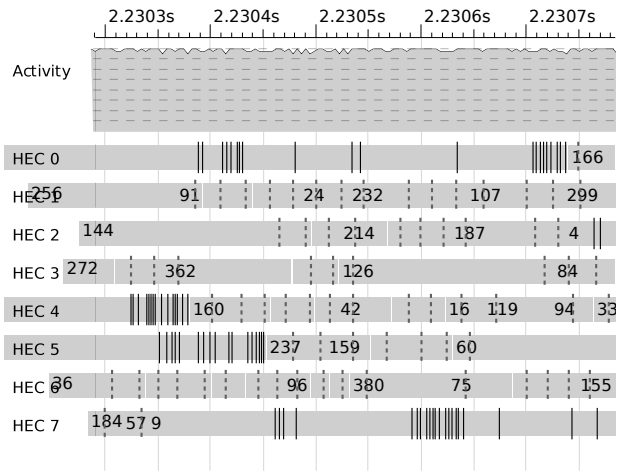

Figure 10. Timeline of `SimpleServer` with per-core dispatcher threads. All HECs are busy and dispatcher threads are running on several HECs.

By performing detailed profiling, we observed that the poor scaling is due to a global Linux kernel lock in the `epoll` subsystem used by the IO manager on our Linux server. This lock, named `epmutex`, is taken whenever an event is newly registered or deleted from an `epoll` object. This lock is required to allow `epoll` instances to be nested without forming cycles and is a truly global lock affecting operations on distinct `epoll` instances. Unfortunately, in the GHC IO manager, every register and unregister call invokes an `epoll` operation that causes `epmutex` to be taken. Specifically, $threadWait$ adds a new event to its `epoll` instance. Later, when the event is retrieved from the OS and the callback is invoked, the callback unregisters the subscription, causing the IO manager to delete the subscription from the `epoll` object. Therefore, every $threadWait$ call requires a global lock to be taken twice. This results in increased contention as more cores are used. Furthermore, a contended lock may cause the native thread running the capability to be descheduled by the OS (which does not, unfortunately, show up in the event log trace).

We avoid this bottleneck by *reusing* `epoll` registrations and thereby avoiding taking `epmutex` on every operation. In particular, when we register an event, we first attempt to modify an existing registered event. Only if that fails – and indeed it will fail on the first registration on the file – we register a new event. Moreover, we register the event subscription in *one-shot mode*; event subscriptions registered in one-shot mode are automatically disabled by `epoll` after the application retrieves the first event for the subscription. Using one-shot mode allows us to safely leave the existing registration in place and has the added benefit that no extra `epoll_ctl` call is needed to delete the event registration.

The curve labelled "Final" in Figure 5, shows the result of applying this optimization on top of the previous improvements. We see greatly improved scaling through 20 cores, serving up to 668,000 requests per second at 20 cores, more than 18 times throughput improvement over the same program using the threaded RTS with the GHC IO manager. Furthermore, Mio improves the single core performance of the server by 1.75x: with Mio, the server serves 58,700 requests per second while with the GHC IO manager it serves only 33,200 requests per second.

4. Implementation

Mio has been fully implemented, has been incorporated into the GHC code base, and will be released as part of GHC 7.8.1. Mio includes 874 new lines of code and 359 old lines of code deleted. Of these changes, 21 lines were added in the RTS code written in C. Most of these 21 lines are routine boilerplate while 7 are needed to support dynamically changing number of capabilities. In the following sections we describe our support for non-Linux OSes, Mio's treatment of timers and several minor GHC RTS bugs and performance problems that we helped to address while implementing Mio.

4.1 BSD and Windows Support

On BSD systems such as FreeBSD, NetBSD and OpenBSD, Mio uses the `kqueue` subsystem, as `epoll` is unavailable on these platforms. The `kqueue` API provides two C functions as listed in Figure 11. The `kqueue` function creates a new `kqueue`. The `kevent` function takes a `changelist` array to specify events to be registered, unregistered, or modified and an `eventlist` array to receive events. Hence, this function may carry out both event registration and event polling at the same time. If `eventlist` is a `NULL` pointer, `kevent` performs event registration only, similar to the function performed by the `epoll_ctl`. Similarly, `kevent` can play the same role as `epoll_wait` by providing a `NULL` pointer for the `changelist` argument. Moreover, the `kqueue` subsystem also supports *one-shot* registrations. Hence, BSD variants can be supported using essentially the same approach we described in Section 3.

```
int kqueue(void);
int kevent(int kq,
  const struct kevent *changelist, int nchanges,
  struct kevent *eventlist, int nevents,
  const struct timespec *timeout);
```

Figure 11. `Kqueue` API.

Mio also uses `kqueue` on OS X, since Darwin, the foundation for Apple's OS X, is a variant of BSD. However, we encountered problems running parallel builds of the GHC compiler using Mio as described above. We have been unable to uncover the underlying source of the problem to our satisfaction. However, several Internet discussions suggest that the implementation of `kqueue` and `pipe` on OS X are unstable. We were able to resolve the observed problems on OS X by avoiding the use of one-shot mode on OS X, instead unregistering events after they are delivered, and by sending wakeup writes to a pipe monitored by the dispatcher threads' kqueue instances on every event registration. With these changes, the behavior on OS X, including parallel builds of GHC, has been stable.

For UNIX platforms not supporting either `epoll` or `kqueue`, Mio utilizes the `poll` backend. Since `poll` does not provide a separate registration function, a Haskell thread must register interest for an event by adding it to a queue of events that should be registered and then interrupting the dispatcher if it is currently in a blocking `poll` call. The registration function performs this by sending a control message on a pipe monitored by the dispatcher thread's `poll` calls. This mechanism is unchanged from the implementation in the GHC IO manager for the `poll` backend.

GHC does not have an IO manager for Windows. For reading from and writing to files and sockets, blocking FFI calls are issued. Implementing a high-performance IO manager for Windows is challenging because Windows does not have scalable `poll`-like API. Windows I/O Completion Ports (IOCP) provides a scalable

asynchronous API and substantial progress has been made in building an IO manager based on IOCP[3]. However, various complications prevented this work from being integrated into GHC. For this reason, we have not yet implemented Mio for Windows.

4.2 Timers

In the GHC IO manager, timers are monitored in the dispatcher loop. In Mio, the dispatcher loop has been modified to yield, which can lead to the dispatcher thread being delayed by other Haskell threads earlier on the run queue. Hence, the dispatcher loop can no longer be used for dispatching functions waiting on timers. Therefore, Mio moves timer monitoring into a separate timer dispatcher thread. This timer dispatching thread is identical to the GHC IO manager dispatcher thread, except that it no longer monitors file descriptors.

4.3 Support for Event-driven Programs

Although the $GHC.Event$ module in the GHC IO manager is considered "private", a survey of hackage applications uncovered several programs that use this module to implement event-driven programs directly, rather than use Haskell threads and the $threadWaitRead$ and $threadWaitWrite$ functions. The GHC IO manager leaves a subscription registered until the client deregisters the subscription. However, as we described earlier, the typical use (via the $threadWait$ function) is to deregister the subscription immediately after the callback is invoked, and Mio optimizes for this case by using the one-shot mode, automatically deregistering the subscription. To continue to support clients using the private Manager interface directly, we allow the Mio manager to be initialized with a flag indicating that it should *not* deregister subscriptions automatically after receiving an event for the subscription. Therefore, such programs can continue to use the Mio manager directly by simply adding a single flag when initializing a Mio manager.

4.4 GHC RTS Issues

Although GHC's threaded RTS provides a high-performance parallel garbage collector, we observed that garbage collection increasingly becomes an impediment to multicore scaling. One commonly used method for reducing the overhead of GC is to reduce the frequency of collections by increasing the allocation area (aka nursery) used by the program. In particular, in our experience, providing a large allocation area (for example, 32MB) improves multicore performance, confirming the observations of other researchers [24].

Each capability of GHC's threaded RTS has a private nursery that must be *cleared* at the end of a GC. This clearing consists of traversing all blocks in the nursery, resetting the free pointer of the block to the start of the block. In the parallel GC, the clearing of all capabilities' nursery blocks is done sequentially by the capability that started the GC, causing many cache lines held by other capabilities to be moved. Instead, we change the behavior such that each capability clears its own nursery in parallel.

Furthermore, many network programs use *pinned* memory regions, e.g. for $ByteString$ values, in order to pass buffers to C functions, such as `send()` and `recv()`. The threaded RTS allocates such objects from the global allocator, which requires the allocating thread to acquire a global lock. Our insight led to a commit by Simon Marlow to allocate necessary blocks for small pinned objects from the capability-local nursery, which can be done without holding any global lock[4]. This change improves performance for multicore programs that allocate many small $ByteString$s.

Our dispatcher thread makes use of the $yield$ function to place the dispatcher thread on the *end* of the run queue when it finds no ready events after polling once. GHC's RTS had a bug in which $yield$ placed the thread back on the front of the run queue. This bug was uncovered by our use of $yield$ which requires that the thread be placed at the end of the run queue.

All of these issues have been addressed by commits that are part of the GHC master development branch.

5. OS Bottlenecks & Bugs

While implementing Mio, we encountered hardware performance problems and a Linux kernel bug. To eliminate the possibility that Haskell (or rather GHC) was causing these problems, we implemented a C version of our `SimpleServer` program. This program, called `SimpleServerC`[5], is implemented in essentially the same way as `SimpleServer`: the main thread accepts connections and a configurable number of *worker native threads* service connections. The accepted connections are assigned to worker threads in round-robin order. Each worker thread uses a distinct epoll instance to monitor the connections it has been assigned and uses the `epoll` API in the same way that `SimpleServer` with Mio uses `epoll`. On the other hand, it does not use a garbage collector, a thread scheduler and other subsystems of the GHC RTS that may introduce performance problems.

5.1 Load Balancing CPU Interrupts

In order to avoid interrupt handling from becoming a bottleneck, Linux (in its default configuration) evenly distributes interrupts (and hence workload) from network interface cards (NICs) to all of the CPU cores in the system. Unfortunately, this feature interacts poorly with power management features of our server. To save power, modern CPU cores can aggressively enter deep sleep states. Specifically, every CPU core on our multicore server can enter (1) C0 state in which the CPU core keeps busy in running code, (2) C1 state in which the CPU core turns off clocks and stops executing instructions, (3) C3 state in which the CPU core flushes the core's L1 and L2 caches into the last level cache, and (4) C6 state in which the CPU core saves all processor state in a dedicated SRAM and then completely removes voltage from the core to eliminate leakage power. When a CPU core wakes up from C6 state, it restores core state from the SRAM, activates L1 and L2 caches and core clocks. This expensive operation causes undesired application delays [1]. Experimentally, we found that, by distributing interrupts well, the workload of each CPU core is reduced to a level which triggers the CPU cores to enter deep sleep states. Later, when new packets arrive, the CPU cores have to wake up before they start processing packets, an expensive operation which leads to undesired delays and low throughput.

To verify this behavior, we compare `SimpleServerC`'s performance on our server in the default configuration with the default configuration with power-saving disabled, while varying the number of cores from 1 to 20. We disable power-saving by specifying the maximum transition latency for the CPU, which forces the CPU cores to stay in C0 state. Figure 12 shows the results, with the curves labelled "Default" and "NoSleep" showing the performance in the default configuration and the default configuration with power-saving disabled, respectively. Without limiting the CPU sleep states (curve "Default"), `SimpleServerC` cannot benefit from using more CPU cores and the throughput is less than 218,000 requests per second. In contrast, after preventing CPU cores entering deep sleep states (curve "NoSleep"), `SimpleServerC` scales up to 20 cores and can process 1.2 million requests per second, approximately 6 times faster than with the default configuration.

[3] `http://ghc.haskell.org/trac/ghc/ticket/7353`

[4] `thread.gmane.org/gmane.comp.lang.haskell.parallel/218`

[5] `github.com/AndreasVoellmy/epollbug`

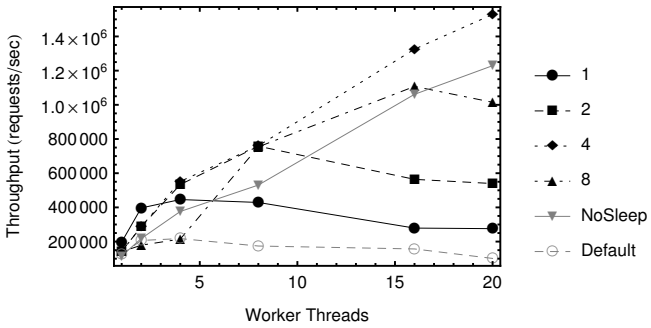

Figure 12. Throughput scaling of `SimpleServerC` when using different number of cores for interrupt handling and different power-saving settings.

By profiling our server with `perf` in Linux and `i7z`[6] from Intel, we observed that the CPU cores of the server in the default configuration frequently entered deep sleep states. For example, when running `SimpleServerC` with 16 worker threads in the default configuration, on average 51% of the CPU time is spent on C6 state (including transition between C6 and running state), 33% on C3 state, 9% on C1 state, and only 7% of the CPU time is spent on running programs. In contrast, with power-saving disabled, the CPUs remain in C0 state throughout the program run.

Disabling sleep modes, however, is not a desirable solution for real systems due to power consumption. Rather than distribute interrupts of 10G NICs among all of the CPU cores in the system and then disable power saving, we limit the number of CPU cores used to handle hardware interrupts to a reasonably small number of CPU cores (In Linux, this can be achieved by adjusting `/proc/irq/N/smp_affinity` settings appropriately). In addition, we set the CPU affinity (using `taskset`) of `SimpleServerC` threads so that they run on the same CPU (NUMA) nodes as the interrupt handlers , which improves stability of the results.

Curves labelled "1", "2", "4", and "8" in Figure 12 show the performance of `SimpleServerC` when interrupts are handled by 1, 2, 4, and 8 cores respectively with default power-saving settings. By limiting the number of CPU cores to handle interrupts, `SimpleServerC` can scale as well or better as when power-saving is disabled. In particular, we see that using 4 cores for interrupt handling is optimal. With this configuration, `SimpleServerC` scales up to 20 workers and can process 1.5 million requests per second, with 100% of the CPU time of CPU cores handling interrupts spent on C0 state. With too few CPU cores (curves 1 and 2), the server cannot scale to more than 8 worker threads because the interrupt handling CPUs become a bottleneck. With too many CPU cores (e.g. curve 8) deep sleep is triggered frequently because the workload of each CPU core is reduced below a critical level.

We therefore prevent our server from entering deep sleep states in all evaluations in this paper by using 4 CPU cores to handle hardware interrupts from our 10 Gbps NICs.

5.2 Linux Concurrency Bug

After removing various bottlenecks in our system, `SimpleServer` scaled to 20 cores and serves nearly 700,000 requests per second. This workload places an unusual burden on the Linux kernel and triggers a bug in Linux; Under such a heavy load, the `epoll` subsystem occasionally does not return read events for a socket, even though the socket has data ready, causing worker threads in `SimpleServer` to wait for data indefinitely. To verify that this problem is due to a kernel bug, we verified that `SimpleServerC` also triggers the same problem (in fact, debugging this problem was the initial motivation for developing `SimpleServerC`).

We reported the bug to the Linux community. Kernel developers quickly identified the problem as a missing memory barrier in the `epoll` subsystem. In particular, when subscribing for an event on a socket, the interest mask for the socket is first updated and *then* the socket is checked for data to see if it is already ready. On the other hand, when processing incoming data, the Linux network stack first writes data to the socket and *then* checks the interest mask to determine whether to notify a waiter. To ensure that these operations occur in the correct order even when executing on different CPUs, memory barriers are required in both sequences. Unfortunately, a memory fence was missing in the registration code. This long-standing bug affects all Linux kernels since 2.4 and a patch fixing the issue has been accepted into the Linux kernel[7].

6. Evaluations

6.1 Methodology

We use a set of benchmark applications and workloads to characterize the performance of Mio. We use the following hardware, system software, and GHC and Haskell library versions.

Hardware: We run Haskell server programs on a SuperMicro X8OBN server, with 128 GB DDR3 memory and 8 Intel Xeon E7-8850 2 GHz CPUs, each having 10 cores with a 24 MB smart cache and 32 MB L3 cache. This server has four 10 Gbps Intel NICs. In the experiments, we turn off hyper-threading to prevent the system scheduler from scheduling multiple native threads that should run in parallel to a single physical CPU core. Client and workload generator programs run on Dell PowerEdge R210 II servers, with 16 GB DDR3 memory and 8 Intel Xeon E3-1270 CPUs (with hyper-threading) running at 3.40 GHz. Each CPU core has 256 KB L2 cache and 8 MB shared L3 cache. The servers communicate over a 10 Gbps Ethernet network and are configured so that each client server has a dedicated 10 Gbps path the main server. This level of network bandwidth was required to avoid network bottlenecks.

Software: The server software includes Ubuntu 12.04, Linux kernel version 3.7.1 on the SuperMicro server and version 3.2.0 on the Dell server, and Intel ixgbe driver (version 3.15.1). We use the latest development version of GHC at the time of this writing. For comparison of the threaded RTS of GHC without Mio, we use the development version of GHC with Mio patches removed.

We run `weighttp`[8] on the Dell servers to benchmark HTTP servers. `weighttp` simulates a number of clients making requests of an HTTP server, with each client making the same number of requests. Each client establishes a connection to the server and then repeatedly requests a web page and then waits for the response. In addition to recording the throughput for a given run, we extend `weighttp` to uniformly and randomly sample the latency for a fraction of the requests.

6.2 Web Server Performance

We evaluate two web servers written in Haskell, `acme`[9] and `mighty` [23], with the GHC IO manager and with Mio manager. `acme` is a minimal web server which does basic request and HTTP header parsing and generates a fixed response without performing any disk I/O, whereas `mighty` is a realistic, full-featured HTTP server, with realistic features such as slowloris protection. For comparison, we also measure the performance of `nginx`, arguably the

[6] `http://code.google.com/p/i7z/`

[7] `https://patchwork.kernel.org/patch/1970231/`

[8] `http://redmine.lighttpd.net/projects/weighttp/wiki`

[9] `http://hackage.haskell.org/package/acme-http`

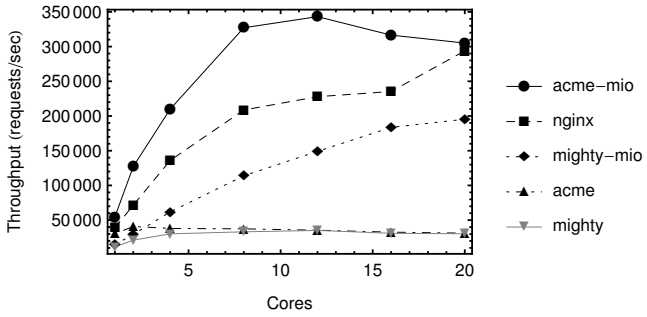

Figure 13. Throughput of Haskell web servers `acme` and `mighty` with GHC IO manager and Mio manager and `nginx` in HTTP requests per second as a function of number of capabilities used.

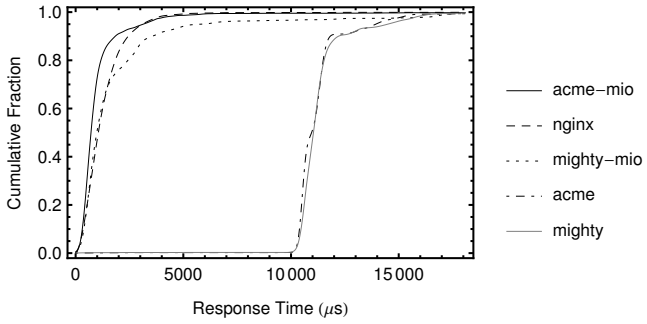

Figure 14. Cumulative Distribution Function (CDF) of response time of `acme` server with GHC IO manager and with Mio manager at 12 cores and 400 concurrent clients.

world's fastest web server, written in C specifically for high performance. We evaluate all three servers with 400 concurrent clients, a total of 500,000 requests and sample the latency of 1% of the requests.

Figure 13 shows the result. We see that when using the GHC IO manager both `acme` and `mighty` scale only modestly to 4 cores and then do not increase performance beyond 30,000 requests per second. On the other hand, with Mio, `acme` scales well to 12 cores serving up to 340,000 requests per second and `mighty` scales up to 20 cores serving 195,000 requests per second at peak performance, resulting in a 8.4x and 6.5x increases (respectively) in peak server throughput using Mio. The graph also demonstrates that a realistic web server in Haskell, `mighty`, performs within a factor of 2.5x of `nginx` for every number of cores and performs within 2x of `nginx` for 8 cores and higher.

Figure 14 shows the cumulative distribution function (CDF) of the response time for the `acme` and `mighty` servers when run with 12 capabilities with and without Mio and for the `nginx` server when run with 12 worker processes. The expected response time for `acme` and `mighty` is 1.1 ms and 2.0 ms, respectively, using the Mio manager and are both 11.3 ms with the GHC IO manager. Hence, Mio improves the expected response time for `acme` and `mighty` by 10.3x and 5.7x. The 95th percentile response time of `acme` and `mighty` with Mio are 3.1 ms and 5.9 ms, whereas with the GHC IO manager they are 13.9 ms and 14.5 ms, representing a 4.4x and 2.5x reduction in 95th percentile response time. We also observe that the response time distribution of the Haskell servers closely matches that of `nginx`.

Figure 15 shows the throughput of `acme` and `mighty` on a FreeBSD server. For this experiment, we use different hardware

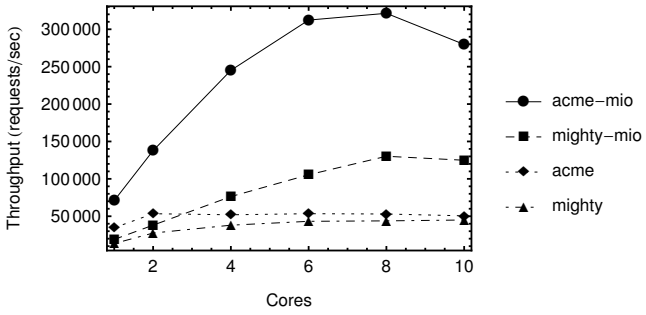

Figure 15. Throughput of Haskell web servers `acme` and `mighty` with GHC IO manager and Mio manager on FreeBSD.

than other benchmarks. In particular, two 12 core (Intel Xeon E5645, two sockets, 6 cores per 1 CPU, hyper-threading disabled) servers are connected with 1 Gpbs Ethernet. One server runs Linux version 3.2.0 (Ubuntu 12.04 LTS) while the other runs FreeBSD 9.1. The web servers are running on the FreeBSD server and they are measured from the Linux server using the same benchmarking software as our earlier evaluations. `acme` and `mighty` scale up to 7 and 8 cores, respectively, with Mio manager, but scale poorly with the GHC IO manager. `acme` achieves a peak throughput of 330,000 requests per second, which saturates the 1 Gbps network connection between these two servers and inhibits further scaling. Using this same hardware, we evaluated `acme` and `mighty` on the Linux server with clients on the FreeBSD server. The resulting throughputs were nearly identical to those in Figure 15 demonstrating that performance with Mio on FreeBSD is comparable to that on Linux up to 8 cores.

6.3 SDN Controllers

A major recent development in computer networking is the notion of a *Software-Defined Network* (SDN), in which network behavior is programmed through centralized policies at a conceptually centralized network controller. In particular, the Openflow protocol [13] has established (1) flow tables as a standard abstraction for network switches, (2) a protocol for the centralized controller to install flow rules and query states at switches, and (3) a protocol for a switch to forward to the controller packets not matching any rules in its switch-local flow table. Openflow provides a critical component for realizing the vision that an operator configures a network by writing a simple, centralized network control program with a global view of network state, decoupling network control from the complexities of managing distributed state.

The simplicity of a logically centralized controller, however, can come at the cost of control-plane scalability. As the network scales up — both in the number of switches and the number of end hosts — the SDN controller can become a bottleneck. Therefore, several extensible Openflow controllers have been developed that use multicore servers to scale event processing for Openflow controllers.

In particular, we compare the throughput and latency of three Openflow controllers: McNettle[20], written in Haskell, Beacon[10] written in Java, and NOX-MT [19] written in C++ with Boost libraries for concurrent event processing. We compare their performance on a standard network control algorithm, called the "learning switch controller" and generate load from several servers which simulate flow requests from 100 switches using a modified version of the `cbench`[11] tool. Each simulated switch generates Openflow events simulating 1000 attached hosts and is limited so that at any

[10] `https://openflow.stanford.edu/display/Beacon/Home`

[11] `http://www.openflow.org/wk/index.php/Oflops`

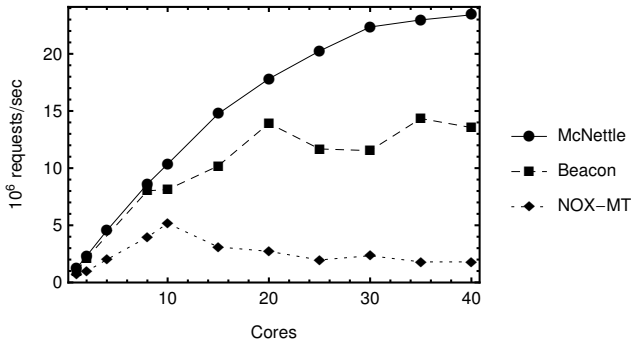

Figure 16. Throughput of SDN controllers written in Haskell, Java, and C++.

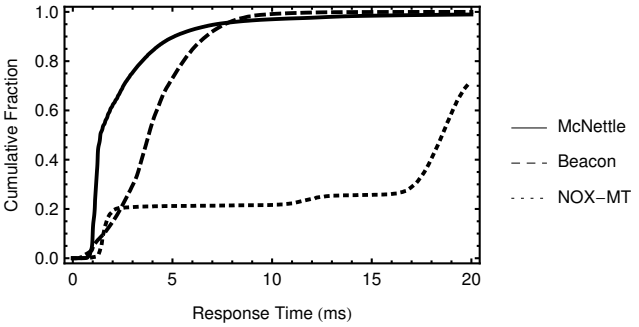

Figure 17. Latency comparison of SDN controllers

time it can have no more than 512 messages for which it has not yet received a response from the controller.

Figure 16 shows the throughput as a function of the number of cores used for all three systems. We observe that McNettle serves over 20 million requests per second using 40 cores and scales substantially better than Beacon or NOX-MT. In particular, Beacon scales to less than 15 million requests per second, and NOX-MT achieves only 5 million requests per second. Figure 17 shows the latency CDF for all three systems run with 30 cores. The median latency of McNettle is 1 ms, Beacon is almost 4 ms, and NOX-MT reaches as high as 17 ms. The 95-percentile latency of McNettle is still under 10 ms.

6.4 Threaded RTS Overhead

CloudHaskell [5] developers have noted that the threaded RTS of GHC often introduces significant latency to their CloudHaskell programs[12] for light loads which do not overwhelm the single-threaded execution. Mio substantially reduces the latency overhead of the threaded RTS for these loads.

We evaluate the threaded RTS overhead with a benchmark program based on a similar program developed by the CloudHaskell community[13], which consists of a server which simply echoes a message sent by a client. We run this program using a single client and measure the round-trip time for each echo request. Figure 18 shows the CDF of the response times for the non-threaded RTS, the threaded RTS using the GHC IO manager, and the threaded RTS using the Mio manager, with both threaded RTSs using a single capability. The expected latencies are 63 microseconds (μs),

[12] `http://www.edsko.net/2013/02/06/performance-problems-with-threaded/`

[13] `github.com/haskell-distributed/network-transport-tcp`

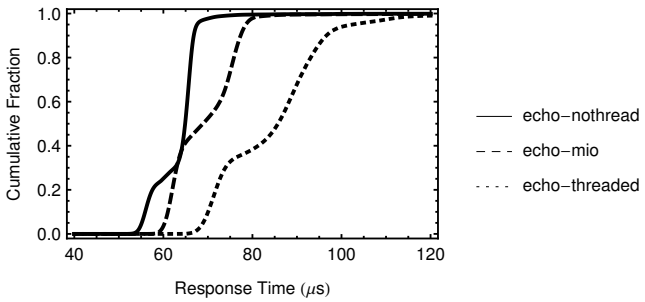

Figure 18. Cumulative Distribution Function (CDF) of response time of echo server with non-threaded RTS, threaded RTS with GHC IO manager and threaded RTS with Mio manager.

69 μs and 84 μs for the non-threaded RTS, threaded RTS with Mio and threaded RTS with GHC IO manager, respectively, while the 99th percentile latencies are 73 μs, 83 μs and 119 μs, respectively. The Mio manager therefore significantly reduces latency over the GHC IO manager and adds just 6 μs latency on average over the non-threaded RTS for this workload.

7. Related Work

7.1 Lightweight Components

Several prominent programming languages provide lightweight threads or processes. Erlang [3] provides lightweight processes that communicate only using message passing. The Go programming language [4] provides sophisticated coroutines, called goroutines. The Go scheduler also uses N (one per cpu) poll servers. In upcoming releases, Go's IO event notification will be integrated with the scheduler, whereas with Mio, the event notification is implemented entirely in Haskell.

7.2 Event Libraries

Event libraries such as `libevent`[14], `libev`[15], and `libuv`[16] also provide abstractions over platform-specific event notification APIs. These APIs require programmers to express programs as event handlers, while Mio is designed and optimized to support a lightweight thread programming model. Furthermore, these libraries do not support concurrent registration of events and notification; the program is *either* polling the OS for events *or* executing a user's event handlers, where new registrations may occur. In contrast, with Mio, a native thread may register event callbacks with a dispatcher while that dispatcher is polling the OS for notifications.

7.3 Prefork Technique

A typical event-driven program runs one event loop in a single process resulting in good utilization of one core but not scaling on multiple cores. For web servers, the *prefork* technique is used to utilize multiple cores. Historically, the prefork technique worked as follows. After setting up a listen socket and before starting the service, the main process forks multiple processes, called *workers*, so that the workers share the same listen socket. When a new connection arrives, one of the workers wins the race to accept the new connection and then serves the connection until completion.

Recently, systems such as `nginx` [2] and `Node.js`[17] have adapted the prefork technique by creating N worker processes

[14] `http://libevent.org/`

[15] `http://software.schmorp.de/pkg/libev.html`

[16] `https://github.com/joyent/libuv`

[17] `http://nodejs.org/`

(where N is the number of cores) with each process running an event-driven loop. The prefork technique can also be applied to Haskell servers by simply forking several identical processes. For instance, `mighty` can use the prefork technique to achieve good multicore performance [23] when using the GHC IO manager.

However, there are two disadvantages when adopting the prefork technique for event-driven programming: 1) *boilerplate*: code for interprocess communication is necessary to manage workers such as stopping services and reloading configuration files. 2) *thundering herd*: all workers are unnecessarily rescheduled when a new connection arrives [18]. Fortunately, the thundering herd problem for original perfork-style servers has been addressed in recent OSes (such as Linux 2.4 or later) by waking up only one process [14] that is blocking on an `accept()` call.

Unfortunately, a thundering herd problem can still occur with the newer style of prefork servers, even on new OSes. In this case, multiple processes watch the same listen socket using `epoll` or `kqueue` (as opposed to blocking on `accept()` calls). These event systems notify all listeners when an event occurs. High performance servers should avoid unnecessary overhead to have resistance to Slashdot effect (or flash crowd). With Mio, servers may avoid thundering herd by using a single Haskell thread for accepting new connections, as both `acme` and `mighty` do.

8. Conclusions and Future Work

We presented and evaluated a new *multicore IO manager* that improves realistic web server throughput by over 6.5x and expected response time by 5.7x. As a result, the performance of network servers in Haskell rivals that of servers written in C.

There are multiple directions for future work. In particular, timer management has been reported as a bottleneck when handling massive number of connections, each with a timeout. For instance, Warp [17], a high performance web server, avoids using $System.Timeout.timeout$ for each connection and alternatively creates one designated Haskell thread to keep track of all timeouts. Improving the timer manager's performance may make such special techniques unnecessary.

Acknowledgments

We would like to thank Bryan O'Sullivan and Johan Tibell for providing critical guidance and code reviews. We also thank Simon Marlow for his technical advice. We are grateful to Edsko de Vries for giving his benchmark tools which our tools are based on. We would like to thank the anonymous referees for their valuable feedback. Andreas Voellmy was primarily supported by a gift from Futurewei, and partially supported by NSF grant CNS-1017206. Paul Hudak was supported in part by a gift from Microsoft Research.

References

[1] Intel xeon processor e7- 8800/4800/2800 product families, volumn 2. 2011. URL `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e7-8800-4800-2800-families-vol-2-datasheet.pdf`.

[2] A. Alexeev. nginx. In *The Architecture of Open Source Applications, Datasheet Volume 2*. 2012. URL `http://www.aosabook.org/en/nginx.html`.

[3] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. 2007.

[4] I. Balbaert. *The Way to Go: A Thorough Introduction to the Go Programming Language*. 2012.

[5] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, 2011.

[6] D. Jones, Jr., S. Marlow, and S. Singh. Parallel performance tuning for haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 81–92, 2009.

[7] J. Lemon. Kqueue - A Generic and Scalable Event Notification Facility. In *the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153, 2001.

[8] D. Libenzi. *Improving (network) I/O performance*, 2001. URL `http://www.xmailserver.org/linux-patches/nio-improve.html`.

[9] S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume 2*. 2012. URL `http://www.aosabook.org/en/ghc.html`.

[10] S. Marlow, S. Peyton Jones, and W. Thaller. Extending the Haskell Foreign Function Interface with Concurrency. In *Proceedings of the Haskell Workshop*, 2004.

[11] S. Marlow, S. Peyton Jones, and S. Singh. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 65–78, 2009.

[12] S. Marlow et al. *Haskell 2010 Language Report*, 2010.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[14] S. P. Molloy and C. Lever. Accept() scalability on Linux. In *ATEC '00 Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000.

[15] B. O'Sullivan and J. Tibell. Scalable I/O Event Handling for GHC. In *Proceedings of the 2010 ACM SIGPLAN Haskell Symposium (Haskell'10)*, pages 103–108, 2010.

[16] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, 1996.

[17] M. Snoyman. *Warp: A Haskell Web Server*, 2011. URL `http://steve.vinoski.net/pdf/IC-Warp_a_Haskell_Web_Server.pdf`.

[18] W. Stevens, B. Fenner, and A. M. Rudoff. *UNIX Network Programming*. Addison-Wesley Professional, 2004.

[19] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *Hot-ICE*, 2012.

[20] A. Voellmy and J. Wang. Scalable software defined network controllers. *SIGCOMM Comput. Commun. Rev.*, 42(4):289–290, Aug. 2012.

[21] R. von Behren, J. Condit, and E. Brewer. Why Events Are A Bad Idea (for High-Concurrency Servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, 2003.

[22] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.

[23] K. Yamamoto. Mighttpd - a High Performance Web Server in Haskell. In *The Monad.Reader Issue 19*. 2011.

[24] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 361–376, 2009.