Scalable Event Handling for GHC

Bryan O'Sullivan
Serpentine
bos@serpentine.com

Johan Tibell
Google
johan.tibell@gmail.com

Abstract

We have developed a portable new event manager for the Glasgow Haskell Compiler (GHC) that scales to the needs of modern server applications. Our new code is transparently available to existing Haskell applications. Performance at lower concurrency levels is comparable with the existing implementation. In tests, we can support hundreds of thousands of concurrent network connections, with hundreds of thousands of active timeouts, from a single multithreaded program, levels far beyond those achievable with the current I/O manager. In addition, we provide a public API to developers who need to create event-driven network applications. We are currently integrating our work into GHC itself.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Runtime-environments

General Terms Languages, Concurrency, Performance, Networking

1. Introduction

The concurrent computing model used by most Haskell programs has been largely stable for almost 15 years[9]. Despite the language's many innovations in other areas, networked software is written in Haskell using a programming model that will be familiar to most programmers: a thread of control synchronously sends and receives data over a network connection. By *synchronous*, we mean that when a thread attempts to send data over a network connection, its continued execution will be blocked if the data cannot immediately be either sent or buffered by the underlying operating system.

The Glasgow Haskell Compiler (GHC) provides an environment with a number of attractive features for the development of networked applications. It provides composable synchronization primitives that are easy to use[3]; lightweight threads; and multicore support[2]. However, the increasing demands of large-scale networked software have outstripped the capabilities of crucial components of GHC's runtime system.

Haskell Symposium 2010 September 30 2010, Baltimore.

We have rewritten GHC's event and timeout handling subsystems to be dramatically more efficient. With our changes, a modestly configured server can easily cope with networking workloads that are several orders of magnitude more demanding than before. Our new code is designed to accommodate both the thread-based programming model of Concurrent Haskell (with no changes to existing application code) and the needs of event-driven applications.

2. Background: the GHC concurrent runtime

GHC provides a multicore runtime system that uses a small number of operating system (OS) threads to manage the execution of a potentially much larger number of lightweight Haskell threads. The number of operating system threads to use may be chosen at program startup time, with typical values ranging up to the number of CPU cores available. (GHC also supports an alternative "unthreaded" runtime, which does not support multiple CPU cores. In this paper, we are concerned only with the threaded runtime.)

Although from the programmer's view, programming in Concurrent Haskell is appealing due to the simplicity of the synchronous model, the fact that Haskell threads are lightweight, and do not have a one-to-one mapping to OS threads, complicates the implementation of the runtime system. When a Haskell thread must block, this cannot lead to an OS-level thread also being blocked, so the runtime system uses a single OS-level I/O manager thread (which *is* allowed to block) to provide an event notification mechanism.

The standard Haskell file and network I/O libraries are written to cooperate with the I/O manager thread. When one of these libraries acquires a resource such as a file or a network socket, it immediately tells the OS to access the resource in a non-blocking fashion. When a client attempts to access (e.g. read or write, send or receive) such a resource, the library performs the following actions:

1. Attempt to perform the operation. If it succeeds, resume immediately.
2. If the operation would need to block, the OS will instead cause it to fail and indicate (via `EAGAIN` or `EWOULDBLOCK` in Unix parlance) that it must be retried later.
3. The thread registers with the I/O manager to be awoken when the operation can be completed without blocking. The sleeping and waking are performed using the lightweight `MVar` synchronization mechanism of Concurrent Haskell.
4. Once the I/O manager wakes the thread, return to step 1. (The operation may fail repeatedly with a would-block error, e.g. due to a lost race against another thread for resources, or an OS buffer filling up.)

As this sketch indicates, GHC provides a synchronous programming model using a lower-level event-oriented mechanism. It does so via a semi-public API that client (e.g. the file and networking libraries) can use to provide blocking semantics.

```
-- Block the current thread until data is available
-- on the given file descriptor.
threadWaitRead, threadWaitWrite :: Fd → IO ()
```

2.1 Timeout management

Well designed network applications make careful use of timeouts to provide robustness in the face of a number of challenges. At internet scale, broken and malicious clients are widespread. As an example, a defensively written application will, if a newly connected client doesn't send any data within a typically brief time window, unilaterally close the connection and clean up its resources.

To support this style of programming, the `System.Timeout` module provides a `timeout` function:

```
timeout :: Int → IO a → IO (Maybe a)
```

It initiates an `IO` action, and if the action completes within the specified time limit, returns `Just` its result, otherwise it aborts the action and returns `Nothing`.

Concurrent Haskell also provides a `threadDelay` function that blocks the execution of a thread for a specified amount of time.

Behind the scenes, the I/O manager thread maintains a queue of pending timeouts. When a timeout fires, it wakes the appropriate application thread.

3. Shortcomings of the traditional I/O manager

Although the I/O manager in versions of GHC up to 6.12 is stable and performs well for small applications, it has some severe shortcomings that make it inapplicable at scale.

The I/O manager uses the venerable `select` system call for two purposes. It informs the OS of the resources it wishes to track for events, and the time until the next pending timeout should be triggered, and blocks until either an event occurs or the timeout fires.

The `select` system call has well-known problems. Most obvious is the distressingly small fixed limit on the number of resources it can handle even under modern operating systems, e.g. 1,024 on Linux. In addition, the programming style enforced by `select` can be inefficient. The sizes of its programmer-visible data structures are linear in the number of resources to watch. They must be filled out, copied twice across the user/kernel address space boundary, and checked afresh for *every* system call performed. Since the most common case for server-side applications on the public Internet is for connections from clients to mostly be idle, the amount of useful work performed per invocation of `select` dwindles as the number of open connections increases. This repetitious book-keeping rapidly becomes a noticeable source of overhead.

The I/O manager incurs further inefficiency by using ordinary Haskell lists to manage both events and timeouts. It has to walk the list of timeouts once per iteration of its main loop, to figure out whether any threads must be woken and when the next timeout expires. It must walk the the list of events twice per iteration: once to fill out the data structures to pass to `select`, and again after `select` has returned to see which threads to wake.

Since `select` imposes such a small limit on the number of resources it can manage, we cannot easily illustrate the cost of using lists to manage events, but in section 9.2, we will demonstrate the clear importance of using a more efficient data structure for managing timeouts.

4. Related work

Li and Zdancewic[8] began the push for higher concurrency in Haskell server applications with an application-level library that provides both event- and thread-based interfaces. We followed their lead in supporting both event-based and thread-based concurrency, but unlike their work, our threading implementation is source-level compatible with existing Haskell applications. We can thus improve the scalability of existing code with a recompile.

In the context of the Java Virtual Machine, Haller and Odersky unify event- and thread-based concurrency via a Scala implementation of the actor concurrency model[1]. Actor-style programming is already supported in Concurrent Haskell via the `Chan` mechanism. Much of their work is concerned with safely implementing lightweight threads via continuations on top of Java's platform-level threads, resulting in an environment similar to the two-level threading of GHC's runtime, with comparable concurrency management facilities.

For several years, C programmers concerned with client concurrency have enjoyed the `libev` and `libevent` libraries. These enable an event- and callback-driven style of development that can achieve high levels of both performance and concurrency.

5. Our approach

When we set out to improve the performance of GHC's I/O manager, our primary goal was to increase the number of files, network connections, and timeouts GHC could manage by several orders of magnitude. We wanted to achieve this in the framework of the existing Concurrent Haskell model, retaining complete source-level compatibility with existing Haskell code, and in a manner that could be integrated into the main GHC distribution with minimal effort.

Secondarily, we wanted to sidestep the long dispute over whether events or threads make a better programming model for high-concurrency servers[10]. Since we needed to implement an event-driven I/O manager in order to provide synchronous semantics to application programmers, we might as well design the event API cleanly and expose it publicly to those programmers who believe they need events[1].

We desired to implement as much as possible of the new I/O manager in Haskell, rather than delegating to a lower-level language. This desire was partly borne out of pragmatism: we initially thought that it might have been more efficient to base our work atop a portable event handling library such as `libev` or `libevent2`. These libraries are intended to be used in a callback-driven style, such that the library executes application code to handle an event. Although Haskell's foreign function interface supports calls from C code back into Haskell, experimentation convinced us that the overhead involved was higher than we were happy with. With performance and aesthetic considerations pushing us in the same direction, we were happy to forge ahead in Haskell.

Architecturally, our new I/O manager consists of two components. Our event notification library provides a clean and portable API, and abstracts the system-level mechanisms used to provide efficient event notifications (`kqueue`, `epoll`, and `poll`). We have also written a shim that implements the semi-public `threadWaitRead` and `threadWaitWrite` interfaces. This means that neither the core file or networking libraries, nor other low-level I/O libraries, require any changes to work with—and transparently benefit from the performance improvements of—our new code.

6. Interface to the event manager

Our event manager is divided into a portable front end and a platform-specific back end. The interface to the back end is simple,

[1] In our experience, even in a language with first-class closures and continuations, writing applications of anything beyond modest size in an event-driven style is painful enough to reward only the most discerning and persistent of masochists.

and is only visible to the front end; it is abstract in the public interface.

```
data Backend = forall a. Backend {
    -- State specific to this platform.
    _beState :: !a,

    -- Poll the back end for new events.  The callback
    -- provided is invoked once per file descriptor with
    -- new events.
    _bePoll :: a
            -> Timeout                  -- in milliseconds
            -> (Fd -> Event -> IO ()) -- I/O callback
            -> IO (),

    -- Register, modify, or unregister interest in the
    -- given events on the specified file descriptor.
    _beModifyFd :: a
                -> Fd       -- file descriptor
                -> Event    -- old events to watch for
                -> Event    -- new events to watch for
                -> IO (),

    -- Clean up platform-specific state upon destruction.
    _beDestroy :: a -> IO ()
    }
```

A particular back end will provide a `new` action that fills out a `Backend` structure.

```
module System.Event.KQueue (new) where
new :: IO Backend
```

On Unix-influenced platforms, more than one back end will typically be available. For instance, on Linux, `epoll` is the most efficient back end, but `select` and `poll` are available. On Mac OS X, `kqueue` is usually the back end of choice, but again `select` and `poll` are also available.

Our public API thus provides a default back end, but allows a specific back end to be used (e.g. for testing).

```
-- Construct the fastest back end for this platform.
newDefaultBackend :: IO Backend

newWith :: Backend -> IO EventManager

new :: IO EventManager
new = newWith =<< newDefaultBackend
```

For low-level event-driven applications, a typical event loop involves running a single step through the event manager to check for new events, handling them, doing some other work, and repeating. Our interface to the event manager supports this approach.

```
init :: EventManager -> IO ()
-- Returns an indication of whether the event manager should
-- continue, and a modified timeout queue.
step :: EventManager -> TimeoutQueue
     -> IO (Bool, TimeoutQueue)
```

To register for notification of events on a file descriptor, clients use the `registerFd` function.

```
-- Cookie describing a registration.
data FdKey

-- A set of events to wait for.
newtype Event
instance Monoid Event
evtRead, evtWrite :: Event

type IOCallback = FdKey -> Event -> IO ()

registerFd :: EventManager -> IOCallback -> Fd -> Event
           -> IO (FdKey, Bool)
```

Because the event manager has to accommodate being invoked from other threads as well as from the same thread in which it is running, `registerFd` sends a wakeup message to the event manager when invoked. We provide a variant, `registerFd_`, which does not wake the event manager. Clients needing a little more efficiency can queue a series of requests to the manager without waking it, then wake it once afterwards via a `wakeManager` call.

A client remains registered for notifications until it explicitly drops its registration, and is thus called back on every step into the event manager as long as an event remains pending. We have found this level-triggered approach to event notification to be much easier for client applications to use than an edge-triggered approach, which forces more complicated book-keeping into application code.

```
unregisterFd :: EventManager -> FdKey -> IO ()
```

7. Implementation

By and large, the story of our efforts revolves around appropriate choices of data structure, with a few extra dashes of context-sensitive and profile-driven optimization thrown in.

7.1 Economical event management

GHC's original I/O manager has to walk the entire list of blocked clients once per loop before calling `select`, and mutate the list afterwards to wake and filter out any clients that have pending events. A step through the I/O manager's loop thus involves $O(n)$ of traversal and mutation, where n is the number of clients.

Our new event manager registers file descriptors persistently with the operating system, using `epoll` on Linux and `kqueue` on Mac OS X, so the event manager no longer needs to walk through all clients on each step through the list. Instead, we maintain a finite map from file descriptor to client, which we can look up for each triggered event. This map is based on Leijen's implementation of Okasaki and Gill's purely functional Patricia tree[6]. The new event manager's loop thus involves $O(m \log n)$ traversal, and negligible mutation, where m is the number of clients with events pending. This works well in the typical case where $m \ll n$.

7.2 Cheap timeouts

In the original I/O manager, GHC maintains pending timeouts in an ordered list, which it partly walks and mutates on every iteration. Inserting a new timeout thus has $O(n)$ cost per operation, as does each step through the I/O manager's loop.

The event manager needs to perform two operations efficiently during every step: remove all timeouts that have expired, and find the next timeout to wait for. A priority queue would suffice to fill this need with good performance, but we also need to allow clients to add and remove timeouts. Since we need both efficient update by key and efficient access to the minimum value, we use a priority search queue instead.

Our purely functional priority search queue implementation is based on that of Hinze[4], so insertion and deletion have $O(\log n)$ cost, while a step through the list has $O(m \log n)$ cost, where m is the number of expired timeouts. Since typically $m \ll n$, we win on performance.

8. War stories, lessons learned, and scars earned

Writing fast networking code is tricky business. We have variously encountered:

- Tunable kernel variables (15 at the last count) that regulate obscure aspects of the networking stack in ways that start to matter at scale;

- Abstruse kernel infelicities (e.g. Mac OS X lacking the `NOTE_EOF` argument to `kqueue`, even though it has been present in other BSD variants since 2003);
- Performance bottlenecks in GHC that required expert diagnosis;
- An inability to stress the software enough, due to lack of 10-gigabit Ethernet hardware (gigabit Ethernet is a breeze to saturate with even the most puny of modern laptops)

In spite of these difficulties, we are satisfied with the level of performance we have achieved to date. To give a more nuanced flavour of the sorts of problems we encountered, we have chosen to share a few in more detail.

8.1 Efficiently waking the event manager

In a concurrent application with many threads, the event manager thread spends much of its time blocked, waiting for the operating system to notify it of pending events. A thread that needs to block until it can perform I/O has no way to tell how long the event manager thread may sleep for, so it must wake the event manager in order to ensure that its I/O request can be queued promptly.

The original implementation of event manager wakeup in GHC uses a Unix pipe, which clients use to transmit one of several kinds of single-byte control message to the event manager thread. The delivery of a control message has the side effect of waking the event manager if it is blocked. Because several kinds of control message may be sent, the original event manager reads and inspects a single byte from the pipe at a time. If several clients attempt to wake the event manager thread before it can service any of their requests, it acts as if it has been woken several times in succession, thus performing unneeded work.

More damagingly, this design is vulnerable to the control pipe filling up, since a Unix pipe has a fixed-size buffer. If control messages are lost due to a pipe overflow, an application may deadlock. Indeed, we wrote a microbenchmark that inadvertantly provided a demonstration of how easy it is to provoke this condition under heavy load.

As a result, we invested some effort in ameliorating the problem. We use substantially the same mechanism for waking our event manager thread, but we have considerably improved both its performance and its reliability. Our principal observation was that under both the original and our new design, the most control common message by far is a simple "wake up." We have accordingly special-cased the handling of this message.

On Linux, when possible, we use the kernel's `eventfd` facility to provide very fast wakeups. No matter how many clients send wakeup requests in between checks by the event manager, it will receive only one notification.

While other operating systems do not provide a comparably fast facility, we still have a trick up our sleeves. We dedicate a pipe to delivering *only* wakeup messages. To issue a wakeup request, a client issues a non-blocking write of a single byte to this pipe. When the event manager is notified that data is available on this pipe, it issues a single `read` system call to gather all currently buffere buffered wakeups. It does not need to inspect any of the data it has read, since they must all be wakeups, and the fixed size of the pipe buffer guarantees that it will not be subject to unnecessary wakeups, regardless of the number of clients requesting. This means that we no longer need to worry about wakeup messages that cannot be written for want of buffer space.

8.2 Bunfight at the GC corral

When a client application registers a new timeout, we must update the data structure that we use to manage timeouts. Originally, we stored the priority search queue inside an `IORef`, and each client manipulated the queue using `atomicModifyIORef`. Alas, this led to a bad interaction with GHC's generational garbage collector.

Since our client-side use of `atomicModifyIORef` did not force the evaluation of the data inside the `IORef`, the `IORef` would accumulate a chain of thunks. If the I/O manager thread did not evaluate those thunks promptly enough, they would be promoted to the old generation and become roots for all subsequent minor garbage collections (GCs).

When the thunks eventually got evaluated, they would each create a new intermediate queue that immediately became garbage. Since the thunks served as roots until the next major GC, these intermediate queues would get copied unnecesarily in the next minor GC, increasing GC time. We had created a classic instance of the generational "floating garbage" problem.

The effect on performance of the floating garbage problem was dramatic. We wrote a microbenchmark to gauge the performance of `threadDelay`, and it would unpredictably run hundreds or thousands of times slower, with the probability of catastrophic performance loss rising as we increased the problem. For example, one small benchmark that usually ran in 0.2 seconds would spike to 20 seconds.

We addressed this issue by having clients store a list of *edits* to the queue, instead of manipulating it directly.

```
type TimeoutEdit = TimeoutQueue → TimeoutQueue

applyTimeoutEdits = foldl' (flip ($))
```

While maintaining a list of edits doesn't eliminate the creation of floating garbage, it reduces the amount of copying at each minor GC enough that calamitous slowdowns no longer occur.

8.3 The great black hole pileup

Our use of an `IORef` to manage the timeout queue yielded another head-scratcher, again with a huge and unpredictable performance impact.

In our `threadDelay` benchmark, thousands of threads compete to update the single timeout management `IORef` using `atomicModifyIORef`. If one of these threads was pre-empted while evaluating the thunk left in the `IORef` by `atomicModifyIORef`, then the thunk would become a "black hole," i.e. a closure that is being evaluated. From that point on, all the other threads would become blocked on black holes: as one thread called `atomicModifyIORef` and found a black hole inside, it would deposit a new black hole inside that depended on its predecessor. Because a black hole is a special kind of thunk that is invisible to applications, we could not play any of the usual `seq` tricks to jolly evaluation along.

At the time we encountered this problem, the black hole queue was implemented as a linear list, which was scanned during every GC. Most of the time, this choice of data structure was not a problem, but with thousands of threads on the list it became very painful.

After several weeks of research, heated discussion, hacking, epic debugging, and with casualties including one whiteboard, Simon Marlow performed a wholesale replacement of GHC's black hole mechanism. Instead of a single global black hole queue, GHC now queues a blocked thread against the closure upon which it is blocking. Simon's fix for our problem brought several other benefits, such as improving the wakeup performance threads blocked on black holes, less sharing of state across cores, and cleaner architectural separation inside GHC's runtime system.

9. Empirical results

Linux results were gathered on commodity quad-core server-class hardware with 4GB of RAM, and 2.66GHz Intel® Xeon® X3230 CPUs running 64-bit Debian 4.0. We used version 6.12.1 of GHC

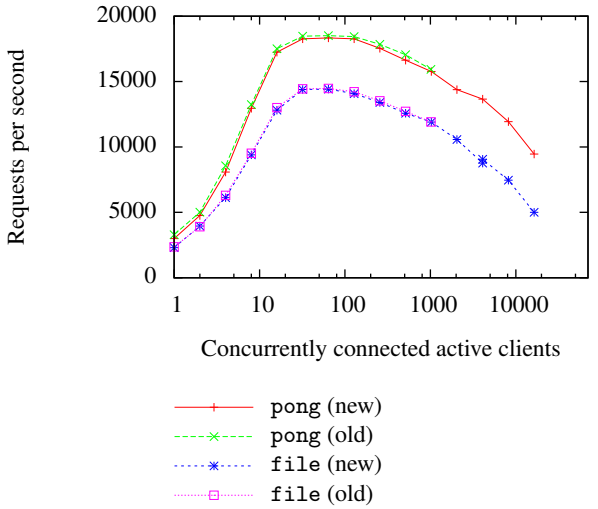

Figure 1. Requests served per second by two HTTP server benchmarks, with all clients busy, under old and new I/O managers.

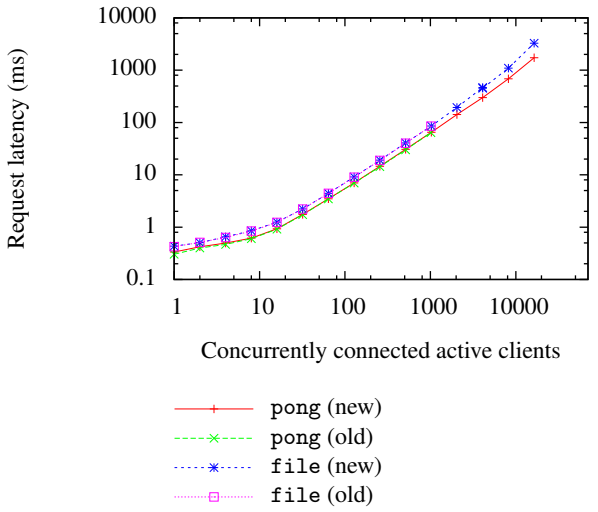

Figure 2. Latency per request, with all clients busy.

for all measurements, running server applications on three cores with GHC's parallel garbage collector disabled. When measuring network application performance, we used an idle gigabit Ethernet network, connected through a switch to an identically configured system.

9.1 Performance of event notification

To evaluate the raw performance of event notification, we wrote two HTTP servers. Each uses the usual Haskell networking libraries, and we compiled each against both the original I/O manager (labeled "(old)" in graphs) and our rewrite (labeled "(new)"). The first, `pong`, simply responds immediately to any HTTP request with a response of "`Pong!`". The second, `file`, opens and serves the contents of a file. For our measurements, we chose a file of 4,332 bytes in size. We used the battle-tested ApacheBench tool to measure performance under varying levels of client concurrency.

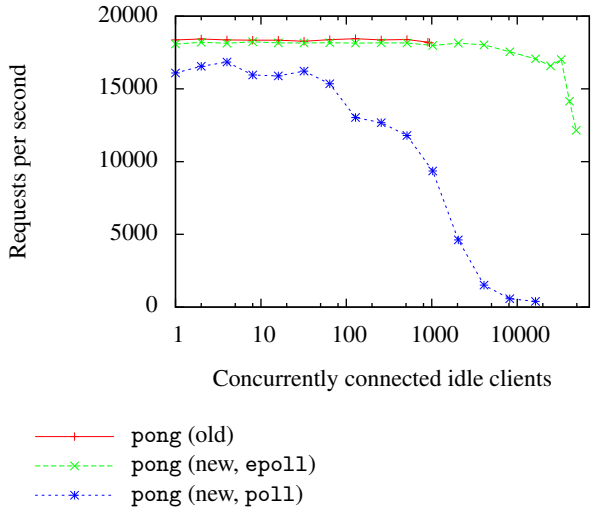

Figure 3. Requests served per second, with varying numbers of idle connections.

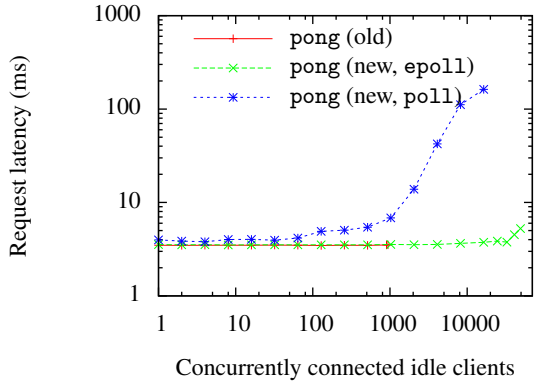

Figure 4. Latency per request, with varying numbers of idle connections.

In figures 1 and 2, our setup has all client connections active simultaneously; there are no idle connections. The graphs demonstrate that under these conditions of peak load, the `epoll` back end performs with throughput and latency comparable to the original I/O manager. Notably, the new event manager handles far more concurrent connections than the 1,016 or so that the original I/O manager is capable of. (We simply stopped measuring at 16,384.)

To create a workload that corresponds more closely to the conditions under which real applications must execute, we open a variable number of idle connections to the server, then measure the performance of a series of requests for which we always use 64 concurrently active clients. Figures 3 and 4 illustrate the effects on throughput and latency of the `pong` microbenchmark of varying the number of idle clients.

For good measure, we measured the performance of both the `epoll` and `poll` back ends to the new event manager. The original and `epoll` managers demonstrate comparable performance up to the 1,024 limit that `select` can handle, but while `poll` is erratic and disastrous, the performance of the `epoll` back end does not begin to fall off significantly until we have 50,000 idle connections open.

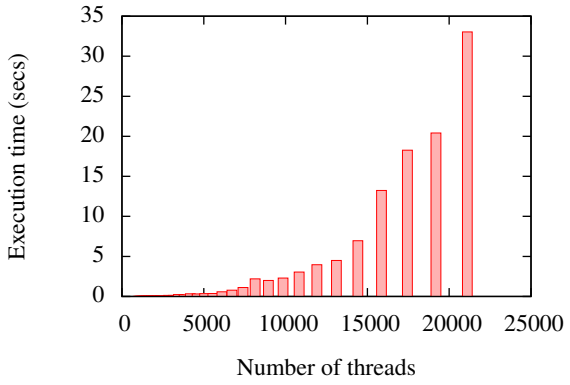

Figure 5. Performance of our `threadDelay` microbenchmark, run under the default I/O manager.

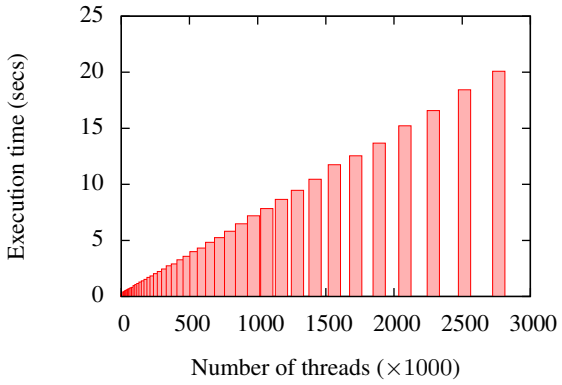

Figure 6. Performance of the `threadDelay` benchmark, run under our rewritten I/O manager. Note that the units on the x-axis are *thousands* of threads running.

We have tested the new event manager with up to 300,000 concurrent client connections. As expected, it works, though we might hesitate to refer to it as a paragon of speed at that level of concurrency.

In general, the small limit that `select` imposes on the number of concurrently managed resources prevents us from seeing any interesting changes in the behaviour of the original I/O manager at scale, because applications fall over long before any curves have an opportunity to change shape. We find this disappointing, as we were looking forward to a fair fight.

9.2 Performance of timeout management

We developed a simple microbenchmark to measure the performance of the `threadDelay` function, and hence the efficiency of the timeout management code. It spawns a number of threads, each of which delays for one millisecond and increments a counter. When the counter reaches its limit, the benchmark exits. We measured the execution time of the program as a whole, with the runtime system using 2 OS-level threads.

```
spawn :: Int → TVar Bool → IO ()
spawn numThreads done = do
  count ← newTVarIO 0
  replicateM_ numThreads ∘ forkIO $ do
    threadDelay 1000
    atomically $ do
      a ← readTVar count
      let !b = a+1
      writeTVar count b
      when (b == numThreads) $ writeTVar done True

main = do
  done ← newTVarIO False
  runInUnboundThread $ do
    spawn numThreads done
    atomically $ do
      b ← readTVar done
      when (not b) retry
```

As figure 5 indicates, GHC's traditional I/O manager exhibits $O(n^2)$ behaviour in the presence of a large number of timeouts. Since many server-side network applications maintain at least one active timeout per connected client, this implementation would pose a performance problem at high levels of concurrency.

In comparison, the measurements of figure 6 demonstrate that the rewritten timeout managament code has no problem coping with millions of simultaneously active timeouts, with available memory the only practical limit on the number that can be managed. For instance, the performance of our microbenchmark did not begin to degrade until we had three million threads and timeouts active on a system with 4GB of RAM.

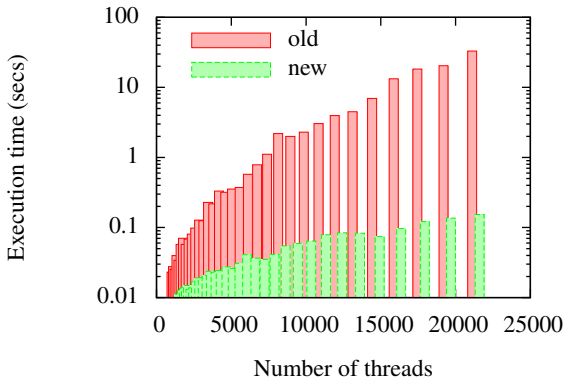

Figure 7. Comparative performance of old and new I/O managers on the `threadDelay` microbenchmark. Note the logarithmic scale on the y-axis.

Even for smaller numbers of threads, the new timeout management code is considerably more efficient than the old, as illustrated in figure 7. Since I/O management is performed by a single thread under both designs, the greater efficiency of our design should improve both throughput and latency under conditions of heavy event and timeout load.

Although using more than one core greatly improves the performance of this benchmark (figure 8), we have not yet investigated why using two cores seems to provide the best performance.

10. Future work

We initially developed our event management code as a standalone library. Our current focus is on integrating it into the GHC source tree and runtime system, so that it is available to all applications.

10.1 Lower overhead

We were a little surprised that `epoll` is consistently slightly slower than `select`. We posit that part of the reason for this might be that we currently issue two `epoll_ctl` system calls per event notification: one to queue it with the kernel, and one to dequeue

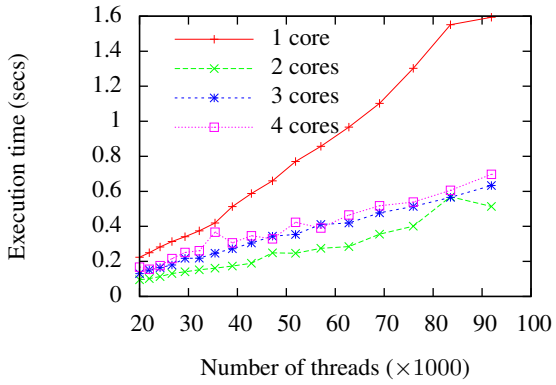

Figure 8. Executing on different numbers of CPU cores affects the performance of the `threadDelay` benchmark under our rewritten I/O manager.

it. In contrast, the original I/O manager has no added system call overhead. If we used `epoll` in edge-triggered mode, we could eliminate the need to call `epoll_ctl` to dequeue an event.

As a side note, the BSD `kqueue` mechanism is cleaner than `epoll` in this one respect, combining queueing, dequeueing, and checking for multiple events into a single system call. However, the smaller number of trips across the user/kernel address space boundary does not appear to result in better performance, and the `kqueue` mechanism is otherwise more cluttered and difficult to use than `epoll`.

10.2 Windows support

The Windows operating system presents a difficulty to our current design. Its support for scalable event notification is based around I/O completion ports, which are integrated tightly with the Windows threading mechanism, and which would require a considerable amount of replumbing to support.

10.3 Improved scaling to multiple cores

At least in theory, an application should be able to improve both throughput and latency by distributing its event management load across multiple cores. Our low-level event manager can be instantiated multiple times, with each instance managing a disjoint set of files or network connections.

We made a brief stab at using multiple event manager threads behind the scenes to handle the notification needs of the standard network library, but when we were unable to quickly demonstrate an improvement in performance, we returned our attention to the more practical concern of integrating our formerly standalone library into the GHC source tree. It appears difficult to create a benchmark that stresses the event manager in such a way that we could demonstrate a performance improvement via multicore scaling.

10.4 Better performance tools

When we were diagnosing performance problems with the event manager, we made heavy use of existing tools, such as the Criterion benchmarking library[7], GHC's profiling tools, and the ThreadScope event tracing and visualisation tool[5].

As useful as those tools are, when we made our brief foray into multicore event dispatching, we found that we had nothing that could help us to determine where our performance bottleneck might be. We speculate that if we could integrate the new Linux `perf` analysis tools with ThreadScope, we might gain a broader system perspective on where performance problems are occurring. We would also like to spend some time with Apple's Instruments tools for Mac OS X.

A. Additional materials

The source code to the original, standalone version of our event management library is available at `http://github.com/tibbe/event`.

Acknowledgments

We owe especial gratitude to Simon Marlow for his numerous detailed conversations about performance, and for his heroic fixes to GHC borne of the tricky problems we encountered.

We would also like to thank Brian Lewis and Gregory Collins for their early contributions to the new event code base.

References

[1] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of the International Conference on Coordination Models and Languages*, 2007.

[2] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61, .

[3] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 48–60, .

[4] R. Hinze. A simple implementation technique for priority search queues. In *Proceedings of the 2001 International Conference on Functional Programming*, pages 110–121, 2001.

[5] D. Jones Jr., S. Marlow, and S. Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2009 Haskell Symposium*, 2009.

[6] C. Okasaki and A. Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[7] B. O'Sullivan. Criterion, a new benchmarking library for Haskell. `http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/`, 2009.

[8] L. Peng and S. Zdancewic. A language-based approach to unifying events and threads. Technical report, University of Pennsylvania, 2006.

[9] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ASPLOS '96: Proceedings of the 1996 Annual Symposium on Principles of Programming Languages*, pages 295–308, 1996.

[10] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS IX: 9th Workshop on Hot Topics in Operating Systems*.