A Scalable I/O Manager for GHC

Johan Tibell
29 January 2010

http://github.com/tibbe/event

Server applications

- Performance matters
  - Servers cost (a lot of) money
  - We want as high throughput as possible
- Scalability: Performance shouldn't degrade (too much) when the number of clients increase
- Observation: In most (HTTP) servers, the majority of clients are idle at any given point in time

Why Haskell?

- Simple programming model:
  - Light-weight threads (forkIO)
  - Blocking system calls

```
server = forever $ do
  sock <- accept serverSock
  forkIO $ talk sock >> sClose sock

talk sock = do
  req <- recv sock
  send sock (f req)
```

Why Haskell?

- Performance:
  - Lots of concurrency
  - Statically compiled; should perform favorably in comparison with e.g. Python and Ruby
  - Alternative to C++ or Java when performance matters
- Correctness:
  - Pure functions
  - Strong static typing

What we are missing

- Support for a large number of concurrent connections
- Support for a large number of active timeouts
    - Typically one per connection

Implementing light-weight threads

- Schedule many light-weight threads across a set of OS threads.
- To avoid blocking the OS threads, use the select system call to monitor multiple file descriptors using a single OS thread.

Non-blocking I/O refresher

- select: a system call for polling the status of multiple file descriptors.
    - A call to select returns when one or more file descriptors are ready for reading writing, or
    - a timeout occurs.
- Only call a potentially blocking system call (e.g. recv) when we know it won't block!

Reading

```
data IOReq = Read Fd (MVar ())
           | Write Fd (MVar ())

read fd = do waitForReadEvent fd
             c_read fd

waitForReadEvent fd = do
  m <- newEmptyMVar
  atomicModifyIORef watechedFds (\xs ->
    (Read fd m : xs, ()))
  takeMVar m
```

Sleeping/timers

```
data DelayReq = Delay USecs (MVar ())

threadDelay time = waitForDelayEvent time

waitForDelayEvent usecs = do
  m <- newEmptyMVar
  target <- calculateTarget usecs
  atomicModifyIORef delays (\xs ->
    (Delay target m : xs, ()))
  takeMVar m
```

I/O manager event loop

```
eventLoop delays watchedFds = do
  now <- getCurrentTime
  (delays', timeout) <- expire now delays
  readyFds <- select watchedFds timeout
  watchedFds' <- wakeupFds readyFds watchedFds
  eventLoop delays' watchedFds'

expire _   [] = return ([], Never)
expire now ds@(Delay d m : ds')
  | d <= now  = putMVar m () >> expire now ds'
  | otherwise = return (ds, Timeout (d - now))
```

I/O manager event loop cont.

```
wakeupFds readyFds fds = go fds []
  where
    go []                    fds' = return fds'
    go (Read fd m : fds) fds'
      | fd `member` readyFds =
          putMVar m () >> go fds fds'
      | otherwise = go fds (Read fd m : fds')
    go (Write fd m : fds) fds'
      | fd `member` readyFds =
          putMVar m () >> go fds fds'
      | otherwise = go fds (Read fd m : fds')
```

The problem

- select:
  - ~O(watched file descriptors)
    - Most file descriptors are idle!
  - Limited number of file descriptors (FD_SETSIZE)
- Iterating through all watched file descriptors every time around the event loop.
- Timeouts are kept in a list, sorted by time
  - Insertion: O(n) as we need to keep the list sorted

A scalable I/O manager

- Scalable system calls
  - epoll, kqueue, and some Windows thing...
- Better data structures
  - Trees and heaps instead of lists

Timeouts

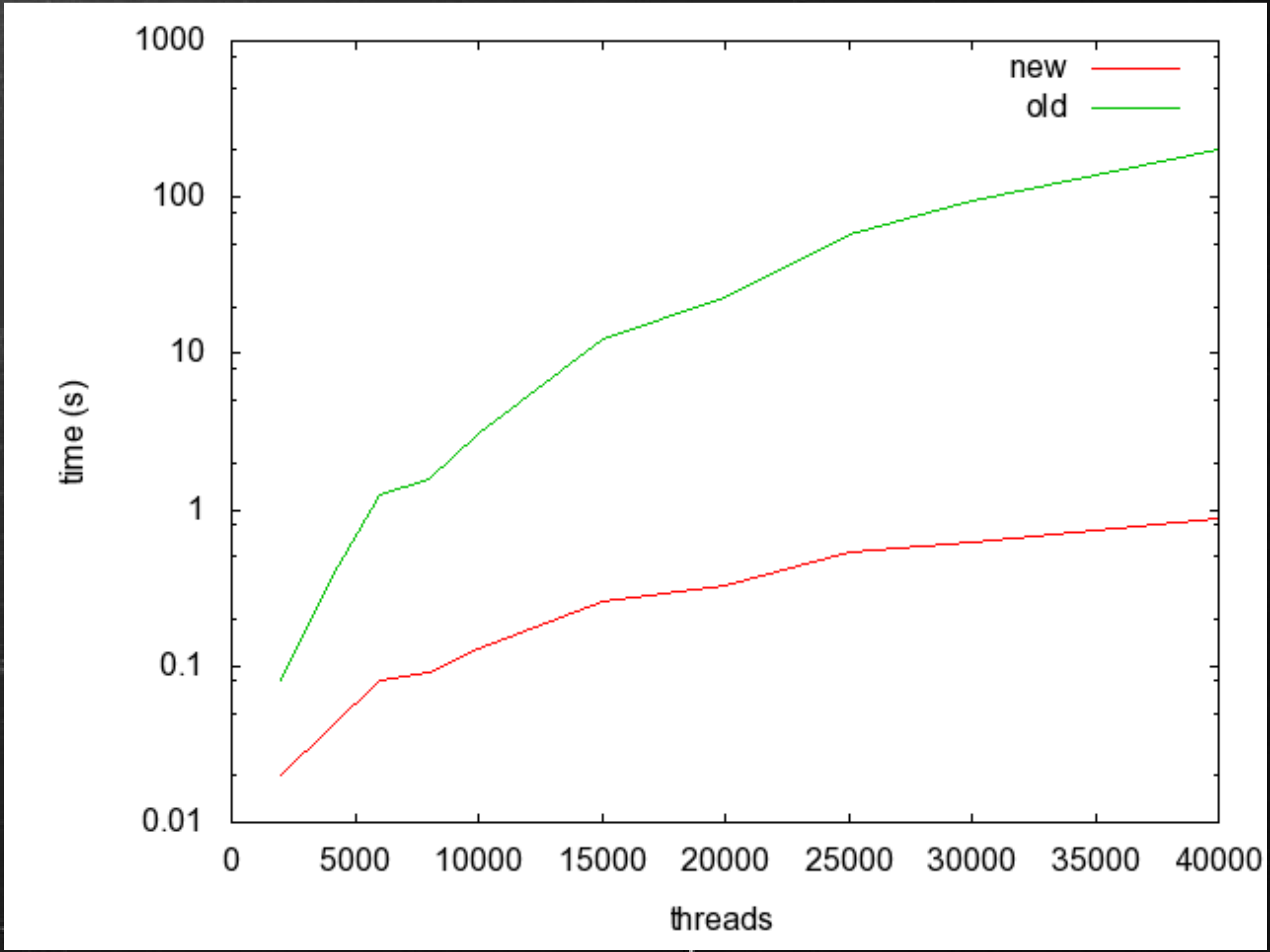
- New I/O manager uses a *priority search queue*
  - Insertion: O(log n)
  - Getting all expired timeouts: O(k*(log n - log k)), where k is the number of expired timeouts
- The API for timeouts is quite limited (to say the least!)
  - One function: threadDelay
- Priority search queues allows us to
  - adjust/cancel pending timeouts

Priority search queue performance

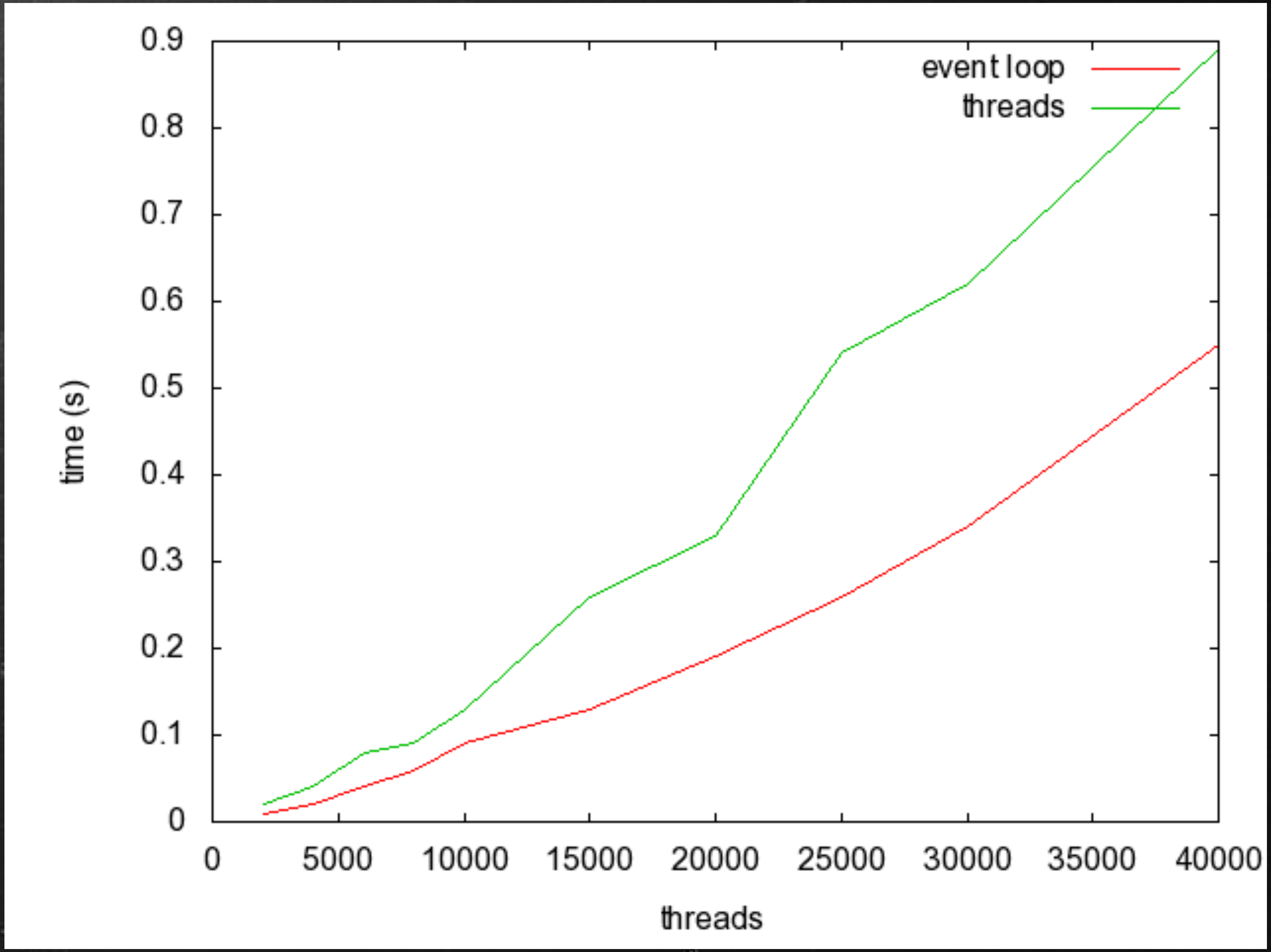
- Used Criterion extensively to benchmark and verify micro optimizations.

- Biggest performance gains:
  - Specialized to a single type of key/priority
  - Strict sub-trees
  - Unpacked data types

- Used QuickCheck to make sure that the optimizations didn't break anything.



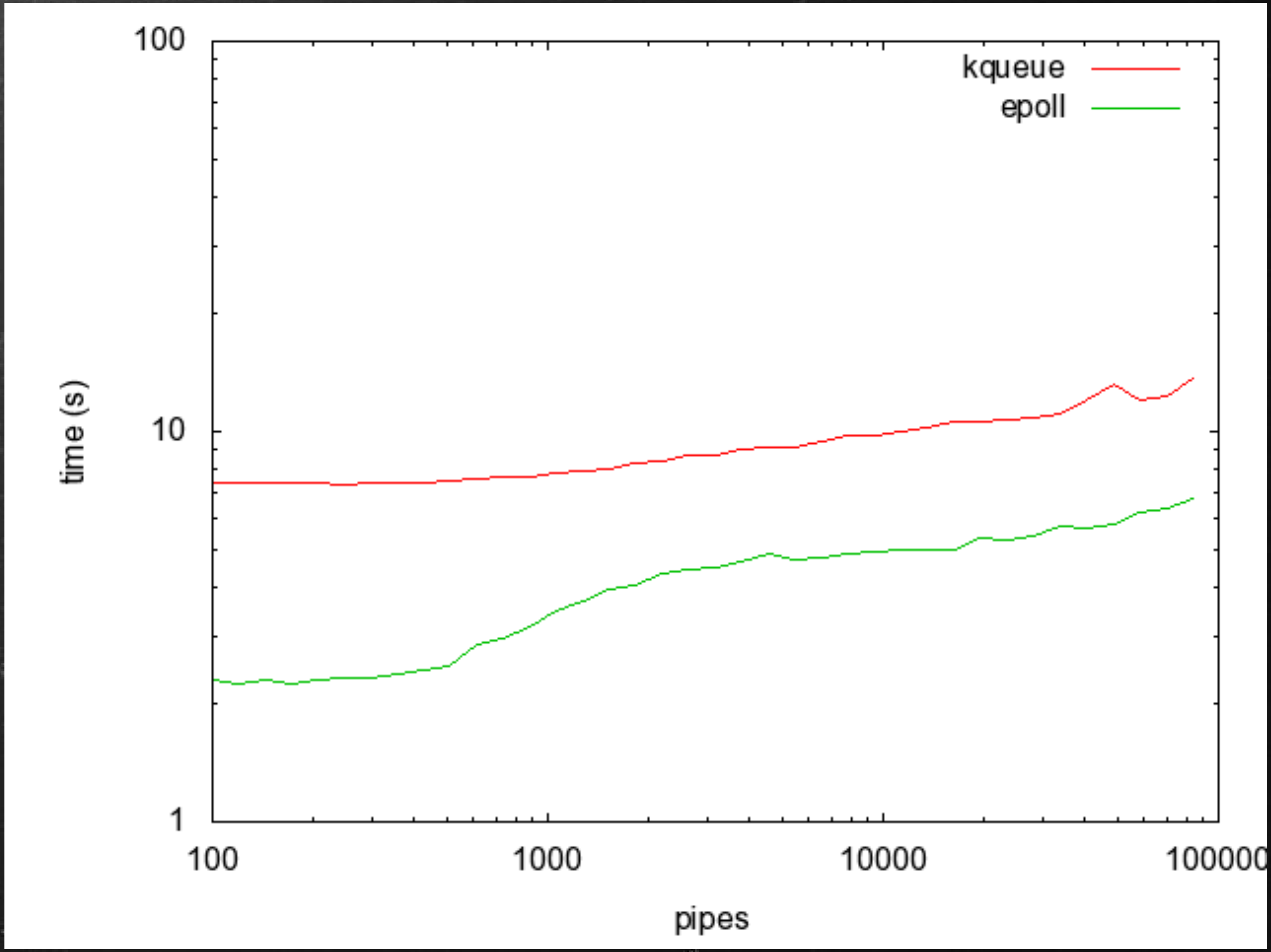
threadDelay 1ms



Light-weight threads vs event loop

Reading/writing

- Scalable system calls
  - epoll/kqueue: O(active file descriptors)
- Unblocking threads that are ready to perform I/O
  - O(log n) per thread, using an IntMap from file descriptor to MVar
  - Total running time for k active file descriptors is O(k * log n) instead of O(n).



Send 1M 1-byte messages through pipes

Aside: Good tools are important

- ThreadScope
  - Helped us find a pathological case in an interaction between atomicModifyIORef and GHC's scheduler

Current status

- Close to feature complete
- Needs more
  - testing
  - benchmarking
- Final step remaining: Integrate into GHC

Conclusions

- Haskell is (soon) ready for the server!
- We still need:
    - High performance HTTP server
    - High performance HTML combinator library
    - Composable and secure HTML form generation
        - Formlets + cross-site scripting protection
    - Scalable and distributed data store
        - We could just write binding to an existing one (but where's the fun in that!)