A High-Performance Multicore IO Manager Based on libuv (Experience Report)

Dong Han
Beijing Bytedance Inc.
Beijing, China
winterland1989@gmail.com

Tao He
Beijing Bytedance Inc.
Beijing, China
sighingnow@gmail.com

Abstract

We present a high performance multicore I/O manager based on libuv for Glasgow Haskell Compiler (GHC). The new I/O manager is packaged as an ordinary Haskell package rather than baked into GHC's runtime system (GHC RTS), yet takes advantage of GHC RTS's comprehensive concurrent support, such as lightweight threads and safe/unsafe FFI options. The new I/O manager's performance is comparable with existing implementation, with greater stability under high load. It also can be easily extended to support all of libuv's callback-based APIs, allowing us to write a complete high performance I/O toolkit without spending time on dealing with OS differences or low-level I/O system calls.

CCS Concepts • **Software and its engineering → General programming languages**; *Software libraries and repositories*;

Keywords Haskell, GHC, libuv, lightweight thread, IO manager, multicore, concurrency, scalability, performance

ACM Reference Format:
Dong Han and Tao He. 2018. A High-Performance Multicore IO Manager Based on libuv (Experience Report). In *Proceedings of the 11th ACM SIGPLAN International Haskell Symposium (Haskell '18), September 27–28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3242744.3242759

1 Introduction

GHC has a long history using lightweight threads to do concurrent programming [7]. The thread-based programming model greatly simplifies concurrent program's control flow (e.g. within a Haskell thread user can block on I/O activities synchronously), and is widely used to construct concurrent applications, such as TCP server, or web data crawler, etc.

Haskell '18, September 27–28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5835-4/18/09…$15.00
https://doi.org/10.1145/3242744.3242759

But the threaded model also gives rise to the I/O multiplexing problem: due to the asynchronous nature of underlining I/O operations, can we effectively schedule those threads doing I/O, so that they won't block the other threads from running. The problem's complexity includes I/O operations differences (for example, network I/O and file I/O may need different treatment), platform gaps and RTS scheduler's integration. The current I/O manager solved this problem using native event mechanism on UNIX and Linux platforms, but its design is not compatible with Windows, and introduced complexity into GHC code base.

Fortunately libuv provides an unified API for doing asynchronous I/O on different platforms. Combined with GHC's comprehensive concurrency support, we have implemented a new I/O manager, which can be used as a basis to write a complete I/O toolkit without dealing with different OS directly, thus saving a great amount of work in I/O library construction. We also benchmarked our new I/O manager and compared its performance with MIO [9], the current I/O manager in GHC, and I/O multiplexer designs from other languages/runtimes. Our benchmark showed that our new I/O manager's performance is comparable to others with higher throughput under high concurrency load.

In this paper we will focus on network I/O, but the techniques described here apply to other types of I/O operations as well. Section 2 presents the background of I/O multiplexing, including the event-driven I/O model in modern OSes, libuv's asynchronous I/O interface and I/O multiplexer in other languages/runtimes. We also give a brief review on GHC's runtime system, and MIO, the current I/O manager in GHC. Section 3 describes our new I/O manager's implementation in detail, including some interesting design choices. Section 4 gives the benchmark of our techniques and result analysis. Sections 5 presents the conclusion and some future work possibilities.

2 Background

Modern OS schedules users' programs with kernel threads, which provides an ordered control flow within a process. To do concurrent I/O operations, for example serving multiple TCP clients, we can simply spawn multiple kernel threads and serve each client with one thread. But with a high concurrent clients number, this architecture requires spawning a large number of kernel threads, which impose scheduling

difficulties and memory pressure. Thus an event-driven I/O model is often prefered, which enables processing concurrent requests within one kernel thread.

2.1 I/O Event Handling on Different Platforms

There are three major event-driven I/O frameworks to be discussed here: epoll [2] on Linux, kqueue [4] on BSD and IOCP[1]/overlapped I/O on Windows. The former two frameworks are very similar, userspace programs could interact with the framework through following steps:

1. Create a notification file descriptor, register events on the notification file descriptor.
2. Poll events by blocking on polling the notification file descriptor, optionally with a timeout limit.
3. Loop to process events received during step 2, perform actual I/O operations, register or modify events during processing.
4. Go back to step 3 and wait for next upcoming events.

On Windows scalable network I/O is done quite differently, the main steps are:

1. Create an I/O completion port, associate sockets with that completion port.
2. Use overlapped I/O call to read/write sockets, i.e. a pointer to an `OVERLAPPED` structure is passed as a parameter to such I/O call.
3. Block current thread by polling on that completion port.
4. Retrieve `OVERLAPPED` structures, process I/O results, associate or modify sockets during processing, make new overlapped I/O calls.
5. Go back to step 3 and wait for next completed overlapped I/O operations.

The main differences between IOCP model on Windows and the others is that on Windows, the overlapped I/O calls ask for pre-allocated buffers, after polling finished, notified completed I/O operations are already performed. But under epoll or kqueue framework, users have to perform the actual read or write manually. To provide an unified APIs in a cross-platform manner, a pre-allocated model must be chosen since there is no way to separate I/O operations from event notifications on Windows, while with epoll and kqueue, one can simply do the I/O operations with pre-allocated buffers after events are received.

2.2 Design Overview of libuv

libuv implements previously discussed pre-allocated model using different native event backend on different OS, its network interface can be illustrated as follows:

```
// creating an event loop instance
uv_loop_t *uvloop = uv_init(...);
// open a new socket, bind it with a loop instance
uv_tcp_t *socket = uv_tcp_init(uvloop, ...);
// attach some custom context data
socket->data = ...
// start receive event on that socket
uv_read_start(socket,allocation_callback,read_
    callback);
// wait on platform's poller
uv_run(uvloop);
```

The `allocation_callback` and `read_callback` will be called by libuv during `uv_run`. Inside these callbacks users can read `socket`'s data field to retrieve context data. libuv supports two polling modes: the `UV_RUN_NOWAIT` mode which can be taken as an event check step, it's implemented by giving native poller a zero timeout, so that `uv_run` returns even without receiving any events. Another mode `UV_RUN_ONCE` is implemented by setting the timeout to infinite, under this mode `uv_run` will not return until some events are polled. These two modes are especially useful when we integrate libuv with GHC's lightweight threads.

It's important to emphasize that most of libuv's function are not thread safe. The event loop has to be run single-threadedly to avoid corrupting libuv's internal states. Waking up a polling thread from other threads safely is supported via `uv_async_t`: user should initialize a `uv_async_t` on the loop thread first, then a `uv_async_send` call on that `uv_async_t` from another thread will wake up the loop thread immediately. This design is different from some of the native event backends. For example, with epoll it's safe to block a thread on an epoll file descriptor with `epoll_wait`, while other threads add events to that epoll file descriptor concurrently.

2.3 I/O Multiplexer in Other Languages

There are some interesting I/O multiplexer designs in other languages and runtime systems. For example, Node.js is widely known for its integration of google v8 javascript runtime and libuv. Another example is golang's netpoll module, which is designed specifically for multiplexing concurrent I/O operations between goroutines (the golang's lightweight thread implementation) [1].

Node.js embed google v8 runtime directly with libuv's event loop, so it shares the same callback-based asynchronous programming model of libuv. A Node.js process contains a single kernel thread which runs libuv's event loop together with v8 javascript runtime, javascript callbacks are scheduled by this embedder thread, passing between v8 runtime and libuv. To leverage multiple cores, Node.js provides cluster module, which uses multiple worker processes: a server's accept loop runs on the parent process, and distributes new connections to children processes via IPC.

Golang's netpoll is designed for monitoring multiple file descriptors to see if I/O operation is possible on any of them, thus it's not compatible with Window's IOCP model. Golang runtime starts a dedicated thread to run the netpoll, which accepts I/O registrations from other goroutines, and unblock

[1] https://msdn.microsoft.com/en-us/library/Windows/desktop/aa365198(v=vs.85).aspx

them after it polled events from underlying event file descriptors.

2.4 GHC Threaded RTS

GHC has a sophisticated RTS for scheduling thousands of lightweight threads to take advantage of modern multicore hardware [6]. The RTS maintains an array of *capabilities* (also called a *Haskell Execution Context (HEC)*), and each HEC is executed by a kernel thread. Thus several HECs can run concurrently on a multicore machine. The native thread running the HEC is responsible for scheduling all runnable Haskell threads live on that HEC, responds to cross-thread messages and balance workload by stealing Haskell threads from other HECs. GHC also support a special FFI mechanism call *safe* FFI [5], which migrates a HEC to another kernel thread when the original kernel thread enter a safe FFI call. This is important for running some blocking system calls since other Haskell threads on the same HEC need to continue to run concurrently with the Haskell thread running that system call, instead of being blocked until the system call returns.

Some thread synchronization primitives are directly provided by `base`, the Haskell standard library. For example we use `MVar` [7] as a single element channel: an empty `MVar` will block the Haskell threads reading it until another Haskell thread fills it with a value. A more complicated notification mechanism based on safe transactional memory [3] is provided by the `stm` package: `retry` will block current Haskell threads until there're changes occur on any transactional variables inside the transaction's context. These synchronization primitives are effective tools when we design the integration of libuv's callback-based asynchronous interface and GHC's lightweight thread runtime system.

2.5 Previous Work

The MIO I/O manager [9] is currently used in GHC since version 7.8. It has two event backends available, which use epoll on Linux, and kqueue on UNIX. This I/O manager do its job as follows:

1. MIO uses file descriptor, the integer value itself as key, and store Haskell callback functions in an `IntTable`.
2. A Haskell thread performing I/O operation on a file descriptor should first allocate an empty `MVar` and store a callback into the `IntTable`, which will fill the `MVar` when be called.
3. Now the Haskell thread may register events with native backend, and blocks itself on taking the previously allocated `MVar` by `takeMVar`.
4. An I/O manager thread is running concurrently with all the other Haskell threads on the same HEC. When run by the scheduler, it will get all available events with native backend poll mechanism, loop through the events and execute corresponding callbacks stored in the `IntTable`, thus previously blocked threads will be resumed, and continue performing their desired I/O operations.

MIO has a subtle logic to choose between safe and unsafe FFI call when calling native backend poll function. It's because when a thread enters a safe FFI call, GHC RTS may relinquish its HEC and trigger a context switch to another native thread [5], this would bring a significant overhead, so in MIO, the I/O manager always prefer doing zero-timeout nonblocking poll, and only if it didn't got any events after a whole RTS scheduler loop, it will start an infinite timeout blocking poll using safe FFI.

3 Implementation

Our libuv based I/O manager also has one I/O manager thread per HEC design. Each I/O manager works on a core data structure defined as follows:

```
data UVManager = UVManager
    { uvmBlockTable
        :: IORef (UnliftedArray (MVar Int))
    , uvmLoop       :: Ptr UVLoop
    , uvmLoopData   :: Ptr UVLoopData
    ...
    }
```

The `uvmBlockTable` is an array holding `MVars`, we use the `UnliftedArray` which appeared first in `primitive` package version 0.6.2.0[2] to reduce one layer of indirection that ordinary boxed array would bring. This array will be enlarged on demand, and perform as a parking lot to park users' Haskell threads.

Whenever a Haskell thread allocates a new libuv's socket struct, for example the `uv_tcp_t`, we will allocate an unique integer we called *slot* as the identity to pass the context between Haskell side and libuv's callbacks, this is done by write the slot number to the `uv_tcp_t`'s `data` field. This integer is also used as the index to find which `MVar` we should block on, thus it should always be smaller than the `uvmBlockTable`'s size, otherwise we will enlarge the table.

The `uvmLoop` field keeps the reference to a `uv_loop_t` struct. We also attach a small struct `uv_loop_data` to the loop's data field to pass data between Haskell and libuv's callbacks, The struct's reference is stored in `uvmLoopData` field for Haskell side fast accessing. It's defined as follows:

```
typedef struct {
   size_t    event_counter;
   size_t*   event_queue;
   char**    buffer_table;
   ssize_t*  buffer_size_table;
} hs_loop_data;
```

The `event_counter` and `event_queue` constitute a queue which we will use to record received events during libuv's `uv_run`. The counter will be reset to zero each time before we enter

[2] http://hackage.haskell.org/package/primitive-0.6.2.0

uv_run, and during callbacks we push the uv_tcp_t's slot into this queue. After uv_run we can peek this queue to find which threads we should unblock. This queue will filled with at most n events, where n is current active uv_tcp_t, so it must be enlarged in the same way as the uvmBlockTable.

When read or write a socket, another information we have to pass to libuv's callback is the receive/send buffer, this is done with the buffer_table and buffer_size_table fields above, which are arrays holding buffer pointer and buffer size respectively. These arrays' size should be synchronized with the uvmBlockTable, and indexed by uv_tcp_t's slot in the same way as the uvmBlockTable.

To combine libuv's asynchronous I/O interface with GHC's lightweight thread runtime, multiple I/O manager threads are started when a Haskell program starts, each one for a HEC. Within each I/O manager thread a UVManager is initialized, including a size N uvmBlockTable, a new uv_loop_t struct, a uv_loop_data struct with event_queue, buffer_table and buffer_size_table, each initialized with size N. Now a typical network I/O operation is performed as follows:

1. Initialize an uv_tcp_t struct on an opened socket, with the event loop on current HEC's UVManager.
2. Allocate a slot integer, write it to the uv_tcp_t's data field. If the slot exceeds UVManager's size, double related data structures, and supply a slot in the new range.
3. Use slot as the index to store the buffer pointer and buffer size from Haskell side into the uv_loop_data struct.
4. Call libuv's asynchronous read or write function, i.e. the uv_read_start or uv_write, with predefined callbacks. Take reading as an example: the allocate callback should fetch buffer from buffer_table and buffer_size_table, and the read callback should push the uv_tcp_t's slot to event_queue, and record read bytes number to buffer_size_table.
5. Block the I/O Haskell thread by takeMVar with the MVar from uvmBlockTable, indexed by slot allocated in previous step. If an asynchronous exception is received during blocking, we should directly close the socket, the slot will be freed by libuv inside close callback. Extra measures are also taken to stop user access closed socket.
6. In I/O manager thread, call uv_run function, with a similar timeout strategy with MIO, and during uv_run, received event on certain uv_tcp_t will be processed with supplied callbacks above.
7. After uv_run returns, read event_counter and event_queue from uv_loop_data struct, loop through slots pushed during uv_run, read I/O result recorded in buffer_size_table, then use the result to unblock corresponding Haskell threads.

There are some design details, for example, the slot allocator is based on GHC's stable pointer implementation, which can give a slot integer in constant time [8]. During each read and write callback, we save the asynchronous operation results in the buffer_size_table so no extra result buffers are needed. A key design choice that has to be made is how can we safely operate on libuv's state since its APIs is not thread-safe. We will describe our current design and potential alternatives in 3.1.

```
data UVManager = UVManager {
    ...
    , uvmRunning :: MVar Bool
    , uvmAsync   :: Ptr UVHandle
    }
```

3.1 Thread-safe Wake-up Mechanism

As we have already pointed out in 2.2 libuv is not a thread-safe library, thus we have to take care of race conditions on the Haskell side. For example, two situations below may raise the risk of race condition:

1. First a Haskell thread associated its uv_handle_t with an uv_loop_t on the same HEC, then after some time the thread is migrated to another HEC run by a different kernel thread, now it can register events on uv_handle_t while original uv_loop_t is being mutated.
2. After I/O manager decides to do a safe blocking poll, thus relinquishes current HEC to another kernel thread in order not to stop other Haskell threads on the same HEC, now it's possible for the other Haskell threads to race with I/O manager's uv_run by doing some event registrations on the same uv_loop_t.

In order to eliminate the race conditions above, two designs are experimented. The first one is based on MVar, namely we add two extra fields to UVManager:

The uvmRunning field is a boolean which indicates if current uv_loop_t is being polled, this boolean is protected with a MVar to stop being accessed concurrently. The uvmAsync field is the pointer to an uv_async_t, which is initialized on current uv_loop_t. During unsafe nonblocking poll, we simply occupy the uvmRunning lock until poll finish. In case of safe blocking poll, we swap the uvmRunning's content for True before entering FFI call, after poll finishes, we swap False back. If other Haskell threads want to mutate the event loop's state, it can use following combinator to ensure the mutation's safety:

```
withUVManager :: UVManager
              -> (Ptr UVLoop -> IO a)
              -> IO a
withUVManager uvm f = do
  r <- withMVar (uvmRunning uvm) $
    \ running ->
        if running
        then do
            uv_async_send (uvmAsync uvm)
            return Nothing
        else do
```

```
            r <- f (uvmLoop uvm)
            return (Just r)
  case r of
    Just r' -> return r'
    _       -> do yield
                  withUVManager uvm f
```

Here we try to acquire the `uvmRunning` lock first, this can eliminate concurrently event registration or race with the unsafe poll. After we successfully acquire the lock, a check on its content is performed: if the boolean indicates there'is a concurrently safe poll, we call `uv_async_send`, then yield current Haskell thread, and go back to this loop again; if there is no concurrently poll, then we directly call user's function `f` with current `uv_loop_t` pointer, and return the result.

An alternative design is based on GHC runtime's STM support. Instead of using `MVar` as the lock, we use a transactional variable to hold concurrent event loop's state, namely, we use following data structure:

```
data UVMState = UVMLocked | UVMPoll | UVMFree

data UVManager = UVManager {
    ...
    , uvmState :: TVar UVMState
    , uvmAsync :: Ptr UVHandle
    }
```

Before safe blocking poll we write `UVMPoll` to `uvState`, and write `UVMFree` after poll finishes. Under this scheme `withUVManager` should be adjusted to use STM instead:

```
withUVManager :: UVManager
              -> (Ptr UVLoop -> IO a)
              -> IO a
withUVManager uvm f = bracket_
  (atomically $ do
    state <- readTVar (uvmState uvm)
      case state of
        UVMLocked -> retry
        UVMPoll -> do
          uvAsyncSendSTM (uvmAsync uvm)
          retry
        UVMFree ->
          writeTVar (uvmState uvm) UVMLocked)
  (atomically $ writeTVar (uvmState uvm) UVMFree)
  (f (uvmLoop uvm))

uvAsyncSendSTM :: Ptr UVHandle -> STM ()
```

In the STM version, we `retry` if current event loop is running, so that the paused Haskell thread will resume when poll finishes, thus the race condition is eliminated. `uvAsyncSendSTM` wraps libuv's `uv_async_t` into STM monad, since it's safe to be called multiple times without any arguments.

Though the STM version provides a nice notification mechanism, in practice it's not chosen over the `MVar` version for several reasons:

1. The transaction's runtime overhead is larger than `MVar` 's, and STM's notification mechanism will wake multiple threads instead of keeping a FIFO order like `MVar`, which costs extra CPU time.
2. When `retry` returns, the waiting Haskell thread is added to the end of HEC's run queue. While in the `MVar` version, we put the waiting Haskell thread to the end of HEC's run queue by calling `yield` explicitly. If our I/O manager thread exits from `uv_run` before the scheduler resumes the waiting thread (which is common), then the STM version will have no latency advantages over the `MVar` version.

4 Benchmarks

We set up a small TCP server benchmark which includes golang's netpoll, node.js' cluster module, MIO in current GHC, and our new I/O manager. The server reads some input (up to 4KB) without parsing, then respond with 500 bytes of zeros in HTTP protocol so that we can use standard HTTP benchmark tools such as `ab`[3] or `wrk`[4]. If the connection is not closed after we responded, then we loop back and start reading again to simulate HTTP keep-alive connections.

Our loader generator has 64 cores (Intel(R) Xeon(R) E5-2683 v4), 128GB memory running Debian 8 with kernel version 3.16. While test server has 48 cores (Intel(R) Xeon(R) CPU E5-2650 v4), 256GB memory running the same OS. The server communicates with loader generator over a 10 Gbps Ethernet network. Load is generated via `wrk` HTTP benchmark tools, with thread option `-t64` to match generator machine's core number. Concurrency level is adjusted with `wrk`'s `-c` parameter.

We run this benchmark using different number of cores to measure scalability, with mechanism provided by each runtime. For example the golang version is run with different `GOMAXPROCS` environment variable, and Haskell versions are run with different `-N` RTS options. We also run Haskell programs with appropriate `-H` parameter (the heap size hint) under high concurrency, to keep GHC's GC and mutation time balanced. The exactly parameters used in this benchmark are `-H128M` under `-c1000`, `-H512M` under `-c5000` and `-H1G` under `-c10000`.

Figure 1 compares throughput of different multiplexers. Golang's single multiplexer thread is not sufficient to support this heavy I/O benchmark scaling effectively: the throughput improvement between `GOMAXPROCS=10` and `GOMAXPROCS=40` under high concurrency load (`-c100000`) only reaches 31.64%. The differences between MIO and our libuv based I/O manager is small, with our new I/O manager pushing 3% to 11% more throughput under concurrency level (`-c100000`). Both I/O managers are able to deliver very good performance

[3] https://httpd.apache.org/docs/current/programs/ab.html
[4] https://github.com/wg/wrk

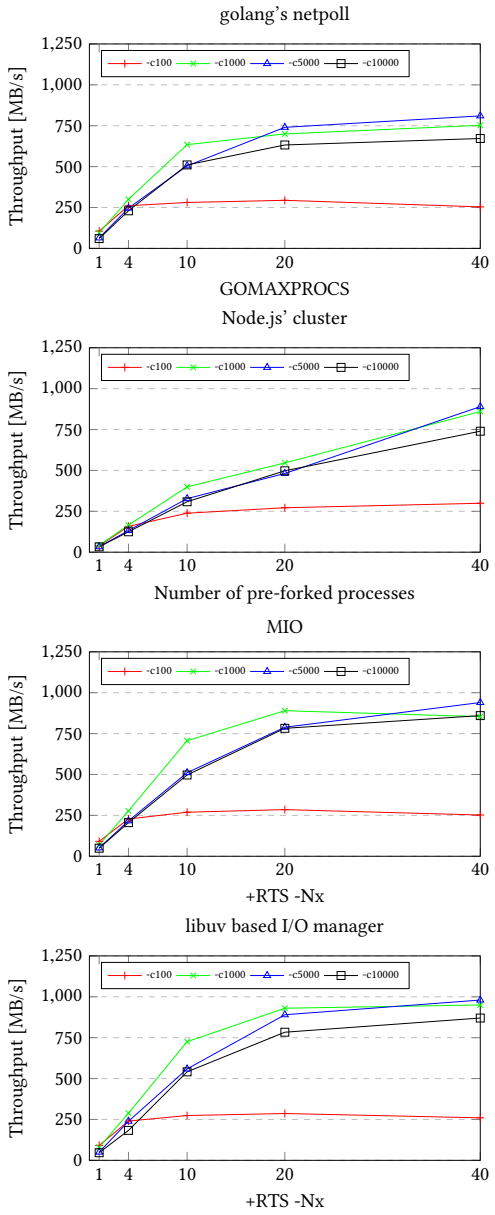

Figure 1. Throughput under different concurrency level, with different cores used.

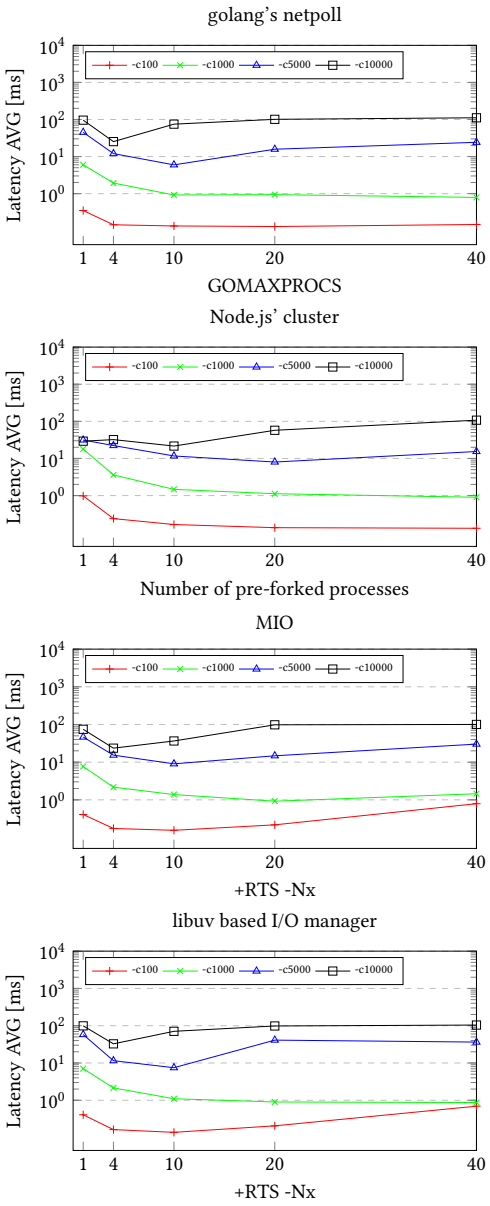

Figure 2. Average latency under different concurrency level, with different cores used.

and scalability. Figure 2 compares latency of different multiplexers. Overall these event-based multiplexers share pretty similar characteristics. Since this workload is I/O bounded, this result is expected.

5 Conclusions and Future Work

We presented and evaluated our new libuv based I/O manager, it provides comparable performance and under high concurrency load comparing with MIO, the current I/O manager in GHC. The design is easy to implement (core manager is less than 500 LOC), modular (without bringing modification to GHC runtime) and robust. The techniques described in this paper are also well suited to be used with many other types of I/O operations provided by libuv. In the future, We plan to write a comprehensive I/O toolkit supporting network, filesystem, tty, named pipe/UNIX domain and more based on this work.

References

[1] Nirmala. R. Deshpande, Erica Sponsler, and Nathaniel Weiss. 2012. Analysis of the Go runtime scheduler.

[2] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. 2004. Comparing and Evaluating epoll, select, and poll Event Mechanisms. (01 2004).

[3] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[4] Jonathan Lemon. 2001. Kqueue - A Generic and Scalable Event Notification Facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 141–153. http://dl.acm.org/citation.cfm?id=647054.715764

[5] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. 2004. Extending the Haskell Foreign Function Interface with Concurrency. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/1017472.1017479

[6] Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. *SIGPLAN Not.* 44, 9 (Aug. 2009), 65–78. https://doi.org/10.1145/1631687.1596563

[7] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. 1997. Concurrent Haskell. In *Proceedings of the 24rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 296–308. https://doi.org/11.1145/237721.237794

[8] Alastair Reid. 1994. Malloc Pointers and Stable Pointers: Improving Haskell's Foreign Language Interface. In *Draft Proceedings of the Glasgow Functional Programming Workshop*.

[9] Andreas Richard Voellmy, Junchang Wang, Paul Hudak, and Kazuhiko Yamamoto. 2013. Mio: A High-performance Multicore Io Manager for GHC. *SIGPLAN Not.* 48, 12 (Sept. 2013), 129–140. https://doi.org/10.1145/2578854.2503790