A Cross-Platform Scalable I/O Manager for GHC

Improving Haskell Networking on Windows

Mikhail Glushenkov
Spring 2016
Bachelor’s thesis, 15 credits
Supervisor: Jerry Eriksson
Examiner: Suna Bensch
Abstract

Haskell is a popular functional programming language. GHC is an industrial-strength implementation of Haskell that has a number of features making it a very attractive platform for writing high-performance network applications. Unfortunately, support for modern scalable network I/O APIs in the GHC runtime system is currently limited to Unix-like platforms. Haskell applications targeting Windows therefore cannot attain the same levels of scalability and performance as their Unix counterparts, and also have some correctness problems.

A redesign of GHC’s Windows I/O subsystem is proposed, making use of the Windows I/O completion ports API. A proof of concept implementation of the design is evaluated and found to be a strict improvement over the current state of affairs.
Acknowledgements

This thesis builds upon the work performed by many different people over the years. Johan Tibell and Bryan O’Sullivan implemented the initial version of the GHC I/O manager for Unix. Andreas Voellmy and coauthors further optimised and improved the Unix I/O manager in many ways. Felix Martini wrote the winio Haskell package for working with Windows asynchronous I/O. Joseph Adams is the author of the first version of Windows I/O manager.

I’d also like to express gratitude to my advisor, Jerry Eriksson, for providing advice and guidance during my work on this thesis.

Everyone else, you know who you are.
Contents

0 Introduction ................................................................. 1

1 Background ................................................................. 2
  1.1 Haskell ................................................................. 2
  1.2 GHC ................................................................. 3
  1.3 Concurrent Haskell ................................................... 3
  1.4 Foreign Calls ......................................................... 5
  1.5 GHC I/O Manager ..................................................... 5
  1.6 Bound threads ........................................................ 7

2 Old Windows I/O Manager ................................................ 8
  2.1 I/O completion ports ................................................ 8
  2.2 Windows Event Loop .................................................. 10
  2.3 Shortcomings of the old I/O Manager .............................. 11

3 New Windows I/O Manager ............................................... 13
  3.1 Removing bound thread pools ...................................... 13
  3.2 Removing stable pointers ........................................... 14
  3.3 Dequeueing multiple completions at once ......................... 14
  3.4 Scalable registrations ............................................... 15
  3.5 Multiple I/O manager threads ...................................... 15
  3.6 Avoiding the overhead of blocking calls ......................... 15

4 Empirical Results ......................................................... 17
  4.1 Proof of concept implementation ................................ 17
  4.2 Experimental setup .................................................. 17
  4.3 Benchmark 1: Pong Server .......................................... 17
  4.4 Benchmark 2: File Server ........................................... 18
  4.5 Latency ............................................................... 19

5 Future Work ............................................................... 20

6 Conclusions .............................................................. 21

References ................................................................. 21

A Accessing the source code ............................................ 24
# Introduction

Threads are a simple and powerful abstraction for implementing network applications such as web servers. Using one thread per connection lets the programmer express client-server communication in a modular way, simplifies exception handling and resource management, and prevents requests that are CPU-intensive to process from causing starvation by taking control over scheduling out of programmer’s hands. However, OS-level threads incur too much of an overhead in terms of context-switching and fixed memory costs to make them practical for high-performance applications; therefore, all modern APIs for scalable network I/O are event-based. In the event-based model communication with all clients is handled by just a single thread processing all incoming I/O events in an infinite loop.

*Haskell threads*, as implemented in GHC, combine the simplicity of the thread-based model with the performance of the event-based approach. Haskell threads have low memory and context-switching overhead, thus allowing to create a large number of them (100K-1M), enough to associate a single thread with each connection. They are multiplexed onto a much smaller number of OS threads, usually set equal to the number of CPU cores. A separate I/O manager thread, implemented as part of the runtime system, handles all I/O under the hood, providing the Haskell programmer with a convenient synchronous API. In the recent releases of GHC the I/O manager has been modified to use modern event-based I/O APIs and heavily optimised, making it competitive with current state of the art.

Sadly, those improvements are available only to GHC users on Unix-like platforms due to differences between I/O event models on Windows (I/O completion ports) and Unix (epoll/kqueue/poll). This thesis proposes a way to solve this problem by redesigning GHC’s Windows I/O subsystem to make use of the I/O completion ports API. The evaluation of this redesign shows it to be a strict improvement over the current state of affairs.

The rest of this document is organised as follows: Chapter 1 provides an overview of the background necessary for understanding the rest of the thesis; Chapter 2 outlines the previous attempt at implementing a Windows I/O manager for GHC; Chapter 3 contains a description of the proposed new design for the Windows I/O manager; Chapter 4 presents an empirical evaluation of a proof of concept implementation of the new design; Chapter 5 reflects on possible future directions; and finally, Chapter 6 contains some concluding words.
1 Background

This chapter provides some necessary background required for understanding the remainder of the thesis. It gives a short overview of Haskell, Glasgow Haskell Compiler and the Concurrent Haskell extension, and describes GHC’s input/output subsystem.

1.1 Haskell

Haskell [Haskell 2010] is a popular programming language with a unique combination of features making it highly attractive for programming high-performance networking applications. Specifically, Haskell has a powerful static type system, focus on immutability, and advanced support for concurrent and parallel programming [Marlow 2013]. GHC, the flagship Haskell compiler, generates fast native code and has a highly optimised network I/O layer [Voellmy 2013]. One unique aspect of Haskell is the non-strict (call-by-value) evaluation strategy.

The origins of Haskell can be traced to a 1987 meeting at a FPCA conference in Portland, Oregon, when a number of programming language researchers decided to agree on a standard for a lazy functional programming language to serve as a common foundation for application development and a research laboratory for testing new ideas [Hudak 2007]. Since then, Haskell continued to evolve and grow in popularity, gaining a measure of industry adoption and a number of advanced language extensions.

Experimental type system features notwithstanding, the attractiveness of Haskell to a working programmer is largely due to three factors: high level of abstraction, statically enforced type safety, and ease of refactoring.

Haskell code is high-level and concise. Its pure nature (by default functions in Haskell cannot have side effects or mutate data) makes it easier to reason about code, since a pure function will always have the same result for a given set of inputs. Its concise syntax makes the cost of creating helper (often local) functions low, and higher-order functions make such helper functions more powerful. This makes idiomatically written Haskell code less repetitive and encourages expressing the problem domain in a more abstract way.

Haskell is type safe. According to Robin Milner’s well-known quote, “well-typed programs cannot go wrong”. While originally that referred only to memory safety, a lot of Haskell-related research has gone into making Haskell programs not “go wrong” in more interesting ways. For example, functions with pure types are statically guaranteed to not have any side effects; for more advanced examples, see [Gibbons 2003]. Type-correct Haskell programs are also widely perceived to not be wrong to begin with: when the code compiles, it often just works\(^1\). Type inference removes the syntactic burden of manually written type annotations.

\(^1\)See https://wiki.haskell.org/Why_Haskell_just_works for more on this topic.
and, again, makes the code more concise.

Ease of refactoring is probably the killer feature of Haskell. Static type checking makes refactoring safe and easy, and therefore large refactorings are much more common in Haskell than in many other languages. This ability means that it’s easier for the code to evolve with changing requirements, thus making maintenance and continued development cheaper.

All this made Haskell one of the most popular languages in its niche (statically typed, garbage collected, functional). Hackage, the online repository of Haskell libraries, holds around ten thousand different packages, and every day there are about 1500 people communicating on the #haskell chat channel on #freenode.

1.2 GHC

GHC [GHC] is a flagship industrial-strength implementation of Haskell developed initially at the University of Glasgow (hence the name, Glasgow Haskell Compiler), and later at Microsoft Research Cambridge and by a large number of contributors worldwide. GHC is by now synonymous with Haskell, and while there is an official Haskell language standard [Haskell 2010], it is considered a bit conservative, and therefore a serious production code base written in Haskell will most likely utilise a number of GHC-specific extensions.

GHC is open source. It is largely written in Haskell itself, though some parts relevant for the topic of this work are in C. It compiles to fast native code – side-effect free nature of idiomatic Haskell makes it possible to implement advanced code transformation passes not possible in other languages. However, the mismatch between the Haskell execution model and the machine model translates into Haskell code usually being somewhat slower than C without manual optimisation.

A large number of target architectures is supported, though only x86 and x86-64 are currently Tier 1. Linux, Windows, OS X, and FreeBSD are Tier 1 supported target operating systems. An extensive runtime system (RTS for short) provides a parallel, generational garbage collector, a load-balancing multicore scheduler for Haskell threads, and support for profiling, exception handling and other tasks associated with running compiled Haskell code.

Besides having an abundance of type system and language extensions and state-of-the-art code optimisation capabilities, GHC is also renowned for its stellar support for concurrency and parallelism. While a number of different models and abstractions for doing parallel and concurrent programming are supported [Marlow 2013], the Concurrent Haskell extension is the most relevant one for the purposes of this thesis.

1.3 Concurrent Haskell

The Concurrent Haskell programming model revolves around the concept of user-level or “green” threads: lightweight threads that are multiplexed onto a much smaller number of operating system (OS) threads, also called Capabilities or Haskell Execution Contexts (HECs) in GHC parlance. The number of HECs is usually set to be equal to the number of CPU cores. The lightweight character makes it possible to create a large number (100K-1M) of Haskell threads; this can be used, for example, to associate a thread with each open
connection in a network server.

The following program listing provides an example of a simple server implemented in Concurrent Haskell:

```haskell
main :: IO ()
main = do
  listenSock <- socket AF_INET Stream defaultProtocol
  bind listenSock (SockAddrInet portNumber iNADDR_ANY)
  forever $ do
    commSock <- accept listenSock
    forkIO $ worker commSock

worker :: Socket -> IO ()
worker commSock = do
  inp <- recvAll commSock
  sendAll commSock inp
```

Here, the `main` IO action creates a socket and binds it to some port. It then starts an infinite loop in which it listens for new connections on that socket, and for each new connection forks off a new lightweight worker thread. The `worker` IO action handles all client communication; in this example it just echoes all input back to the client. An important detail is that all input/output operations are synchronous: `recvAll` and `sendAll` block until all data has been received and sent, respectively.

In addition to the ability to create new threads, Concurrent Haskell provides a number of facilities for inter-thread synchronisation and communication. For the purposes of this work it suffices to only look at MVars – a basic synchronisation primitive on top of which more sophisticated structures such as channels and semaphores can be built. An MVar is essentially a mutable location protected by a mutex; a minimal useful interface for working with MVars looks like the following:

```haskell
data MVar a

newEmptyMVar :: IO (MVar a)
putMVar :: MVar a -> a -> IO ()
takeMVar :: MVar a -> IO a
```

An MVar can be either empty or full. The `newEmptyMVar` operation creates a new empty MVar statically constrained to hold values of some type `a`. The `putMVar` operation puts some value of type `a` into an empty MVar. Trying to put a value into a non-empty MVar generates an exception. The `takeMVar` operation returns the contents of a given MVar, making it empty. If the MVar is empty to start with, `takeMVar` blocks until it’s full; if there’re multiple threads waiting on a single MVar, only one thread is woken up and the remaining ones are served in a fair (FIFO) order.
1.4 Foreign Calls

In the example in the previous section it was shown that an input/output operation can block a Haskell thread. It is clear, however, that an operation that blocks a Haskell thread must not be allowed to block the OS thread that it’s executing on: otherwise the remaining Haskell threads will never get a chance to run.

Safe foreign calls are a mechanism for preventing blocking or long-running calls to foreign (usually C) library code issued by a Haskell thread from blocking the underlying OS thread. A safe foreign call gets compiled to the following native code sequence (using C syntax):

```c
suspendThread();
safeForeignCall();
resumeThread();
```

The `suspendThread()` and `resumeThread()` operations are both primitives that are part of the GHC runtime system. The `suspendThread()` call removes the current Haskell thread from the HEC's run queue, releases the lock associated with the current HEC, wakes up an OS thread from a worker pool associated with the current HEC (or creates a new one in case the worker pool is empty), and hands off the HEC to that worker thread, which now proceeds with executing Haskell code. The actual foreign call is performed in the next step, after which the `resumeThread()` operation puts the current Haskell thread back on the HEC's run queue and adds the current worker thread to the HEC's worker pool and goes to sleep, or just shuts the worker thread down when the HEC's worker pool is full.

One downside of safe foreign calls is that they can’t be interrupted by asynchronous exceptions (such as timeouts). This can be mitigated by using an `interruptible` foreign call (enabled by the `InterruptibleFFI` extension). The runtime system will try to interrupt the thread blocked in such a call with an OS-level mechanism (a `SIGPIPE` signal on Unix systems or `CancelSynchronousIO` on Windows); otherwise they behave like safe ones.

Finally, there are unsafe foreign calls. An unsafe foreign call gets compiled to just a simple `call` instruction, therefore avoiding the runtime bookkeeping overhead described above. The downside is that unsafe foreign calls are both blocking and uninterruptible. Unsafe foreign calls can be useful in highly optimised low-level code, but should be handled with care.

1.5 GHC I/O Manager

The safe foreign call mechanism, while sufficient for relatively infrequent and long-running foreign calls, is not performant enough for highly concurrent network applications. As was shown in the previous section, using one Haskell thread per connection means that a new OS thread has to be allocated for each `recv` or `send` foreign call. This is problematic because a high number of native threads incurs unacceptable context-switching and memory overhead (each native thread requires approximately 1 MiB of memory for the native stack and associated structures).

This problem has been long recognised by the designers of modern operating systems. Modern APIs for doing scalable I/O, such as `kqueue` and `epoll`, use an event-based programming
model, in which a single OS thread can handle multiple concurrent connections. Let’s look at the interface provided by the Linux epoll mechanism to see what this means in practice:

\[
\text{int epoll_create(int size);}\
\]

\[
\text{int epoll_ctl(int epfd, int op, int fd,}\n\quad \text{\textbf{struct epoll_event *event});}\n\]

\[
\text{int epoll_wait(int epfd,}\n\quad \text{\textbf{struct epoll_event *events,}\n\quad \text{int maxevents, int timeout);}}\n\]

The `epoll_create` call creates a new `epoll` object. The client can then associate a number of file descriptors with that object by using the `epoll_ctl` call. For each file descriptor, the client can choose which types of events are watched by the `epoll` object – for example, the client can choose to wait for the file descriptor to become available for reading, writing, or both. Finally, the `epoll_wait` call blocks until any of the events registered with `epoll_ctl` actually happen, and then allows to find out which ones did, and for which file descriptors.

A typical network server using this API will use just a single OS thread for processing all incoming events. During the setup phase the application will create a socket for listening on some port, register it with the `epoll` object, and enter an infinite event loop, in which it will alternate between waiting for new events with `epoll_wait` and processing them. Each new connection will trigger an event on the socket listening on the server port; this will cause the application to register a fresh socket associated with that connection with the `epoll` object. Once a connection socket becomes ready for reading or writing, it will in turn trigger new events that the application will process; and once the application is done with a connection, it will close the connection socket and deregister it from the `epoll` object.

The main advantage of this model is its improved scalability and performance compared with using one OS thread per connection; the main disadvantage is that it is much less convenient to program with. Using one thread per connection lets the programmer express her intent in a modular way, simplifies exception handling and resource management (all resources dedicated to a single connection can be freed when the associated thread exits), and makes it easier to prevent CPU-intensive code processing a single request from starving other request handlers of CPU time (since control over preemption is taken out of the programmer’s hands).

Happily, modern versions of GHC allow to combine the performance of the event-based model with the convenience of the thread-based one. This is achieved by the runtime system component called the I/O manager [O’Sullivan 2010]. Simply put, the I/O manager is just another Haskell thread, implemented as part of the runtime system, that runs an epoll- or kevent-based event loop and notifies all other Haskell threads when new I/O events become available. Since the I/O manager handles all I/O for all Haskell threads, it can afford to use the safe foreign call mechanism for calls to `epoll_wait` and related system functions; since the I/O manager is a part of the runtime and only exits when the whole program shuts down, it doesn’t have to worry about asynchronous exceptions either. The I/O manager exposes the following semi-public API to Haskell code:

\[\text{In fact, as will be shown later, the I/O manager is actually used for implementing timeouts.}\]
A `threadWaitWrite` call creates an empty MVar, registers the provided file descriptor with the I/O manager, and blocks on that MVar. Once that file descriptor becomes ready for writing, the I/O manager runs the callback provided by `threadWaitWrite`, which fills in that MVar, waking the Haskell thread blocked in `threadWaitWrite`. The Haskell thread can then proceed with writing to that file descriptor. The `threadWaitRead` call works in the same way, the only difference is that it blocks until a file descriptor becomes ready for reading instead of writing.

Libraries for doing network I/O in Haskell are written to cooperate with this API, exposing the familiar synchronous `send/recv`-style interface to the programmer. Thus the context-switching and memory overhead associated with using one OS thread per connection is avoided in the GHC implementation of Concurrent Haskell. In recent GHC versions the I/O manager has been further optimised [Voellmy 2013], and its performance is currently on par with the state of the art in the area.

### 1.6 Bound threads

In addition to normal Haskell threads, GHC runtime also supports a variant called bound threads. Those are Haskell threads that have an associated OS thread and are guaranteed to only ever be executed in the context of that OS thread. This is useful for communicating with foreign libraries that are sensitive to the ID of the calling OS thread – for example, libraries that use thread-local-storage internally. Otherwise, bound threads are just like normal Haskell threads, the only difference is that they are created with the `forkOS` operation instead of `forkIO`.
2 Old Windows I/O Manager

GHC on Windows currently lacks an I/O manager [GHC Trac]. All network I/O is performed using the safe foreign call mechanism described in the previous chapter. Besides the performance and scalability problems outlined above, this means that I/O operations such as send and recv cannot be interrupted by asynchronous exceptions (e.g. a timeout or a keyboard interrupt).

However, there does exist an abandoned set of patches\(^1\) implementing a Windows I/O manager for GHC [GHC Trac]. Work described in this thesis builds on that early implementation. This chapter outlines the approach taken during that previous attempt and the problems that prevented it from being merged into mainline GHC.

2.1 I/O completion ports

The main reason that a Windows I/O manager hasn’t yet been added to GHC (besides the lack of resources) is that the approach chosen by the designers of the Windows API for doing scalable event-based I/O is quite different from the one used by Unix-like systems (Linux, FreeBSD, OS X/Darwin). As was shown in the previous chapter, on Unix a program utilising epoll or kevent registers a number of file handles with the OS and waits for them to become available for reading or writing. However, with the I/O completion ports (or IOCP for short), as the Windows API is called, it is the other way around: the program starts all its I/O operations asynchronously, and waits for them to complete.

Let’s look at the IOCP API in more detail. For the purposes of this discussion, it can be boiled down to the following five functions:

```c
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    ULONG_PTR CompletionKey,
    DWORD NumberOfConcurrentThreads
);

int WSAEnd(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,

\(^1\)Original author: Joseph Adams.
An I/O completion port is an abstract OS-level queue object similar to the epoll object created by epoll_create. The IOCP API function CreateIoCompletionPort serves the same purpose as epoll_create and epoll_ctl do for epoll. It is used for both creating new I/O completion ports and associating existing ones with file handles.

Asynchronous I/O operations are initiated by standard system functions like ReadFile and WriteFile. For this example the WSASend function from the Windows Sockets API was chosen because of this thesis’s focus on network I/O. The arguments WSASend accepts are almost the same ones that the send system call accepts on Unix: a socket handle, a buffer pointer, the number of bytes in the buffer, an output argument for recording the number of bytes actually sent, and a flag word. By default WSASend sends some data over a socket, blocking until it is done.

The lpCompletionRoutine argument can be ignored for the purposes of this discussion, leaving only the lpOverlapped argument, which is a pointer to an OVERLAPPED structure. The internals of the OVERLAPPED structure can be, again, ignored for the purposes of this discussion. The important thing is that when this pointer is non-zero and the socket has been marked as overlapped during creation, the send operation is asynchronous.

The GetQueuedCompletionStatus function is IOCP’s analogue of epoll_wait. It blocks until some I/O operation initiated on one of the file handles associated with a given I/O completion port actually completes and returns some data about the completed operation. The dwMilliseconds argument allows to perform a non-blocking poll or block for a limited amount of time.

Finally, the CancelIo function cancels all asynchronous I/O operations on the given file handle initiated by the calling OS thread.

Additional information on the I/O completion ports API is available in Microsoft’s official documentation online [MSDN].

```c
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);

BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytes,
    PULONG_PTR lpCompletionKey,
    LPOVERLAPPED *lpOverlapped,
    DWORD dwMilliseconds
);

BOOL WINAPI CancelIo(
    HANDLE hFile
);
```
2.2 Windows Event Loop

As is the case with the GHC I/O manager on Unix, the heart of the unofficial Windows GHC I/O manager is the event loop. In simplified form, a single iteration of the event loop looks like this:

```haskell
step = do
  runExpiredTimeouts
  m <- getNextCompletion
  case m of
    Nothing -> return ()
    Just (cb, numBytes, errCode) -> cb errCode numBytes
```

Let's ignore `runExpiredTimeouts` for a moment. As can be seen, the event loop is quite simple and mainly involves repeated calls to a Haskell procedure `getNextCompletion`, which is a wrapper over a safe foreign call to `GETQueuedCompletionStatus`. When `getNextCompletion` returns, it either means that some I/O operation has completed or that the time slice specified via the `dwMilliseconds` argument of `GETQueuedCompletionStatus` has been exceeded. In the former case `getNextCompletion` returns a callback associated with the completed operation, which is immediately executed.

In addition to taking care of input and output, the unofficial Windows I/O manager also handles timeouts. This part is, again, similar to the Unix I/O manager, which also has to deal with timeouts\(^2\). Timeout handling is, again, pretty simple: the internal I/O manager state structure contains a priority search queue that records all timeouts currently registered in the system. The `runExpiredTimeouts` operation removes all timeouts that expired during the latest time slice from the queue and executes the callbacks associated with those expired timeouts.

Just like the standard Unix I/O manager, the unofficial Windows I/O manager provides a semi-public interface for implementing synchronous user-level wrappers on top of low-level asynchronous I/O operations. As was explained in the previous section, the `threadWaitRead` / `threadWaitWrite` operations can't be efficiently implemented on Windows. Therefore, this interface looks a bit differently:

```haskell
associateHandle :: Manager -> HANDLE -> IO ()

withOverlapped :: Manager
                -> HANDLE
                -> StartCallback
                -> CompletionCallback a
                -> IO a

  type StartCallback = Overlapped -> IO ()

  type CompletionCallback a = ErrCode
                               -> DwORD
                               -> IO a
```

\(^2\)Discussion of this topic was omitted from the previous chapter in the interests of clarity.
Here, the `Manager` type represents a reference to the internal I/O manager state structure that can be obtained via a call to the `getSystemManager` operation. The `associateHandle` operation simply associates a file handle with the I/O manager's I/O completion port (as explained in the previous section, that file handle must be properly set up for doing asynchronous I/O).

The `withOverlapped` operation takes a file handle, a callback for starting an asynchronous I/O operation, and a callback that should be called on completion. It allocates a new `OVERLAPPED` structure, invokes the start callback, creates a new empty MVar, registers the I/O operation with the I/O manager and blocks on the MVar it has just created. When the I/O operation completes, the I/O manager invokes the completion callback and signals the MVar, causing `withOverlapped` to return control to the caller.

There are two further complications that the Windows I/O manager has to deal with: canceling asynchronous I/O operations and sensitivity to the ID of the calling thread. When a Haskell thread blocked in `withOverlapped` receives an asynchronous exception, it must cancel the unfinished I/O operation it has just initiated. As was shown in the previous section, the `cancelIo` function simply cancels all asynchronous I/O operations on a given file handle issued by the calling OS thread; this can be problematic if two different Haskell threads have initiated I/O operations on the same file handle from the same OS thread, but only one of the Haskell threads needs to be canceled.

Similarly, asynchronous I/O operations in Windows are also sensitive to the ID of the calling OS thread. When an OS thread that initiated an asynchronous I/O operation exits, that operation is canceled. As was explained in Section 1.4, GHC runtime maintains a fixed-size pool of native worker threads that are considered interchangeable for all intents and purposes. An OS thread that has at some point initiated an asynchronous I/O operation can be simply shut down by the runtime at any moment.

Therefore the unofficial Windows I/O manager enforces the following invariant: each file handle registered with the I/O manager has an associated pool of bound threads (see Section 1.6), and each bound thread in the pool can have at most one pending I/O operation per file handle. However, bound threads can be shared between pools associated with different file handles. Calls to `cancelIo` are always issued from the same bound thread (and therefore OS thread) that initiated the I/O operation. A bound thread that is a part of a pool associated with some file handle never exits until all pending I/O operations initiated from that thread have completed.

### 2.3 Shortcomings of the old I/O Manager

The unofficial Windows I/O manager, as described above, is reasonably complete and functional; it even solves the problem of network I/O operations being uninterruptible by asynchronous exceptions. However, the overhead of maintaining multiple pools of bound threads makes it unsuitable for practical use. A simple benchmark measuring the performance effect of enabling the unofficial Windows I/O manager by default (two Haskell threads exchanging messages over a socket) showed an order of magnitude difference compared with the standard Haskell network library (implemented on top of safe foreign calls on Windows).
Another problem of the unofficial Windows I/O manager is that one has to add support for the `WITH/VERLAPPED` interface to all standard Haskell I/O libraries in order to actually take advantage of it. This, unfortunately, is a fundamental limitation forced by using IOCP; but see Chapter 6 for some discussion on this topic.
3 New Windows I/O Manager

As was shown in the previous chapter, the existing attempt at implementing a Windows I/O manager for GHC suffers from performance problems making it unusable in practice and unsuitable for inclusion in mainline GHC. This chapter describes a new Windows I/O manager design that attempts to solve or at least alleviate those problems. Most ideas discussed in this chapter were originally proposed in the [Voellmy 2013] paper. A partial proof of concept implementation of this design is the focus of the next chapter.

3.1 Removing bound thread pools

As was explained in Section 2.3, the main problem with the old Windows I/O manager is the overhead of maintaining multiple pools of bound threads and the associated context switching costs. Getting rid of the bound thread pools would probably make the I/O manager code at least as fast as the standard network library, if not faster.

The first reason for implementing the bound thread pool scheme was the limited functionality of the Canec1Io function: it cannot cancel specific I/O operations, and it cannot cancel I/O operations initiated by other OS threads. Fortunately, in Windows Vista Microsoft introduced the following function that lifts both of those restrictions:

```c
BOOL CancelIoEx(
    HANDLE hFile,
    LPOVERLAPPED lpOverlapped
);
```

This function is not sensitive to the ID of the calling thread, and the optional OVERLAPPED argument that it accepts allows to identify a specific I/O operation previously initiated by some thread in the current process. Not supporting Windows versions earlier than Vista used to be a concern for some time, but it appears that starting with the GHC 8 release pre-Vista versions of Windows are no longer supported anyway\(^1\).

The second reason for having a bound thread pool scheme was that all asynchronous I/O initiated by a given OS thread is canceled when that thread exits. Given that the GHC runtime starts and shuts down native worker threads as it pleases, this means that any operation initiated by a Haskell thread can in principle be randomly canceled at any moment. Working around that restriction is a bit more burdensome. What is needed is a way to prevent the GHC runtime from prematurely terminating native worker threads that have pending I/O operations. This can be implemented by adding a reference-counting system for native worker threads to the runtime system with the following Haskell interface:

The `incrTaskRefCount` operation increases the reference count for the OS thread (or Task in GHC RTS parlance) that the current Haskell thread is running on and returns an action that decreases the reference count for that thread when run (this is needed because Haskell thread can migrate between Tasks). A Task whose reference count is non-zero can never get forcibly shut down by the run time system even if the number of Tasks in the HEC worker pool exceeds the maximum threshold. To prevent the number of Tasks in the HEC worker pool from growing uncontrollably the runtime system tries to shut down the excess Tasks whose reference count is zero during the garbage collection phase.

The `withIncrTaskRefCount` operation is just an exception-safe Haskell wrapper for `incrTaskRefCount`. It increases the reference count of the current Task for the duration of the IO action given to it as an argument. It is used in the implementation of `withOverlapped` for managing the reference counts of Tasks with pending I/O operations:

```haskell
withOverlapped = withIncrTaskRefCount $ do ...
```

### 3.2 Removing stable pointers

Old Windows I/O manager associated a stable pointer with each registered completion callback. A stable pointer is reference to a Haskell object that is not affected by garbage collection and so can be used to refer to objects on Haskell heap from C. That stable pointer was then saved adjacent to the OVERLAPPED structure and then passed back to the Haskell code by the C wrapper for `GetQueuedCompletionStatus`.

The implementation of stable pointers in GHC RTS is not very scalable since it uses a global lock internally. The new design removes stable pointers completely and manages completion callbacks using a hash table on the Haskell side indexed by pointers to OVERLAPPED structures.

### 3.3 Dequeuing multiple completions at once

The `GetQueuedCompletionStatusEx` Windows API function allows to dequeue multiple completions from an I/O completion port with a single foreign call. Using it instead of `GetQueuedCompletionStatus` improves the performance in the case where the application needs to handle large volume of completions. It is supported since Windows Vista.

---

3.4 Scalable registrations

The remaining features of the new Windows I/O manager design are aimed at improving its performance and scalability beyond the current status quo on Windows. They are based on the pioneering work in the [Voellmy 2013] paper. Not all of these optimisations are currently implemented (see Section 4.1 for a review of the implementation status). However, it is nevertheless instructive to briefly discuss them here.

The first bottleneck identified in the [Voellmy 2013] paper is the callback registration mechanism used by the I/O manager. In the original implementation it was just a hash table protected by a mutex; when a large number of Haskell threads tried to update that hash table at the same time that caused contention and degraded performance. That problem was solved by an additional level of indirection: using a small fixed number ($2^5$) of mutex-protected hash tables instead of just one, plus a read-only hash table on top. To register a callback in this scenario the client first maps the file handle it is interested in to one of the $2^5$ mutable hash tables using the top-level read-only one, and then modifies the appropriate mutable hash table. The I/O manager uses the same method to find out which callback to invoke.

This technique was found to be directly applicable for the Windows I/O manager with a modest modification to its internal state structures and the code of \texttt{WITH/VERLAPPED}.

3.5 Multiple I/O manager threads

As discussed in the [Voellmy 2013] paper, a single I/O manager thread in a highly concurrent server application simply cannot saturate all HECs with work beyond 4 cores. This was resolved by parallelising the I/O manager and setting the number of I/O manager threads equal to the number of HECs (which is usually the same as the number of cores). This technique should be also possible to apply in the case of the Windows I/O manager. The \texttt{WITH/VERLAPPED} function will have to be modified to register the completion callback with the I/O manager thread associated with the current HEC instead of the global one. This optimisation should improve core utilisation (by distributing the I/O manager load more evenly across cores) and locality (by minimising the need for cross-HEC communication).

3.6 Avoiding the overhead of blocking calls

Another optimisation discussed in the [Voellmy 2013] paper is avoiding expensive safe foreign calls in the event loop by using a couple of initial fast non-blocking unsafe calls to poll for ready events before proceeding with the slow blocking call. After each non-blocking poll the I/O manager thread voluntarily yields the control to the RTS scheduler, causing the I/O manager thread to be moved to the end of the current HEC’s run queue. This speeds things up in the case where a large number of events is being generated and processed by avoiding an expensive context switch incurred by the safe foreign call. Voluntary yielding of control, however, increases the latency of timeout handling, so that component has to be split into its own Haskell thread.
This optimisation should be directly applicable for the Windows I/O manager as well with minimal modifications to the event loop. The timer manager code from the Unix version can probably be reused.

Finally, the [Voellmy 2013] paper discusses a technique for working around an internal synchronisation bottleneck in the Linux kernel. This optimisation seems to be Linux-specific; however, it may provide insight into further optimisation of the Windows I/O manager.
4 Empirical Results

A proof of concept version of the design described in the previous chapter was implemented and experimentally evaluated using a number of simple benchmarks. This chapter discusses the results obtained during the experimental evaluation. The new design is shown to be competitive with the status quo on Windows, and thus potentially appropriate for inclusion in GHC at some point in the future.

4.1 Proof of concept implementation

The proof of concept implementation includes most of the design elements described in Chapter 3, except multiple I/O manager threads (Section 3.5) and non-blocking event polling (Section 3.6). Additionally, instead of implementing the full reference counting scheme as described in Section 3.1, the proof of concept implementation just sets the maximum spare worker limit to a high number (500); this shouldn’t affect the results as the number of spare worker threads created is expected to be small. See Appendix A for instructions on how to access the source code of the proof of concept implementation.

4.2 Experimental setup

The benchmarks were run on a Windows 10/Ubuntu Linux 16.04 machine with a 3.4 GHz four-core Intel Core i5 4670K processor and 16 GiB of RAM. The same machine was used as both client and server. ApacheBench\(^1\) was used to test the network I/O performance. Each test run performed 50 thousand requests total (ApacheBench’s \(-N\) parameter) with a varying number of concurrent connections (ApacheBench’s \(-c\) option). Appendix A contains more detailed instructions for replicating the experiment.

4.3 Benchmark 1: Pong Server

The Pong benchmark implements a simple HTTP server that sends a response with a constant five-byte body (“Pong!”) in answer to any HTTP request. The pong version of the server is implemented using the new I/O manager prototype, and the pong-baseline version is implemented using the standard Haskell network library.

Table 1 shows the number of requests per second as reported by ApacheBench as a function of the number of concurrent connections.

\(^1\text{https://httpd.apache.org/docs/2.4/programs/ab.html}\)
Table 1 Number of requests per second against concurrent connections. Higher is better.

<table>
<thead>
<tr>
<th>Conc. Connections</th>
<th>pong</th>
<th>pong-baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10738</td>
<td>10526</td>
</tr>
<tr>
<td>100</td>
<td>10810</td>
<td>10191</td>
</tr>
<tr>
<td>1000</td>
<td>8290</td>
<td>8602</td>
</tr>
<tr>
<td>10000</td>
<td>4885</td>
<td>4726</td>
</tr>
</tbody>
</table>

As can be seen from Table 1, code using the new I/O manager is competitive with code that uses the standard network library on this benchmark.

Table 2 Requests per second and latency (ms) against concurrent connections on Linux. For RPS, higher is better; for latency, lower is better.

<table>
<thead>
<tr>
<th>Conc. Connections</th>
<th>Requests per second</th>
<th>Time per request, ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>42369</td>
<td>0.23</td>
</tr>
<tr>
<td>100</td>
<td>39756</td>
<td>2.5</td>
</tr>
<tr>
<td>1000</td>
<td>35282</td>
<td>28.3</td>
</tr>
</tbody>
</table>

For comparison, Table 2 shows the results of the same experiment performed on Linux, also including the latency numbers. As can be seen from this table, performance of the Windows implementations still has room for improvement.

### 4.4 Benchmark 2: File Server

The File Server benchmark is a simple HTTP server that reads and serves a small static file from disk in response to any HTTP request. The file version of the server is the one implemented using the new I/O manager prototype and asynchronous file I/O, and the file-baseline version is implemented using the standard network library and synchronous file I/O.

Table 3 shows the number of requests per second measured with ApacheBench on Windows as a function of the number of concurrent connections.

Table 3 Requests per second against concurrent connections. Higher is better.

<table>
<thead>
<tr>
<th>Conc. Connections</th>
<th>file</th>
<th>file-baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10289</td>
<td>9275</td>
</tr>
<tr>
<td>100</td>
<td>10094</td>
<td>9142</td>
</tr>
<tr>
<td>1000</td>
<td>8510</td>
<td>8163</td>
</tr>
<tr>
<td>10000</td>
<td>4347</td>
<td>2844</td>
</tr>
</tbody>
</table>

As can be seen from Table 3, the new I/O manager is competitive with the old one on this benchmark as well, overall being slightly faster.
4.5 Latency

Finally, Table 4 shows the latency numbers (average time per request) for both Pong and the File Server benchmarks on Windows.

<table>
<thead>
<tr>
<th>Conc. Connections</th>
<th>Pong</th>
<th>Pong-baseline</th>
<th>File</th>
<th>File-baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.925</td>
<td>0.922</td>
<td>0.953</td>
<td>1.0</td>
</tr>
<tr>
<td>100</td>
<td>9</td>
<td>9.5</td>
<td>9.6</td>
<td>10.6</td>
</tr>
<tr>
<td>1000</td>
<td>115</td>
<td>114</td>
<td>115</td>
<td>128.7</td>
</tr>
<tr>
<td>10000</td>
<td>2062</td>
<td>2225</td>
<td>2206</td>
<td>3318</td>
</tr>
</tbody>
</table>

As can be seen from these results, the new implementation is competitive with the old one even when it comes to latency.
5 Future Work

The prototype Windows I/O manager implementation was shown above to be competitive with current status quo in GHC. However, a number of issues still need to be resolved before the new implementation can be declared production-ready.

First, support for the new Windows I/O manager must be integrated with the standard network library so that existing Haskell applications could take advantage of it. There are some bits and pieces missing from the GHC RTS’s low-level I/O device abstraction that need to be implemented first.

Second, the new Windows I/O manager code must be extended to also handle file I/O in addition to network I/O. This will likely require extensive changes to the GHC runtime support library to make it use native Windows API types and functions instead of the CRT wrapper.

Additionally, performance of the current implementation could be improved. As discussed in Chapter 4, there are two elements of the design not yet implemented in the current prototype: per-core event manager threads and non-blocking event polling that are expected to seriously boost the performance. The implementation could be also profiled with e.g. ThreadScope\(^1\) to weed out the possible remaining bottlenecks. If the full design were to be implemented, testing its performance would require a more advanced setup akin to the one described in [Voellmy 2013] to saturate the network; Amazon EC2 can probably be used for that purpose.

Finally, it’d be nice to share more code among the Windows and Unix I/O manager implementations. It should be possible to come up with an API that combines threadWaitRead / threadWaitWrite and withOverlapped by implementing an IOCP-style interface on top of epoll/kqueue. The timeout manager code can probably be reused wholesale.

\(^1\)https://wiki.haskell.org/ThreadScope.
6 Conclusions

The existing attempt at implementing a Windows I/O manager for GHC was evaluated and found to be inadequate from the performance point of view. A new design for the Windows I/O manager for GHC was developed, taking advantage of the improved asynchronous I/O API available since Windows Vista and various optimisations developed in the [Voellmy 2013] paper.

A prototype implementation of the new design was evaluated using a number of benchmarks and was found to be an improvement over the status quo. At this stage, the main advantage of the new implementation is improved correctness (proper handling of asynchronous exceptions); when it comes to performance, the new implementation is usually competitive with the old one and often slightly faster.

The proposed design was therefore shown to be viable in practice; with more effort, it should be possible to optimise the prototype implementation further and make it more complete with respect to functionality. Once the new Windows I/O manager is included in GHC, the improvements will be transparently available to existing Haskell applications.
References

Simon Marlow et al.
https://www.haskell.org/onlinereport/haskell2010/
URL accessed June 8, 2016.

[Hudak 2007] A History of Haskell: being lazy with class
Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler
The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III) San Diego, California, June 9-10, 2007.

[Marlow 2013] Parallel and Concurrent Programming in Haskell
Simon Marlow
O’Reilly 2013, ISBN: 978-1449335946

[Gibbons 2003] The Fun of Programming
ed. by Jeremy Gibbons and Oege de Moor

[GHC] The Glasgow Haskell Compiler: a technical overview
SL Peyton Jones, CV Hall, K Hammond, WD Partain, and PL Wadler
March 1993.

[O’Sullivan 2010] Scalable I/O Event Handling for GHC
Bryan O’Sullivan & Johan Tibell.
Proceedings of the 2010 ACM SIGPLAN Haskell Symposium (Haskell’10).

[Voellmy 2013] Mio: A High-Performance Multicore IO Manager for GHC
Andreas Voellmy, Junchang Wang, Paul Hudak, Kazuhiko Yamamoto.
Proceedings of the 2013 ACM SIGPLAN symposium on Haskell (Haskell’13).

[Marlow 2004] Extending the Haskell Foreign Function Interface with Concurrency
Simon Marlow, Simon Peyton Jones, Wolfgang Thaller
Proceedings of the ACM SIGPLAN workshop on Haskell, pages 57–68, Snowbird, Utah, USA, September 2004

[Marlow 2009] Runtime Support for Multicore Haskell
Simon Marlow, Simon Peyton Jones, Satnam Singh
ICFP ’09: Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, Edinburgh, Scotland, August 2009
[GHC Trac] *GHC Trac issue #7353*
https://ghc.haskell.org/trac/ghc/ticket/7353
URL accessed June 8, 2016.

[MSDN] *I/O Completion Ports*
Microsoft Developer Network.
URL accessed June 8, 2016.
A Accessing the source code

Source code for the proof of concept implementation described in this thesis is available on GitHub at https://github.com/23Skidoo/haskell-iocp/. Building the code requires a patched version of GHC, which is available at https://github.com/23Skidoo/ghc.

Follow these steps to build the source code:

- Clone the Git repository with patched GHC from https://github.com/23Skidoo/ghc.
- Switch to the ghc-8.0-max-spare-workers branch.
- Follow the instructions on GHC Wiki\(^1\) to build GHC from source.
- Make sure that you have cabal-install installed. A Windows binary can be downloaded from https://www.haskell.org/cabal/download.html. Make sure that the cabal tool is in PATH.
- Clone the haskell-iocp Git repository from https://github.com/23Skidoo/ghc.
- Run cabal sandbox init.
- Run cabal install -w c:/path/to/patched/ghc/inplace/bin/ghc-stage2.exe --only-dependencies.
- Run cabal configure -w c:/path/to/patched/ghc/inplace/bin/ghc-stage2.exe.
- Run cabal build.

This will build the I/O manager library and a number of benchmarks and test programs. To run the test programs, use cabal run. For example, cabal run pong starts the Pong HTTP server. Use ApacheBench or httpperf for performance tests. Example invocation of ApacheBench: ab -n 10000 -c 100 http://127.0.0.1:8080/. This makes ApacheBench send ten thousand requests using one hundred simultaneous connections.

Windows x64 binaries for the pong and file benchmarks are also available from https://github.com/23Skidoo/haskell-iocp/releases/tag/0.1.

\(^1\)https://ghc.haskell.org/trac/ghc/wiki/Building