Experience Report: Developing High Performance HTTP/2 Server in Haskell

Kazuhiko YAMAMOTO
Internet Initiative Japan Inc., Japan
kazu@iij.ad.jp

Abstract

While the speed of the Internet has been increasing, HTTP/1.1 has been plagued by head-of-line blocking, low concurrency and redundant headers. To solve these problems, HTTP/2 was standardized. This paper summarizes our experience implementing HTTP/2 in Haskell. We found several techniques to improve the performance of the header compression and identified a suitable data structure for HTTP/2 priority. Also, we showed that Haskell lightweight threads are useful for HTTP/2 where the common tactics of one lightweight thread per connection cannot be used. The HTTP/2 implementation of Warp, the popular HTTP server library in Haskell, ultimately provides better throughput than its HTTP/1.1 counterpart.

Categories and Subject Descriptors C.2.2 [*Computer Communication Networks*]: Network Protocols—HTTP/2; D.1.3 [*Programming Techniques*]: Concurrent Programming—Lightweight thread

Keywords HTTP/2, Header Compression, Priority, Lightweight Thread

1. Introduction

HTTP is a killer application protocol for the deployment of the Internet. The flagship version HTTP/1.1 (Fielding et al. 1997) was standardized in 1997 and has been used for around 20 years without revisions. In the years since its standardization, available bandwidth has grown by orders of magnitude, dynamically generated web content has become the norm, and diversity among client devices has increased drastically. These changes have revealed several problems with the aging protocol:

- HoL (head-of-line) blocking — Since HTTP/1.1 requests are served in FIFO order, the next response cannot be sent until the previous response is finished. If the server spends a long time processing a request, the next request can be subject to arbitrary delays.
- Low concurrency — A TCP connection can host only a single request at a time. To increase concurrency, major browsers make up to 6–8 TCP connections. For services for which this concurrency is not sufficient, content is distributed onto multiple servers. This practice, known as *domain sharding*, makes it complex to manage content.
- Redundant headers — Because HTTP/1.1 is a stateless protocol, it is necessary to convey similar headers (cookies in particular) for each request to implement state. It is said that the average size of request headers in 2015 is about 800 bytes. This wastes network bandwidth, which is the most expensive resource in browser-server communications.

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...
http://dx.doi.org/10.1145/2976002.2976006

To solve these problems, after three years of discussions, Internet Engineering Task Force (IETF) standardized HTTP/2 (Belshe et al. 2015), based on SPDY[1], in 2015. HTTP/2 was designed to maintain semantics such as headers and to achieve high performance communication by introducing a new transport layer. HTTP/2 has the following features:

- It exchanges data in *frames* asynchronously. Since frames have IDs, browsers can associate response frames with their corresponding requests. Therefore, servers are able to send responses in any order.
- It uses only one TCP connection and can multiplex multiple requests and responses, up to a configurable concurrency limit. The minimum recommended value for this limit is 100 (Belshe et al. 2015).
- It provides a header compression mechanism, called HPACK (Peon and Ruellan 2015).

Other HTTP/2 features include window-based flow control both for the connection as a whole and for each stream, server-initiated streams (*server push*), and client-specified prioritization of requests. Here *stream* is an HTTP/2 technical term which means a set of request/response frames for specific content. A stream consists of multiple frames with the same ID.

We joined the standardization process of HTTP/2 in 2013 and continue to develop HTTP/2 in Haskell (Marlow et al. 2010). As a result, we have already released the HTTP/2 library[2] which provides a frame encoder/decoder, header compression, and nested priority. The author has also developed HTTP/2 functionality based on the HTTP/2 library in Warp (Yamamoto et al. 2013) – a popular HTTP server library – and released HTTP/2 enabled Warp in July 2015.

This paper summarizes our experiences with implementing HTTP/2 in Haskell and is organized as follows: findings about HPACK and priority are described in Section 2 and Section 3, respectively. Section 4 demonstrates how to implement an HTTP/2 server using lightweight threads, and Section 5 evaluates performance.

[1] http://dev.chromium.org/spdy/

[2] http://hackage.haskell.org/package/http2

2. Header Compression

The element technologies of HPACK are as follows:

- Static table — a predefined table whose entry is either a header name or a pair of header name and header value, chosen in the specification (Peon and Ruellan 2015). Each entry can be accessed by index. Since the typical length of indices is seven bits, sending indices instead of strings saves bandwidth.
- Dynamic table — similar to the static table, but with entries dynamically registered. The dynamic table has an upper limit for its size. If the registration of a new entry causes overflow, old entries are removed. Each endpoint uses two dynamic tables for sending and receiving per connection.
- Huffman encoding — encoding more frequently used letters in header names and header values with fewer bits. The mapping is statically defined.

2.1 High Performance Implementation of HPACK

We developed HPACK with simple data structures in purely functional settings initially, but this implementation was slow. By profiling a program which encodes headers and decodes the result, we found the following two bottlenecks:

- Converting a header to an index in HPACK encoding
- Huffman decoding in HPACK decoding

For the first bottleneck, we introduced reverse indices instead of searching the static/dynamic tables directly. It is necessary to find an index from both a header name and a header name-value pair. Logically speaking, we can make use of a finite map of finite maps where the outer keys are header names and the inner keys are header values.

The cause of the second bottleneck is bit by bit transition in the Huffman binary tree. To improve performance, we adopted a method to calculate transition destinations by n bits basis in advance (Pajarola 2003). Logically, this converts the Huffman binary tree to an 2^n-way tree. As the encoded headers always have byte boundary with padding, 2, 4 and 8 are reasonable candidates for n. If n gets larger, the performance gets better, but more memory is necessary. We chose 8 for the best performance. Soon thereafter, this technique was adopted by nghttp2[3], the de facto reference implementation of HTTP/2, followed by other HTTP/2 implementations.

Even with these improvements, performance was not satisfactory. Therefore, we gave up the purely functional implementation and switched to direct buffer manipulation. One technique which we found is copy avoidance. The format of Huffman encoded headers are length-value. The byte count of an integer is variable, and the length of Huffman encoded headers is not known in advance. So, a naive implementation encodes a header in a temporary buffer, obtains the length of the result, encodes the length in the target buffer, and copies the result from the temporary buffer. We found that this copy is avoidable by guessing the length of the result. In our experiments, Huffman encoding can compress by 20 percent on average for sample data provided from IETF. Thus, we can estimate the length of the result using a factor of 0.8. If the estimated length occupies the same number of bytes as the true length, no copy is necessary. Otherwise, the result must be moved within the target buffer. No temporary buffer is needed. This technique was also inherited by other HPACK implementations.

The finite map of finite maps for reverse indices above was not sufficient. We were suggested to introduce *tokens* for header names to improve the performance of reverse indices (Oku). The tokens are an enumeration whose members are organized by the header names defined in the static table. A token holds an index to reverse index arrays and other useful information statically defined. Since header names used in HPACK are lower-case only, one string comparison is necessary to convert a header name to a token:

```
toToken :: ByteString -> Token
toToken bs = case len of
    3 -> case lst of
        97  | bs == "via" -> tokenVia
        101 | bs == "age" -> tokenAge
        _                 -> makeTokenOther bs
    4 -> case lst of
        101 | bs == "date" -> tokenDate
        103 | bs == "etag" -> tokenEtag
        ...
        _                  -> makeTokenOther bs
    ...
    _  -> makeTokenOther bs
  where
    len = Data.ByteString.length bs
    lst = Data.ByteString.last bs
```

With these tokens, we introduced the three reverse indices:

A An array of finite maps for the static table. The array is accessed with the tokens, and the finite maps are accessed with header values. Indices for header name-value pairs and header names can be resolved. Note that most finite maps in this reverse index are empty because header values are not defined for most header names in the static table.

B An array of *references* to finite maps for dynamic table. The array is accessed with the tokens, and the finite maps are accessed with header values. Indices for header name-value pairs whose header names are defined in the static table can be resolved.

C A reference to finite maps for dynamic table. The finite maps are accessed with header name-value pairs. Indices for header name-value pairs whose header names are not defined in the static table can be resolved.

The look-up algorithm is as follows: for a header name defined in the static table, reverse index B is queried first. If an index for the name-value pair is found, it is returned. Otherwise, reverse index A is queried to obtain an index for the header name or the name-value pair. For a header name not defined in the static table, reverse index C is queried to obtain an index for the name-value pair. While two lookups in $\mathcal{O}(\log n)$ are necessary for the original scheme, essentially one lookup is used for most cases for this new scheme.

With the above improvements and some others, the performance became satisfactory. Figure 1 shows the performance of the initial implementation (1), the middle one with reverse indices/2^n way trees (2) and the final one (3) for encoding (Enc) and decoding (Dec) with 646 sets of headers provided by IETF. For measurement, we used Xeon E5-2650Lv2 (1.70GHz/10 core/25MB) without hyper threading x 2 with 64G memory for hardware, CentOS 7.2 for OS, and GHC (Marlow and Peyton Jones 2012) 7.10.3 for Haskell compiler. The benchmark framework is `criterion`. To stabilize benchmark results, CPUs are set to the performance mode.

3. Priority

HTTP/2 multiplexes multiple streams in a TCP connection. It is undesirable for content needed immediately for rendering to be delayed while other less-crucial content occupies the connection bandwidth. Thus, HTTP/2 provides a way of prioritizing individual streams. Priority for a stream consists of a weight value (from 1 to 256) and a stream ID of dependency. The larger the weight, the greater the preference. A set of priorities forms a dependency tree (Figure 2).

[3] https://nghttp2.org/

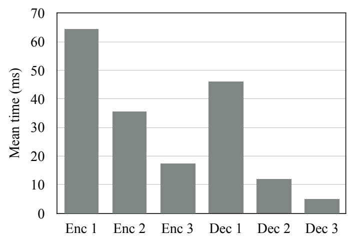

Figure 1. The performance progression of HPACK encoding and decoding. The smaller, the better.

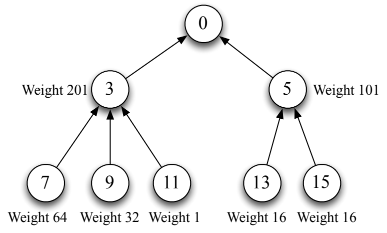

Figure 2. An example of HTTP/2 priority tree. A circle and its number indicate a stream and a stream ID, respectively. ID 0 is a special stream ID to represent a root. For instance stream 3 and 5 depend on stream 0 and they share the resource of stream 0 proportionally to weight 201 and 101, respectively. Stream 7, 9, 11 have stream 3 as their parent and they share the resource of stream 3 proportionally to weight 64, 32 and 1, respectively.

Streams with the same parent share resources proportionally based on their weight. Implementors can define what a resource is. Though few HTTP/2 servers implement priority, those that do use either the byte count of sent data or the number of sent frames.

In this paper, we use the technical term *HTTP/2 priority queue* to refer to data structures which implement a priority tree of one parent with one or more children. General priority trees, whose height is more than 2, can be implemented based on HTTP/2 priority queues.

3.1 HTTP/2 Priority Queues

If we implement HTTP/2 priority queues with *max heaps* whose precedence is weight, proportion is achieved but fairness is not guaranteed. For example, consider stream A with weight 200 and stream B with weight 100. Each corresponding content is logically divided into fragments fit to a send buffer. Suppose the first fragment of stream A and that of stream B are enqueued. In this case, the fragment of stream A is dequeued and sent first. Suppose that the next fragment of stream A is enqueued with weight 199. This time, the fragment of stream A will again be dequeued. In this scheme, the fragment B is not dequeued until 100 fragments of stream A are dequeued. What we want to implement is a sequence that is proportional and fair to weight such as A, A, B, A, A, B, A, A, B and so on.

3.2 Random Skew Heap

As the first attempt, we implemented fairness by pseudo-random number generation. That is, an entry to be dequeued is selected by generating a random number with total weight as maximum. To implement enqueueing and dequeueing in $\mathcal{O}(\log n)$, we used skew heaps (Sleator and Tarjan 1986) as the base data structure. The core operation of skew heaps is merge. An entry can be inserted by merging the singleton of the entry and the heap. Extracting the maximum entry can be done by removing the root and merging the two sub heaps. We introduced randomness to the merge operation. The following describes the core algorithm in Haskell[4]:

```
type Weight = Int
data Heap a = Tip
            | Bin Weight -- total weight
                  a      -- element
                  Weight -- element weight
                  !(Heap a) !(Heap a)

merge :: Heap t -> Heap t -> Heap t
merge t Tip = t
merge Tip t = t
merge l@(Bin tw1 x1 w1 ll lr) r@(Bin tw2 x2 w2 rl rr)
  | g <= tw1  = Bin tw x1 w1 lr $ merge ll r
  | otherwise = Bin tw x2 w2 rr $ merge rl l
  where
    tw = tw1 + tw2 -- total weight of two trees
    -- pseudo random
    g = unsafePerformIO $ uniformR (1,tw) gen
```

Using this code on the Internet, we noticed that the deletion operation is also necessary. For instance, let's suppose that a user is browsing with multiple tabs. Unselected tabs are given low weights. When the user selects one of the unselected tabs, its weight increases. The new weight is sent from the browser to the server. To reflect this change quickly on the server side, the old entry of the newly selected tab should be deleted and enqueued again with the new, higher weight. A skew heap does not provide the delete operation. So, for deletion, we need to re-construct the entire heap, resulting in complexity of $\mathcal{O}(n \log n)$.

3.3 Weighted Fair Queueing

The second attempt was *weighted fair queueing* (WFQ) (A.Demers et al. 1989). WFQ is a queue which is proportional and fair with respect to weights. It can be implemented by *min heaps* with inverted weights as precedence. The smaller the precedence, the greater the preference. In addition to the entries, a WFQ holds the minimum precedence among them as the current precedence. When a new entry is enqueued, its precedence is calculated by adding the inverted value of its weight (multiplied by a certain constant to round up to an unsigned integer) to the current precedence. In the case where a dequeued entry is enqueued again, the new precedence is calculated by adding the inverted value of its weight to its previous precedence.

Since precedence values grow linearly, they can overflow. It is not necessary to worry about this overflow if precedence is defined as an unsigned integer (Oku). All precedence values are in a certain range and arithmetic operations of unsigned integers is based on modular arithmetic. So, the magnitude relationship holds even in overflow cases if a proper comparison is used. The following is a comparison example with a constant value 65536:

```
newtype Precedence = Precedence Word deriving Eq
instance Ord Precedence where
  Precedence w1 <  Precedence w2 = w1 /= w2
                               && w2 - w1 <= 65536
  Precedence w1 <= Precedence w2 = w2 - w1 <= 65536
```

3.4 Comparing Data Structures

To study suitable data structures for HTTP/2 priority queues, we implemented the following data structures in Haskell.

- Random skew heap

[4] `unsafePerformIO` is used just because of simple implementation and cloud be removed by passing a list of random values.

- WFQ with a bootstrapped skew binomial heap (Brodal and Okasaki 1996)
- WFQ with a priority search queue (PSQ) (Hinze 2001)

Table 1. Complexity on priority queue operations in worst-case complexity

	enqueueing	dequeueing	deletion
Random skew heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$
Skew binomial heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
PSQ	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Table 1 shows complexity of enqueueing, dequeueing and deletion for the three implementations. With the benchmark environment above, we measured the performance as follows:

- Preparing an HTTP/2 priority queue with 100 entries generated by a pseudo random generator, and repeating dequeueing-and-enqueueing 10,000 times (the left hand side of Figure 3).
- Preparing an HTTP/2 priority queue as mentioned above and removing all entries (the right hand side of Figure 3).

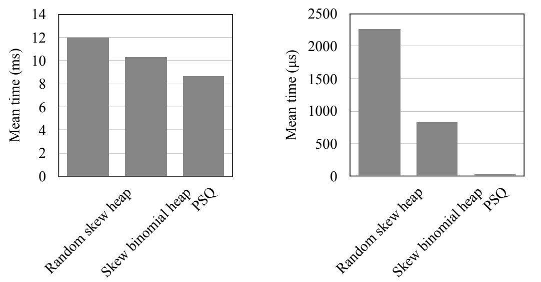

Figure 3. Performance of enqueueing/dequeueing (left) and deletion (right). The smaller, the better.

According to these figures, PSQ provides high performance. PSQ has the heap characteristic, and the complexity of its enqueueing and dequeueing operations is $\mathcal{O}(\log n)$. It also has the search tree characteristic, and the deletion operation is in $\mathcal{O}(\log n)$. PSQ in purely functional settings can be combined with STM to create a deadlock-free data structure in concurrent environments (Marlow 2013). Note that all control frames share the stream ID 0; multiple control frames cannot be enqueued into PSQ. So, it is necessary to prepare another simple queue for control frames and to process them preferentially.

Based on these considerations, we changed our implementation of HTTP/2 priority queues from random skew heaps to WFQ with PSQ.

4. HTTP/2 Implementation in Warp

This section discusses how to implement HTTP/2 servers with Haskell's lightweight threading model. The author, one of the main developers of Warp, has developed and maintained HTTP/2 functionality in Warp as described below.

The common tactics for implementing servers for a simple application protocol such as HTTP/1.1 in Haskell is to use *one lightweight thread per connection* (Marlow 2002). Unlike event-driven programming, this allows us to write HTTP/1.1 code in a straightforward manner. A lightweight thread in Warp's HTTP/1.1 implementation repeatedly receives an HTTP request, parses the request to produce a request value, passes it to a web application to get a response value, composes it into an HTTP response, and sends it.

Our question was whether lightweight threads were useful to implement servers for HTTP/2, which includes its own transport layer. Since an HTTP/2 server multiplexes contents, a sender thread is needed to multiplex the application's streams onto the socket. The sender repeatedly dequeues a response value from an nested HTTP/2 priority queue (called output queue), compresses its headers, encodes its data to frames until the buffer is filled or a flow-control window is exhausted, sends the frames, and if data remains to be sent, re-enqueues the response on the output queue. In a symmetric fashion, we also prepare a receiver. The receiver repeatedly decodes received frames, uncompresses headers, produces a request value, and enqueues it to an input queue. Figure 4 illustrates this skeleton.

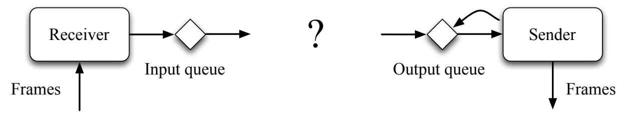

Figure 4. The skeleton of HTTP/2 implementation in Warp. Rounded-edged rectangles indicate lightweight threads.

One challenge is the organization of the components between the input queue and the output queue. Our method is *one lightweight thread per stream*. We call this lightweight thread a *worker*. The role of workers is to dequeue a request value from the input queue, pass it to a web application, and enqueue the application's response value onto the output queue.

We found that throughput is not satisfactory if a worker is spawned on demand. So, we introduced a worker pool where workers are created in advance. If many workers are prepared, many context switches and high contention against the two queues occur. On the other hand, HoL is not avoidable if there is only one worker. Through a trial and error process by measuring memory footprint and throughput, we take 3 as the number of workers currently.

We also observed that using a bounded queue as the input queue results in increased memory footprint. Thus, we adopted a simple queue for the input queue. Since the only component enqueueing to the input queue is the receiver, flow rate can be controlled to some extent. That is, when the receiver enqueues a certain number of requests to the input queue, it requests a context-switch to the scheduler of lightweight threads.

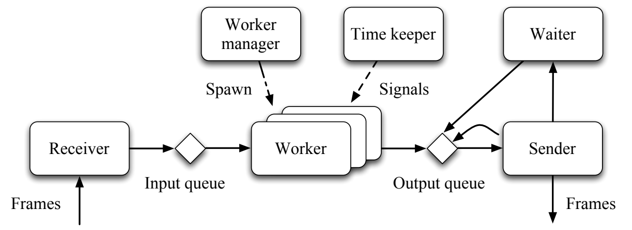

Figure 5. The HTTP/2 architecture in Warp. Rounded-edged rectangles indicate lightweight threads.

Figure 5 illustrates the current HTTP/2 architecture in Warp. Components not described yet are as follows:

- Worker manager — Spawning workers at boot time and when requested.
- Time keeper — If a worker does not finish its procedure, the time keeper sends a signal to the worker so that it can move to another request.

- Waiter — This thread is spawned when a response value is not ready to be sent (e.g. due to per-stream flow control). It waits for the response to be sendable, enqueues it onto the output queue, and goes away.

4.1 Optimistic Enqueueing

Warp is the primary handler for the Web Application Interface (WAI)[5]. This API defines the following three response types:

- File — a static file
- Builder — in-memory content
- Streaming — data flow continuously generated by a web application

For the file and builder types, when a worker receives a response value, the corresponding web application has already finished. So, after enqueueing the response value to the output queue, the worker can serve another request.

Initially, we adopted pessimistic enqueueing. In this scheme, the worker checks the stream window before enqueueing the response onto the output queue. If the flow control window is closed, the worker instead spawns a temporary waiter to enqueue the response once the window becomes open.

For streaming responses, an application continues to work even after returning a response value. The corresponding worker cannot move on to serve another request until the streaming completes. This could result in no workers being available to dequeue new requests, if all workers are occupied by streaming responses. Thus, the worker asks the worker manager to spawn another worker before running the stream. When the streaming is finished, the worker exits.

A worker serving a streaming response creates a queue for response fragments and acts as a bridge between the application and this stream queue. A response value containing a reference to the stream queue is enqueued onto the output queue. This enqueueing was also initially pessimistic. That is, to ensure that the output queue could not contain empty stream queues, a dedicated waiter would wait until data was available on the stream queue before placing it onto the output queue. The sender would dequeue this response value (containing the stream queue), send all or part of it over the connection, and enqueue it to the output queue again if necessary. So, the sender needs to synchronize with the waiter in this case through a communication channel.

As it appeared that pessimistic enqueueing resulted in poor throughput, optimistic enqueueing is applied currently. That is, a worker enqueues a response value to the output queue without checking the stream window or the stream queue. After dequeueing a stream response value, the sender checks that the stream window is open and there is data available on the stream queue. If either condition is not met, the sender spawns a temporary waiter, which waits until data can be sent on the stream, then re-enqueues the response value to the output queue and exits. Optimistic enqueueing simplifies the architecture and improves throughput.

4.2 Packing Output Frames

The HTTP/1.1 implementation has a 16 KiB send buffer per connection. For the builder type and the streaming type, the content is repeatedly sent using the send buffer until the entire content is transmitted.

For the file type, a file is sent without copying it between the kernel space and the user space thanks to the `sendfile` system call on UNIX. When a file is opened, a file descriptor is created with the global lock of the file descriptor list in the kernel. Hence, if a file is opened everytime when a file response is sent, throughput becomes poor. To improve performance, Warp caches and reuses file descriptors when appropriately configured. The `sendfile` system call is designed to preserve the offset of a file descriptor in the kernel. For this, the system takes an offset value and a length value as arguments. Thanks to this clean API, multiple lightweight threads can share file descriptors.

The sender of the HTTP/2 implementation fills the 16 KiB send buffer with a fragment of stream data only once per entry retrieved from the output queue, to implement priority at a granularity of 16 KiB buffers. If the entire content cannot be sent in one fragment, a response value representing the remainder of the response is re-enqueued, so that it will be delayed appropriately according to its priority. If the send buffer still has room after a response value is completed, the sender tries to fill the send buffer with other fragments. In this scheme, the `sendfile` system call cannot be used for the file type. Thus, we adopt the `pread` system call, which, like the `sendfile` system call, also preserves offsets in the kernel.

The following is the algorithm for dequeueing from the control queue and the output queue based on STM:

- If the control queue is not empty, dequeue control frames.
- If the control queue is empty:
  - Wait until the connection window is open.
  - If the output queue is not empty, dequeue a response value.
  - If the output queue is empty:
    - If the send buffer is empty, retry this procedure.
    - If the send buffer is not empty, give the flush instruction.

The connection window is consumed only by responses which have bodies. Responses without bodies should be sent even if the connection window is closed. The algorithm above cannot cover this case, but we consider this case to be rare.

5. Evaluation

To evaluate the performance of Warp's HTTP/2 implementation, we measured the throughput of the following three servers:

- Warp version 3.2.3 with a simple web application which responds to all requests with the same in-memory 612-byte HTML. This software shows the maximum performance of Warp.
- Mighty[6] version 3.3.0 — a practical HTTP server to provide static files, CGI and reverse proxies implemented as a web application on Warp version 3.2.3
- nginx 1.9.9[7] implemented in event driven programming in C

We carefully configured Mighty and nginx to behave as similarly as possible, such as reusing file descriptors, no recording logs, etc. Our benchmark environment is two computers as described above directly connected via a pair of 10 Gigabit Ethernet. As a benchmark tool, we used `h2load` in nghttp2, which can measure throughput for both HTTP/1.1 and HTTP/2.

For HTTP/2, 100 connections were created with 100 as the maximum value of concurrency for each, and an HTTP file of 612 bytes (nginx's default index.html) was downloaded 1,000,000 times in total. For HTTP/1.1, 600 connections were used to increase the HTTP/1.1 concurrency to 6 and the same file was downloaded the same number of times. The numbers of used cores were 1, 2, 4, 8 and 16. For each configuration, we took the median of 5 measurements. The result is shown in Figure 6.

[5] http://hackage.haskell.org/package/wai

[6] http://hackage.haskell.org/package/mighttpd2

[7] http://nginx.org/

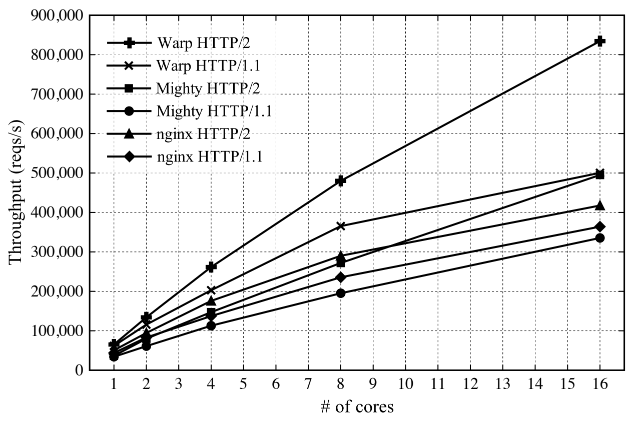

Figure 6. Throughput of Warp (in-memory), Mighty and nginx for both HTTP/1.1 and HTTP/2. The larger, the better.

As far as this benchmark is concerned, the HTTP/2 implementation achieves better performance than the HTTP/1 implementation for both Warp and Mighty. It is difficult to compare different HTTP servers fairly since each server implements a different set of features. Though nginx shows better performance than Mighty on small numbers of cores, its specification conformance is lacking: while nginx passes only 37 of the 71 test cases provided by h2spec[8], Mighty passes all 71. The performance gap between the two servers may be partially attributed to their differing levels of conformance to the specification: the additional logic needed for each case is likely to come with a performance cost.

One obstacle to high performance network servers in Haskell relates to file paths. A file path (`FilePath`) is defined as a list of Unicode characters (`[Char]`) in Haskell. So, file path operations, such as comparing, concatenating and converting to other data structures, are inefficient. We hope that file paths will be defined with a proper data structure in the future.

Next we took heap profiles of Warp to investigate memory footprint using `h2load` three times with the same parameters above on 16 cores. Figure 7 and 8 shows heap profiles for HTTP/1.1 and HTTP/2, respectively. As described above, seven lightweight threads are spawned per HTTP/2 connection while one lightweight thread is used per HTTP/1.1 connection. Almost the same amount of memory is used. HTTP/2 uses more memory for data other than stacks. Note that GHC allocates stacks of lightweight threads in the heap area. Considering the dynamic tables of HPACK and so on, this is reasonable.

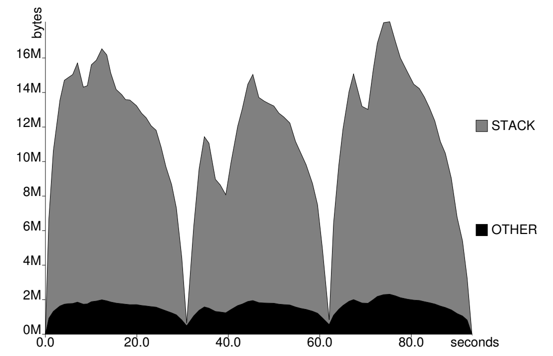

Figure 7. Heap profile of Warp handling HTTP/1.1

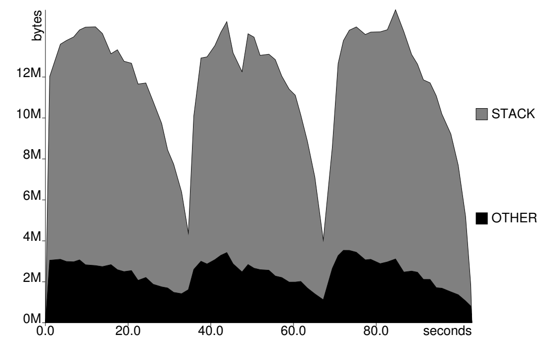

Figure 8. Heap profile of Warp handling HTTP/2

Acknowledgments

We deeply thank Moto Ishizawa, Tatsuhiro Tsujikawa and Kazuho Oku. Without their insight and discussions with them, this research would be impossible. We would like to extend our appreciation to Michael Snoyman and Andrew Pritchard for their thorough proofreading of this paper. We sincerely thank anonymous referees for their useful comments to improve draft versions of this paper. Lastly, we would express gratitude for the HTTP/2 community in Japan.

References

A.Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review*, 19, 1989.

M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (http/2), 2015. RFC7540.

G. Brodal and C. Okasaki. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, 6:839–857, 1996.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1, 1997. RFC2068, RFC2616 and RFC7230–7235.

R. Hinze. A Simple Implementation Technique for Priority Search Queues. In *Proceedings of ICFP*, 2001.

S. Marlow. Developing a high-performance web server in Concurrent Haskell. *Journal of Functional Programming*, 12(4+5):359–374, 2002.

S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. In *the Architecture of Open Source Applications*, volume 2. 2012. http://www.aosabook.org/en/ghc.html.

S. Marlow et al. *Haskell 2010 Language Report*, 2010.

K. Oku. The auther of h2o, DeNA Co., Ltd. Private communication.

R. Pajarola. Fast Prefix Code Processing. In *Proceedings IEEE ITCC Conference*, 2003.

R. Peon and H. Ruellan. HPACK: Header Compression for HTTP/2, 2015. RFC7541.

D. D. Sleator and R. E. Tarjan. Self-Adjusting Heaps. *SIAM Journal on Computing*, 15:52–69, 1986.

K. Yamamoto, M. Snoyman, and A. Voellmy. Warp. In *The Performance of Open Source Applications*. 2013. http://www.aosabook.org/en/posa/warp.html.

[8] https://github.com/summerwind/h2spec