Web Server in Rust

04/25/2016

Team Members

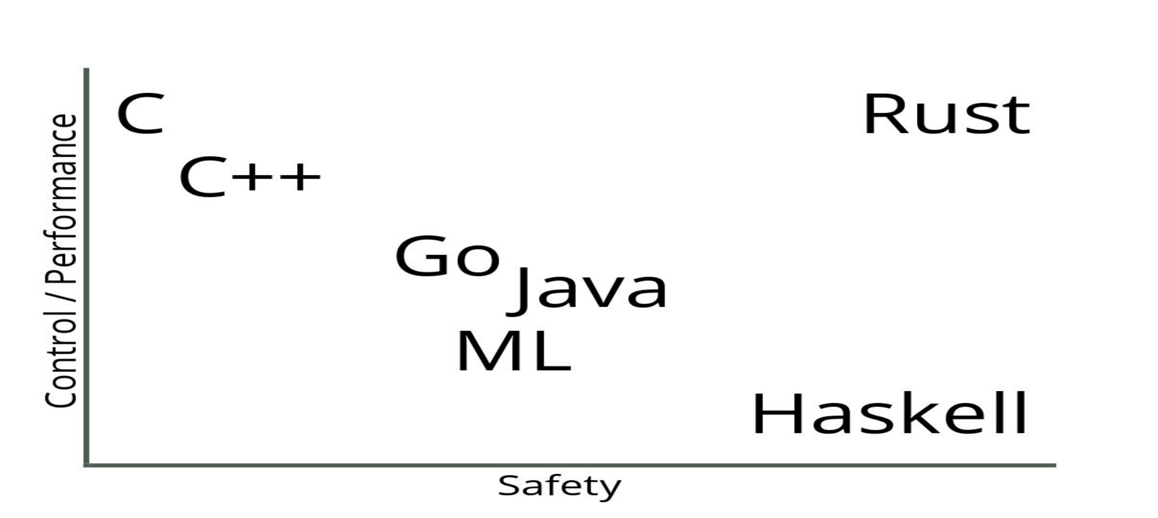
Arulselvan Madhavan

Jimmy Ly

Agenda

- Introduction
- Why Rust?
- Basic web server design
- Initial thread pool design
- Improved thread pool design
- Web server cache
- Logging
- Demo
- Web server performance
- Final thoughts/next steps

Why Rust?

- Rust is a systems programming language focused on three goals: safety, speed, and concurrency.
- It maintains these goals without having a garbage collector.
- Key Feature: Rust's Ownership Model
- 

Basic Web Server Tasks

Parallelizable Tasks

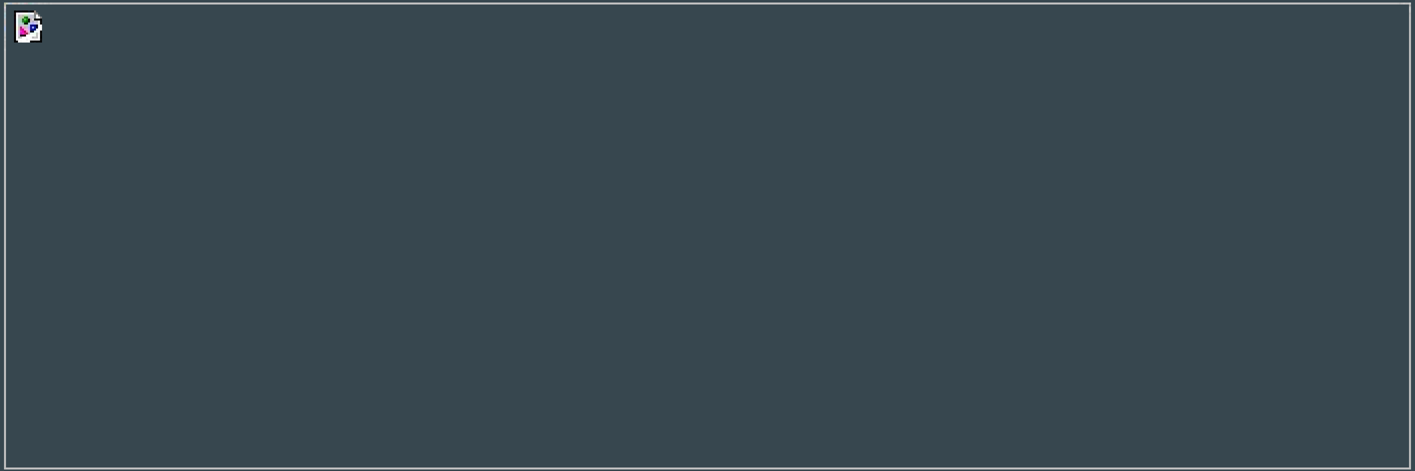
Thread Pool Design - 1

- Spawn a new thread for each incoming request.
- Advantages:
  - No need to worry about data sharing and safe concurrency.
  - All the data required by a thread is self-contained.
- Disadvantages:
  - Not scalable.

Thread Pool Design - 2

Thread Pool based design - 2

- Challenge: Requests should be safely shared among the threads in the threadpool.
- Rust provides channels based concurrency support.



Channel based Concurrency

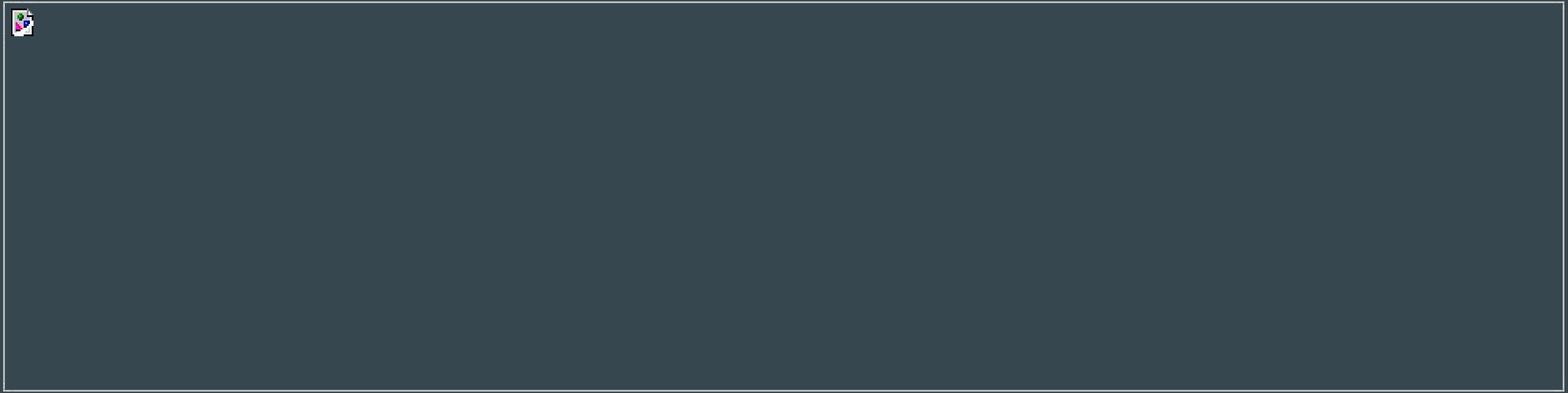
- When a channel is created, a transmitter and receiver is created in the memory.
- There can be 1 transmitter and multiple receivers or multiple transmitters and 1 receiver.
- By default, there can be only 1 owner for any data in memory.
- Rust allows multiple owners to the receiver end by using Atomic Reference Counting.
- Each thread gets an atomic reference count to the receiver end.

Thread Pool Design - 2

- Solved the scalability problem with design 1.
- Problem: Cannot enforce any priority over the jobs that needs to be served.

Thread Pool Design - 2

- Solved the scalability problem with design 1.
- Problem: Cannot enforce any priority over the jobs that needs to be served.



Thread Pool Design - 3

- Solution: Add an additional layer of indirection.

Web Server Cache

- Motivation: File IO is a costly operation and some files are served so often(ex: 404 page)
- Cache file contents in web server for files that are smaller than some threshold (ie 50kb)
- Associative cache using rust-concurrent-hashmap library
- Keys are path to file, values are file contents

Web Server Cache

Web server Cache - Design 2

- Issue: Each thread needs to acquire lock to update cache
- Solution: Create cache thread such that only cache thread updates cache. Worker threads send cache request through channel

Web Server Cache - Design 2

Logger thread

- Issue: All threads should write to the same log file.
- Create a separate thread for logging.
- Each log file write requests is routed to the logger threads.
- Logger thread own the file and write to the file

Logger Thread

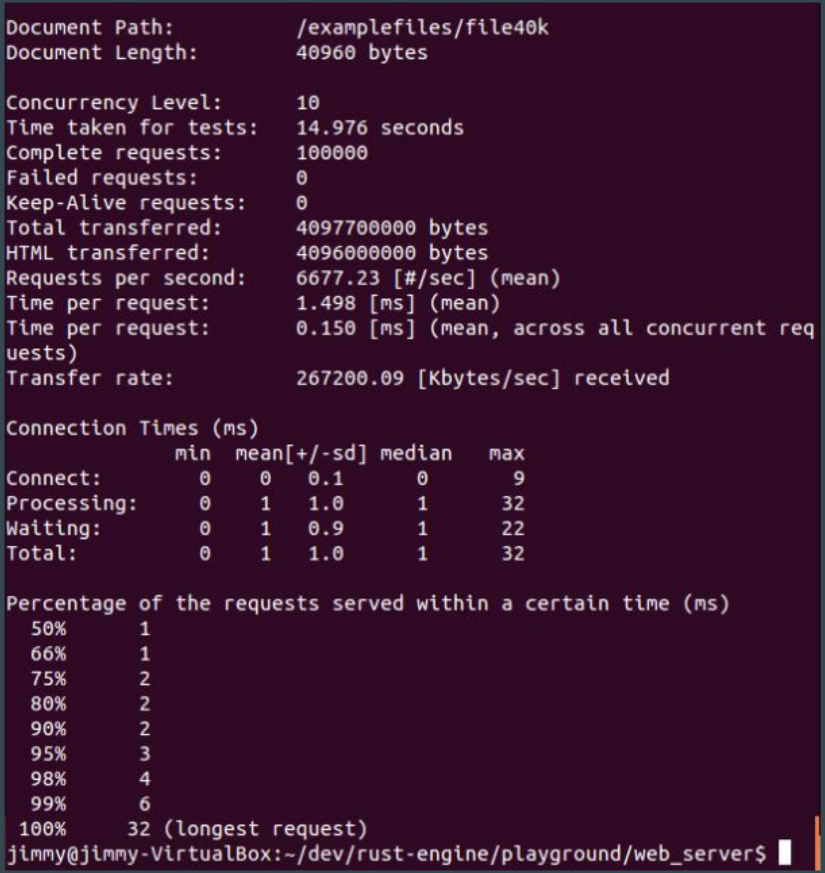
Experience with Programming in Rust

- Positive experience
- Steep learning curve
- Not worried about breaking old features when adding new ones
- Runtime errors (panicking) are still possible, if we choose to be careless.
- Many functions return a Result type(An enum of Ok(value) and Err(err)).
- Handle Results with match statements.

Auto Restart of threads

- We have several threadpools in our web server architecture.
- Each thread is provided with the capability to restart on its own, in the event of a panic(Seg fault).

Demo

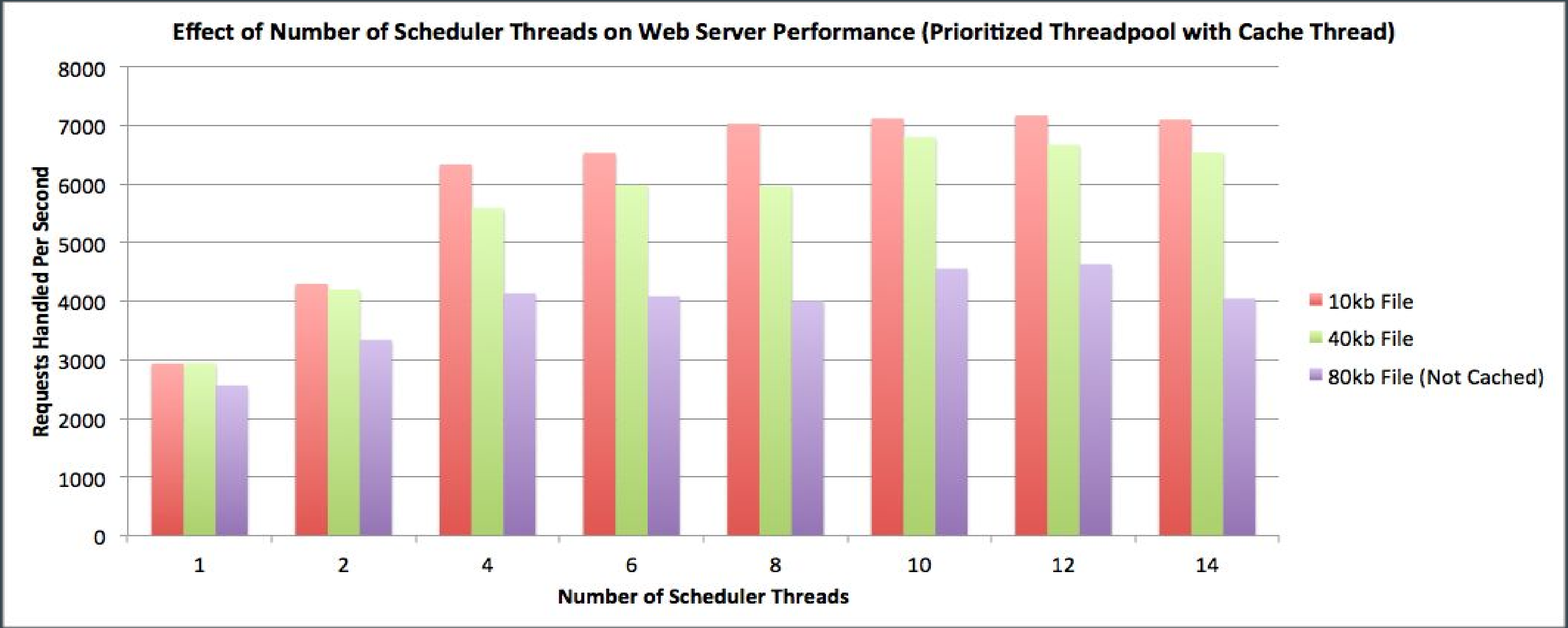


```
Document Path:          /examplefiles/file40k
Document Length:        40960 bytes

Concurrency Level:      10
Time taken for tests:   14.976 seconds
Complete requests:      100000
Failed requests:        0
Keep-Alive requests:    0
Total transferred:      4097700000 bytes
HTML transferred:       4096000000 bytes
Requests per second:    6677.23 [#/sec] (mean)
Time per request:       1.498 [ms] (mean)
Time per request:       0.150 [ms] (mean, across all concurrent req
uests)
Transfer rate:          267200.09 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       9
Processing:     0    1   1.0      1      32
Waiting:        0    1   0.9      1      22
Total:          0    1   1.0      1      32

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      2
  80%      2
  90%      2
  95%      3
  98%      4
  99%      6
 100%     32 (longest request)
jimmy@jimmy-VirtualBox:~/dev/rust-engine/playground/web_server$ █
```

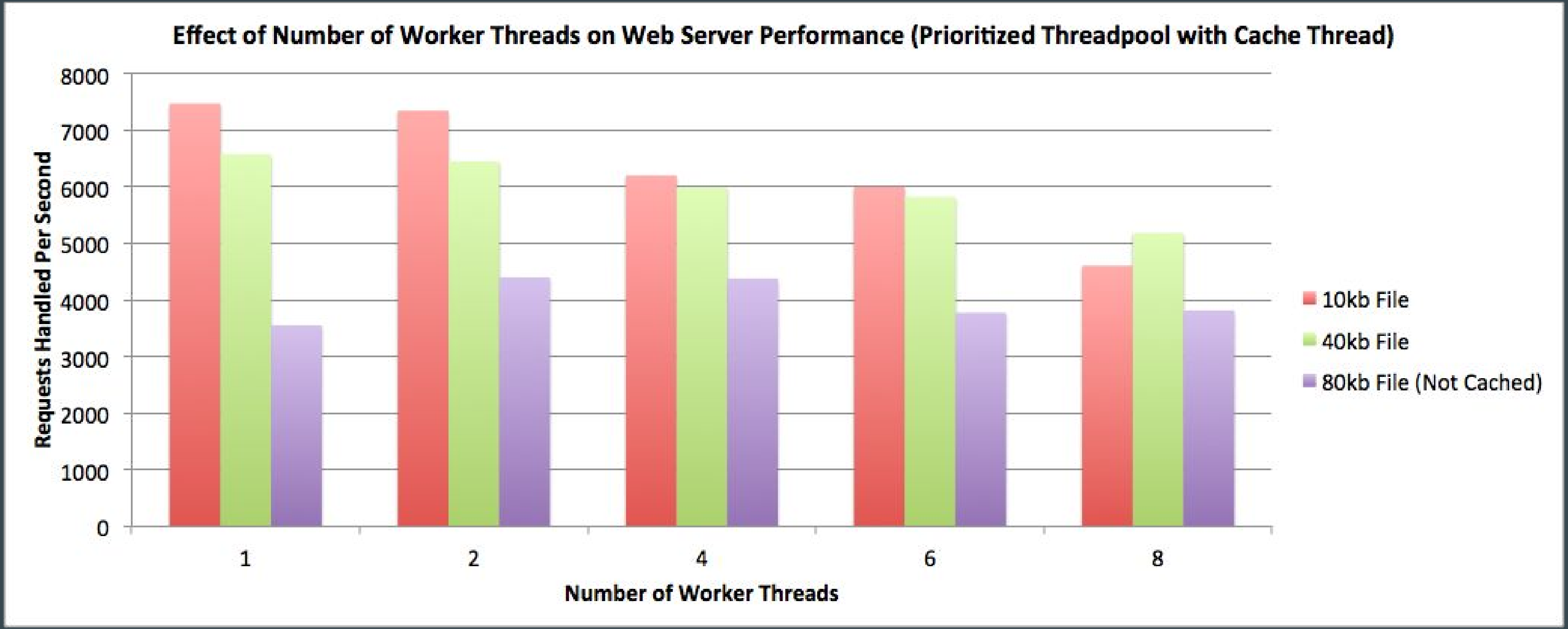
Web Server Performance

- Used ApacheBench tool to perform benchmarking on the different versions of the server.
- Different combinations of scheduler thread and worker thread counts were observed, as well as a comparison between this Rust web server and the Apache web server
- 10 concurrent connections making 100,000 total requests serving either a 10kb file, 40kb file, or 80kb file (not cached).

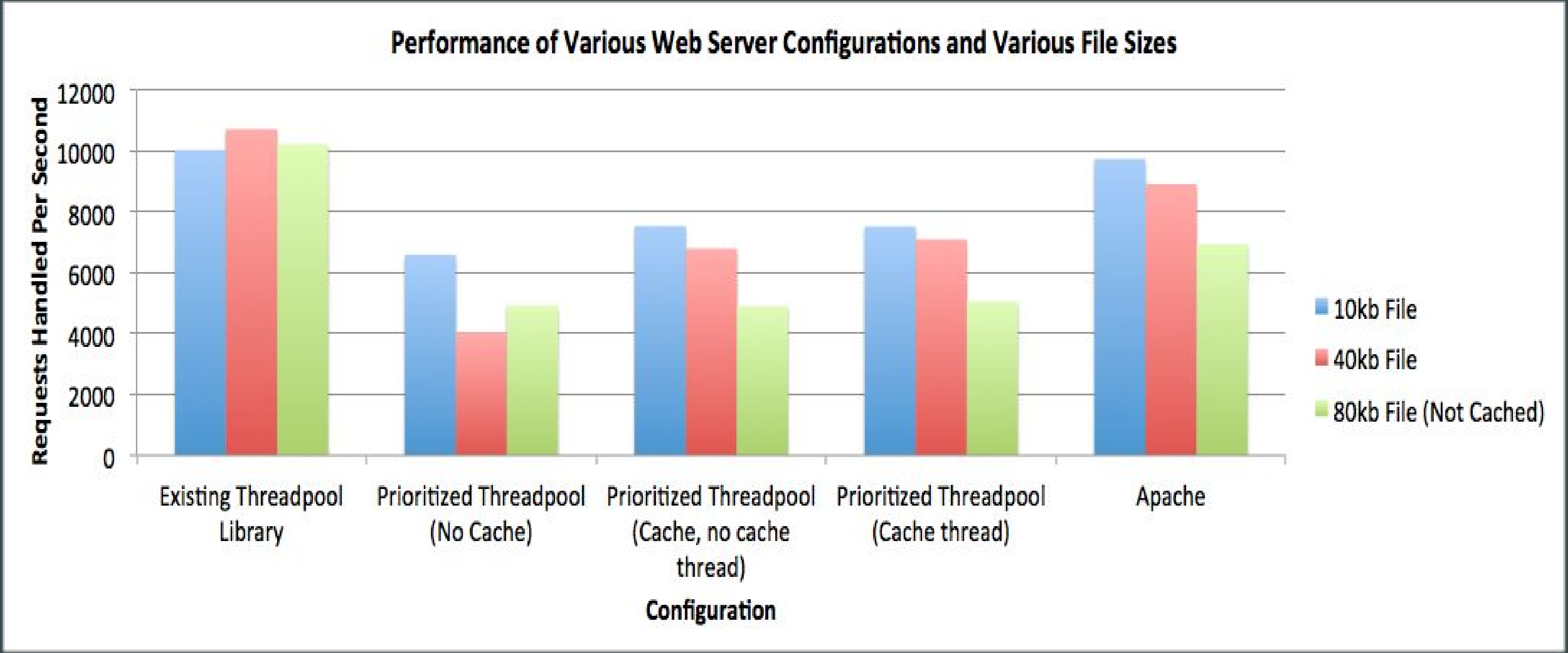
Web Server Performance



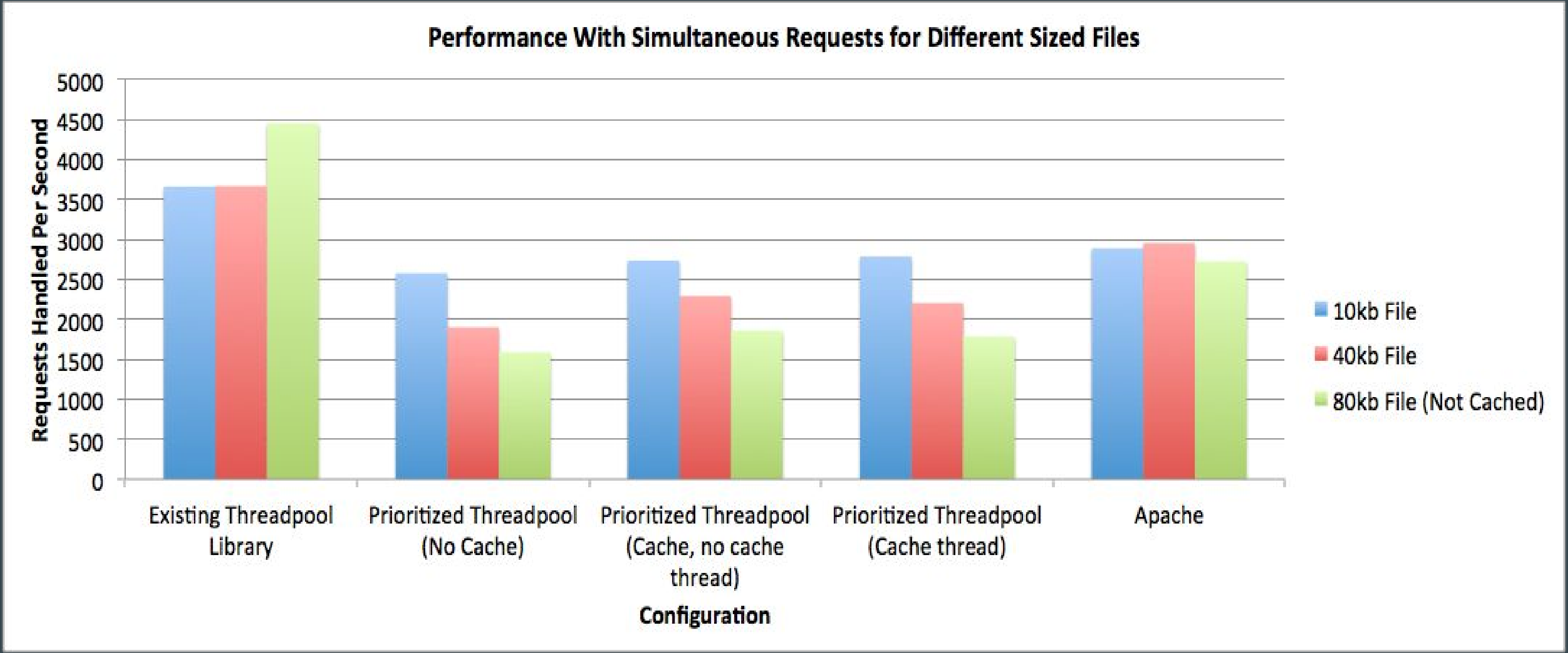
Web Server Performance



Web Server Performance



Web Server Performance



Web Server Performance

	Existing Threadpool Library	**Prioritized Threadpool (No Cache)**	**Prioritized Threadpool (Cache, no cache thread)**	**Prioritized Threadpool (Cache thread)**	**Apache**
10kb File	3659.71 requests/sec **9.108 sec total**	2575.21 requests/sec **12.944 sec total**	2734.61 requests/sec **12.189 sec total**	2785.85 requests/sec **11.965 sec total**	2885.02 requests/sec **11.554 sec total**
40kb File	3665.25 requests/sec **9.094 sec total**	1899.65 requests/sec **17.547 sec total**	2289.95 requests/sec **14.556 sec total**	2205.26 requests/sec **15.115 sec total**	2950.97 requests/sec **11.296 sec total**
80kb File (not cached)	4447.61 requests/sec **7.495 sec total**	1585.92 requests/sec **21.019 sec total**	1856.65 requests/sec **17.954 sec total**	1783.35 requests/sec **18.692 sec total**	2722.04 requests/sec **12.246 sec total**

Final Thoughts/Next Steps

- Further tune the performance of the prioritized thread pool
- Eviction policy on cache
- Update file contents in cache if one of the files is modified
- Support for HTTP/2, IPV6, WebSockets, Directory Browsing, Virtual Hosts

Thank You