Load Testing Report: Server and Function Performance

1. Aim/Purpose of the Experiment

The primary aim of this experiment was to evaluate the performance of a custom web server that mimics some basic Flask-like functionality. Specifically, we tested the server's capacity to handle concurrent HTTP requests for three different functions:

- **Square Function** (`/square?num=5`): Computes the square of a number.
- **Fibonacci Function** (`/arithmetic/fibonacci?k=10`): Computes the k-th Fibonacci number.
- **Index Page** (`/`): Returns a simple "Hello World" response.

The objective was to measure two key performance metrics:

- **Throughput**: The number of requests per second the server can handle.
- **Concurrency**: The capacity of the server to process concurrent requests.

The experiment aimed to analyze how these metrics change with increasing request concurrency and different server thread pool sizes.

2. Setup and Execution Details

- **Environment**: The server and client were hosted on the same physical machine to simulate load under limited hardware conditions. The server used CivetWeb, and functions were implemented in C.
- **Tools**:
    - **Apache Benchmark (ab)** was used as the load generator. The command parameters included:
        - `-n 9000`: Total number of requests.
        - `-c [variable]`: The concurrency level, which ranged from 100 to 3000 in increments of 250.
    - **Script**: The provided Bash script automated the load testing and recorded the results.
- **Independent Variable**: The **concurrency level** (`-c`) was increased from `100` to `3000` to understand its impact on performance.
- **Dependent Variables**:
    - **Requests per Second (Throughput)**
    - **RequestRate (Concurrency)**

- **Thread Pool Settings**: The server was run with thread pool sizes of 8, 32, 64, and 128 to evaluate the impact of threading.
- **Output**: Metrics were recorded in CSV files for plotting and analysis.

3. Hypothesis/Expectation

- **Throughput**: Throughput is expected to **increase** with the request rate initially, as more threads and higher concurrency can handle more requests concurrently. However, throughput will **plateau** or even decrease as the server's capacity reaches its limit and starts experiencing resource contention.

4. Observations from the Data/Plots

- **Throughput vs. Concurrency Level**:
    - For **low concurrency levels** (100-500), throughput increased proportionally for all three functions tested. The server was able to keep up with the incoming request rate, demonstrating the benefit of parallel processing.
    - For the **Square Function**, throughput began to **plateau around a concurrency level of 1500**, regardless of the thread pool size. The plateau occurred because the server reached its maximum capacity to process requests.
    - The **Fibonacci Function** showed a **lower peak throughput** compared to the other functions due to its computational intensity, which resulted in longer processing times.
    - The **Index Page** generally had the highest throughput, as expected, since it involved minimal computation and responded quickly even at high concurrency levels.
- **Throughput Graphs**:
    - The graphs for the **Square Function**, **Fibonacci Function**, and **Index Page** (as seen in the provided images) indicate significant variability in throughput across different concurrency levels and thread pool sizes.
    - For the **Square Function**: The throughput peaked around 19,000 requests/sec for a thread pool size of 64 and started decreasing with higher concurrency levels due to resource contention.
    - For the **Index Page**: Throughput initially peaked around 42,000 requests/sec with a thread pool size of 8, showing a steady decline beyond a concurrency level of 1000 due to server resource limitations.
    - For the **Fibonacci Function**: The throughput consistently decreased as concurrency increased, with the best results observed at lower concurrency levels, emphasizing the computationally intensive nature of this function.

5. Explanation of Behavior and Inferences

- **Thread Pool Size Impact**:

    - Increasing the thread pool size allowed the server to handle more requests concurrently, resulting in **higher throughput** for lower concurrency levels. However, at very high concurrency levels (beyond 2000), the benefits diminished due to **resource contention** and **overhead** from managing too many threads.
    - For the **Index Page**, the throughput benefits of a smaller thread pool size were noticeable at lower concurrency levels, while larger thread pool sizes (e.g., 128) performed better when the concurrency increased, though with a fluctuating pattern.
- **Function Complexity**:
    - The **Fibonacci Function** had lower throughput and higher variability due to its computationally intensive nature, indicating that server performance is not only dependent on concurrency but also on the **complexity of the function being served**.
    - The **Square Function** showed a better balance between throughput and concurrency, though it still plateaued and then decreased as the load increased.
    - The **Index Page** consistently provided the best performance metrics, showing high throughput even at higher concurrency levels due to its minimal computational needs.
- **Single Machine Setup**:
    - Hosting both the client and server on the same machine likely introduced **CPU and memory contention**, which impacted the overall performance results. Running the load tests in a **distributed setup** (client and server on different machines) could yield higher throughput values and more stable response times.

6. Graphs and Results Summary

- **Throughput Graphs**: The three graphs representing **Square Function**, **Index Page**, and **Fibonacci Function** show distinct patterns:
    - The **Square Function** graph showed fluctuations, with a notable peak in throughput around 19,000 requests/sec for a thread pool size of 64.
    - The **Index Page** graph had the highest throughput, peaking around 42,000 requests/sec with smaller thread pools, but the performance gradually decreased as concurrency increased.
    - The **Fibonacci Function** graph indicated a steady decline in throughput with increasing concurrency, reflecting the high computation demands.

Key Observations:

- The **optimal performance** for all functions occurred with a thread pool size of **8 or 32** and a concurrency level between **500-1000**.
- Beyond a concurrency level of **1500**, **thread contention** and **server resource limitations** led to significantly fluctuating and higher response times, particularly for computationally intensive functions like Fibonacci.

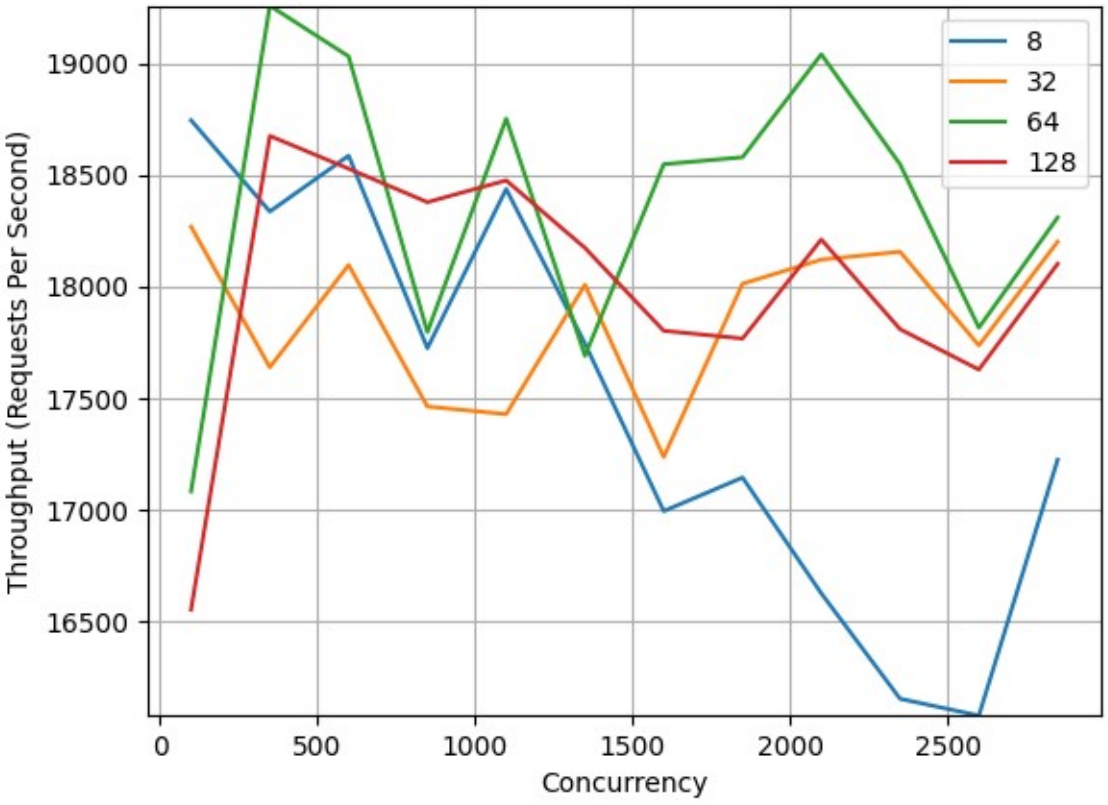
7. Conclusion

The experiment demonstrated the importance of understanding the interplay between **concurrency**, **thread pool size**, and **function complexity** in a web server. The results showed that:

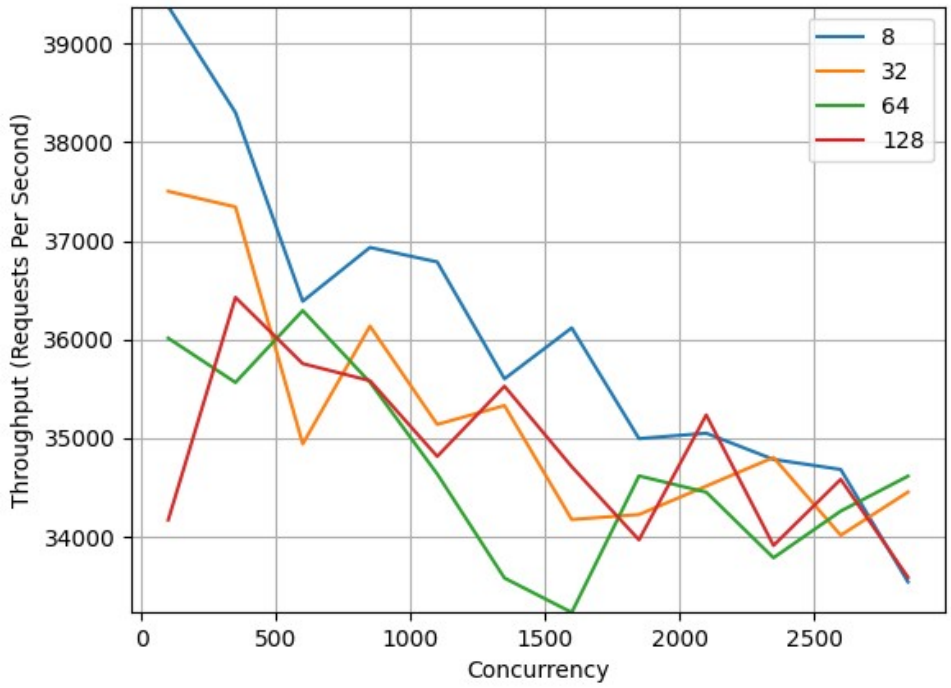
- **Thread pools** provide significant performance benefits up to a certain point, after which adding more threads yields diminishing returns due to increased management overhead.
- **Computationally intensive functions** like Fibonacci reduce server capacity, emphasizing the need for careful load management and optimization for CPU-bound tasks.
- **Static content** and lightweight functions are easier to scale and maintain low response times, even at high concurrency levels.

8. Images

- **Square**



- **Fibonacci**



- **Index page**