Stock Exchange System Scalability Performance Analysis

Name: Hangming Ye, Huidan Tan

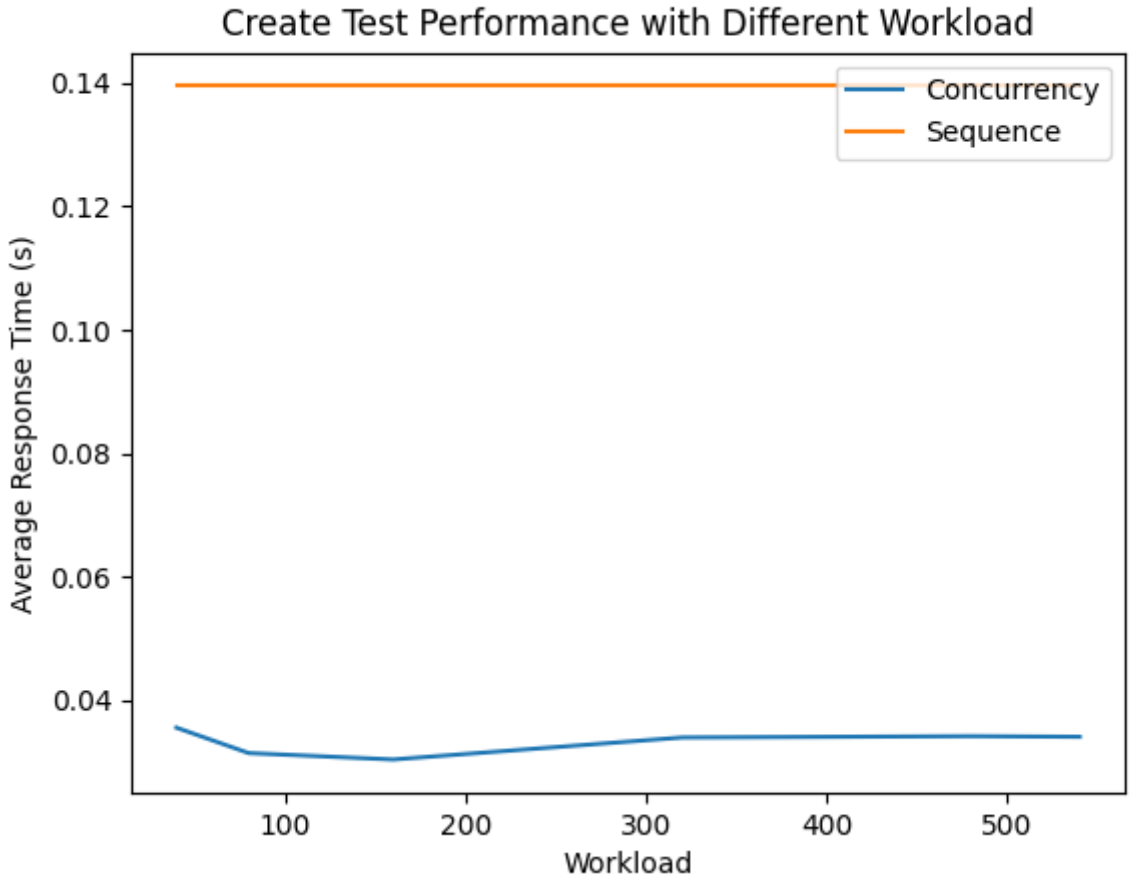
NetID: hy201, ht175

1. Different Functions Performance with Different Workload

Test Environment

1. CPU Count: 4
2. Concurrent Process Count: 4
3. Method: Use multithreading to create clients on another virtual machine to make requests (reduce interference caused by client resource occupancy), where workload represents the concurrency level.
4. Description: This is a description of the testing environment. The system under test is running on a machine with 4 CPU cores. To simulate concurrent user traffic, we will use a separate virtual machine to create multiple client instances using multithreading. This approach reduces interference caused by the client's resource usage on the system under test. The workload parameter will be used to control the level of concurrency during testing.

1.1 Create Test



Test Type:

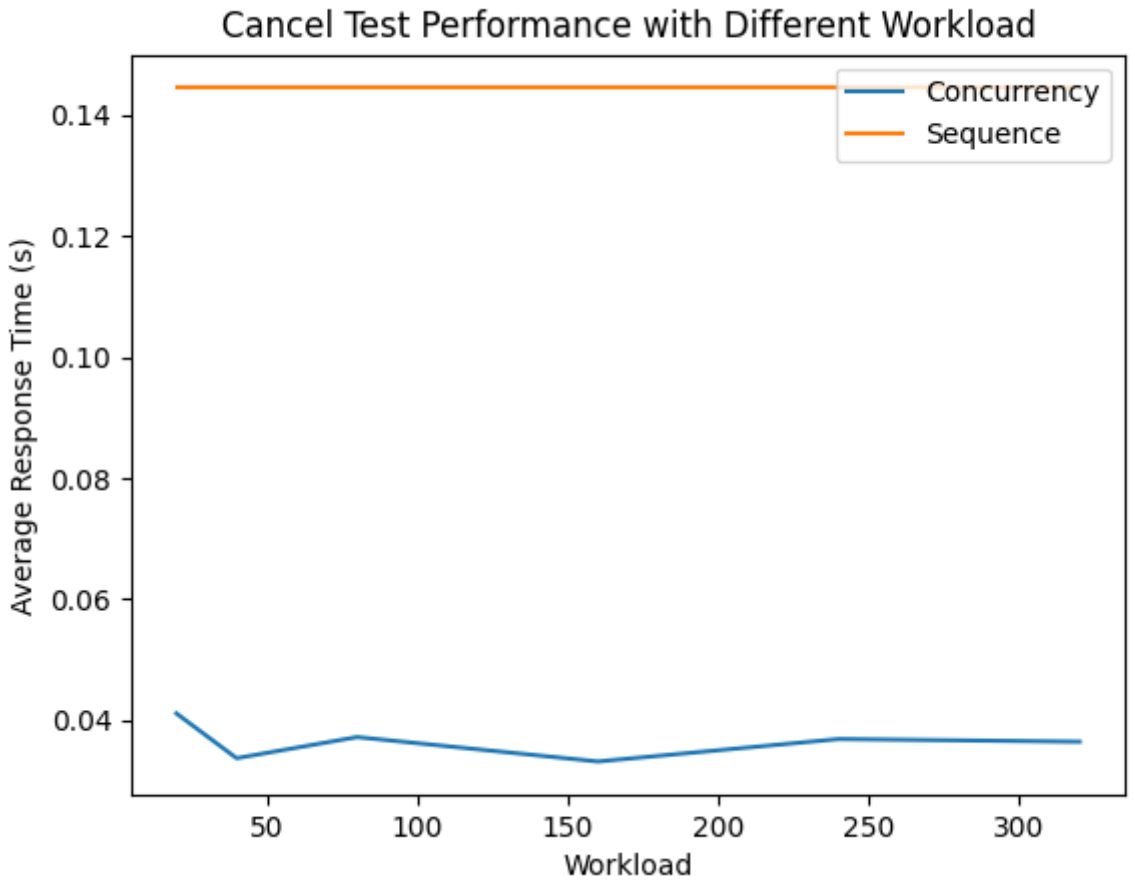
This test is primarily focused on non-blocking, pure concurrency operations, such as creating an account.

picture description：

From the comparison of the latency lines between the sequential and concurrent tests, it can be observed that the concurrent latency is significantly lower than the sequential latency.

In terms of the concurrency latency line, at the beginning of the test, the latency is relatively high. This is due to the fact that the initial workload is small, and the CPU's concurrency performance is not fully utilized. As the workload increases, the CPU utilization rate increases, resulting in a reduction in latency. However, as the workload continues to increase, the latency gradually increases with the workload.

1.2 Cancel Test



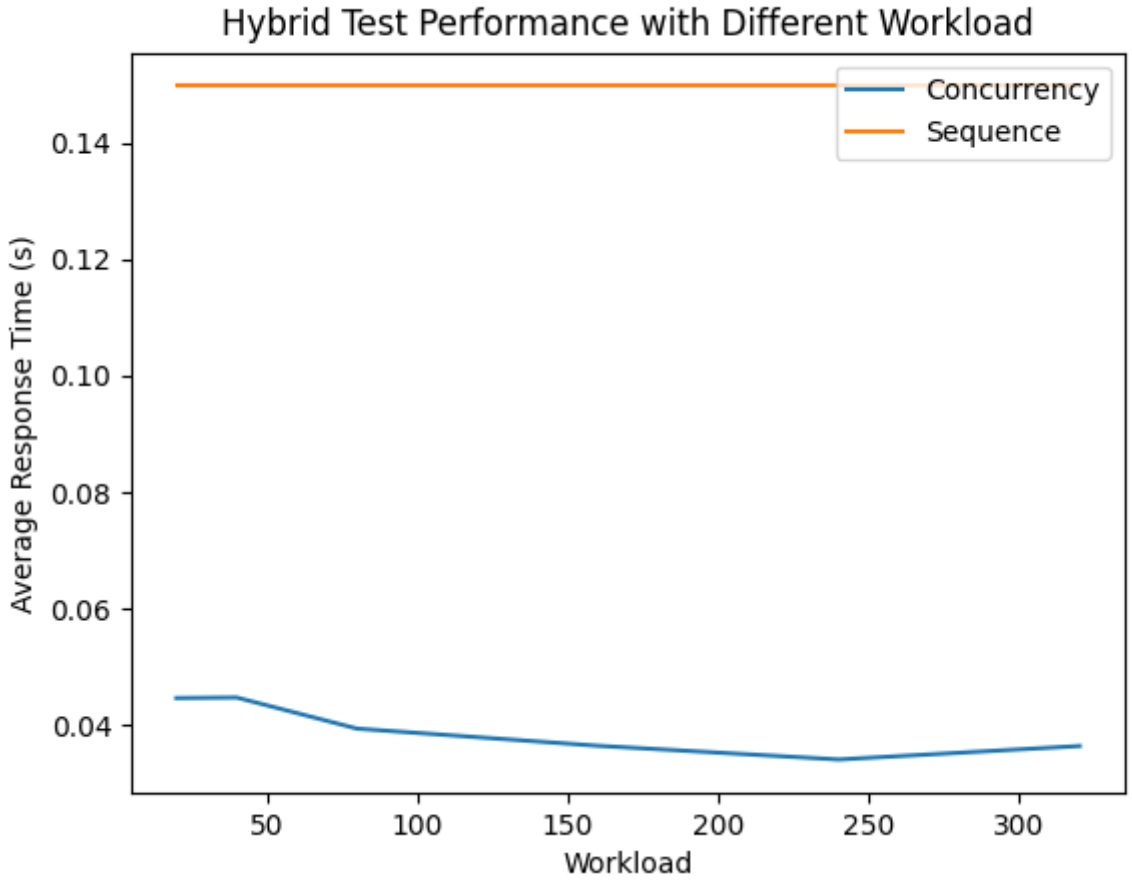
Test Type:

This test primarily focuses on pure blocking operations, such as canceling an order. To conduct concurrent testing of pure blocking operations, we first create orders simultaneously and then separately measure the cancellation of orders. This approach helps reduce testing errors.

Image Description:

The concurrency line shows a generally stable trend. This is because we have locked the process, and the initial fluctuations are due to errors caused by the small data volume.

1.3 Hybrid Test



Test type:

In contrast to the previous two unit tests, this test is a mixed test that simulates real-world concurrency scenarios. Odd-numbered clients create batches of buy orders, while even-numbered clients execute sell orders and queries. The involved functions include user creation, position creation, trade execution, and order querying.

Image description:

In terms of concurrency, the average latency line shows a higher value at the beginning, as the workload is small and does not fully utilize the CPU's concurrency capability. However, as the workload increases, the CPU utilization rate also rises, resulting in a decrease in latency (for example, between a workload of 50-240). Nevertheless, when the workload surpasses 250, the latency gradually increases as the workload increases.

2. Performance of Different Process Number

Test Environment:

1. CPU Number: 4
2. Test Method: Hybrid Test

3. Workload: 100 clients * 3 requests / client

4. Testing Method: Modifying the size of the process number in the pool.

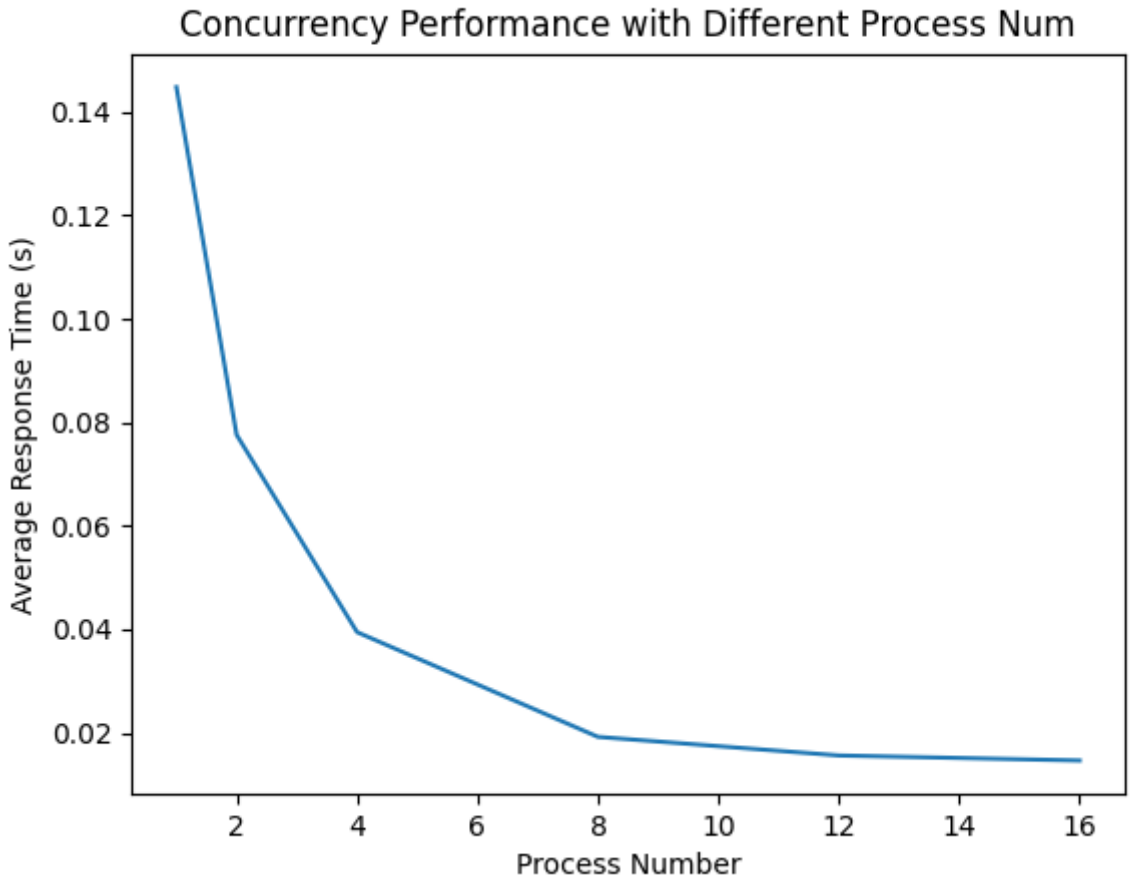


Image description:

The curve in the graph shows an inverse function. In the early stages, the performance of the system improves significantly as the number of process increases (latency decreases significantly). However, in the later stages, the curve becomes flatter, and increasing the number of threads does not lead to a significant improvement in performance.

Analysis:

The reason for this behavior could be that as the number of processes increases, the overhead associated with managing context switches between processes also increases. This could lead to diminishing returns as the number of processes increases beyond a certain point, resulting in a flattening of the performance curve.

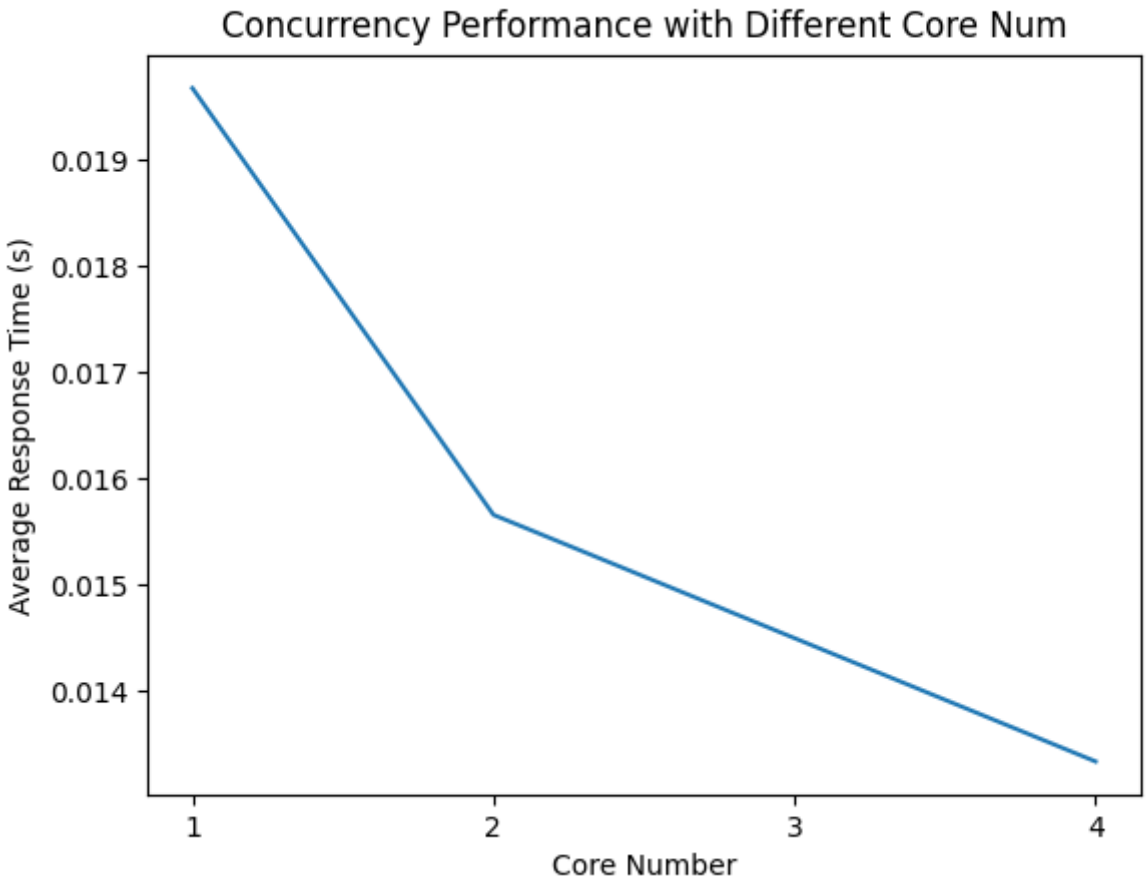
3. Performance of Different Core Number

Test Environment:

1. Process Number: 16
2. Test Method: Hybrid Test
3. Workload: 100 clients * 3 requests / client
4. Method

   change directory to the ./docker_deploy/src and use task set to specify core number used to run server

```
taskset -c 0 python3 server.py
taskset -c 0-1 python3 server.py
taskset -c 0-3 python3 server.py
```



Core Number	1	2	4
Average Latency (s)	0.01968	0.01566	0.01333

Core Number	1	2	4
Throughput (request/s)	152.4	191.6	225.1

Image description:

- As the number of cores increases from 1 to 4, the average execution time (latency) decreases (i.e., throughout improves).
- However, the improvement is not proportional to the increase in cores. In other words, doubling the number of cores does not lead to a doubling of performance (i.e., halving of response time).
- The response times for 1, 2, and 4 cores are 00.01968, 0.01566 and 0.01333 seconds, respectively. This means that the speedup from 1 to 2 cores is about 125%, and the speedup from 2 to 4 cores is about 117.5%.

4. Scalability & Bottleneck Analysis

With the above result, we could find that adding more cores could have a speedup, which means it actually improves the performance. However, during the test, we find that the scale could achieve a better performance when adding more processes in the pool since it could increase the utilization of the CPU.

In the meanwhile, the contest and non-contest requests show no significant differences in the average response time. While adding more processes and more cores could improve the throughput in the whole system, which means that the network I/O (socket amount to receive and send) and parallel task like parse XML is the main part to be improved. Also, the database i/o may be the bottleneck of the system since the contest and non-contest request takes almost the same time.