Diploma Thesis

PERFORMANCE EVALUATION OF TAOBENCH IN THE DCPERF FRAMEWORK

Andreas Karagiorgis

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2026

UNIVERSITY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

PERFORMANCE EVALUATION OF TAOBENCH IN THE DCPERF FRAMEWORK

Andreas Karagiorgis

Supervisor
Dr. Yiannakis Sazeides

The Individual Diploma Thesis was submitted in partial fulfillment of the requirements for the award of the degree of Bachelor in Computer Science at the Department of Computer Science of the University of Cyprus.

May 2026

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor, Dr. Yiannakis Sazeides, for his continuous guidance, support, and valuable feedback throughout the development of this thesis. Our weekly meetings helped me stay focused, organize my work, and approach the research process in a more structured and critical way. His advice was extremely important, both for the technical direction of this thesis and for helping me understand how to interpret the experiments and their results more clearly.

I am also grateful to the Department of Computer Science at the University of Cyprus for providing the academic environment and knowledge that supported my studies. The skills I gained during my degree were essential for completing this work.

Furthermore, I would like to thank my family for their constant support, patience, and encouragement throughout my studies. Their belief in me helped me continue even during stressful and difficult periods. I would also like to thank my friends and classmates for their support, motivation, and presence during these years.

I would also like to thank Panteleimonas Chatzimiltis for his support and willingness to help whenever I had questions or faced technical difficulties. His explanations, advice, and encouragement helped me overcome several challenges during the development and experimental part of this thesis.

Finally, I would like to acknowledge the open-source tools and research platforms used in this work, including DCPerf, TaoBench, and CloudLab, which made the experimental evaluation possible.

Abstract

Modern datacenter applications require high throughput, low latency, and efficient use of system resources. As these applications become more complex, realistic benchmarking frameworks are necessary to evaluate how server systems behave under production-like workloads. DCPerf is an open-source benchmark suite designed for datacenter workload evaluation, and TaoBench is one of its workloads. TaoBench models the behavior of a large-scale caching system inspired by Meta's TAO infrastructure, where requests are processed through fast cache-hit paths and slower cache-miss paths.

This thesis presents an experimental performance evaluation of TaoBench within the DCPerf framework. The experiments were conducted on CloudLab using a distributed three-node setup consisting of one server node and two client nodes. The evaluation focuses on both application-level metrics, including throughput, average latency, tail latency, and cache hit ratio, and system-level metrics, including CPU, memory, network, and disk utilization.

A key part of the methodology was the use of a fixed warm-up phase. Since TaoBench performance depends strongly on cache population and hit-ratio stabilization, the benchmark workflow was modified so that each experiment started from a comparable cache state before entering the measurement phase. This improved the fairness of comparisons between different client loads and server thread configurations.

The first part of the evaluation studied the impact of client load on TaoBench performance. A full client load sweep with five repetitions per load showed that the best operating load for the current setup was 40 clients per thread. At this load, TaoBench achieved approximately 564K QPS with 3.68 ms average latency, 9.58 ms p99 latency, and 18.41 ms p99.9 latency. Increasing the load beyond this point did not significantly improve throughput and mainly increased average and tail latency.

The second part of the evaluation studied the impact of server-side fast/slow thread configuration. The final repeated ratio plateau sweep showed that the best performance region was reached around FAST_THREADS_RATIO = 4.5–5.0 and SLOW_TO_FAST_RATIO = 0.125. In this region, TaoBench achieved approximately 956K–957K QPS with about 2.17 ms average latency and 13.8 ms p99 latency. The

configuration FAST_THREADS_RATIO = 5.0 and SLOW_TO_FAST_RATIO = 0.125 achieved 13.77 ms p99 latency and 24.56 ms p99.9 latency, while the highest server-side throughput was observed at FAST_THREADS_RATIO = 4.5 and SLOW_TO_FAST_RATIO = 0.125. Increasing the fast-thread ratio further to 6.0 reduced performance, indicating a throughput plateau.

Overall, the results show that TaoBench performance is strongly affected by both workload intensity and server-side thread allocation. The workload is fast-path dominated because most requests are cache hits, and it benefits from a high number of fast threads combined with a small number of slow threads. The experiments also show that excessive threading can introduce scheduling and coordination overhead, limiting further performance gains.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Motivation

Modern online services depend on large datacenter infrastructures. Applications such as social networks, video platforms, web services, data analytics systems, and cloud applications must handle a large number of requests while still responding quickly to users. As these services continue to grow, it becomes increasingly important to understand how datacenter systems behave under load.

Evaluating datacenter performance is not simple, because real applications use many parts of the system at the same time. Their performance depends on the CPU, memory, network, storage, operating system scheduling, caching layers, and the way requests are processed by the application. For this reason, traditional benchmarks that focus only on one part of the system are often not enough to represent real datacenter behavior. More realistic benchmark suites are needed in order to study how complete applications behave.

DCPerf is an open-source benchmark suite designed for evaluating datacenter workloads using applications that are inspired by real production services [1], [2]. It includes workloads for different types of services, such as feed generation, video transcoding, web services, analytics, and caching systems. One of these workloads is TaoBench, which models the behavior of a distributed caching system inspired by Meta's TAO infrastructure [3], [4], [10]. Caching systems are important in datacenters because they help reduce repeated access to slower backend storage systems [4], [9]. By keeping frequently requested data in memory, caches can improve response time and reduce the load on backend services. However, the performance of a caching system depends on several factors, such as the number of client requests, the cache hit rate, the server thread configuration, and the use of system resources.

The motivation of this thesis is to study TaoBench within the DCPerf framework and understand how its performance changes under different experimental conditions. More specifically, this work focuses on how client load and fast/slow server thread configurations affect throughput, latency, hit rate, and system utilization.

1.2 Problem Statement

Although benchmark suites such as DCPerf provide realistic workloads for datacenter evaluation, understanding the behavior of a specific workload still requires careful experimentation. TaoBench includes several configurable parameters that can significantly affect performance, including client load, memory size, warm-up duration, fast thread ratio, and slow-to-fast thread ratio.

A key problem is that increasing client load does not always lead to higher throughput. At low load, the system may have available capacity and throughput may increase as more requests are generated. However, after the system reaches saturation, additional load may only increase queueing delay and tail latency without improving throughput. Therefore, it is necessary to identify the operating region where the system performs efficiently and the point where additional load starts to degrade performance.

Another challenge is the configuration of TaoBench's server-side threading model. TaoBench separates request processing into fast-path and slow-path components. Cache-hit requests are handled by fast threads, while cache-miss requests are handled through the slow path using dispatcher and slow-thread mechanisms. Since most requests are expected to be cache hits, the balance between fast and slow thread capacity can strongly affect performance. A default configuration may not always be optimal for a specific workload or hardware setup.

Finally, experimental comparability is also an important issue. If each run starts from a different cache state, the resulting throughput and latency measurements may not be directly comparable. This is especially relevant for TaoBench, where the warm-up phase is needed to populate the cache and stabilize the hit ratio. For this reason, the thesis also considers the importance of using a consistent warm-up methodology before collecting measurement results.

Therefore, this thesis focuses on how TaoBench performance changes under different client loads and fast/slow thread configurations, and how these changes help identify saturation, latency issues, and server-side bottlenecks.

1.3 Objectives of the Thesis

The main objective of this thesis is to evaluate the performance of TaoBench within the DCPerf framework using a distributed CloudLab experimental environment. The evaluation focuses on both application-level performance metrics and system-level resource utilization. The specific objectives of the thesis are:

1. To study the DCPerf framework and understand the role of TaoBench as a representative caching workload.
2. To deploy and execute TaoBench in a distributed setup consisting of one server node and two client nodes.
3. To collect and analyze key performance metrics, including throughput, average latency, p50 latency, p99 latency, hit ratio, CPU utilization, memory utilization, and network utilization.
4. To evaluate how different client load levels affect TaoBench performance.
5. To identify the saturation point of the system and analyze how latency behaves when the offered load increases beyond that point.
6. To investigate how different fast/slow thread configurations affect throughput, latency, and resource utilization.
7. To determine which tested configuration provides the best performance for the selected workload and experimental setup.
8. To explain the observed performance trends using the internal architecture of TaoBench, especially its fast-path and slow-path processing model.

Overall, these objectives define the experimental direction of the thesis and support the analysis of TaoBench under controlled and repeatable conditions.

1.4 Thesis Contributions

The main contributions of this work are:

1. **Experimental Deployment of TaoBench on CloudLab.** A structured deployment of TaoBench was created within the DCPerf framework using a distributed CloudLab setup. The experimental environment consists of one server node and two client nodes, allowing TaoBench to be evaluated under networked conditions instead of only in standalone mode.
2. **Reproducible Experiment Automation.** A separate experiment repository was created to organize the CloudLab profile, setup scripts, execution scripts, monitoring tools, and documentation used in the experiments. These scripts automate server startup, client execution, load sweeps, utilization collection, and result aggregation, making the experimental workflow more repeatable and reducing manual execution errors.
3. **Fixed Warm-up Methodology for Comparable Runs.** A fixed warm-up methodology was introduced and used for the TaoBench experiments. Since TaoBench depends on cache population and hit-ratio stabilization, each run uses the same warm-up load before entering the measurement phase. This helps ensure that different experimental runs start from a comparable cache state.
4. **Characterization of TaoBench under Different Client Loads.** TaoBench performance was evaluated under different client load levels. The analysis shows how throughput, average latency, tail latency, and hit ratio behave as the load increases, and helps identify the efficient operating region and saturation point of the system.
5. **Evaluation of Fast/Slow Thread Configurations.** The impact of TaoBench's server-side fast/slow thread configuration was studied by varying FAST_THREADS_RATIO and SLOW_TO_FAST_RATIO. This analysis shows how the allocation of server threads between cache-hit and cache-miss processing paths affects throughput, latency, and resource utilization.
6. **Combined Performance and Utilization Analysis.** Application-level metrics, such as throughput, latency, tail latency, and hit ratio, were analyzed together with system-level utilization metrics, such as CPU, memory, network, and disk utilization. This

makes it possible to better understand whether performance limitations are caused by client-side load, server-side processing, network behavior, or storage activity.

1.5 Thesis Structure

The rest of the thesis is organized as follows.

Chapter 2 introduces the background concepts needed to understand the thesis. It covers benchmarking, performance metrics, client-server architectures, caching, key-value stores, workload generation, and datacenter workloads.

Chapter 3 reviews related work in datacenter benchmarking. It discusses benchmark suites such as CloudSuite, TailBench, and explains how TaoBench fits into this area.

Chapter 4 focuses on the DCPerf framework and the TaoBench workload. It explains the TaoBench architecture, including the server, clients, request processing flow, fast path, slow path, thread ratio configuration, and warm-up/execution phases.

Chapter 5 describes the experimental methodology. It presents the CloudLab environment, software setup, experimental parameters, scripts, metrics collection, and data processing approach.

Chapter 6 presents the results and evaluation. It analyzes the baseline results, the impact of client load, throughput and latency behavior, resource utilization, and fast/slow thread ratio configurations.

Chapter 7 concludes the thesis by summarizing the main findings, discussing the limitations of the work, and suggesting possible directions for future work.

Chapter 2: Background

2.1 Benchmarking Computer Systems

Benchmarking is the process of evaluating the performance of a computer system using a controlled and repeatable workload. The goal is to measure how well a system performs under specific conditions and to compare different systems, configurations, or optimization techniques. In computer systems research, benchmarks are commonly used to study CPU performance, memory behavior, network performance, storage activity, and application-level behavior.

A benchmark usually includes three main parts: a workload, a set of input parameters, and a set of metrics. The workload represents the type of task or service being tested. The input parameters control the scale of the experiment, such as the number of clients, request rate, dataset size, or test duration. The metrics describe the observed behavior of the system, such as throughput, latency, resource utilization, or error rate.

Traditional benchmarks often focus on specific parts of the system. For example, some benchmarks measure only CPU performance, memory bandwidth, or disk throughput. These benchmarks are useful for understanding individual hardware components, but they may not fully represent the behavior of modern datacenter applications. Real datacenter workloads usually involve several components working together, including CPU processing, memory access, networking, caching, synchronization, and request scheduling.

For this reason, modern benchmark suites try to model more realistic workloads [1], [5], [6]. Instead of measuring only one component, they evaluate complete applications or application-like services. DCPerf follows this approach by using workloads that model production-style datacenter services, such as feed generation, video transcoding, web services, analytics, and caching systems [1], [2]. This makes it more suitable for studying the end-to-end behavior of servers running realistic workloads.

In this thesis, benchmarking is used to evaluate the performance of TaoBench under different experimental conditions. The goal is not only to measure maximum throughput, but also to understand how latency, hit rate, and resource utilization change when the client load and server thread configuration are modified.

2.2 Performance Metrics

Performance metrics are measurements used to describe how a system behaves during execution. In this thesis, the most important metrics are throughput, latency, tail latency, hit rate, CPU utilization, memory utilization, network utilization, and disk activity. Each metric shows a different part of the system's behavior.

Throughput shows how much useful work the system completes per unit of time. Latency shows how long individual requests take to complete. Tail latency focuses on the slowest requests, which are especially important in systems that must respond quickly to users. Hit rate describes how effectively the cache serves requests from memory. CPU, memory, network, and disk utilization show how much of the available system resources are being used.

Using only one metric can be misleading. For example, a system may achieve high throughput but still suffer from very high tail latency. Similarly, high CPU utilization does not always mean that the system is doing useful work efficiently. The CPU may also be used for overhead such as scheduling, synchronization, or thread coordination. Therefore, this thesis evaluates TaoBench using multiple metrics together, in order to better understand both performance and possible bottlenecks.

2.2.1 Throughput

Throughput measures how much work a system completes per unit of time. In server benchmarks, throughput is often reported as requests per second, operations per second, or queries per second. In TaoBench, throughput is mainly reported as QPS, which means queries per second.

Throughput is important because it shows the capacity of the system. Higher throughput means that the system can serve more client requests in the same amount of time. In datacenter workloads, this is important because services often need to handle a large number of requests from users and applications.

However, throughput must be interpreted carefully. A system may initially increase throughput as the client load increases. However, after the system reaches saturation, additional load may no longer improve throughput. Instead, it may only increase queueing

delays and latency. For this reason, throughput should be analyzed together with latency and resource utilization.

In TaoBench, throughput can be separated into fast-path throughput and slow-path throughput. Fast-path throughput corresponds to cache-hit requests, while slow-path throughput corresponds to cache-miss requests. The total throughput is the combination of both. This distinction is important because it helps explain whether performance is mainly dominated by fast cache access or by slower backend-like processing.

2.2.2 Latency

Latency measures the time required to complete a request. In a client-server system, latency usually refers to the time between sending a request and receiving the corresponding response. It includes client-side processing, network delay, server-side processing, queueing delay, and response transmission.

In benchmarking, latency is usually reported in milliseconds. Lower latency means that requests are completed faster. This is especially important for interactive applications, where users expect quick responses. Even if a system achieves high throughput, it may still provide poor user experience if request latency becomes too high.

Average latency provides a general indication of response time, but it does not show the full distribution of request delays. Some requests may complete very quickly, while others may experience longer delays because of queueing, contention, or slow-path processing. For this reason, average latency is useful, but it is not enough on its own.

In the TaoBench experiments, latency is measured from the client side. This is important because client-side latency captures the end-to-end behavior of the system. It includes the time spent in the network, the server processing time, possible TCP queueing, and client-side effects. Therefore, client-side latency is useful for evaluating the complete behavior of the distributed TaoBench setup.

2.2.3 Tail Latency

Tail latency refers to the latency experienced by the slowest requests in a system. It is commonly measured using percentiles such as p95, p99, or p99.9 [7]. For example, p99 latency means that 99% of requests complete within that time, while the slowest 1% take longer.

Tail latency is important in datacenter systems because large-scale services handle many requests, and even a small fraction of slow requests can affect user-perceived performance [7]. Even if only a small percentage of requests are slow, a large number of users may still be affected. For this reason, tail latency is often more important than average latency in user-facing systems.

Tail latency can increase for several reasons, such as queueing delays, CPU scheduling overhead, lock contention, network congestion, memory stalls, and slow backend operations. In a caching workload such as TaoBench, tail latency can also be affected by the slow path, because cache misses require additional processing compared to cache hits.

In this thesis, p99 and p99.9 latency are especially important for identifying saturation. When client load increases beyond the system's efficient operating region, throughput may remain almost the same while tail latency increases sharply. This indicates that the system is no longer processing additional load efficiently and that requests are spending more time waiting in queues.

2.2.4 Hit Rate

Hit rate measures the fraction of requests that are served successfully from the cache. In a caching system, a cache hit occurs when the requested item is found in memory and can be returned quickly. A cache miss occurs when the item is not available in the cache and requires additional backend-like processing.

The hit rate is usually expressed as a percentage or fraction. For example, a hit rate of 0.90 means that 90% of requests are served from the cache, while 10% are misses. A higher hit rate generally improves performance because more requests can be handled through the fast path.

In TaoBench, hit rate is a central metric because the benchmark is designed to model a cache-dominated workload. A stable hit rate near the target value indicates that the workload is behaving as expected. If the hit rate is too low, then too many requests follow the slow path, which can reduce throughput and increase latency. If the hit rate is too high, the slow path may not be exercised enough, making the workload less realistic.

The warm-up phase is strongly related to hit rate. At the beginning of a run, the cache may be cold, meaning that many requested items are not yet present. During warm-up, the cache is populated and the hit rate gradually stabilizes. In the experiments of this thesis, a stable hit rate close to 0.90 was used as an indication that the cache had been warmed properly before the measurement phase. Without sufficient warm-up, the results could be unstable or misleading, because the measurement phase would include too many cold-cache misses.

2.2.5 CPU Utilization

CPU utilization measures how much of the processor's capacity is being used during execution. It is usually expressed as a percentage. For example, 80% CPU utilization means that the CPU is busy for 80% of the measurement period and idle for the remaining 20%.

CPU utilization is useful for understanding whether the processor is a possible bottleneck. If throughput stops increasing while CPU utilization is high, the system may be CPU-bound. If CPU utilization is low while throughput is also low, then the bottleneck may be somewhere else, such as the network, memory subsystem, disk, or synchronization overhead.

However, high CPU utilization does not always mean efficient performance. A system may use a lot of CPU time because of scheduling overhead, lock contention, interrupt processing, or inefficient thread coordination. Therefore, CPU utilization should be analyzed together with throughput and latency. A good configuration should use CPU resources to complete useful requests, not mainly to handle overhead.

In this thesis, CPU utilization is collected for the server and both client nodes. This allows a comparison between server-side and client-side load. If the clients have relatively low CPU utilization while the server has high CPU utilization, then the bottleneck is more likely to be on the server side. This distinction is important for interpreting both the client load experiments and the fast/slow thread ratio sweep.

2.2.6 Network Utilization

Network utilization measures how much data is transmitted and received over the network during execution. In a distributed benchmark such as TaoBench, the clients and the server communicate over the network, so network behavior can affect both throughput and latency. Network utilization is often reported as receive throughput and transmit throughput, usually in MB/s or Gbps. On the server side, receive traffic mainly corresponds to incoming client requests, while transmit traffic mainly corresponds to responses sent back to the clients. On the client side, the opposite pattern is expected: clients transmit requests and receive responses.

Network measurements are useful for checking whether the network is a bottleneck. If the network interface is close to saturation, then throughput may be limited by network capacity instead of server processing. If network utilization remains below saturation while throughput stops increasing, then the bottleneck is likely somewhere else.

In the TaoBench experiments, network utilization is used to check consistency between client and server measurements. For example, the total traffic transmitted by the clients should approximately match the traffic received by the server. This helps confirm that the utilization monitoring is reliable and that the observed performance trends are not caused by measurement errors.

2.3 Client-Server Architectures

A client-server architecture is a distributed system model where one or more client machines communicate with a server to request services or data. The clients generate requests, while the server processes those requests and returns responses. This model is widely used in modern computing systems, especially in web applications, cloud platforms, databases, and caching systems.

In a typical client-server system, a client sends a request over the network to the server. The server receives the request, performs the required computation or data lookup, and sends a response back to the client. The total response time depends on several factors, including client-side processing, network delay, server-side processing, queueing delay, and the size of the response.

This architecture is important for benchmarking because the performance of the system depends on both the clients and the server. If the clients cannot generate enough requests, then the server may not be fully stressed. On the other hand, if the server cannot process requests quickly enough, requests begin to queue and latency increases. Therefore, a good experimental setup must ensure that the clients generate enough load, while also measuring whether the server becomes the bottleneck.

In this thesis, TaoBench is executed using a distributed client-server setup. The server node runs the TaoBench server, while two client nodes generate requests. This setup allows the benchmark to evaluate the server under networked conditions instead of running everything on a single machine. It also makes it possible to observe how throughput, latency, and resource utilization behave across multiple nodes.

The client-server model is especially relevant to caching workloads. In these workloads, clients continuously request data from a cache server. If the requested data is found in memory, the server can respond quickly. If the data is not found, additional processing is required. TaoBench models this behavior through its fast-path and slow-path request processing mechanism.

2.4 Caching and Key-Value Stores

Caching is a technique used to improve system performance by storing frequently accessed data in a faster storage layer, usually memory. Instead of retrieving the same data repeatedly from a slower backend system, such as a database or remote storage service, the system can serve common requests directly from the cache. This reduces latency and lowers the load on backend components.

A cache is especially useful in read-heavy systems, where many requests access the same or similar data. Social networks, web services, recommendation systems, and content platforms often rely heavily on caching because users repeatedly request profiles, posts, media metadata, relationships, and other frequently accessed objects.

A key concept in caching is the difference between a cache hit and a cache miss. A cache hit occurs when the requested item is already present in the cache. In this case, the system can

return the data quickly. A cache miss occurs when the requested item is not in the cache. In that case, the system must retrieve the data, or simulate retrieving it, from a slower backend source. Cache misses are usually more expensive and can increase latency.

Key-value stores are commonly used for caching. In a key-value store, each data item is stored using a unique key and an associated value. The client sends a key to the server, and the server returns the corresponding value if it exists. This model is simple but efficient for high-throughput data access.

For example, a simplified TaoBench-style key-value request can be represented as: GET key_12345

In this example, key_12345 represents the requested object key. If the key exists in the cache, the server can return the associated value through the fast path. If the key is not found, the request is treated as a cache miss and follows the slow path.

TaoBench models this type of workload. It uses key-value requests to simulate a caching layer inspired by Meta's TAO system. Most requests are expected to be cache hits and are handled by the fast path. A smaller fraction of requests are cache misses and are handled by the slow path. This allows TaoBench to study not only raw key-value throughput, but also the interaction between cache efficiency, request scheduling, and backend-like delay.

In practice, key-value caching systems are often evaluated using tools such as Memcached and Memtier Benchmark. Memcached is an in-memory key-value store that can be used as a cache server [9], while Memtier Benchmark is a workload generator that sends key-value requests to a server and measures throughput and latency. TaoBench follows a similar client-server caching model, but it uses modified components to reproduce TAO-like cache behavior more closely.

2.5 Workload Generation

Workload generation is the process of creating requests that stress a system during a benchmark. A workload generator controls how many requests are sent, how many clients are active, what type of operations are performed, and how long the experiment runs. The goal is to reproduce a realistic or controlled pattern of system usage.

In benchmarking, workload generation is important because the results depend heavily on the input load. A system may perform well under light load but behave differently under heavy load. As the number of requests increases, the system may reach saturation. At that point, throughput may stop increasing, while latency and tail latency continue to grow.

A workload generator can control several parameters, such as the number of clients, the number of connections, the request type, the request size, the duration of the experiment, the read/write ratio, and the key distribution.

In TaoBench, the clients act as workload generators. They create many concurrent connections to the server and issue key-value requests. The load is mainly controlled using the `CLIENTS_PER_THREAD` parameter. A higher value increases the number of connections and creates more pressure on the server.

Workload generation must be designed carefully. If the load is too low, the server may not be stressed enough and the results will not show the system's full capacity. If the load is too high, the system may enter an overload region where latency increases sharply and throughput does not improve. Therefore, load sweeps are useful because they show how the system behaves across different load levels.

In this thesis, workload generation is used in two main ways. First, different client loads are tested in order to identify the saturation point of TaoBench. Second, a fixed client load is used while changing the server thread configuration. This allows the experiments to separate the effect of client-side load from the effect of server-side thread allocation.

2.6 Data Center Workloads

Datacenter workloads are applications and services that run on large-scale server infrastructure. These workloads support many modern online services, such as social networks, video platforms, search engines, cloud storage, data analytics, and artificial intelligence services. They often serve a large number of users and must provide high throughput, low latency, and reliable performance.

Unlike small standalone programs, datacenter workloads usually involve many interacting system components. A single user request may require network communication, cache lookups, database access, application logic, storage operations, and sometimes machine learning inference. Because of this complexity, evaluating datacenter performance requires benchmarks that can capture realistic behavior across the full system [1], [5], [6].

Datacenter workloads can stress different resources depending on their purpose. Some workloads are CPU-intensive, such as video transcoding. Others are memory-intensive, such as caching systems. Some are network-sensitive, especially distributed services with many client-server interactions. Others are latency-critical, where even small delays can affect user experience.

Common characteristics of datacenter workloads include high concurrency, large request volume, strict latency requirements, heavy memory and cache usage, network communication between services, and scalability across many CPU cores.

TaoBench represents a caching-oriented datacenter workload. Caching workloads are important because they are often on the critical path of large-scale services. A slow cache can increase latency for many dependent services, while an efficient cache can reduce backend load and improve overall system responsiveness.

This thesis focuses on TaoBench because it provides a controlled way to study a realistic class of datacenter workload. By changing client load and fast/slow thread configurations, the experiments analyze how a caching workload behaves under pressure and how server-side configuration affects throughput, latency, and resource utilization.

Chapter 3: Related Work

3.1 Benchmark Suites for Data Center Systems

Benchmark suites are widely used in computer systems research to evaluate the performance of hardware, software stacks, and complete application environments. In the context of datacenters, benchmark suites are especially important because modern services are complex and cannot be evaluated accurately using only simple synthetic tests. A realistic benchmark should capture application behavior, concurrency, memory usage, network communication, latency requirements, and resource utilization.

Earlier benchmark suites, such as CloudSuite and TailBench, helped move systems research away from purely synthetic workloads and toward more realistic cloud and latency-sensitive applications [5], [6]. CloudSuite focuses on scale-out cloud services, while TailBench focuses on latency-critical applications and tail-latency behavior. These suites are important because they introduced workloads that better resemble real services compared to traditional CPU-only benchmarks.

However, modern hyperscale datacenters have become increasingly complex. Production systems now involve many interacting services, large memory footprints, high concurrency, strict latency requirements, and different types of workloads running on the same infrastructure. As a result, newer benchmark suites aim to model production behavior more accurately. DCPerf follows this direction by providing workloads inspired by real datacenter applications and by collecting both application-level and system-level metrics [1], [2].

The following sections discuss CloudSuite, TailBench, and DCPerf, which are the benchmark suites most relevant to this thesis.

3.2 CloudSuite

CloudSuite is a benchmark suite designed to evaluate scale-out cloud workloads [5]. It was one of the earlier benchmark suites that focused on cloud services instead of only traditional CPU or memory microbenchmarks. Its workloads represent common cloud application categories, such as web search, data caching, graph analytics, media streaming, and other service-oriented workloads.

The main contribution of CloudSuite is that it helped make cloud workload evaluation more realistic and reproducible. Instead of measuring only isolated processor performance, CloudSuite evaluates applications that behave more like real cloud services. This makes it useful for studying how modern servers handle scale-out workloads, where many client requests must be processed at the same time.

CloudSuite is also useful in academic environments because its workloads are open and reproducible. Researchers can use it to compare system configurations, study architectural bottlenecks, and evaluate cloud-service behavior under controlled conditions. In this sense, CloudSuite helped establish the importance of workload realism in computer systems benchmarking.

However, CloudSuite also has some limitations when compared with newer production-oriented benchmark suites. Its workloads are based on open-source applications and benchmark configurations, which may not fully capture the complexity of modern hyperscale datacenter services. This can limit how closely it represents production behavior, especially in terms of concurrency, memory pressure, and production-specific software configurations.

For this thesis, CloudSuite is relevant because it represents an earlier generation of cloud benchmarking. It provides background for understanding why benchmark suites evolved toward more realistic and production-inspired workloads such as DCPerf and TaoBench.

3.3 TailBench

TailBench is a benchmark suite focused on latency-critical applications [6]. Unlike benchmarks that mainly evaluate total throughput, TailBench emphasizes response-time behavior, especially tail latency. Tail latency refers to the slowest requests in the system and is usually measured using percentiles such as p95 or p99.

This focus is important because many modern online services are interactive. In these systems, average latency may look acceptable, but a small percentage of very slow requests can still negatively affect user experience. For example, if a service handles thousands of requests per second, even 1% slow requests can affect many users. Therefore, measuring only average latency is not enough.

TailBench includes workloads that represent latency-sensitive applications such as search, speech recognition, machine translation, and image recognition. These workloads are useful for studying service-level objectives, request scheduling, quality of service, and tail-latency behavior. Its main value is that it highlights the importance of worst-case or near-worst-case response times in systems evaluation.

TailBench is related to this thesis because TaoBench also reports latency metrics, including high-percentile latency values such as p99 and p99.9. However, the focus of the two benchmark suites is different. TailBench is mainly designed for latency-critical workload analysis, while DCPerf and TaoBench are used in this thesis to study throughput, latency, hit rate, and utilization in a production-inspired caching workload.

For this reason, TailBench provides useful background for understanding why tail latency is important in this thesis, especially when analyzing TaoBench saturation under increasing client load.

3.4 Summary of Related Work

The benchmark suites discussed in this chapter show how datacenter performance evaluation has evolved over time. CloudSuite helped move evaluation beyond simple synthetic benchmarks by introducing more realistic scale-out cloud workloads. TailBench highlighted the importance of latency-sensitive applications and tail-latency behavior. Together, these benchmark suites show that modern system evaluation should consider not only average performance, but also realistic workloads and latency behavior.

Overall, the related work shows that realistic benchmarking is important for understanding modern cloud and datacenter systems. These ideas provide the background for this thesis, which focuses on evaluating a production-inspired datacenter workload. The next chapter presents DCPerf and TaoBench in detail, since they form the main benchmarking framework and experimental workload used in this thesis.

Chapter 4: DCPerf Framework and TaoBench Workload

4.1 Overview of DCPerf

DCPerf is an open-source benchmark suite designed to evaluate the performance of modern datacenter systems using realistic, production-inspired workloads [1], [2]. Unlike traditional synthetic benchmarks, which often focus on isolated CPU or memory operations, DCPerf aims to model the behavior of real large-scale services. It includes workloads that represent applications such as feed generation, video transcoding, web services, data analytics, and caching systems [1], [2]. This makes DCPerf suitable for studying how server hardware and software behave under conditions that are closer to real datacenter environments.

The main objective of DCPerf is to provide a realistic and reproducible framework for evaluating datacenter workloads. It collects both application-level and system-level metrics, such as throughput, latency, CPU utilization, memory behavior, and cache behavior. These measurements allow researchers to analyze not only how fast a system performs, but also how efficiently it uses its available resources. In this way, DCPerf can be used to compare different server configurations, study possible bottlenecks, and evaluate how workload characteristics affect system performance.

A key strength of DCPerf is that its workloads are inspired by applications and configurations used in production datacenters. This improves the usefulness of the benchmark results, because the workloads are designed to reflect realistic concurrency, memory access patterns, and request processing behavior. As a result, DCPerf provides a useful framework for both academic research and practical performance evaluation of datacenter systems.

In this thesis, DCPerf is used as the benchmarking framework for evaluating TaoBench, one of its included workloads. The focus is not on evaluating all DCPerf workloads, but on studying the performance behavior of TaoBench under different client loads and server thread configurations.

4.2 Workloads Included in DCPerf

DCPerf includes several workloads that represent different types of datacenter applications [1], [2]. Each workload stresses different parts of the system, such as CPU processing, memory bandwidth, network communication, storage activity, or request scheduling. This variety allows DCPerf to provide a wider view of datacenter performance compared to benchmarks that focus on only one application type.

Examples of workloads included in DCPerf are FeedSim, VideoTranscodeBench, MediaWiki, SparkBench, DjangoBench, and TaoBench. These workloads cover different application categories. FeedSim represents social media feed generation, where users request personalized content. VideoTranscodeBench represents video encoding and transcoding pipelines, which are CPU-intensive and important for media services. MediaWiki represents collaborative web applications with database-backed content access. SparkBench models large-scale data analytics workloads using Apache Spark, while DjangoBench represents backend web service request handling using the Django framework.

TaoBench, which is the main workload studied in this thesis, represents a caching workload inspired by Meta's TAO system [3], [4], [10]. It models key-value cache behavior and is used to evaluate throughput, latency, hit rate, and server-side processing efficiency. Compared to some other DCPerf workloads, TaoBench is practical to deploy and analyze, while still representing an important class of datacenter workloads high-throughput, low-latency caching systems.

The workloads in DCPerf are useful because modern datacenters do not run only one type of application. Some workloads are compute-intensive, some are memory-intensive, some are latency-sensitive, and others are affected by networking or synchronization overhead. Therefore, DCPerf provides a framework for evaluating systems under a range of realistic conditions. In this thesis, TaoBench is selected as the primary workload because it allows controlled experimentation with client load, cache behavior, and fast/slow server thread configurations.

4.3 Overview of TaoBench

TaoBench is a benchmark workload included in DCPerf that simulates the behavior of a large-scale distributed caching system [3], [10], [11]. It is inspired by Meta's TAO system, which is used to serve social graph data efficiently at large scale [4]. In production systems, caching layers are important because they reduce the need to repeatedly access slower backend storage systems. By serving frequently requested data from memory, caching systems can improve throughput and reduce response latency.

In this thesis, TaoBench is used as a representative read-heavy caching workload. It generates key-value requests over a large key space and makes it possible to study how a caching system behaves under different client loads and server-side configurations. This is useful for evaluating throughput, average latency, tail latency, cache hit ratio, and resource utilization.

A key characteristic of TaoBench is that most requests are expected to be served from the cache. In the experiments of this thesis, the target hit ratio is approximately 0.9. This means that most requests follow the fast cache access path, while a smaller fraction exercises slower request processing. As a result, TaoBench is suitable for studying both high-throughput cache behavior and the performance impact of a smaller number of slower requests.

TaoBench reports several important metrics, including total queries per second, fast-path throughput, slow-path throughput, cache hit ratio, average latency, and tail latency. These metrics are used in the experimental analysis to understand how different client loads and server thread ratios affect the overall behavior of the system.

4.4 Comparison of TaoBench with Related Benchmark Suites

TaoBench is a specific workload within the broader DCPerf framework. While CloudSuite, TailBench, and DCPerf are benchmark suites, TaoBench is a workload that focuses specifically on caching behavior [1], [3], [10]. It models a key-value caching system inspired by Meta's TAO infrastructure [3], [4] and is designed to reproduce high-throughput, low-latency cache access patterns.

CloudSuite is broader in scope because it includes different types of cloud workloads [5]. It is useful for evaluating scale-out cloud services in general. TaoBench, on the other hand, focuses on the cache tier. This makes it narrower in scope, but more suitable for detailed analysis of cache hit rate, fast-path processing, slow-path behavior, and thread configuration.

TailBench has a different focus. It is mainly designed for latency-critical applications and tail-latency analysis [6], [7]. TaoBench is not mainly a tail-latency benchmark, but tail latency is still important in its evaluation. Cache misses, queueing, and server-side contention can increase p99 and p99.9 latency. For this reason, TailBench provides useful background for understanding why tail latency should be included in the evaluation, while TaoBench provides the workload evaluated in this thesis.

DCPerf provides the overall benchmarking framework, while TaoBench provides the specific caching workload used for the experimental analysis [1], [2], [10]. This thesis focuses on TaoBench because it is practical to deploy, directly related to datacenter caching systems, and suitable for controlled experiments involving client load and fast/slow thread configuration.

TaoBench is appropriate for the goals of this thesis because it generates key-value request streams and reports metrics such as throughput, latency, tail latency, hit ratio, and CPU utilization. These metrics make it possible to study how changes in client load and server-side thread allocation affect the performance of a datacenter caching workload.

4.5 TaoBench Architecture

Figure 4.1 presents the high-level architecture of TaoBench as used in this thesis.

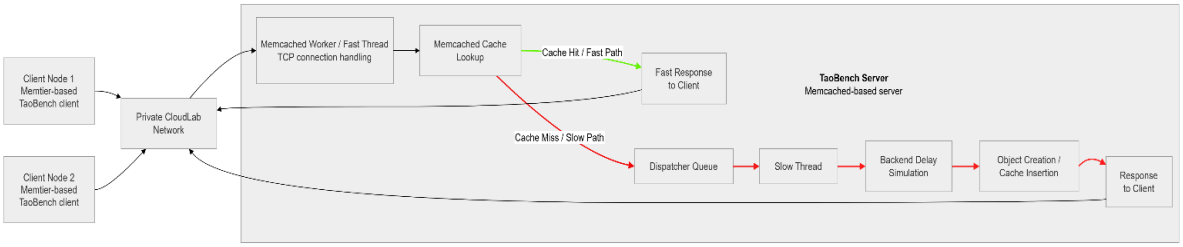

Figure 4.1: High-level TaoBench architecture showing the Memtier-based client nodes, the private CloudLab network, and the Memcached-based TaoBench server with fast-path and slow-path request processing.

The TaoBench setup used in this thesis follows a client-server architecture. The client side consists of two TaoBench client nodes connected to the server through the private CloudLab network. Each client runs a modified workload generator that creates read-dominated key-value requests and sends them to the TaoBench server over TCP connections. In the experiments, the client-side offered load is controlled mainly through the CLIENTS_PER_THREAD parameter.

On the server side, TaoBench is based on a Memcached-like caching server. Incoming requests are first handled by the server's request-processing threads, which perform the cache lookup. If the requested item is found in the cache, the request follows the fast path and a response is returned directly to the client. This path corresponds to cache hits and represents the common case in the workload, since the target hit ratio is approximately 0.9.

If the requested item is not found in the cache, the request follows the slow path. In this path, the request is passed through the dispatcher queue and handled by slow-processing threads. The slow path simulates the additional work that would normally be required for a backend lookup or data generation. After this simulated backend work is completed, the result can be inserted into the cache and a response is returned to the client.

This separation between fast-path and slow-path processing is central to the experiments in this thesis. TaoBench exposes server-side configuration parameters that affect how processing capacity is divided between these two paths. In particular, FAST_THREADS_RATIO affects the amount of processing capacity dedicated to fast-path requests, while SLOW_TO_FAST_RATIO controls the relative slow-path capacity. By

varying these parameters, the experiments evaluate how server-side resource allocation affects throughput, latency, tail latency, hit ratio, and resource utilization.

The architecture also helps explain the saturation behavior observed in the results. As client load increases, more requests arrive at the server and the pressure on the request-processing paths increases. Once the server reaches its efficient operating region, additional client load does not significantly increase throughput. Instead, it mainly increases queueing delay and tail latency. This behavior is analyzed in detail in the experimental results chapter.

4.5.1 **TaoBench Server**

The TaoBench server is responsible for receiving client requests, processing key-value operations, and producing the main server-side performance metrics. It is based on a modified version of Memcached, extended to support TaoBench-specific behavior [9], [10]. The server stores data in memory and responds to client requests either by serving items directly from the cache or by forwarding them through the slow-path mechanism when a cache miss occurs. During execution, the server uses several thread groups. Fast threads handle cache-hit requests, slow threads handle simulated backend operations, and dispatcher threads coordinate the movement of slow-path requests between queues. This structure is important because it allows TaoBench to model the behavior of a caching system where most requests are served quickly, while a smaller fraction requires additional processing. The number of fast and slow threads depends on configuration parameters such as FAST_THREADS_RATIO and SLOW_TO_FAST_RATIO.

The server also collects metrics during the benchmark run. These include fast QPS, slow QPS, total QPS, hit ratio, and the number of valid measurement samples. These metrics are later used to evaluate the behavior of each run and to compare different configurations. In the experiments of this thesis, the server was also monitored externally to collect system-level utilization metrics, such as CPU utilization, memory usage, network receive/transmit throughput, and disk activity.

4.5.2 TaoBench Clients

The TaoBench clients are responsible for generating the workload sent to the server, following the distributed TaoBench setup described in the DCPerf TaoBench documentation [10]. Each client creates multiple worker threads and network connections, depending on the selected load parameter. The clients issue key-value requests to the server and measure client-side performance metrics such as throughput, average latency, p50 latency, p99 latency, and p99.9 latency.

The client-side component is important because it controls the amount of pressure applied to the server. In the experiments, the client load is mainly controlled through the CLIENTS_PER_THREAD parameter. This parameter does not directly represent the total number of clients. Instead, it defines how many connections are created per client worker thread.

For example, if a client runs 26 worker threads and the selected load is 50 clients per thread, then that client creates:

$26 \times 50 = 1300$ connections

With two client nodes, the total number of connections becomes:

$1300 \times 2 = 2600$ connections

The clients also participate in the warm-up phase. During warm-up, they generate requests that help populate the cache and bring the system to a stable state before measurement begins. This is important because measuring performance before the cache is populated would produce low hit rates and unstable results. In this thesis, the warm-up phase was modified so that all experiments start from a more comparable cache state.

4.5.3 Request Processing Flow

Figure 4.2 shows the request processing flow in TaoBench for both cache-hit and cache-miss requests.

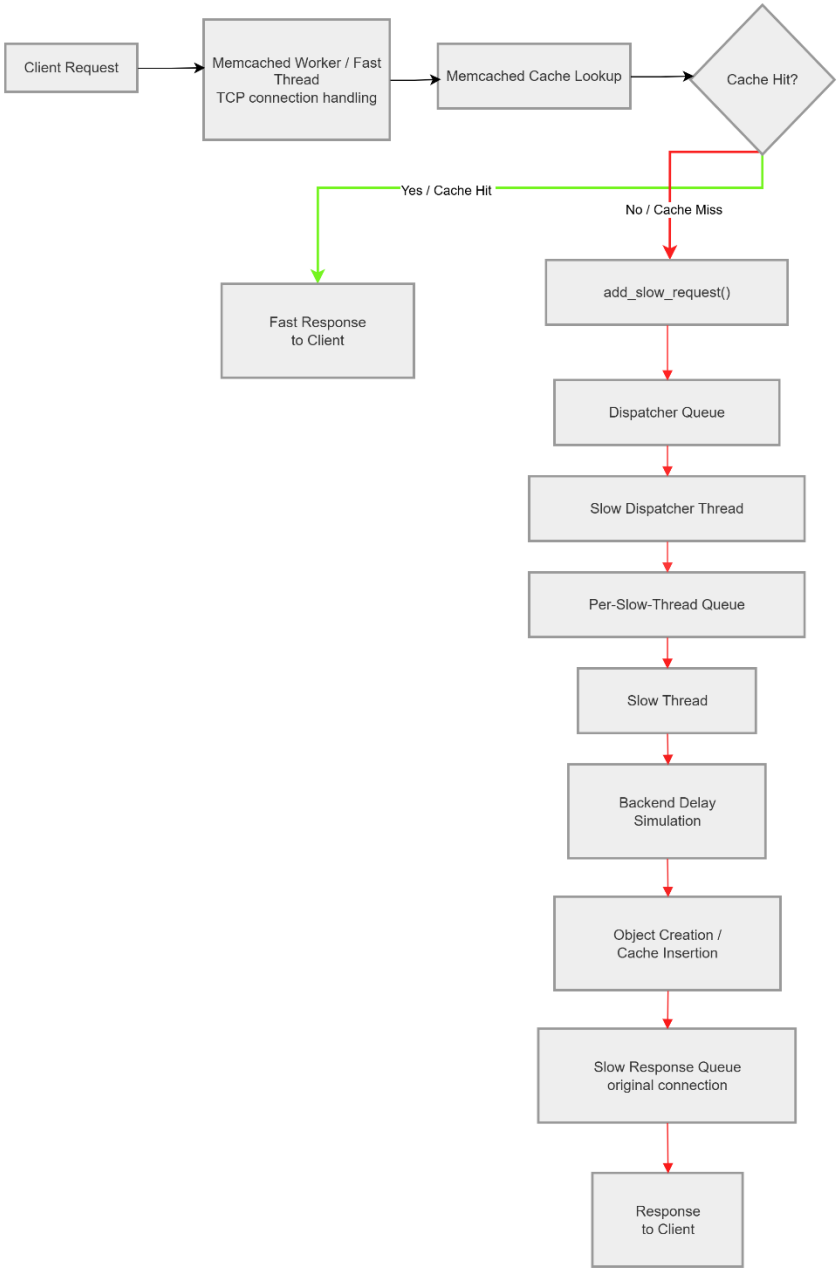

Figure 4.2: TaoBench request processing flow. Cache-hit requests are served directly by the Memcached worker/fast-thread path, while cache-miss requests are dispatched through the slow-path mechanism before the response is returned to the client.

The request processing flow in TaoBench begins when a client sends a key-value request to the server.

The request arrives through a TCP connection that is handled by a Memcached worker thread, which

is part of the fast-path side of TaoBench. This worker thread performs the cache lookup and determines whether the request is a cache hit or a cache miss.

If the item is found in the cache, the request is considered a cache hit and is handled through the fast path. The server retrieves the item from memory and sends the response back to the client. This is the dominant case when the hit ratio is close to the target value of approximately 0.9.

If the requested item is not found in the cache, the request is treated as a cache miss and follows the slow path. In the DCPerf version of TaoBench used in this thesis, cache misses are not served by an external database or a real backend storage system. Instead, backend-like behavior is represented inside the benchmark through a synthetic slow-path mechanism. The cache-miss request is inserted into the slow path using add_slow_request(). It is first placed in a dispatcher queue and then forwarded by a slow dispatcher thread to a per-slow-thread queue. A slow thread then processes the request by performing backend-like work, such as object generation, cache insertion, and response handoff back to the original connection.

An important detail is that TaoBench can also emulate additional backend lookup delay through a configurable parameter, tao_slow_path_sleep_us. This parameter can add a fixed sleep time, in microseconds, to slow-path requests. However, in the experiments of this thesis, this value was set to zero. Therefore, no additional fixed backend-delay sleep was added during the measurements.

As a result, the slow-path overhead observed in this work mainly comes from TaoBench's internal slow-path behavior, including queueing, dispatcher and slow-thread scheduling, object generation, optional compression, cache insertion, and response handoff. For this reason, the results should be interpreted as the performance of TaoBench's in-memory cache and slow-path processing model, rather than as the performance of a complete storage system with a real external database. The experiments do not separately model realistic database latency or cross-region access latency.

TaoBench also includes a separate option related to how slow threads wait for incoming work. In the experimental autoscale-v2 configuration, the slow_threads_use_semaphore parameter can make slow threads use semaphores instead of repeatedly waiting with nanosleep(). This parameter does not define the amount of backend delay. Instead, it affects the implementation overhead of the slow-path waiting mechanism. Therefore, tao_slow_path_sleep_us is the parameter related to explicit backend-delay simulation, while slow_threads_use_semaphore is related to slow-thread waiting and scheduling behavior.

This request flow is important because it allows TaoBench to model both the high-throughput behavior of cache hits and the performance impact of cache misses. Since the measured hit ratio in this thesis remains close to 0.9, most requests are handled by the fast path. Therefore, increasing fast-

path processing capacity can improve throughput. However, the slow path remains important, because around 10% of requests still require slow-path processing and can affect tail latency.

4.6 Fast Path and Slow Path

A central design feature of TaoBench is the separation between fast-path and slow-path request processing [3], [10]. This separation allows the benchmark to model the behavior of a realistic caching system, where most requests are served directly from memory, while a smaller fraction requires additional processing.

The fast path represents cache-hit requests. When a requested key is found in the server's in-memory cache, the request can be served immediately by retrieving the corresponding value and returning it to the client. This path is expected to have low latency and high throughput, because it avoids expensive backend operations. In a well-warmed TaoBench run, the majority of requests follow this path.

The slow path represents cache-miss requests. When the requested item is not available in the cache, the request cannot be completed directly through the fast path. Instead, TaoBench forwards the request to a slow-path mechanism that simulates the delay of accessing a backend system, such as a database or remote storage service. This does not mean that TaoBench necessarily connects to a real database. Instead, it models the effect of backend delay and thread waiting behavior, which is important for reproducing more realistic server-side performance characteristics.

This distinction is important because real datacenter caching systems are not limited only by raw memory lookup speed. They are also affected by queueing, synchronization, request dispatching, backend delays, and thread scheduling. TaoBench captures this behavior by reporting separate metrics for fast-path and slow-path throughput, usually shown as fast_qps and slow_qps. The total throughput of the system is the combination of both paths.

In the experiments of this thesis, the target cache hit ratio is approximately 0.9. This means that around 90% of requests are expected to be served through the fast path, while around 10% follow the slow path. This ratio is important because it creates a realistic workload balance: the workload is dominated by fast cache hits, but still includes enough slow-path requests to stress backend-like processing and affect tail latency.

The separation between fast and slow paths is also directly related to the thread ratio experiments performed later in this thesis. Since the workload is mostly cache-hit dominated, allocating more processing capacity to the fast path can improve throughput. However, if the slow path is under-provisioned, cache misses may experience queueing delays and increase tail latency. Therefore, the balance between fast and slow processing is one of the main performance factors studied in this work.

4.7 Fast Threads and Slow Threads

TaoBench implements fast-path and slow-path processing using different groups of server threads [3], [10]. These thread groups are configured according to the number of available CPU cores and the selected thread ratio parameters. The main thread categories are fast threads, dispatcher threads, and slow threads.

Fast threads are responsible for handling client requests and serving cache-hit requests. These threads process the dominant part of the workload, since most requests are expected to hit in the cache. Their work includes receiving the request, performing the cache lookup, retrieving the value from memory, and sending the response back to the client. Because cache hits are frequent and latency-sensitive, the number of fast threads has a strong effect on overall throughput.

Slow threads are responsible for handling cache-miss requests. When a request misses in the cache, it is not completed by the fast worker thread. Instead, the worker thread adds the request to the slow-path queue using add_slow_request(). The specific client request then waits until the slow-path processing is completed. The worker thread itself does not perform the slow work; after forwarding the request, it can continue handling other ready client requests. Slow threads simulate backend processing delay and eventually return the response through the appropriate connection. Although slow-path requests are fewer than fast-path requests, they are important because they can affect tail latency and system stability.

Dispatcher threads coordinate the movement of slow-path requests between the fast threads and the slow threads. When a fast thread identifies a cache miss, the request is inserted into a dispatcher queue. There is no fixed mapping between fast threads and dispatchers. Instead, dispatcher selection is based on a per-connection round-robin mechanism. This allows cache-

miss requests to be distributed across dispatcher queues without requiring strict centralized coordination.

Each dispatcher then forwards slow-path requests to slow-thread queues. The dispatchers use local scheduling to distribute requests across slow threads. This design keeps fast threads from blocking on backend-like work and allows cache-hit requests to continue being processed efficiently. It also helps the system balance slow-path work while keeping synchronization overhead relatively low.

The number of threads in each category is controlled by configuration parameters. In TaoBench, the fast thread count is derived from the number of CPU cores and the selected FAST_THREADS_RATIO. The slow thread count is then derived from the number of fast threads and the selected SLOW_TO_FAST_RATIO. Dispatcher threads are also derived from the fast thread count. This thread model is central to TaoBench's execution behavior because it determines how much processing capacity is allocated to cache hits, cache misses, and request dispatching.

This thread structure is important for performance analysis. If too few fast threads are available, cache-hit requests may queue and throughput may be limited. If too many slow threads are created, the system may suffer from scheduling overhead, queue contention, or synchronization overhead. Therefore, the thread configuration must match the workload characteristics.

4.8 Thread Ratio Configuration

TaoBench exposes parameters that control how server-side processing capacity is divided between fast-path and slow-path work [10]. The two most important parameters for this thesis are FAST_THREADS_RATIO and SLOW_TO_FAST_RATIO.

The FAST_THREADS_RATIO parameter determines the number of fast threads relative to the available CPU cores. A higher value allocates more processing capacity to fast-path requests. Since TaoBench is configured with a high cache hit ratio, increasing the fast thread ratio can improve throughput when the workload is dominated by cache hits. However, adding more fast threads also changes the overall balance of the system and may introduce overhead after a certain point.

The SLOW_TO_FAST_RATIO parameter determines how many slow threads are created for each fast thread. Slow threads are used to handle cache misses and simulate backend-like processing. A higher slow-to-fast ratio provides more slow-path capacity, but it can also introduce overhead if too many slow threads are created.

Dispatcher threads are also part of the TaoBench server thread model. They coordinate slow-path requests by moving cache-miss requests from dispatcher queues to slow-thread queues. In this thesis, dispatcher threads are not treated as a separate experimental parameter. However, they are still relevant because they are part of the internal request flow between the fast and slow paths.

The thread counts can be understood using the simplified relationships shown in Table 4.1.

Table 4.1: Simplified calculation of TaoBench server thread counts.

Thread type	Simplified relationship
Fast threads	fast_threads = CPU_cores × FAST_THREADS_RATIO
Dispatcher threads	dispatcher_threads = fast_threads × DISPATCHER_TO_FAST_RATIO
Slow threads	slow_threads = fast_threads × SLOW_TO_FAST_RATIO

Table 4.2: Example thread count calculation on a 32-logical-CPU system.

Parameter / Thread type	Calculation	Result
Fast threads	32 × 0.75	24
Dispatcher threads	24 × 0.25	6
Slow threads	24 × 3	72

Thread ratio configuration is one of the main experimental variables in this thesis. After studying the effect of client load on TaoBench performance, the experiments evaluate how different fast/slow thread combinations affect throughput, latency, and resource utilization. This is important because a default or previous configuration may not be optimal for every workload or hardware setup.

Since the target hit ratio is approximately 0.9, most requests are served by the fast path. Therefore, configurations with more fast-path capacity may perform better for the selected workload. However, the best configuration is not necessarily the one with the highest total number of threads. Adding too many threads can increase scheduling overhead, queue contention, and synchronization costs.

In the ratio sweep experiments, this trade-off is important. Configurations with too many slow threads may use more CPU without producing higher useful throughput. Therefore, thread ratio configuration provides a way to study the balance between fast-path capacity, slow-path capacity, and server-side overhead.

4.9 Warm-up and Execution Phases

TaoBench experiments are divided into two main phases: the warm-up phase and the execution, or measurement, phase. This separation is essential for obtaining meaningful and reproducible performance results.

The warm-up phase prepares the system before measurement begins. During this phase, the cache is populated and the system moves from a cold state toward a steady state. At the beginning of a run, the cache contains little or no useful data, so many requests may miss in the cache. If measurements were collected immediately, the results would not represent steady-state caching behavior. Instead, they would mainly reflect cache population effects. For this reason, the warm-up phase is not used as the final measurement window.

A successful warm-up should bring the cache hit ratio close to the target value. In the experiments of this thesis, the target hit ratio is approximately 0.9. This means that the system should reach a condition where around 90% of requests are served from the cache. Warm-up is also important because it helps stabilize the server threads, network connections, and CPU behavior before measurement begins.

The execution phase is the actual measurement period. During this phase, throughput, latency, hit ratio, and utilization metrics are collected and analyzed. This phase represents the steady-state behavior of the system under the selected experimental configuration. In this thesis, the measurement phase is used to compare different client load levels and different fast/slow thread ratio configurations.

One important issue identified during the experimental work was that using the same load for both warm-up and measurement can make comparisons unfair. For example, a run with low client load would warm up the cache more slowly than a run with high client load. As a result, different experiments could start their measurement phase from different cache states. To address this, a fixed warm-up methodology was introduced in the experimental scripts. In this approach, every run uses the same warm-up load before switching to the selected measurement load.

This fixed warm-up methodology improves comparability across experiments. It ensures that the measurement phase begins from a similar cache state, regardless of the selected client load. In the experimental results, this was important for comparing low-load and high-load configurations fairly. The fixed warm-up approach is described in more detail in Chapter 5, while its impact is discussed in Chapter 6.

Overall, the warm-up and execution phases are critical for TaoBench evaluation. Without proper warm-up, the benchmark may report unstable hit rates, misleading throughput, and inflated latency. With a stable warm-up, the measurements better reflect the steady-state behavior of the caching system.

Chapter 5: Experimental Methodology

5.1 Experimental Environment

The experiments in this thesis were conducted using CloudLab [8]. CloudLab was selected because it provides access to real physical machines and allows the creation of repeatable experimental topologies. This is important for performance evaluation, because benchmarks such as TaoBench can be affected by hardware resources, network behavior, CPU utilization, and system configuration.

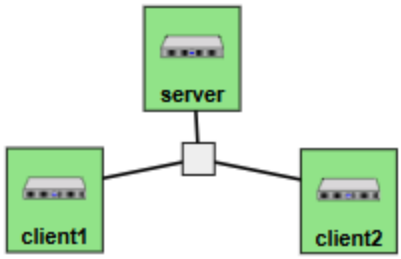

Figure 5.1: CloudLab experimental topology with one TaoBench server and two client nodes.

Figure 5.1 shows the CloudLab topology used in the experiments. The setup consists of one TaoBench server node and two client nodes connected through the private CloudLab network.

The experimental setup used a three-node topology. One node was used as the TaoBench server, while two additional nodes were used as clients. The clients generated requests toward the server and were used to apply different load levels. This distributed setup allowed TaoBench to be evaluated under networked conditions, instead of running the server and clients on the same machine.

The purpose of this environment was to reproduce a simplified datacenter-style client-server setup. Although the setup is much smaller than a real production datacenter, it is sufficient for studying important performance trends such as throughput, latency, cache hit behavior, and server-side bottlenecks.

5.1.1 CloudLab Infrastructure

CloudLab is a research platform that allows users to allocate physical machines for networking, cloud computing, and systems experiments [8]. Unlike virtualized public cloud environments, CloudLab provides more control over the experimental infrastructure. This makes it suitable for benchmarking studies where reproducibility and low-level system behavior are important.

For this thesis, CloudLab was used to create a private experimental topology consisting of one server node and two client nodes. All nodes were connected through a private network. This allowed the TaoBench traffic to remain separate from external network traffic and made the measurements more controlled.

Using CloudLab also made it possible to repeat the experiments with the same topology and configuration. This was important because the thesis required multiple runs for each configuration, especially for the load sweep and the final ratio plateau sweep. Repeating the experiments helped reduce the effect of random variation and improved confidence in the final results.

5.1.2 Server and Client Nodes

The TaoBench experiments used three nodes with fixed roles:

Table 5.1: Server and client nodes used in the TaoBench experiments.

Node name	Role	Private IP address
server	TaoBench server	192.168.1.10
client1	TaoBench client	192.168.1.11
client2	TaoBench client	192.168.1.12

The server node was responsible for running the TaoBench server process. It handled incoming client requests, maintained the in-memory cache, processed fast-path and slow-path operations, and generated server-side metrics such as fast QPS, slow QPS, total QPS, and hit ratio.

The two client nodes were responsible for generating the workload. Each client executed the TaoBench client component and created multiple worker threads and connections. Both

clients were started for each experiment so that the server received traffic from two independent sources. This setup was used consistently across all final experiments.

Using two client nodes helped ensure that the server could be stressed sufficiently. It also made it possible to check whether the clients were becoming bottlenecks. In the utilization analysis, client CPU usage was monitored separately from server CPU usage. This helped determine whether performance limitations were caused by the clients or by server-side processing.

5.1.3 Hardware Configuration

The experiments were executed on CloudLab machines with x86_64 architecture. The server used 32 logical CPUs, which was important for configuring TaoBench's fast and slow thread ratios. In TaoBench, the number of server threads is derived from the logical CPU count and the selected thread ratio parameters, as described in Chapter 4.

The main hardware and configuration values used in the experiments are shown in Table 5.2.

Table 5.2: Main hardware and configuration values used in the experiments.

Parameter	Value
Architecture	x86_64
Server logical CPUs	32
TaoBench memory size (memsize)	16 GB
NUM_CLIENTS	2
Default server port	11211

The TaoBench memory size was fixed at 16 GB in all experiments. This value refers to the benchmark memory configuration, or memsize, used by TaoBench, rather than the total physical memory of the CloudLab server. Keeping this value fixed was important because changing the TaoBench memory size would also change the cache capacity and could affect the cache hit ratio. By keeping the benchmark memory size fixed at 16 GB, the experiments focused on the impact of client load and thread configuration rather than changes in cache size.

For example, in the final best observed configuration, FAST_THREADS_RATIO = 5 produced approximately 160 fast threads on the 32-logical-CPU server. With

SLOW_TO_FAST_RATIO = 0.125, this resulted in approximately 20 slow threads. This relationship was important for the final thread ratio experiments, where different allocations of fast and slow server threads were evaluated.

5.1.4 Network Configuration

The server and clients communicated through the private CloudLab network. The server used the private IP address 192.168.1.10, while the clients used 192.168.1.11 and 192.168.1.12. The TaoBench clients connected to the server using the configured server IP address and server port.

The default TaoBench server port used in the experiments was: 11211. This is the standard Memcached port and was used for client-server communication. Network utilization was monitored during the experiments. The collected metrics included server receive throughput, server transmit throughput, and the corresponding client-side traffic. This made it possible to check whether network traffic was consistent across nodes. For example, the total traffic transmitted by the clients should approximately match the traffic received by the server.

Network monitoring was also useful for bottleneck analysis. If the network interface had become saturated, throughput could have been limited by network capacity. However, the final utilization results showed that the clients were not saturated and that the main bottleneck was more likely related to server-side processing and thread scheduling.

5.2 Software Setup

The software setup was designed to make the TaoBench experiments easy to repeat and consistent across runs. A separate experiment repository was created to organize the CloudLab profile, setup scripts, execution scripts, documentation, and monitoring utilities used throughout the thesis. The repository includes automated bootstrap scripts for the server and clients, client launch automation, and configuration files for running TaoBench through DCPerf on CloudLab.

For reproducibility, the scripts and configuration files used in this thesis are available in a public GitHub repository [12]. This repository contains the CloudLab profile used to create the three-node topology, setup documentation, bootstrap scripts for the server and client nodes, TaoBench launch scripts, load sweep and fast/slow ratio experiment scripts, utilization monitoring scripts, and the fixed warm-up patch used during the experiments. It was used to organize the experimental workflow and make the setup easier to repeat.

The experiments used Ubuntu 22.04 on x86_64 CloudLab nodes. DCPerf was installed on the server and client nodes, and the environment was configured using a Python virtual environment. To avoid compatibility problems with TaoBench, NumPy was pinned to version 1.26.4. The system setup also included numactl and an increased open file limit using ulimit -n 65536, which is important for high-concurrency experiments.

A key reproducibility decision was to pin the DCPerf version to a specific commit: 9308c3e3c404e0466f0a2929f15ddcf62b2215f6. This ensured that all experiments used the same DCPerf codebase and avoided changes caused by future updates to the repository. During the bootstrap process, DCPerf was cloned at this pinned commit, the Python virtual environment was created, the warm-up patch was applied, and the runtime scripts were installed under ~/DCPerf/.

5.2.1 DCPerf Repository Setup

The DCPerf framework was installed on all three nodes: the server, client1, and client2. The setup process was automated through bootstrap scripts. On the server node, the bootstrap script prepared the DCPerf environment, installed the required dependencies, applied the TaoBench warm-up patch, and placed the experiment scripts under the ~/DCPerf/ directory. On the client nodes, the bootstrap script installed the required runtime environment and prepared the machines for running the TaoBench client workload.

The setup supported two installation methods. The recommended method was to clone the experiment repository and then run the appropriate bootstrap script on each node. The alternative method was to execute the bootstrap script directly using a one-line command. This made it easier to repeat the setup in fresh CloudLab experiments.

SSH connectivity from the server node to the two client nodes was also required, because the client launch script starts the TaoBench clients remotely. For this reason, SSH keys were configured between the server and the clients before running the distributed experiments. Connectivity was verified by testing SSH access from the server to both client nodes.

5.2.2 TaoBench Setup

TaoBench was installed as part of the DCPerf setup. The benchmark was executed using a distributed topology with one TaoBench server and two TaoBench clients. The server was

responsible for running the TaoBench server process, while the clients generated load and sent requests to the server over the private CloudLab network.

The TaoBench server was launched from the server node using the tao_server.sh script. This script used environment variables to configure important runtime parameters, such as the private LAN interface, server IP address, number of clients, memory size, warm-up duration, test duration, server port, and open file limit. Before starting the benchmark, the script checked that INTERFACE_NAME was set, activated the Python virtual environment, applied the configured open-file limit, and then ran benchpress_cli.py with the tao_bench_autoscale workload.

The clients were launched using the tao_client.sh script. This script configured the server hostname, server memory size, warm-up duration, test duration, server port, wait time after warm-up, open file limit, and CLIENTS_PER_THREAD. It then executed the TaoBench client workload and stored the client output and latency summary in the client_results directory.

The launch_clients.sh script was used to start both clients from the server node through SSH. It passed the required environment variables to both client machines and launched the client script on client1 and client2 in parallel. This helped ensure that both clients were started consistently and reduced the risk of manual timing errors.

5.2.3 Configuration Files and Scripts

A separate experiment repository was created to organize the configuration files, documentation, and automation scripts used in this thesis. The repository contains three main directories: cloudlab/, docs/, and scripts/. The cloudlab/ directory contains the CloudLab profile used to create the three-node experiment. The docs/ directory contains the experiment runbook, while the scripts/ directory contains the shell scripts used for node setup, TaoBench execution, experiment automation, utilization monitoring, and warm-up modification.

The repository structure includes the main files shown in Table 5.3.

Table 5.3: Experiment repository files and their purpose.

Directory / File	Purpose
cloudlab/profile.py	Defines the CloudLab three-node experiment topology.
docs/runbook.md	Contains setup and execution notes for the experiments.
scripts/bootstrap_common.sh	Installs common dependencies and prepares the shared environment.
scripts/bootstrap_server.sh	Prepares the server node and installs server-side scripts.
scripts/bootstrap_client.sh	Prepares the client nodes for running TaoBench clients.
scripts/tao_server.sh	Launches the standard TaoBench server.
scripts/tao_server_ratio.sh	Launches the TaoBench server with fast/slow thread ratio parameters.
scripts/tao_client.sh	Launches the TaoBench client workload.
scripts/launch_clients.sh	Starts both clients remotely through SSH.
scripts/run_all_experiments.sh	Runs the client load sweep experiments.
scripts/run_all_experiments_uti.sh	Runs load sweep experiments with utilization monitoring.
scripts/run_ratio_experiments.sh	Runs the fast/slow thread ratio sweep experiments.
scripts/run_ratio_experiments_uti.sh	Runs ratio experiments with utilization monitoring.
scripts/collect_utilization.sh	Collects CPU, memory, network, and disk utilization.
scripts/patch_warmup.sh	Applies the fixed warm-up modification to TaoBench.

The load sweep experiments were automated using run_all_experiments.sh. This script executed TaoBench across multiple CLIENTS_PER_THREAD values and repeated each measurement for a configurable number of runs. It also cleaned previous processes, started

the server and clients, collected logs, extracted client and server metrics, and generated summary.csv and per-load aggregate CSV files.

The utilization-enabled version, run_all_experiments_uti.sh, extended this workflow by collecting CPU, memory, network, and disk I/O activity from the server and both client nodes during each run. It started the utilization monitors before the benchmark execution, stopped them after the run completed, collected the utilization files from the client nodes, and stored the CSV files inside each run directory.

Disk I/O was collected only as an auxiliary system-level metric. In this setup, disk activity does not represent access to a real backend storage system, because TaoBench models backend-dependent behavior internally through the slow path. Any observed disk activity mainly comes from local experiment overheads, such as writing benchmark logs, utilization CSV files, and other run output files, as well as normal operating-system activity.

The collect_utilization.sh script was used to collect system-level metrics. It generated separate CSV files for CPU, memory, network, and disk activity, using node-specific filenames such as server_cpu.csv, client1_network.csv, and client2_disk.csv. These files were later used to analyze whether the bottleneck was related to the server, the clients, the network, or disk I/O.

The patch_warmup.sh script was used to apply the fixed warm-up methodology. This script modifies TaoBench's run.py so that the warm-up phase always uses a fixed CLIENTS_PER_THREAD value, independent of the measurement load. By default, this fixed warm-up load is 380 clients per thread. This modification was important because it helped ensure that the server cache was warmed in a consistent way before each measurement phase.

The fast/slow thread ratio experiments were automated using run_ratio_experiments.sh and run_ratio_experiments_uti.sh. These scripts executed TaoBench across different FAST_THREADS_RATIO and SLOW_TO_FAST_RATIO combinations. The utilization-enabled version also collected CPU, memory, network, and disk utilization during each ratio experiment. The tao_server_ratio.sh script was used to launch the TaoBench server with the selected fast/slow thread configuration. These scripts extended the same experimental workflow used in the load sweep experiments and were part of the final process used to identify the best server-side thread configuration.

5.3 Experimental Parameters

The TaoBench experiments were controlled through environment variables and script parameters, following the configurable options provided by the TaoBench workload [10]. This made the experimental process repeatable and allowed different configurations to be tested without manually changing the benchmark code for every run. The most important parameters controlled the memory size, number of clients, client load, warm-up duration, measurement duration, server port, open file limit, and fast/slow thread ratios.

The main fixed parameters used across the final experiments are shown in Table 5.4.

Table 5.4: Main fixed parameters used in the TaoBench experiments.

Parameter	Value
MEMSIZE	16 GB
NUM_CLIENTS	2
WARMUP	600 seconds
TEST	300 seconds
PORT	11211
OPEN_FILES_LIMIT	65536

The server node used the private IP address 192.168.1.10, while the client nodes used 192.168.1.11 and 192.168.1.12. These values were used consistently in the experiment scripts. The server and client scripts also allowed these values to be overridden using environment variables, which made the setup flexible for different CloudLab deployments.

5.3.1 Memory Size

The memory size allocated to TaoBench was fixed at 16 GB. This value was passed to the TaoBench server using the MEMSIZE_GB or MEMSIZE parameter, depending on the script. Keeping the memory size fixed was important because changing it would also change the cache capacity and could affect the hit ratio.

By using a fixed memory size, the experiments focused on the impact of client load and thread configuration rather than changes in cache size. This made the comparison between different runs more controlled.

5.3.2 Number of Clients

The experiments used two client nodes. This value was configured using:

NUM_CLIENTS = 2

Both clients generated workload toward the same TaoBench server. The two-client setup was used throughout the experiments to provide enough load to stress the server, while also allowing client utilization to be monitored separately. This helped determine whether the observed bottleneck was caused by the clients or by server-side processing.

5.3.3 Clients per Thread

The main load parameter in the experiments was CLIENTS_PER_THREAD. This parameter controls how many connections are created per client worker thread. Therefore, it does not directly represent the total number of clients in the experiment, but rather the number of connections created per worker thread on each client node.

For the load sweep experiments, different CLIENTS_PER_THREAD values were tested in order to observe how TaoBench behaves as the offered load increases. The final full load sweep included: 1, 5, 10, 25, 40, 50, 60, 75, 100, 200, 300, 400

Each load was repeated five times in the final load sweep. This allowed average throughput, latency, and variability to be calculated more reliably.

5.3.4 Warm-up Time

The warm-up phase was one of the most important parts of the methodology. TaoBench requires a warm-up phase because the cache must be populated before performance measurements are meaningful. Without sufficient warm-up, the hit ratio may be unstable and the benchmark may overrepresent slow-path behavior.

A fixed warm-up methodology was used in this thesis. The patch_warmup.sh script modifies TaoBench's run.py so that the warm-up phase always uses a fixed CLIENTS_PER_THREAD value. By default, this value is: WARMUP_CLIENTS_PER_THREAD = 380

This means that every run warms up with the same load, regardless of the measurement load. For example, a run with CLIENTS_PER_THREAD = 1 and a run with

CLIENTS_PER_THREAD = 400 both warm up using 380 clients per thread before switching to their selected measurement load.

This was important because using the same load for both warm-up and measurement would create different cache states across experiments. Low-load experiments would warm the cache more slowly, while high-load experiments would warm it more aggressively. The fixed warm-up patch helped ensure that all measurement phases started from a comparable cache state.

The final experiments used: WARMUP = 600 seconds. The official server logs confirmed that the hit ratio remained close to the expected value of approximately 0.9, indicating that the fixed warm-up methodology worked correctly.

5.3.5 Test Duration

The duration of the measurement phase was fixed at: TEST = 300 seconds**.** During this phase, performance metrics such as throughput, average latency, p50 latency, p99 latency, p99.9 latency, and hit ratio were collected. The 300-second measurement window provided enough time to observe steady-state behavior, while also keeping the total experiment time manageable.

5.3.6 Client Load

Client load was varied during the load sweep experiments. The purpose of the load sweep was to identify the load level that provided the best balance between throughput and latency. The final load sweep used five repetitions per load to improve confidence in the results.

The selected load for the final ratio experiments was: CLIENTS_PER_THREAD = 40

This load was chosen because it produced the best operating point in the load sweep: high throughput with relatively low average and tail latency. Higher loads did not significantly improve throughput and mainly increased latency.

5.3.7 Fast Threads Ratio

The FAST_THREADS_RATIO parameter controls the number of fast-path threads relative to the logical CPU count. Fast threads handle cache-hit requests, which represent the majority of requests in TaoBench when the hit ratio is close to 0.9.

This parameter was one of the main variables in the ratio sweep experiments. The initial broad ratio sweep tested lower and medium fast-thread ratios: FAST_THREADS_RATIO = 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 2.

After the broad sweep showed that higher fast-thread ratios improved throughput, an additional extension sweep tested: FAST_THREADS_RATIO = 2, 2.5, 3

Finally, the repeated plateau sweep focused on the high-fast-ratio region and tested: FAST_THREADS_RATIO = 3.5, 4, 4.5, 5, 6.

The goal was to identify whether increasing the fast-thread ratio would continue to improve throughput or whether the system would eventually reach a plateau.

5.3.8 Slow-to-Fast Threads Ratio

The SLOW_TO_FAST_RATIO parameter controls how many slow threads are created for each fast thread. Slow threads handle cache misses and simulate backend-like processing. Since only about 10% of requests are expected to follow the slow path, this parameter must be chosen carefully.

This parameter was varied across the ratio sweep experiments. In the initial broad ratio sweep, a wide range of slow-to-fast ratios was tested: SLOW_TO_FAST_RATIO = 0.25, 0.5, 1, 2, 3, 4

After the broad sweep showed that lower slow-to-fast ratios performed better, the extension sweep tested: SLOW_TO_FAST_RATIO = 0.25, 0.5, 1

Finally, the repeated plateau sweep focused on the lower slow-to-fast-ratio region and tested: SLOW_TO_FAST_RATIO = 0.125, 0.25, 0.5

The purpose was to evaluate whether a small number of slow threads was sufficient, and whether increasing slow-thread capacity introduced unnecessary overhead.

5.4 Experimental Procedure

Each experiment followed a structured procedure to ensure that the runs were repeatable and consistent. The same general workflow was used for both the load sweep experiments and the fast/slow thread ratio experiments.

Before each run, any previous TaoBench or Memcached processes were terminated on the server and client nodes. This cleanup step ensured that each run started from a clean state and that old benchmark processes did not interfere with the new measurements.

After the cleanup step, the TaoBench server was started on the server node. The server was launched using the selected memory size, warm-up time, test duration, server IP address, number of clients, port, and network interface. Before starting the benchmark, the server script also activated the Python virtual environment and set the open file limit.

The experiment then waited for a short startup period to allow the server to initialize. After this, the client processes were started remotely on both client nodes using SSH. Both clients used the same benchmark parameters and generated workload toward the server at the same time.

The benchmark then entered the warm-up phase. During this phase, the fixed warm-up patch forced the clients to use the fixed warm-up load of 380 clients per thread. This helped populate the cache and allowed the system to reach a stable hit ratio before measurement began.

After warm-up, the benchmark entered the measurement phase. During this phase, the clients switched to the selected experimental load. For the load sweep experiments, this was the current CLIENTS_PER_THREAD value being tested. For the ratio experiments, the client load was fixed at 40 clients per thread while the server thread configuration was varied.

After the measurement phase completed, the scripts collected server logs, client logs, latency summaries, and benchmark metric bundles. The scripts also extracted the most important metrics into CSV files, including client QPS, average latency, p50 latency, p99 latency, p99.9 latency, server total QPS, fast QPS, slow QPS, hit ratio, and number of data points.

For utilization-enabled experiments, CPU, memory, network, and disk monitoring were started before the clients and stopped after the benchmark completed. The utilization CSV files were collected from the server and both clients and stored inside each run directory.

Finally, a cooldown period was used before the next run. This helped reduce interference between consecutive experiments and made the repeated runs more consistent.

5.5 Metrics Collection

The experiments collected both application-level benchmark metrics and system-level utilization metrics. This was necessary because TaoBench performance cannot be understood only from throughput or latency. A complete analysis also requires observing how the server and clients use CPU, memory, network, and disk resources during execution.

Application-level metrics were collected from the TaoBench server and client logs. The client-side logs provided throughput and latency information, including average latency, p50 latency, p99 latency, and p99.9 latency. The server-side logs provided total QPS, fast QPS, slow QPS, hit ratio, and the number of valid measurement samples.

The main application-level metrics used in the analysis were total client QPS, server QPS, fast QPS, slow QPS, average latency, p50 latency, p99 latency, p99.9 latency, hit ratio, and number of measurement samples.

Client-side latency was especially important because it represented end-to-end request behavior. It included not only server processing time, but also network delay, possible queueing, and client-side effects. Tail latency metrics such as p99 and p99.9 were used to identify saturation and queueing effects under higher load.

In addition to benchmark metrics, utilization data was collected from all three nodes: the server, client1, and client2. The utilization monitoring was performed using the collect_utilization.sh script. This script collected CPU, memory, network, and disk measurements at a configurable interval. In the experiments, the interval was one second. The script wrote each metric group to a separate CSV file, using the node label as a filename prefix.

The utilization files generated for each node are summarized in Table 5.5.

Table 5.5: Utilization files generated during the experiments.

Node	CPU file	Memory file	Network file	Disk file
server	server_cpu.csv	server_memory.csv	server_network.csv	server_disk.csv
client1	client1_cpu.csv	client1_memory.csv	client1_network.csv	client1_disk.csv
client2	client2_cpu.csv	client2_memory.csv	client2_network.csv	client2_disk.csv

The CPU CSV file stored the timestamp, node label, and CPU utilization percentage. The memory CSV file stored memory utilization percentage. The network CSV file stored the selected interface, receive bytes per second, and transmit bytes per second. The disk CSV file stored the root device and disk read/write throughput in KB/s. These outputs allowed the analysis to compare application performance with system resource usage.

Network utilization was especially useful for checking traffic consistency. Since the clients send requests to the server, the total traffic transmitted by the clients should approximately match the traffic received by the server. Similarly, the traffic transmitted by the server should correspond to the traffic received by the clients. This helped confirm that the measurements were consistent across nodes.

The utilization-enabled experiment script, run_all_experiments_uti.sh, automatically started utilization monitoring on the server and both clients during each run. It then stopped the monitors after the benchmark completed and copied the client utilization CSV files back into the run directory. This ensured that every experiment run contained both benchmark logs and system utilization data.

5.6 Data Processing and Analysis

After each experiment, the collected logs and CSV files were processed to extract the final metrics used in the results chapter. The experiment scripts stored each run in a separate directory. Each run directory contained the server console output, client console outputs, client run logs, latency summaries, server metrics, benchmark metric bundles, run information, and utilization files.

For the load sweep experiments, the scripts created a central summary.csv file. This file contained one row per run and included the selected load, run number, run directory, client-side QPS and latency metrics, and server-side QPS and hit-ratio metrics. The scripts also generated per-load aggregate CSV files, which summarized the results for each tested CLIENTS_PER_THREAD value. Client metrics were extracted from the client logs, while server metrics were extracted from server logs or JSON metrics blocks.

For repeated experiments, average values were calculated across runs. In the final load sweep, each load was executed five times. This made it possible to compute average throughput, average latency, p99 latency, p99.9 latency, and QPS variability. This was important because single benchmark runs may be affected by temporary system noise or small execution differences.

For utilization analysis, only the measurement phase was considered. Since the utilization script collected data continuously during each run, the final analysis selected the samples corresponding to the 300-second measurement window. This avoided mixing warm-up behavior with steady-state measurement behavior. In practice, the last 300 utilization samples were used, because the sampling interval was one second and the test duration was 300 seconds.

The analysis separated the results into two main experimental groups. The first group was the client load sweep, where CLIENTS_PER_THREAD was varied while the server configuration remained fixed. This experiment was used to identify the best operating load and to study saturation behavior. The second group was the fast/slow thread ratio sweep, where the client load was fixed and the server-side thread ratios were varied. This experiment was used to study how server thread allocation affects performance.

The analysis focused on several aspects of the experiments. These included throughput comparison across different loads, average and tail latency comparison, hit-ratio stability, server and client CPU utilization, network RX/TX consistency, fast/slow thread ratio comparison, identification of the best observed configuration, and identification of throughput plateau behavior.

The hit ratio was verified using the official server logs. These logs confirmed that the hit ratio remained close to 0.9 during the experiments. This supported the fixed warm-up methodology and showed that the cache state was stable during the measurement phase.

The final results were interpreted using both performance metrics and utilization metrics. For example, if throughput stopped improving while latency increased, this indicated saturation. If server CPU utilization was high while client CPU utilization remained moderate, this suggested that the bottleneck was on the server side rather than the client side. If disk activity remained negligible, then disk I/O was not considered a limiting factor. This combined analysis helped identify the most likely causes of TaoBench performance behavior.

Chapter 6: Results and Evaluation

6.1 Baseline Experiment Results

Before performing the final load sweep and thread ratio experiments, baseline TaoBench experiments were executed to verify that the benchmark was running correctly on the CloudLab setup. The goal of these initial experiments was not to identify the best configuration, but to confirm that the distributed TaoBench deployment was functional, stable, and able to produce meaningful performance metrics.

The baseline setup used the same three-node topology described in Chapter 5, consisting of one server node and two client nodes. The server executed the TaoBench server workload, while the two clients generated requests toward the server. The benchmark was executed with a fixed memory size of 16 GB and a measurement duration of 300 seconds. The server and client logs were collected after each run and used to check throughput, hit ratio, and correct execution.

The initial runs showed that TaoBench could successfully execute in the distributed environment. The server produced valid metrics such as total QPS, fast QPS, slow QPS, and hit ratio, while the clients produced throughput and latency metrics. These experiments also confirmed that the benchmark could reach stable cache-hit behavior after warm-up.

An important observation from the baseline phase was that TaoBench performance depends strongly on cache warm-up. When the cache is not sufficiently populated before measurement, the hit ratio may be unstable and the results may not be comparable across runs. This led to the fixed warm-up methodology described in Chapter 5, where every experiment used a fixed warm-up load before switching to the selected measurement load.

Overall, the baseline experiments served three main purposes. First, they verified that the server and clients could communicate correctly over the private CloudLab network. Second, they confirmed that the benchmark produced the required application-level metrics. Third, they motivated the need for a more controlled methodology for repeated experiments, especially with respect to warm-up and hit-ratio stability.

6.2 Client Load Sweep Analysis

After validating the baseline setup, a full client load sweep was performed. The purpose of this experiment was to study how TaoBench performance changes as the offered load increases. The load was controlled using the CLIENTS_PER_THREAD parameter, which defines the number of client connections per worker thread on each client node.

The load sweep tested the following values:

CLIENTS_PER_THREAD = 1, 5, 10, 25, 40, 50, 60, 75, 100, 200, 300, 400

Each load was executed five times in order to improve confidence in the results and reduce the effect of random variation. All runs used the same fixed configuration: MEMSIZE = 16 GB, WARMUP = 600 seconds, TEST = 300 seconds, and NUM_CLIENTS = 2.

The fixed warm-up modification was active for all runs. Therefore, before each measurement phase, the benchmark first warmed up using a fixed warm-up load of 380 clients per thread. This helped ensure that all measurement phases started from a comparable cache state.

The aggregated load sweep results are shown in Table 6.1.

Table 6.1: Aggregated client load sweep results.

Load	***Runs***	***Avg Server QPS***	***Avg Latency***	***p99***
1	5	342,545	0.15 ms	0.40 ms
5	5	526,461	0.49 ms	1.70 ms
10	5	533,318	0.97 ms	2.84 ms
25	5	547,408	2.36 ms	6.25 ms
40	5	564,172	3.68 ms	9.58 ms
50	5	557,913	4.65 ms	12.01 ms
60	5	558,695	5.57 ms	14.46 ms
75	5	548,695	7.09 ms	18.60 ms
100	5	518,356	10.01 ms	27.25 ms
200	5	508,634	20.36 ms	66.71 ms
300	5	511,479	30.51 ms	113.30 ms
400	5	485,941	42.82 ms	167.12 ms

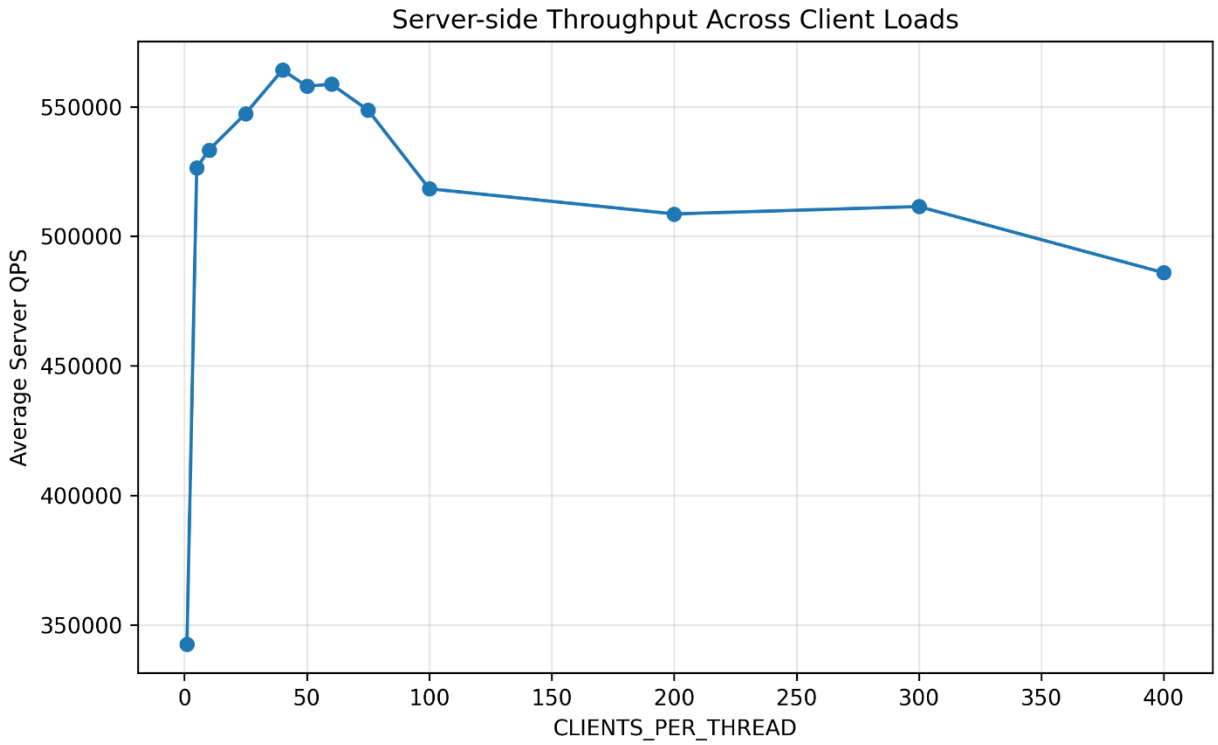

Figure 6.1: Server-side throughput across different client loads.

The results show that throughput increases rapidly at low load. From load 1 to load 5, the average server-side throughput increases from approximately 343K QPS to approximately 526K QPS. Throughput continues to improve more gradually up to CLIENTS_PER_THREAD = 40, where the highest average server-side throughput is observed.

At load 40, TaoBench achieved the best operating point, with approximately 564K server QPS, 3.68 ms average latency, and 9.58 ms p99 latency. After load 40, increasing the load did not significantly improve throughput. Loads 50 and 60 produced similar throughput, but with higher average and tail latency. At higher loads, such as 100, 200, 300, and 400, throughput remained mostly flat or decreased, while latency increased significantly.

This behavior indicates that the system reaches its most efficient operating region around load 40. Beyond this point, additional offered load does not produce more completed requests. Instead, it increases queueing delay and causes higher latency. Therefore, load 40 was selected as the fixed client load for the fast/slow thread ratio experiments.

6.3 Throughput, Latency and Hit Rate Analysis

The load sweep results show a clear relationship between throughput and latency. At low loads, increasing the offered load improves throughput while latency remains relatively low. This indicates that the server still has available processing capacity. However, once the system approaches saturation, throughput no longer increases significantly, while latency continues to rise.

The highest average server-side throughput was observed at CLIENTS_PER_THREAD = 40, with approximately 564K server QPS. Loads 50 and 60 achieved similar throughput, but with higher latency. This means that although these loads were close in throughput, they were less efficient from a latency perspective. For this reason, load 40 represents a better operating point than load 50 or load 60.

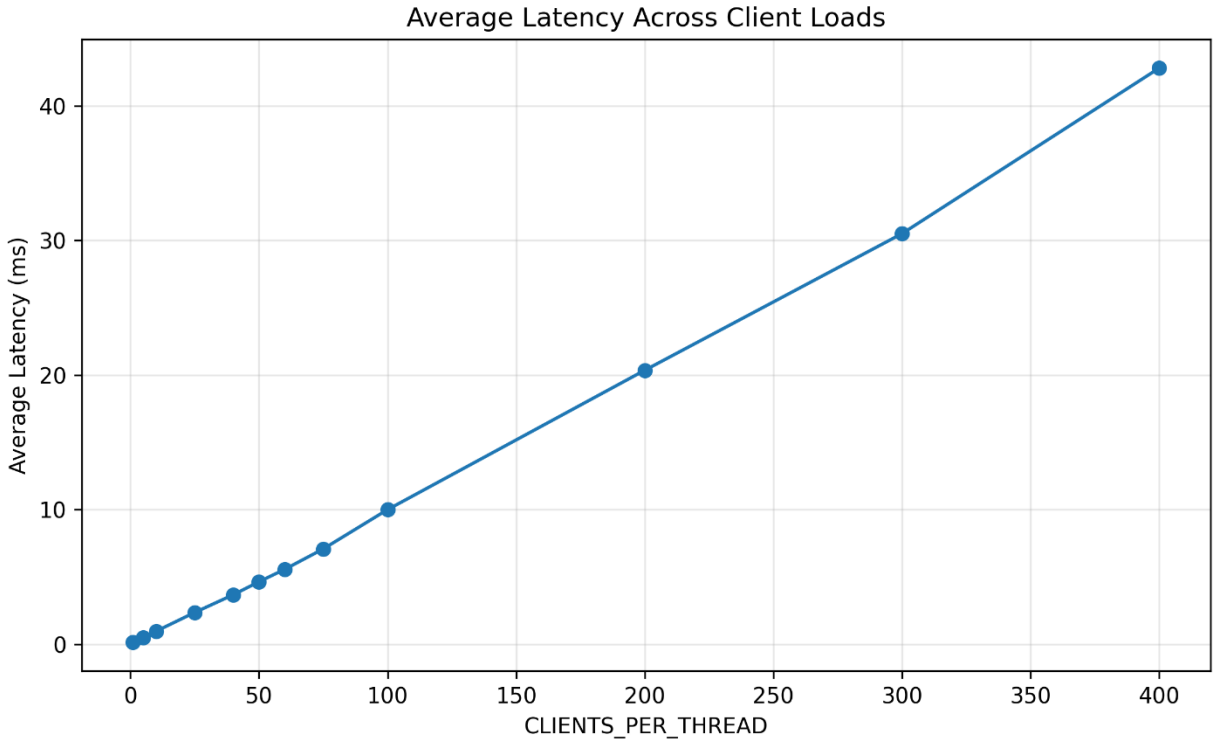

Figure 6.2: Average latency across different client loads.

The latency trend becomes more visible when average latency is compared with p99 latency.

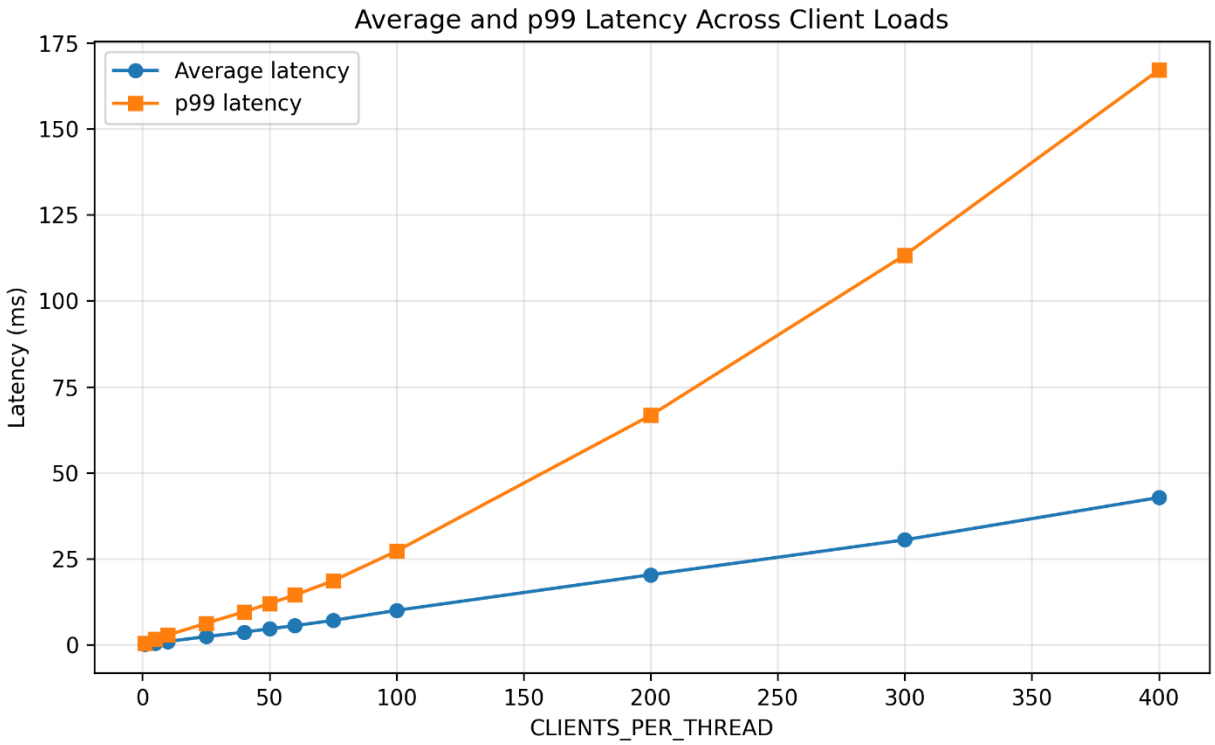

Figure 6.3: Comparison of average latency and p99 latency across different client loads.

Figure 6.3 shows that both average latency and p99 latency increase as the offered load becomes higher. However, p99 latency grows much more sharply than average latency. At load 40, p99 latency was approximately 9.58 ms. At load 100, p99 latency increased to approximately 27.25 ms, and at load 400 it reached approximately 167.12 ms. This shows that tail latency is much more sensitive to overload than average latency.

This is important because datacenter systems often care about high-percentile latency, not only average latency [7]. Even if average latency remains acceptable, high p99 latency means that a fraction of requests experience much slower response times. In real systems, these slow requests can negatively affect user experience.

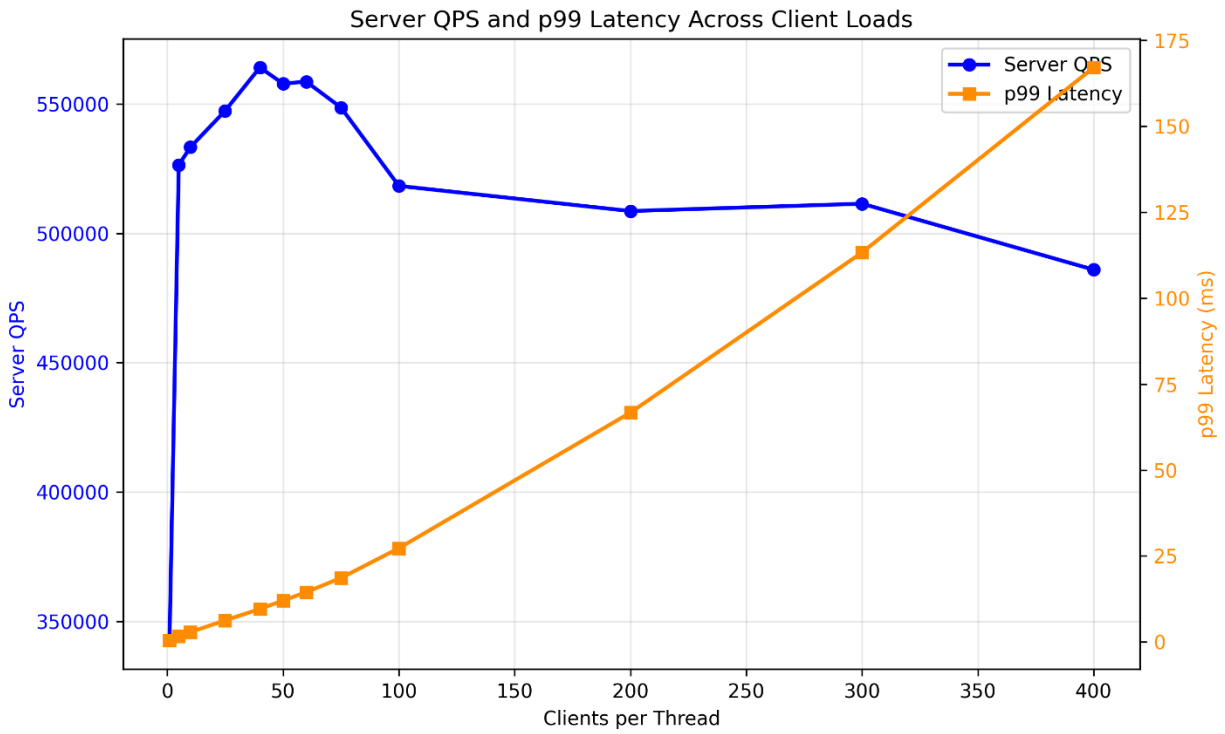

Figure 6.4: Server-side throughput and p99 latency across different client loads.

Figure 6.4 combines server-side throughput and p99 latency in a single view. The figure shows that throughput increases rapidly at low loads and reaches its highest value around CLIENTS_PER_THREAD = 40. Beyond this point, throughput does not improve significantly and eventually decreases, while p99 latency continues to increase sharply. This indicates that after the system reaches its efficient operating region, additional offered load mainly increases queueing delay and tail-latency behavior rather than producing useful throughput gains.

The hit ratio was verified using the server-side metrics. This value was not unexpected, because the DCPerf TaoBench documentation states that the expected hit ratio is approximately in the range of 0.88 to 0.9. Therefore, the observed hit ratio close to 0.9 indicates that the benchmark was operating in its intended cache-dominated region.

The workload characteristics are also important for interpreting this result. In this thesis, the exact per-key access distribution was not independently measured. Instead, the official TaoBench workload generator was used, and the resulting cache behavior was validated through the observed hit ratio in the server-side metrics.

Therefore, the performance trends observed in the load sweep are not explained by unstable cache behavior. Since the hit ratio remained close to the expected range, the increase in latency at higher loads is mainly attributed to load-induced queueing and server-side processing limits.

The main conclusion from this section is that TaoBench reaches a throughput saturation point around CLIENTS_PER_THREAD = 40 in this setup. After that point, higher load does not improve throughput and mainly increases average and tail latency.

6.4 Resource Utilization Analysis

In addition to throughput and latency, resource utilization was collected to better understand how the server and clients behaved during the experiments. This analysis is important because application-level metrics alone do not always explain why performance improves, saturates, or decreases. By observing CPU, memory, network, and disk utilization, it becomes possible to identify whether the bottleneck is related to the server, the clients, the network, or storage.

The utilization data was collected during the load sweep using the utilization-enabled experiment script. For each run, CPU, memory, network, and disk utilization were recorded on the server, client1, and client2. The analysis focused only on the measurement phase, using the samples corresponding to the 300-second test window.

The aggregated utilization results for the load sweep are shown in **Table 6.2**.

Table 6.2: Load sweep utilization results.

Load	***Runs***	***Server CPU Avg***	***Server CPU Max***	***Server RX MB/s***	***Server TX MB/s***	***Client1 CPU Avg***	***Client2 CPU Avg***
1	5	63.16%	71.63%	108.39	100.81	20.06%	20.04%
5	5	69.55%	75.03%	166.78	155.10	30.72%	30.47%
10	5	69.68%	75.45%	169.09	157.25	31.11%	30.72%
25	5	73.04%	78.24%	173.79	161.63	32.07%	31.56%
40	5	74.77%	79.50%	178.70	166.22	34.11%	33.02%
50	5	74.74%	79.93%	176.99	164.60	33.71%	33.27%
60	5	75.03%	80.62%	177.21	164.83	34.87%	33.82%
75	5	74.86%	80.38%	174.10	161.89	35.44%	34.41%
100	5	73.00%	78.72%	164.83	153.10	35.64%	34.68%
200	5	74.79%	80.92%	162.68	150.70	37.22%	36.75%
300	5	76.82%	81.94%	164.08	153.79	40.15%	39.18%
400	5	77.27%	81.91%	157.08	157.24	41.69%	40.39%

Figure 6.5 shows the relationship between server-side throughput and server CPU utilization across different client loads.

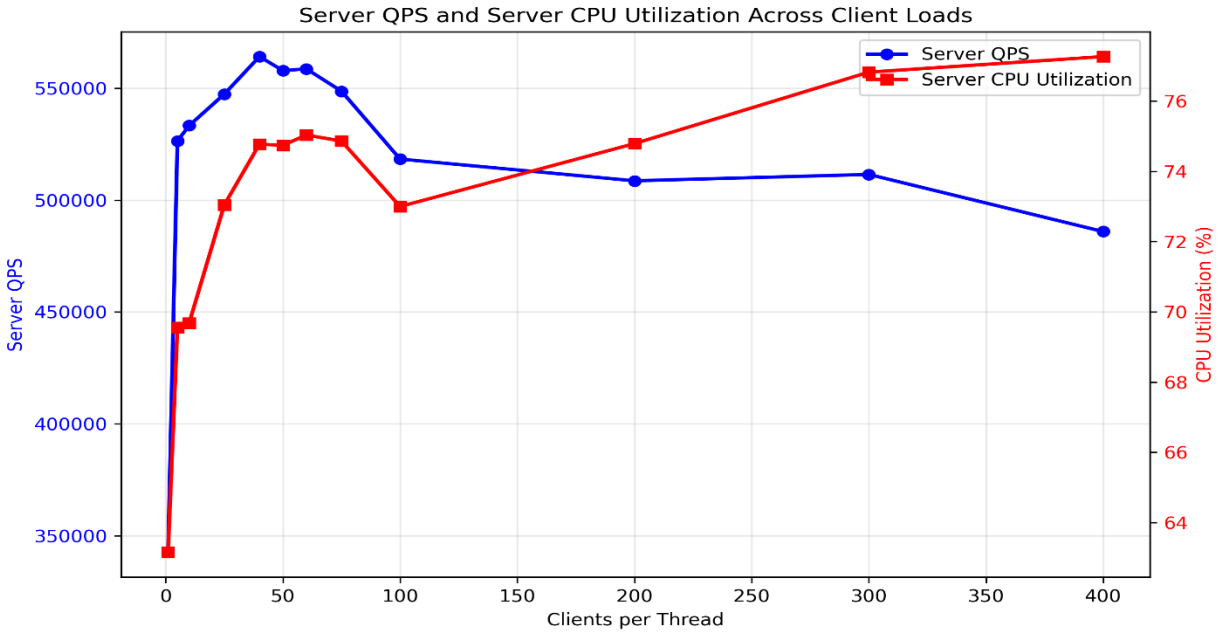

Figure 6.5: Server-side throughput and server CPU utilization across different client loads.

The results show that the server was consistently the most utilized node. Server CPU utilization increased from approximately **63%** at load 1 to approximately **75%** at load 40. At higher loads, server CPU utilization remained in the range of approximately **73% to 77%**. This indicates that the server was under significant load throughout the experiment, especially after the system reached its throughput saturation region.

The clients, however, remained much less utilized than the server. According to Table 6.2, at load 40, client CPU utilization was approximately **34%** on each client. Even at load 400, both clients remained around **40–42%** CPU utilization. This shows that the clients were not saturated and were still capable of generating additional load. Therefore, the main performance limitation was not caused by client-side CPU capacity.

Network utilization followed a similar trend to throughput. Server RX and TX increased at low loads and reached their highest values around the best-performing region. At load 40, server RX was approximately **178.70 MB/s** and server TX was approximately **166.22 MB/s**.

After this point, network throughput did not continue increasing significantly, which is consistent with the observation that application throughput had already reached saturation.

Disk activity remained very low during the experiments. This is expected because TaoBench is primarily an in-memory caching workload. Since disk activity remained low, disk I/O was not considered a bottleneck in this setup.

Overall, the utilization results support the throughput and latency findings. The server was the most heavily utilized component, while the clients and disk were not saturated. Network traffic was consistent with the observed QPS values and did not indicate a clear network bottleneck. Therefore, the results suggest that the main bottleneck was server-side processing, including thread scheduling, request handling, and fast/slow path coordination.

6.5 Fast/Slow Thread Ratio Sweep and Configuration Comparison

After identifying load 40 as the best client load, the next set of experiments focused on TaoBench's server-side thread configuration. The purpose of these experiments was to evaluate how different allocations of fast and slow threads affect throughput, latency, and resource utilization.

In TaoBench, fast threads handle cache-hit requests, while slow threads handle cache-miss requests. Since the hit ratio was approximately 0.9, most requests were expected to follow the fast path. Therefore, the hypothesis was that increasing fast-path processing capacity could improve throughput. At the same time, the slow path could not be removed completely, because around 10% of requests still required slow-path processing.

The ratio experiments varied two main parameters: FAST_THREADS_RATIO and SLOW_TO_FAST_RATIO. The client load was fixed at CLIENTS_PER_THREAD = 40, which ensured that the effect of server-side thread configuration could be studied without changing the offered client load.

The ratio analysis was performed in stages. First, a broad sweep explored a wide range of fast and slow thread ratios. Then, an extension sweep tested higher fast-thread ratios because the best result in the broad sweep occurred at the highest tested fast ratio. After that, a high fast-ratio plateau sweep explored the region where throughput appeared to approach a

maximum. Finally, a repeated plateau sweep focused on the high-fast-ratio region using multiple runs per configuration.

6.5.1 Broad Ratio Sweep

The first ratio experiment was a broad sweep across a wide range of configurations. The tested values were:

FAST_THREADS_RATIO = 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 2.0

SLOW_TO_FAST_RATIO = 0.25, 0.5, 1, 2, 3, 4

This produced 42 configurations. The purpose of this broad sweep was to identify promising regions before running more time-consuming repeated experiments. The server-side throughput results of the broad sweep are shown in Figure 6.6.

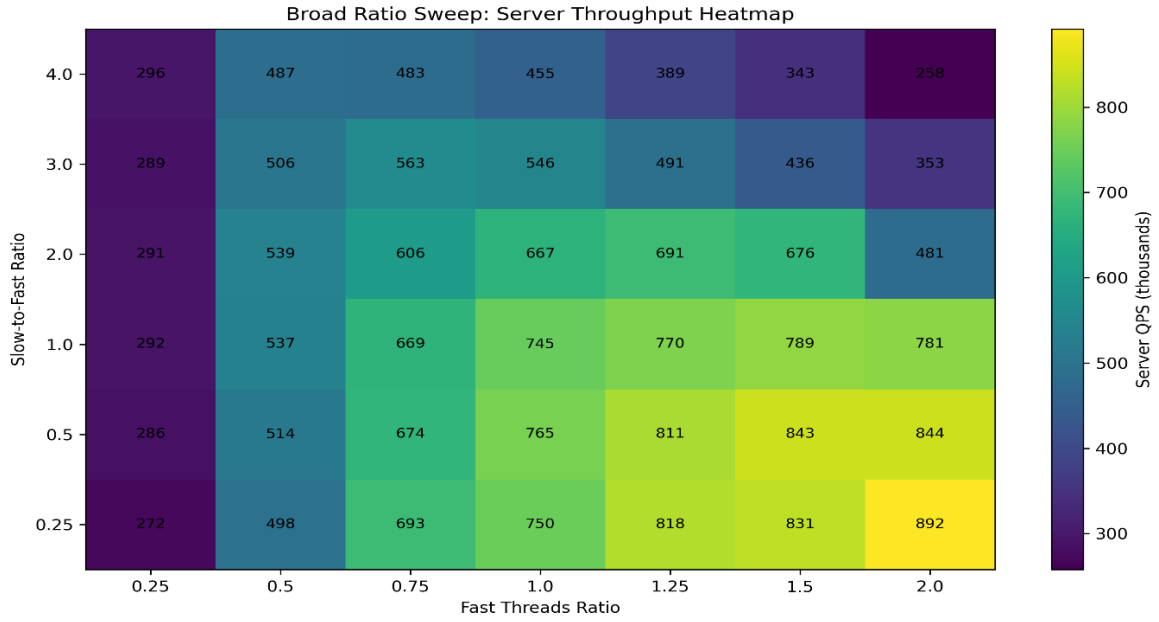

Figure 6.6: Server-side throughput heatmap for the broad fast/slow thread ratio sweep.

The broad sweep showed that the workload benefited from increasing the number of fast threads. Throughput generally improved as FAST_THREADS_RATIO increased, especially when SLOW_TO_FAST_RATIO remained low. The best configuration in this stage was:

FAST_THREADS_RATIO = 2.0

SLOW_TO_FAST_RATIO = 0.25

This configuration achieved approximately 891.5K server QPS, 2.32 ms average latency, 10.43 ms p99 latency, and a hit ratio close to 0.900. It also reached approximately 82.1% average server CPU utilization and 283 MB/s server RX throughput.

This result can be explained by the cache-dominated nature of the workload. Since the hit ratio remained close to 0.9, most requests were cache hits and were served through the fast path. Therefore, increasing FAST_THREADS_RATIO increased the processing capacity available for the dominant part of the workload. In this setup, FAST_THREADS_RATIO = 2.0 on a 32-logical-CPU server corresponds to approximately 64 fast threads. With SLOW_TO_FAST_RATIO = 0.25, this also provided approximately 16 slow threads, which was enough to handle the smaller fraction of cache-miss requests without allocating excessive capacity to the slow path.

Therefore, the best configuration in the broad sweep was not simply the configuration with the highest total number of threads. Instead, it was the configuration that provided a better balance between the dominant fast-path traffic and the smaller slow-path traffic. Configurations with too few fast threads limited cache-hit throughput, while configurations with too much relative slow-path capacity did not improve the common cache-hit case and could introduce additional scheduling and resource overhead.

6.5.2 Fast Ratio Extension Sweep

Because the best result in the broad sweep occurred at the highest tested fast-thread ratio, an additional sweep was performed to test whether increasing the fast-thread ratio further would continue to improve performance.

The tested values were: FAST_THREADS_RATIO = 2.0, 2.5, 3.0 SLOW_TO_FAST_RATIO = 0.25, 0.5, 1.0

This produced 9 configurations. The purpose of this extension sweep was to explore whether the promising region identified in the broad sweep continued beyond FAST_THREADS_RATIO = 2.0.

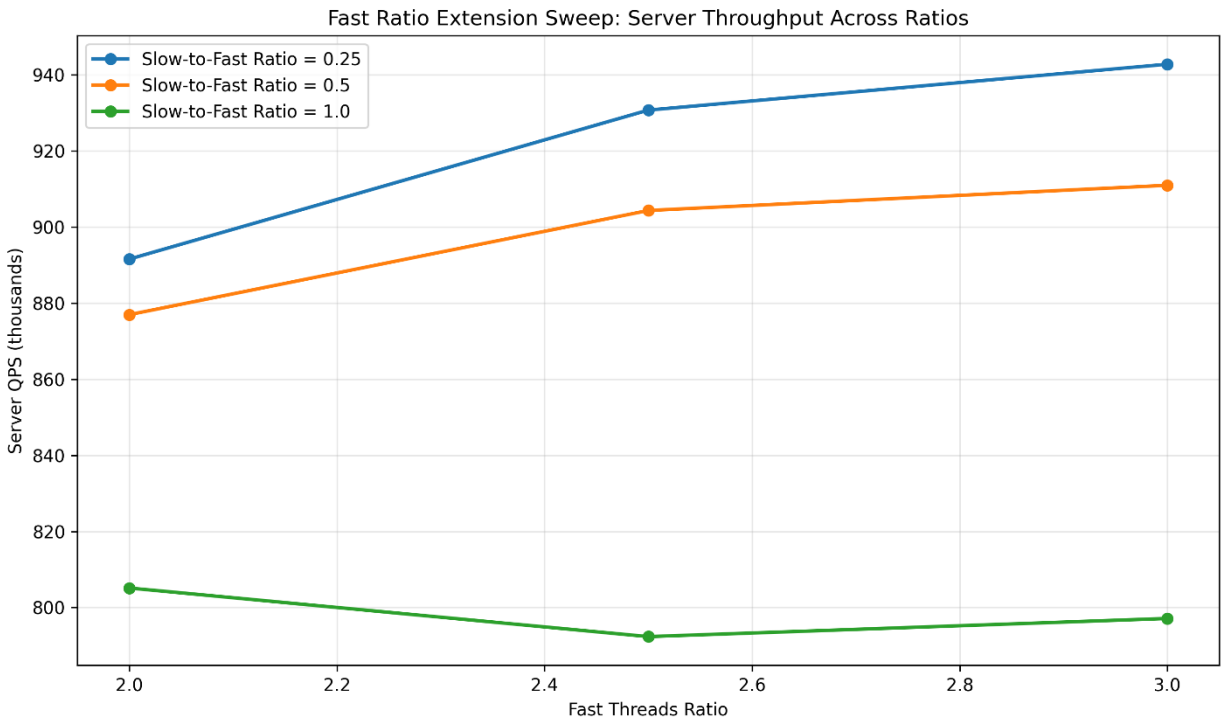

Figure 6.7: Server-side throughput across the fast ratio extension sweep.

The results showed that increasing the fast-thread ratio beyond 2 improved performance. The best configuration in this extension sweep was:

FAST_THREADS_RATIO = 3.0

SLOW_TO_FAST_RATIO = 0.25

This configuration achieved approximately 942.7K server QPS, 2.20 ms average latency, 12.00 ms p99 latency, 88.98% average server CPU utilization, and 298.70 MB/s server RX throughput.

This result confirmed that the workload was strongly fast-path dominated. However, since the best result again appeared at the highest tested fast-thread ratio, the experiment still had not identified a clear throughput plateau. Therefore, a high fast-ratio plateau sweep was required.

6.5.3 High Fast-Ratio Plateau Sweep

The next stage extended the search to larger fast-thread ratios. The purpose of this sweep was to determine whether throughput would continue to improve beyond FAST_THREADS_RATIO = 3.0 and to identify the approximate region where performance started to plateau.

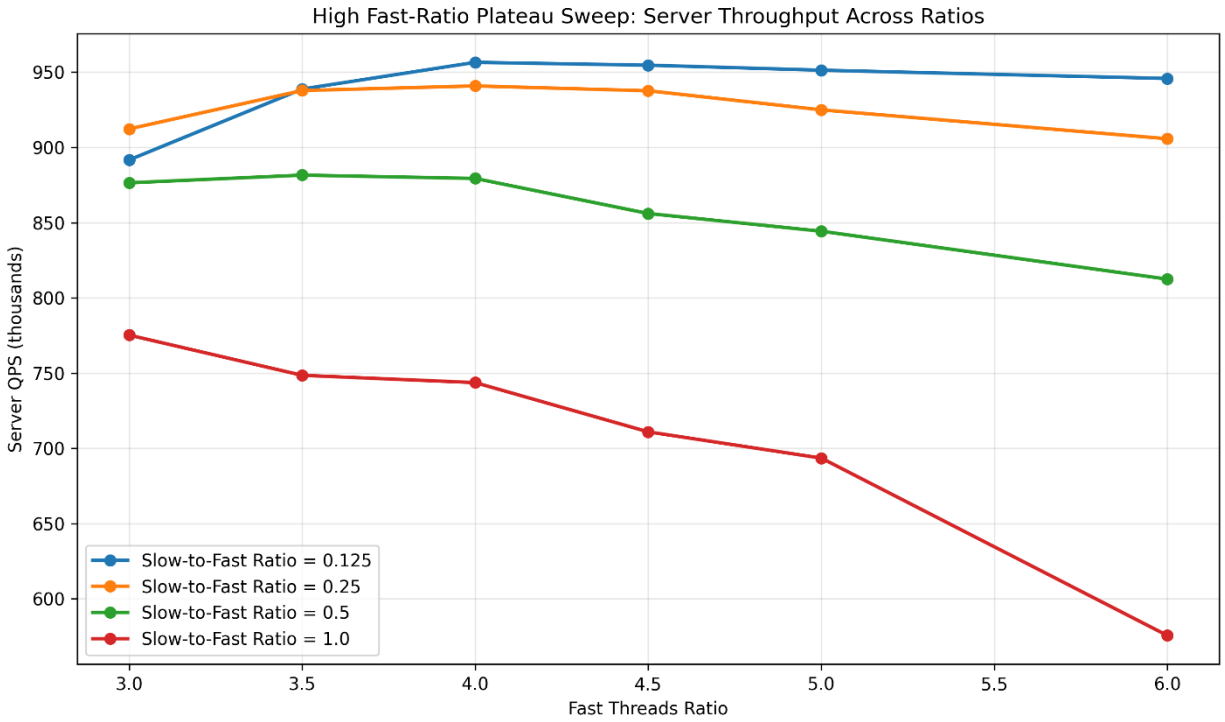

Figure 6.8: Server-side throughput across the high fast-ratio plateau sweep.

The high fast-ratio plateau sweep showed that performance continued to improve when FAST_THREADS_RATIO was increased beyond 3. The best result in this exploratory stage was observed at:

FAST_THREADS_RATIO = 4.0

SLOW_TO_FAST_RATIO = 0.125

This configuration achieved approximately 956.6K server QPS, 2.17 ms average latency, 12.77 ms p99 latency, and 89.8% average server CPU utilization.

However, this sweep used a single run per configuration. Therefore, the result was treated as exploratory. A final repeated plateau sweep was then executed to confirm the best region more reliably.

6.5.4 Final Repeated Plateau Sweep

The final ratio experiment focused on the high-fast-ratio region. The purpose was to confirm the best configuration using repeated runs and to determine where increasing the fast-thread ratio stops improving throughput.

The final plateau sweep tested:

FAST_THREADS_RATIO = 3.5, 4.0, 4.5, 5.0, 6.0

SLOW_TO_FAST_RATIO = 0.125, 0.25, 0.5

Each configuration was repeated five times. This produced:

5 fast ratios × 3 slow ratios × 5 runs = 75 total runs

The aggregated results are shown in Table 6.3.

Table 6.3: Aggregated results of the final repeated fast/slow thread ratio plateau sweep.

Fast Ratio	***Slow Ratio***	***Runs***	***Avg Server QPS***	***Avg Latency***	***p99 Latency***	***Hit Ratio***
4.5	0.125	5	956,798	2.17 ms	13.62 ms	0.897
5.0	0.125	5	955,952	2.17 ms	13.77 ms	0.896
6.0	0.125	5	940,824	2.21 ms	14.26 ms	0.896
4.0	0.25	5	939,816	2.21 ms	13.05 ms	0.898
4.0	0.125	5	939,527	2.21 ms	13.41 ms	0.897
3.5	0.25	5	937,478	2.21 ms	12.75 ms	0.898
4.5	0.25	5	936,490	2.22 ms	13.34 ms	0.898
3.5	0.125	5	935,308	2.22 ms	12.80 ms	0.897
5.0	0.25	5	926,939	2.24 ms	13.65 ms	0.898
6.0	0.25	5	905,921	2.29 ms	14.07 ms	0.898
3.5	0.5	5	882,446	2.35 ms	11.97 ms	0.898
4.0	0.5	5	870,179	2.39 ms	12.49 ms	0.898
4.5	0.5	5	860,306	2.41 ms	12.93 ms	0.898
5.0	0.5	5	838,287	2.47 ms	13.31 ms	0.897
6.0	0.5	5	810,277	2.56 ms	13.92 ms	0.894

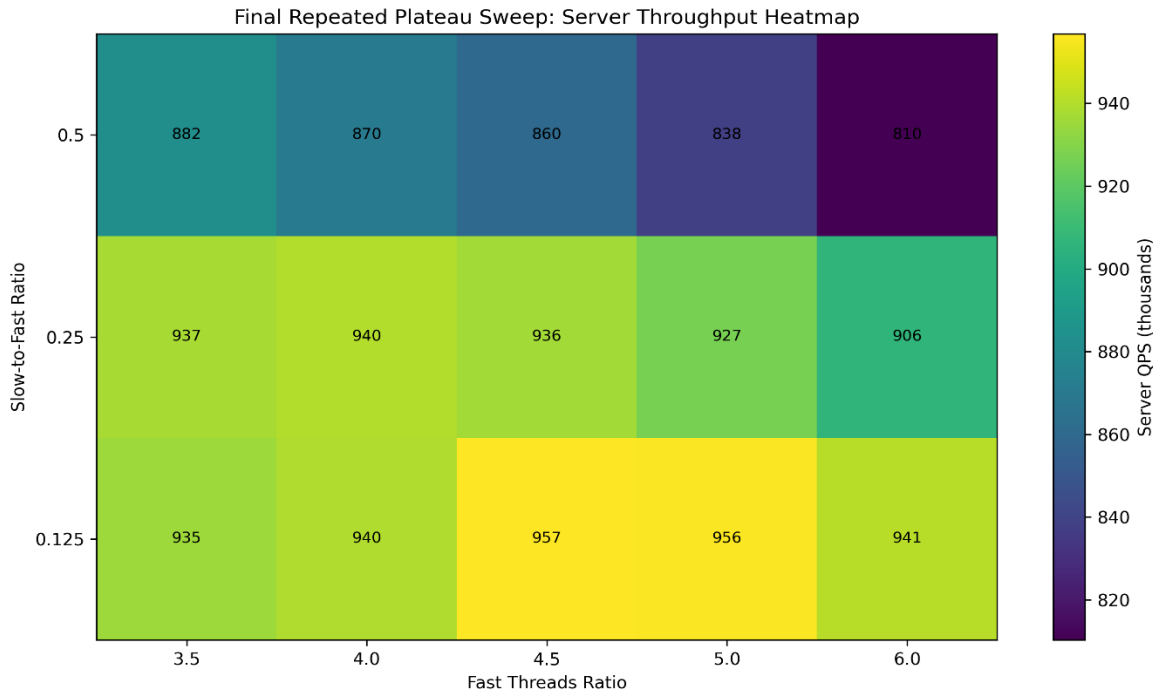

Figure 6.9: Server-side throughput heatmap for the final repeated fast/slow thread ratio plateau sweep.

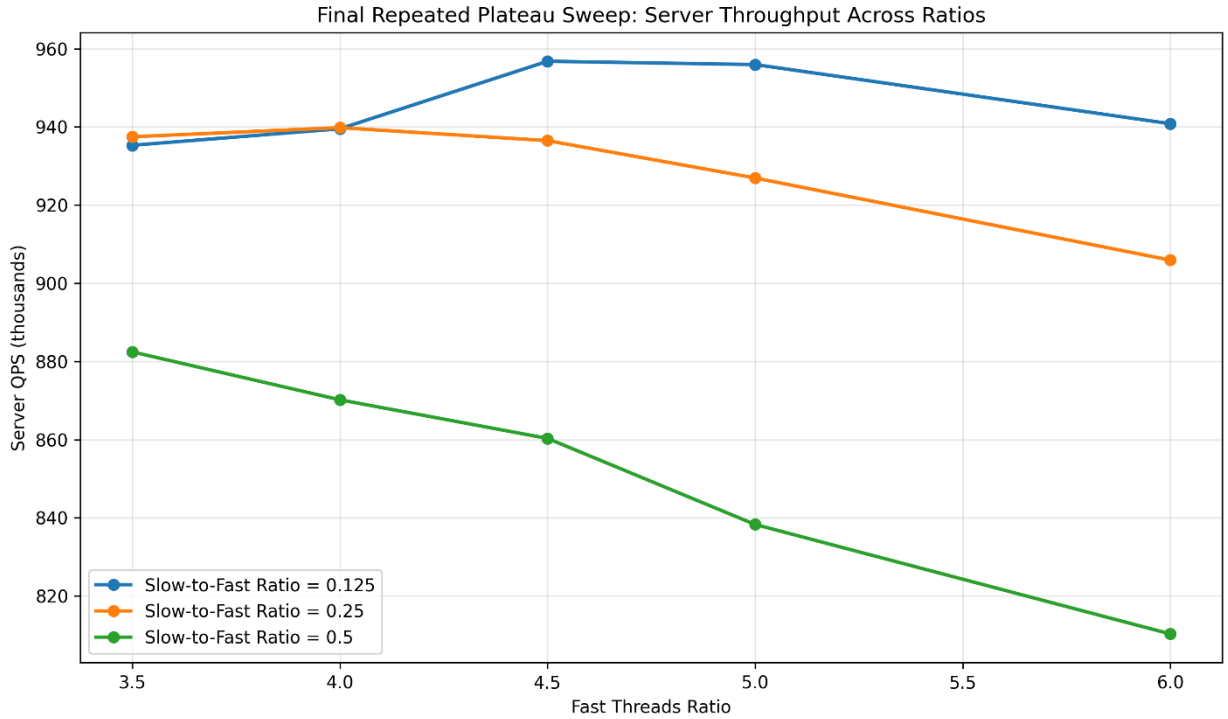

Figure 6.10: Server-side throughput across final repeated fast/slow thread ratio configurations.

The final repeated plateau sweep showed that the best throughput region was reached when SLOW_TO_FAST_RATIO = 0.125 and FAST_THREADS_RATIO was between 4.5 and 5.0. The highest average server-side throughput was observed at:

FAST_THREADS_RATIO = 4.5

SLOW_TO_FAST_RATIO = 0.125

This configuration achieved approximately 956.8K server QPS, 13.62 ms p99 latency, and a hit ratio of 0.8968.

The configuration:

FAST_THREADS_RATIO = 5.0

SLOW_TO_FAST_RATIO = 0.125

achieved nearly identical throughput, reaching approximately 956.0K server QPS, 2.17 ms average latency, and 13.77 ms p99 latency. Therefore, the results should be interpreted as a performance plateau region rather than a single isolated best configuration.

This indicates that increasing the fast-thread ratio improves throughput up to the 4.5–5.0 region. Beyond this point, increasing the ratio further does not improve performance and instead begins to reduce throughput. For example, FAST_THREADS_RATIO = 6.0 with SLOW_TO_FAST_RATIO = 0.125 achieved lower throughput than the 4.5–5.0 region. This suggests that additional fast threads eventually introduce overhead, such as thread scheduling, coordination, CPU contention, or cache pressure.

Given that the server has 32 logical CPUs, FAST_THREADS_RATIO = 4.5 corresponds to approximately 144 fast threads, while FAST_THREADS_RATIO = 5.0 corresponds to approximately 160 fast threads. With SLOW_TO_FAST_RATIO = 0.125, these configurations correspond to approximately 18 and 20 slow threads, respectively. Therefore, the best observed performance region corresponds approximately to 144–160 fast threads and 18–20 slow threads.

6.5.5 Utilization of the Final Ratio Sweep

The utilization results of the final repeated plateau sweep confirm that the best-performing configurations heavily utilized the server.

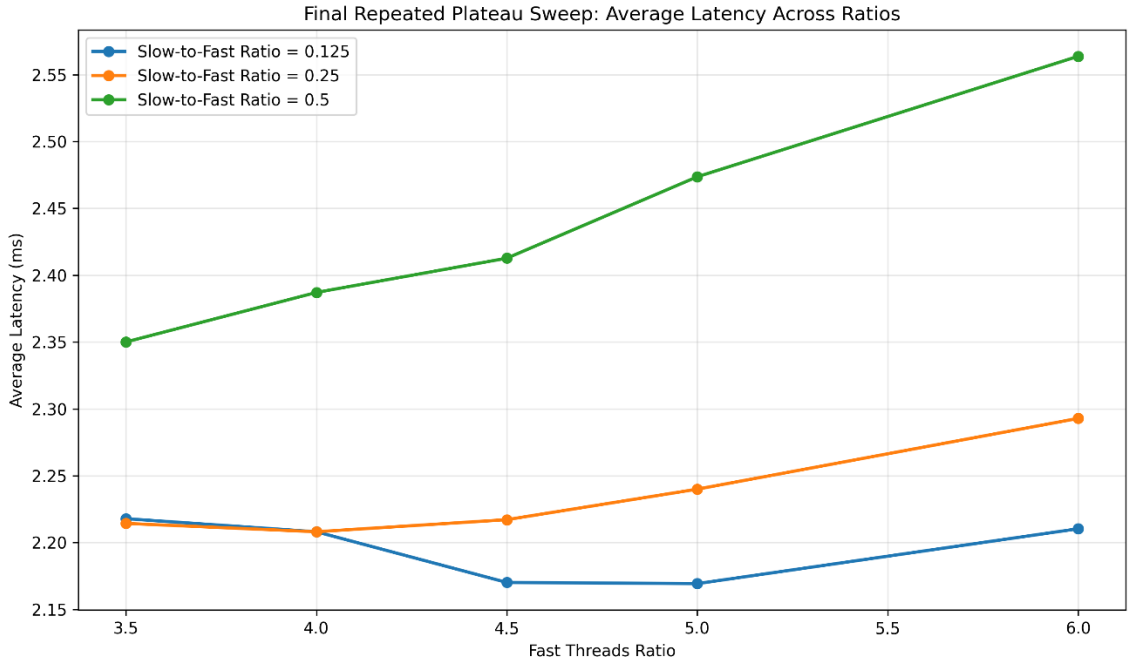

Figure 6.11: Average latency across final repeated fast/slow thread ratio configurations.

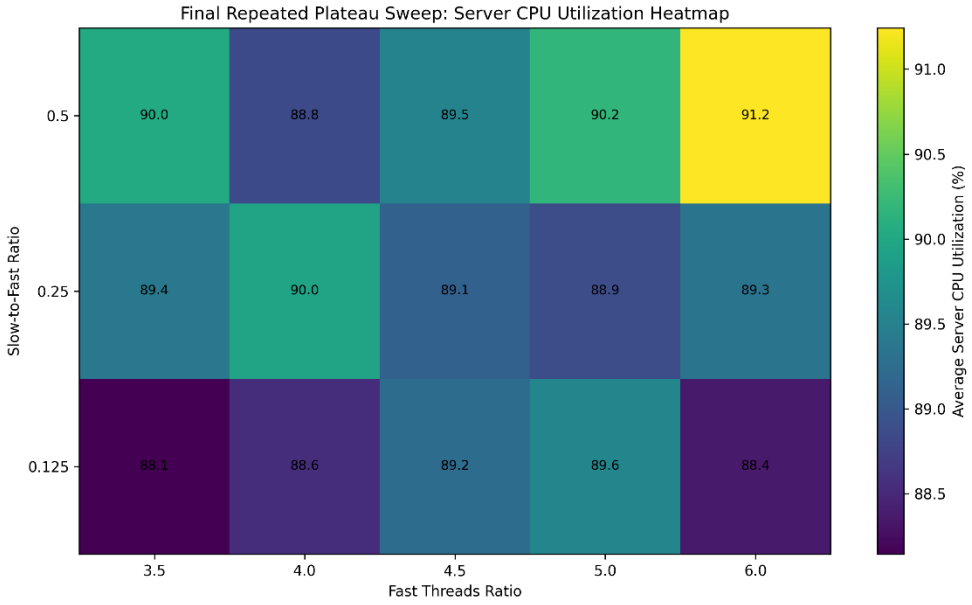

Figure 6.12: Server CPU utilization heatmap for the final repeated fast/slow thread ratio sweep.

The configuration FAST_THREADS_RATIO = 5.0 and SLOW_TO_FAST_RATIO = 0.125 achieved the lowest average latency, approximately 2.17 ms, with 13.77 ms p99 latency, while maintaining high server throughput. This configuration reached 89.56% average server CPU utilization, 98.67% maximum server CPU utilization, 303.15 MB/s server RX throughput, and 283.72 MB/s server TX throughput. The average CPU utilization of the two clients was approximately 45.53%, indicating that the clients were not the limiting factor.

The configuration FAST_THREADS_RATIO = 4.5 and SLOW_TO_FAST_RATIO = 0.125 achieved the highest server-side throughput, while FAST_THREADS_RATIO = 5.0 and SLOW_TO_FAST_RATIO = 0.125 achieved the lowest average latency. Since the difference between these configurations was very small, the final results indicate a performance plateau around FAST_THREADS_RATIO = 4.5–5.0 with SLOW_TO_FAST_RATIO = 0.125.

6.5.6 Plateau and Slow Ratio Behavior

The final ratio sweep confirmed a throughput plateau around FAST_THREADS_RATIO = 4.5–5.0. This comparison is shown in Table 6.4.

Table 6.4: Throughput plateau for SLOW_TO_FAST_RATIO = 0.125.

FAST_THREADS_RATIO	***SLOW_TO_FAST_RATIO***	***Server QPS***
3.5	0.125	935,308
4.0	0.125	939,527
4.5	0.125	956,798
5.0	0.125	955,952
6.0	0.125	940,824

The results show that increasing the fast ratio improves throughput up to approximately 4.5–5.0. After that point, increasing the fast ratio to 6.0 does not improve performance and reduces throughput.

This behavior suggests that additional fast threads eventually introduce overhead. Possible sources of overhead include thread scheduling, context switching, CPU contention, cache pressure, and coordination overhead between the benchmark’s internal thread pools.

The final sweep also showed that the best results were consistently achieved with the lowest tested slow-to-fast ratio, SLOW_TO_FAST_RATIO = 0.125. The effect of changing the slow-to-fast ratio when FAST_THREADS_RATIO = 5.0 is shown in Table 6.5.

Table 6.5: Effect of SLOW_TO_FAST_RATIO when FAST_THREADS_RATIO = 5.0.

FAST_THREADS_RATIO	***SLOW_TO_FAST_RATIO***	***Server QPS***
5.0	0.125	955,952
5.0	0.25	926,939
5.0	0.5	838,287

The results show that increasing the number of slow threads reduced throughput. This behavior is consistent with the high hit ratio of the workload. Since approximately 90% of requests are cache hits, most requests are handled by the fast path. Only a small part of the workload requires slow-path processing. Therefore, a small number of slow threads is sufficient, while too many slow threads introduce unnecessary overhead.

Overall, the fast/slow thread ratio experiments show that TaoBench is strongly fast-path dominated under this workload. The best region was observed at high FAST_THREADS_RATIO values and low SLOW_TO_FAST_RATIO values. In the final repeated plateau sweep, the best throughput region was FAST_THREADS_RATIO = 4.5–5.0 with SLOW_TO_FAST_RATIO = 0.125. This region achieved approximately 956K server QPS, around 2.17 ms average latency, approximately 13.8 ms p99 latency, and close to 90% average server CPU utilization.

6.6 Summary of Experimental Findings

The experiments in this chapter show that TaoBench performance is strongly affected by both client load and server-side thread configuration.

The full load sweep showed that the best operating load for the current setup was CLIENTS_PER_THREAD = 40. At this load, TaoBench achieved approximately 564k server QPS, 3.68 ms average latency, 9.58 ms p99 latency, and 18.41 ms p99.9 latency. After load 40, increasing the offered load did not significantly improve throughput. Instead, it increased average latency and tail latency. This indicates that the system reached a saturation region where additional load mainly caused queueing delay.

The hit-ratio validation showed that the fixed warm-up methodology worked correctly. The official server logs confirmed that the hit ratio remained close to 0.9 across the experiments. Therefore, the observed performance changes were not caused by unstable cache behavior, but mainly by load pressure and server-side processing effects.

The fast/slow thread ratio experiments showed that the workload is fast-path dominated. Increasing the number of fast threads improved throughput until the system reached a plateau. The final repeated plateau sweep identified the best performance region as FAST_THREADS_RATIO = 4.5–5.0 with SLOW_TO_FAST_RATIO = 0.125. The highest average server-side throughput was observed at FAST_THREADS_RATIO = 4.5 and SLOW_TO_FAST_RATIO = 0.125, reaching approximately 956.8K server QPS with 13.62 ms p99 latency. The configuration FAST_THREADS_RATIO = 5.0 and SLOW_TO_FAST_RATIO = 0.125 achieved nearly identical throughput, with approximately 956.0K server QPS, and the lowest average latency, around 2.17 ms and 13.77 ms p99 latency.

These results should be interpreted as a throughput plateau rather than a single isolated best configuration. On the 32-logical-CPU server, FAST_THREADS_RATIO = 4.5 corresponds to approximately 144 fast threads and 18 slow threads, while FAST_THREADS_RATIO = 5.0 corresponds to approximately 160 fast threads and 20 slow threads. The plateau behavior showed that increasing FAST_THREADS_RATIO beyond the 4.5–5.0 region did not improve throughput. In fact, FAST_THREADS_RATIO = 6 reduced performance. This

suggests that additional fast threads eventually introduce scheduling and coordination overhead.

Finally, utilization analysis showed that the server was the main bottleneck. In the 5.0/0.125 configuration, server CPU utilization was approximately 89.56% on average, while both clients remained around 45.5% CPU utilization. Disk activity was negligible, and network throughput was consistent with the achieved QPS. Therefore, the main limitation was server-side processing and thread scheduling or coordination, rather than client capacity, disk I/O, or network saturation.

Overall, the final results demonstrate that careful server-side thread configuration can significantly improve TaoBench performance. The experiments show that a default-like configuration is not necessarily optimal for the selected setup, and that a high fast-thread ratio combined with a low slow-to-fast ratio provides the best observed performance for this workload.

Chapter 7: Conclusion and Future Work

7.1 Summary of the Thesis

This thesis presented an experimental performance evaluation of TaoBench within the DCPerf framework. TaoBench was selected because it represents an important type of datacenter workload: high-throughput, low-latency caching systems. The workload models a cache-based client-server system with separate fast-path and slow-path request processing, making it suitable for studying the effect of cache behavior, client load, and server-side thread allocation.

The work began by studying the DCPerf framework and the internal behavior of TaoBench. Particular attention was given to the benchmark architecture, including the TaoBench server, TaoBench clients, request processing flow, fast path, slow path, fast threads, slow threads, and warm-up behavior. This background was necessary in order to interpret the experimental results correctly.

A distributed experimental environment was deployed on CloudLab using one server node and two client nodes. The server executed the TaoBench server workload, while the two clients generated requests over the private CloudLab network. A separate experiment repository was also created to organize the CloudLab profile, setup scripts, execution scripts, monitoring tools, and documentation used throughout the thesis.

The experiments focused on two main directions. First, a client load sweep was performed to understand how TaoBench behaves as the offered load increases. Second, a set of fast/slow thread ratio sweeps was performed to evaluate how server-side thread allocation affects throughput, latency, hit ratio, and resource utilization.

A key part of the methodology was the fixed warm-up approach. TaoBench was modified so that every experiment used the same warm-up load before the measurement phase. This ensured that different experimental runs started from a comparable cache state. The official server logs confirmed that the hit ratio remained close to the expected value of approximately 0.9, which supported the fixed warm-up methodology.

7.2 Main Findings

The first major finding is that TaoBench reaches a clear throughput saturation point as client load increases. In the final load sweep, the best operating load was found to be CLIENTS_PER_THREAD = 40. At this load, the benchmark achieved approximately 564K QPS, 3.68 ms average latency, 9.58 ms p99 latency, and 18.41 ms p99.9 latency. Increasing the load beyond 40 clients per thread did not significantly improve throughput. Instead, it mainly increased average latency and tail latency. This shows that after the system reaches its efficient operating region, additional offered load mostly creates queueing delay rather than useful performance improvement.

The final repeated ratio plateau sweep identified the best performance region as FAST_THREADS_RATIO = 4.5–5.0 with SLOW_TO_FAST_RATIO = 0.125. The highest average server-side throughput was observed at FAST_THREADS_RATIO = 4.5 and SLOW_TO_FAST_RATIO = 0.125, reaching approximately 956.8K server QPS. The configuration FAST_THREADS_RATIO = 5.0 and SLOW_TO_FAST_RATIO = 0.125 achieved nearly identical throughput, with approximately 956.0K server QPS, the lowest average latency of around 2.17 ms, 13.77 ms p99 latency, and 24.56 ms p99.9 latency. On the 32-logical-CPU server, this performance region corresponds approximately to 144–160 fast threads and 18–20 slow threads. This result shows that careful thread configuration can significantly improve TaoBench performance compared to earlier/default-like configurations.

Another important finding is that throughput reaches a plateau when the fast-thread ratio becomes too high. The final sweep showed that performance improved up to approximately FAST_THREADS_RATIO = 4.5–5.0. However, increasing the fast ratio to 6.0 did not improve performance and instead reduced throughput. This suggests that after a certain point, additional fast threads introduce overhead such as scheduling cost, context switching, CPU contention, cache pressure, and coordination overhead.

The experiments also showed that higher slow-to-fast ratios reduced performance. The best results were consistently achieved with the lowest tested slow-to-fast ratio, SLOW_TO_FAST_RATIO = 0.125. This is consistent with the workload's high hit ratio. Since only a small fraction of requests follow the slow path, a small number of slow threads

is sufficient. Adding too many slow threads creates overhead without improving useful throughput.

Finally, the utilization results showed that the bottleneck was mainly server-side. In the best configuration, the server CPU was heavily utilized, while both clients remained below saturation. Disk activity was negligible, and network throughput was consistent with the achieved QPS. Therefore, the main limitation was not client capacity, disk I/O, or network saturation, but server-side processing and thread scheduling or coordination.

7.3 Limitations

Although the experiments produced clear findings, this work has some limitations.

The evaluation was performed on a specific CloudLab setup with one server node and two client nodes. This topology was enough to study the main performance trends, but it does not represent the scale of a real production datacenter. A larger setup with more client nodes, or even multiple server nodes, could show additional scalability behavior.

Another limitation is that the experiments used a fixed memory size of 16 GB. This helped keep the experiments controlled, but it also means that the effect of cache size was not fully studied. Different memory sizes could affect cache behavior, hit-ratio stability, and throughput.

The thesis also focused only on TaoBench. DCPerf includes other workloads, such as FeedSim, VideoTranscodeBench, DjangoBench, MediaWiki, and SparkBench. These workloads stress different parts of the system and may show different performance behavior.

The final thread ratio results are also specific to the selected hardware, operating system, network configuration, and TaoBench setup. The best observed configuration may not be the best choice on a different machine with a different number of CPU cores, memory hierarchy, NUMA topology, or network interface.

Another limitation is that the analysis focused mainly on throughput, latency, hit ratio, CPU utilization, and network utilization. Other useful measurements, such as hardware performance counters, cache misses, memory bandwidth, context switches, and scheduler-level statistics, were not analyzed in depth.

Finally, the fixed warm-up modification improved comparability across experiments, but it also changed the original benchmark workflow. This modification was useful for the goals of this thesis, but future work could compare the fixed warm-up approach with the original TaoBench warm-up behavior in more detail.

7.4 Future Work

There are several possible directions for future work. One direction would be to repeat the experiments on larger CloudLab topologies. Using more client nodes would allow the server to be stressed more aggressively and would help evaluate whether the same bottlenecks appear at a larger scale. In addition, using multiple server nodes could provide more insight into distributed scaling behavior.

Future work could also evaluate different memory sizes. Since TaoBench is a caching workload, memory capacity can affect the working set, cache pressure, and hit ratio. Testing smaller and larger memory configurations would help explain how cache size influences throughput and latency.

Another useful direction would be to analyze additional DCPerf workloads. This thesis focuses on TaoBench because it represents caching behavior, but DCPerf also includes workloads such as video transcoding, feed generation, web services, and analytics. Comparing TaoBench with these workloads would provide a broader understanding of DCPerf and of how different datacenter workloads stress the system.

The analysis could also be extended with deeper system-level metrics. Hardware performance counters could be used to study cache misses, instructions per cycle, memory bandwidth, branch behavior, and CPU stalls. Scheduler-level statistics could help confirm whether high thread ratios introduce context switching and scheduling overhead.

NUMA effects could also be studied in more detail. Since TaoBench performance depends on memory access and thread placement, experiments with different CPU binding, memory binding, and NUMA locality settings could provide more insight into server-side bottlenecks.

A further direction would be to study the backend-delay component of TaoBench in more detail. In the experiments of this thesis, the explicit slow-path sleep parameter was set to zero. This means that cache misses were handled through the internal slow-path mechanism, but without adding a fixed artificial backend delay.

Such experiments could vary tao_slow_path_sleep_us to evaluate how different backend-delay assumptions affect throughput, average latency, and p99 latency. A more realistic extension would be to replace the fixed delay with a latency distribution based on production-like backend or cross-region access times. Another possible direction would be to connect the slow path to an external storage service, so that cache misses experience more realistic backend behavior.

In addition, future work could compare the slow-path waiting mechanisms, including the semaphore-based option and the nanosleep()-based option. This would help separate the effect of backend-delay modeling from the overhead caused by thread waiting, timer interrupts, and scheduling behavior.

Finally, the experiment repository could be extended with additional automation scripts, graph-generation tools, and analysis notebooks. This would make the workflow easier to repeat, easier to analyze, and more useful for future benchmarking studies.

References

[1] W. Su et al., “DCPerf: An Open-Source, Battle-Tested Performance Benchmark Suite for Datacenter Workloads,” in Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA), 2025, doi: 10.1145/3695053.3731411.

[2] Meta Research, “DCPerf: Benchmark Suite for Hyperscale Cloud Applications,” GitHub repository. [Online]. Available: https://github.com/facebookresearch/DCPerf. Accessed: May 17, 2026.

[3] A. Cheng, X. Shi, A. Kabcenell, S. Lawande, H. Qadeer, J. Chan, H. Tin, R. Zhao, P. Bailis, M. Balakrishnan, N. Bronson, N. Crooks, and I. Stoica, “TAOBench: An End-to-End Benchmark for Social Network Workloads,” Proceedings of the VLDB Endowment, vol. 15, no. 9, pp. 1965–1977, 2022, doi: 10.14778/3538598.3538616.

[4] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, “TAO: Facebook’s Distributed Data Store for the Social Graph,” in Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC), 2013, pp. 49–60.

[5] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” in Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012, pp. 37–48, doi: 10.1145/2150976.2150982.

[6] H. Kasture and D. Sanchez, “TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications,” in Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2016, pp. 1–10, doi: 10.1109/IISWC.2016.7581261.

[7] J. Dean and L. A. Barroso, “The Tail at Scale,” Communications of the ACM, vol. 56, no. 2, pp. 74–80, 2013, doi: 10.1145/2408776.2408794.

[8] R. Ricci, E. Eide, and the CloudLab Team, “Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications,” ;login: The USENIX Magazine, vol. 39, no. 6, pp. 36–38, 2014.

[9] Memcached, “memcached — a distributed memory object caching system.” [Online]. Available: https://memcached.org/. Accessed: May 17, 2026.

[10] Meta Research, “TaoBench README,” DCPerf GitHub repository. [Online]. Available: https://github.com/facebookresearch/DCPerf/tree/main/packages/tao_bench. Accessed: May 17, 2026.

[11] TAOBench Project, “TAOBench Overview.” [Online]. Available: https://taobench.org/docs/overview/taobench_overview. Accessed: May 17, 2026.

[12] A. Karagiorgis, “dcperf-taobench-experiments: Reproducible CloudLab scripts for evaluating TaoBench in the DCPerf framework,” GitHub repository. [Online]. Available: https://github.com/andKarag02/dcperf-taobench-experiments. Accessed: May 29, 2026.