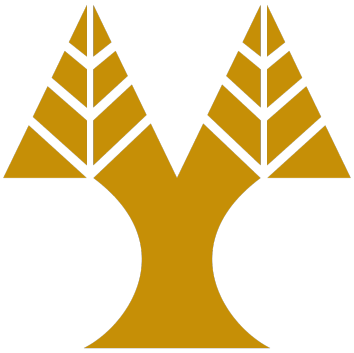
Diploma Project

Reducing Microservice Response Time through an Adaptive and Proactive Processor Wake-up Mechanism

Christoforos Seas



DEPARTMENT OF COMPUTER SCIENCE

May 2024

UNIVERSITY OF CYPRUS

Faculty of Pure and Applied Sciences

DEPARTMENT OF COMPUTER SCIENCE

Reducing Microservice Response Time through an Adaptive and Proactive Processor Wake-up Mechanism

Christoforos Seas

Advisor:

Dr. Haris Volos

The Thesis was submitted in partial fulfillment of the requirements for obtaining the Computer Science degree of the Department of Computer Science of the University of Cyprus

May 2024

Acknowledgements

I want to express my heartfelt gratitude to Dr. Haris Volos for his exceptional coordination and guidance throughout this thesis. Without his support and advice, this work would not have been possible.

I also wish to thank Dr. Yanos Sazeides for his invaluable mentorship during my two-month summer internship in my first year of studies. His insights and professional research skills have significantly contributed to my growth. I am also grateful to him for serving as another supervisor for my thesis.

Lastly, I extend my sincere thanks to Ms. Georgia Antoniou, who assisted me during my internship and this thesis, despite her busy schedule pursuing her Ph.D.

ABSTRACT

In the context of microservices architecture, enhancing operational efficiency is crucial for optimal performance. This thesis, addresses the challenge of latency arising from CPU wake-ups from deep sleep C-States in microservices environments. In Linux systems, an integral component known as the ”Idle governor” manages CPU states, including various sleep modes referred to as C-States, to optimize power consumption while ensuring system responsiveness. The idle governor dynamically adjusts CPU states based on system workload, optimizing power consumption while maintaining responsiveness.

However, one notable outcome of the idle governor's operation is the potential for it to put the CPU into a deep idle C-State, resulting in transition overhead when the CPU needs to wake-up in response to device requests. This transition overhead can contribute to latency, impacting the responsiveness of microservices.

This research endeavors to tackle the latency associated with CPU idleness by proposing a practical solution: implementing a mechanism to awaken CPUs from idle states, prior to processing incoming requests.

By introducing this proactive wake-up mechanism, the study aims to reduce the latency incurred during CPU wake-up from idle states, consequently improving the responsiveness and overall performance of microservices. Through practical implementation and experimentation, the efficacy of the proposed approach is assessed, focusing on latency reduction.

Additionally, the proactive wake-up mechanism is enhanced through training with a hill climbing AI algorithm. This AI algorithm optimizes the timing of sending pre-query requests to awaken CPUs and immediately serve incoming queries, further refining the responsiveness of microservices architecture.

The outcomes of this research offer valuable insights into enhancing microservices efficiency by leveraging CPU wake-up mechanisms. By prioritizing responsiveness through proactive CPU management, this work contributes to the advancement of performance optimization strategies in microservices-based applications, ultimately fostering enhanced operational efficiency and user experience.

Keywords: C-States, Idle governor, **Pre-Query Request**: The request sent before the actual request to wake up the processor

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

- **QPS**: Queries Per Second
- **Avg**: Average
- **PL**: Percentile Latency
- **CPU**: Central Processor Unit
- **API**: Application Programming Interface
- **IRQ**: Interrupt Request
- **PMF**: Probability Mass Function
- **TCP**: Transmission Control Protocol
- **ACK**: Acknowledgment
- **SMT**: Simultaneous Multithreading
- **RNN**: Recurrent Neural Network
- **PDF**: Probability Density Function
- **OLDI**: Online Data Intensive Applications

1 Introduction

The ever-growing demand for agility and scalability in software development has propelled microservices architectures to the forefront. These architectures decompose applications into smaller, independent services that communicate through well-defined APIs. While offering numerous benefits, microservices introduce new challenges, particularly in the realm of operational efficiency. This thesis investigates a specific performance drawback: latency arising from CPU wake-ups from deep C-States in microservices environments.

1.1 Motivation

Our motivation stems from the need to reduce the tail latency within microservices architectures. One significant performance drawback we identified is the latency induced by CPU wake-ups from deep C-States in such environments. To address this issue, our focus lies on proactively waking up the processor before sending the actual query or request. By preemptively activating the processor, we aim to minimize latency and optimize operational efficiency in microservices-based systems.

Expanding on our motivation, the drive to enhance tail latency within microservices architectures stems from the critical importance of latency reduction in modern computing environments. As applications become increasingly distributed and real-time responsiveness becomes a necessity, even minor delays can significantly impact user experience and system performance.

The specific challenge we aim to address is the latency introduced by CPU wake-up overhead, which can occur when processors remain in low-power states while awaiting incoming requests. This C-State transition overhead can result in delays in processing requests, leading to increased tail latency and potentially degraded system performance.

Recognizing this challenge, our research endeavors to proactively mitigate latency by implementing strategies to awaken the processor before the arrival of actual queries or requests. By preemptively activating the processor, we aim to minimize the latency introduced by CPU wake-ups from deep C-States, thereby optimizing the overall operational efficiency of microservices-based systems, with the cost of increased power consumption.

Our motivation is rooted in the belief that by tackling this performance drawback head-on, we can unlock significant improvements in system responsiveness, scalability, and overall user satisfaction within microservices architectures. Through our research efforts, we seek to contribute to the ongoing evolution and refinement of distributed computing paradigms,

ultimately enabling more agile, resilient, and efficient software ecosystems.

1.2 Our Hypothesis

We aim to determine whether sending a preliminary request before the actual data transfer can yield tangible reductions in response latency. The rationale behind this aim stems from the hypothesis that initiating a pre-query request could potentially optimize system responsiveness by priming the underlying infrastructure for incoming data reception. By introducing a preliminary interaction before the main data transfer, we hypothesize that the processor will wake-up and then will either enter a shallow C-State, such as `C1`, or not enter a C-State at all, thereby minimizing the response latency.

1.3 C-States Impact on Latency

The ideal C-state selection depends on the anticipated workload. For microservices, where short bursts of activity are interspersed with periods of low utilization, frequent transitions between active and sleep states can introduce significant latency.

This thesis proposes a novel approach to address this latency challenge. By implementing a proactive wake-up mechanism, we aim to mitigate the latency incurred when CPUs transition from deeper sleep states to service incoming requests in microservices environments. The following sections will delve deeper into the proposed solution, its implementation, and the evaluation of its effectiveness in reducing latency and improving overall microservices performance.

1.4 Contributions

Our research makes several contributions to the field of microservices architecture and system optimization:

- **Introduction of Wake-Up Mechanism:** One of our primary contributions is introducing and exploring a mechanism to proactively awaken the processor before the arrival of the actual query. This mechanism, serves to minimize latency arising from deep sleep C-States wake-up overhead, thereby enhancing the responsiveness of microservices.
- **Hill Climbing Algorithm for Optimal Timing:** We developed a hill climbing algorithm to identify the optimal timing for sending the pre-query wake-up request. This algorithm dynamically adjusts the timing based on the backend server response latency, ensuring that the wake-up request is sent at the most effective moment to minimize latency.

- **Experimental Validation:** Through rigorous experimentation and analysis, we empirically demonstrate the potential benefits of our proposed wake-up mechanism in microservices scenarios. Our experiments showcase a notable reduction (approximately 33% in HDSearch and 18% in Router, see section 2.4 for more details on the experiments) in response latency, highlighting the efficacy of proactive wake-up strategies in optimizing system performance.
- **Insights into Microservices Optimization:** By shedding light on the impact of CPU wake-up latency on microservices performance, our research provides valuable insights into the optimization of microservices architectures. Our findings contribute to a deeper understanding of the challenges and opportunities associated with latency-sensitive distributed computing environments.

Overall, our contributions offer practical insights and solutions aimed at improving the operational efficiency and responsiveness of modern distributed systems.

2 Background

2.1 Microservices and their Latency Concerns

Microservices architectures decompose applications into smaller, independent services, each responsible for a specific functionality and communicating with others through APIs. This modular approach offers several advantages, such as faster development cycles, easier deployment, and improved scalability. By breaking down a monolithic application into discrete services, developers can update or scale individual components without affecting the entire system. This flexibility accelerates development and deployment processes, allowing teams to respond swiftly to changing business requirements or technological advancements.

However, this decomposition into microservices introduces new challenges, particularly concerning communication overhead and latency. Each microservice must interact with others to fulfill a complete user request, often through network calls or inter-process communication. These interactions can introduce significant latency due to the additional network hops and the overhead of serializing and deserializing messages. Furthermore, the increased number of network calls can lead to higher resource consumption and potential bottlenecks, especially under high load conditions.

A critical aspect of optimizing microservices performance involves managing CPU states. Modern CPUs employ various power-saving modes, known as C-states, to conserve energy during periods of low activity. When a CPU is in a deeper C-state, such as C6, it consumes less power but requires more time to transition back to the active state (C0) where it can process tasks. This transition latency can significantly impact the responsiveness of microservices, as each incoming request may need to wait for the CPU to wake up fully.

Consider a typical microservices architecture where services like authentication, user data retrieval, and recommendation engines work together to process a user query. When a request arrives, the CPU handling the request might be in a deep sleep state due to low system load. The time taken to transition from a deep C-state to C0 adds to the total latency experienced by the user, which is particularly problematic for latency-sensitive applications like real-time analytics or high-frequency trading systems.

An example of a microservices architecture, illustrating the interactions between different services is shown in Figure 2.4.

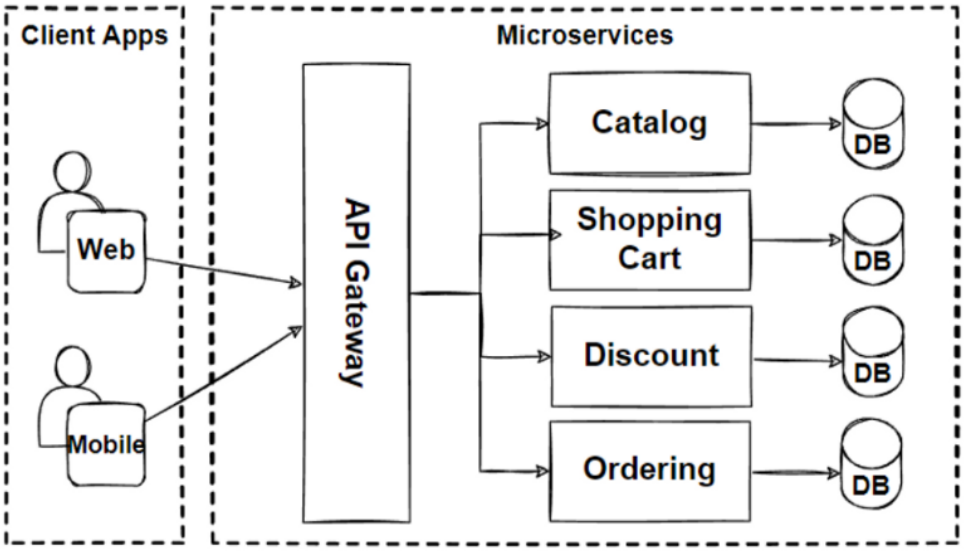


Figure 2.1: Microservices Architecture Example [1]

2.2 gRPCs

gRPC, or Remote Procedure Call, is an open-source remote procedure call (RPC) framework initially developed by Google (the 'g' initial stands for Google). It facilitates communication between distributed systems by enabling clients and servers to invoke methods on remote services as if they were local objects. gRPC leverages HTTP/2 as its transport protocol, providing features such as bidirectional streaming, multiplexing, and header compression, which enhance performance and efficiency in communication between services.

At its core, gRPC utilizes Protocol Buffers (protobuf) as its interface definition language (IDL) for defining the structure of messages and services. Protocol Buffers offer a concise and language-neutral mechanism for serializing structured data, enabling efficient data transmission across network boundaries. This abstraction simplifies the development process by providing a unified interface for defining service contracts and message formats, thereby promoting interoperability across different programming languages and platforms.

gRPC facilitates efficient and scalable communication between distributed systems through a lightweight, high-performance RPC framework. By leveraging Protocol Buffers and HTTP/2, gRPC offers a streamlined approach to building and integrating microservices, enabling developers to create robust and interoperable systems with ease.

2.2.1 Grpc.io Examples

We utilized a straightforward, open-source gRPC example program encompassing both a client and a server component from grpc.io [2,3]. This choice was deliberate, aiming for simplicity to

facilitate ease of understanding, tracking requests, and managing the implementation process. By opting for a simple example, we wanted to handle the intricacies of the gRPC framework while focusing on the core functionality of the client-server communication.

In this example, the client initiates communication by sending a concise “hello from client” grpc message to the server. Upon receiving this message, the server promptly responds with a corresponding “hello from server” grpc message. This streamlined exchange encapsulates the fundamental interaction between a gRPC client and server, allowing for a clear demonstration of communication flow and message handling.

This example has step-by-step instructions to be implemented with the following programming languages:

- C++
- C#
- Dart
- Go
- Java
- Kotlin
- Node
- Objective-C
- PHP
- Python
- Ruby
- WebJS

For the purpose of this research, we used the Python and C++ implementations only.

2.3 Linux Power Management and Idle Governors

This research focuses on the impact of CPU power management on microservices performance in Ubuntu 20.04 LTS. Ubuntu 20.04 LTS was chosen due to its status as a modern and widely used operating system. This choice ensures the utilization of an already-optimized idle governor [4], a crucial component for this research. Idle governor implementations in older versions of Ubuntu, such as 18.04 LTS, may not offer the same level of accuracy in predicting idle states compared to their newer counterparts.

Modern operating systems, like Ubuntu 20.04 LTS, employ a two-pronged approach to manage CPU performance and power consumption. The first layer is handled by the CPU frequency governor [5]. This governor dynamically adjusts the CPU's clock speed based on workload, ensuring optimal performance during high utilization periods and conserving power during low activity.

Complementing the frequency governor, the idle governor focuses specifically on optimizing CPU states when the system is experiencing low utilization. It utilizes various sleep states, known as C-states (C0, C1, C2, etc.). C-states are CPU sleep states that progressively trade off power consumption for wake-up time. C0 represents the fully active state with the highest power consumption and the fastest wake-up time. Deeper C-states (C1, C2, etc.) progressively reduce power consumption but also increase the time it takes for the CPU to wake up from sleep. The idle governor intelligently transitions the CPU between these states, aiming for a balance between power efficiency and responsiveness.

In the Linux kernel, power management plays a crucial role in optimizing energy efficiency while maintaining system responsiveness. One of the key components of Linux power management is the CPU idle governor, which determines the appropriate CPU idle state for each CPU core based on system workload and power-saving objectives.

2.3.1 Menu and Ladder Idle Governors

Two commonly used governors are the `menu` and `ladder` governors. The `ladder` governor is typically utilized in systems reliant on periodic timer-tick interrupts. This governor initiates the idle state transition by first moving the CPU core into a shallow idle state when it becomes idle. If the idle time since the core last woke up exceeds a predefined threshold, the governor progressively transitions the core to deeper idle states in stages. These state transitions are triggered by timer interrupts known as ticks. However, in tickless systems where periodic timer interrupts are disabled, the `ladder` governor's control often remains in shallow states for prolonged periods, thereby hindering the maximization of power savings.

In real-time systems, concerns about performance degradation due to periodic timer interrupts lead to the adoption of tickless configurations. Consequently, the `ladder` governor may not be suitable for such systems. Moreover, the `ladder` governor faces challenges in reducing wake-up latency when transitioning from deep idle states. This latency can impact real-time performance, especially when tasks arrive while the CPU is in a deep idle state, as the governor struggles to quickly recover from this state.

On the other hand, the `menu` governor serves as the default governor in the Linux kernel. This governor maintains records of the last 8 idle times for each CPU core and utilizes this data to estimate the next idle period. Specifically, if the deviation of the recorded idle times falls

below a certain threshold or if the mean is significantly smaller than the standard deviation, the mean is considered a reasonable interval and is adopted as the next idle period. However, if these conditions are not met, the governor rejects the largest observed idle time and continues estimating the next idle time. The estimated idle time is then compared with the wake-up latency from the current idle state. If the two values are approximately equal, it is determined that the idle time in the current state was too short, and the governor transitions the core to a shallower state to reduce latency.

While the `menu` governor is effective for regular workloads, it struggles to accurately estimate idle times for sudden irregular workloads. Additionally, similar to the `ladder` governor, the `menu` governor faces challenges in suppressing wake-up latency when transitioning from deep idle states, especially in scenarios where tasks arrive during these states.

2.3.2 Ubuntu 20.04 Power Management

Ubuntu 20.04 LTS, as a widely used Linux distribution, incorporates power management features aimed at optimizing energy efficiency while ensuring system responsiveness. The default power management settings, including the selection of idle governors, are designed to strike a balance between power savings and performance.

In Ubuntu 20.04, users can customize power management settings through utilities like `cpupower` and `tlp`, allowing fine-tuning of CPU frequency scaling, idle state transitions, and other parameters. Additionally, kernel updates and patches may introduce improvements to power management algorithms and idle governors, further enhancing system efficiency.

2.4 Benchmarks

For our Microservice benchmarks, we utilized a containerized version of the MicroSuite (µSuite) benchmark [6], originally developed at the University of Michigan. The containerized version was created by the University of Cyprus' xilab [7].

µSuite is a suite of OLDI services that are each composed of front-end, mid-tier, and leaf microservice tiers. µSuite includes four OLDI services that incorporate open-source software: a content-based high dimensional search for image similarity — HDSearch, a replication-based protocol router for scaling fault-tolerant key-value stores — Router, a service for performing set algebra on posting lists for document retrieval — Set Algebra, and a user-based item recommender system for predicting user ratings — Recommend. µSuite was originally written to evaluate OS and network overheads faced by microservices.

OLDI applications, such as web search, advertising, and online retail, constitute a significant

portion of data center workloads and are crucial for meeting soft real-time deadlines, often specified as Service Level Objectives (SLOs). Unlike traditional monolithic architectures, modern OLDI applications are composed of numerous distributed microservices interconnected via standardized Remote Procedure Call (RPC) interfaces, such as Google's Stubby and gRPC or Facebook/Apache's Thrift.

Each microservice in the µSuite comprises three tiers: a front-end, mid-tier, and leaf microservice. These microservices are designed to execute within single-digit milliseconds, necessitating sub-millisecond median latencies to meet tight SLOs. For instance, microservices must handle tasks serially, with a single user query often traversing multiple microservices in a pipeline. The pressure to optimize microservice latency continually mounts as OLDI datasets and applications grow, with compositions of increasingly complex interactions.

This µSuite Fork has been amended to achieve the following:

- Correct and confirm all the installation/compilations commands to run on Ubuntu Linux 18.04 and 20.04
- Provide instructions to compile and run docker and prepare a docker image with the complete µSuite for easier deployment
- Provide instructions and the configuration to run the applications on single node using docker-compose.yaml
- Provide intrusctions and the configuration to run the applications on multiple nodes using docker-compose-swarm.yml
- Provide instructions and source code to run the application on single node allowing the system to enter c6.

2.4.1 HDSearch

This research investigates the impact of CPU power management on the performance of microservices architectures. To achieve this, a well-established microservice benchmark, HDSearch [8], was employed. HDSearch implements a core functionality commonly found in search engines.

Service Description: HDSearch simulates a multi-node microservice architecture consisting of three distinct tiers:

Microservice Architecture:

- **Client:** The client tier represents the user and initiates the search query by providing an image. It expects a response containing a list of similar images.

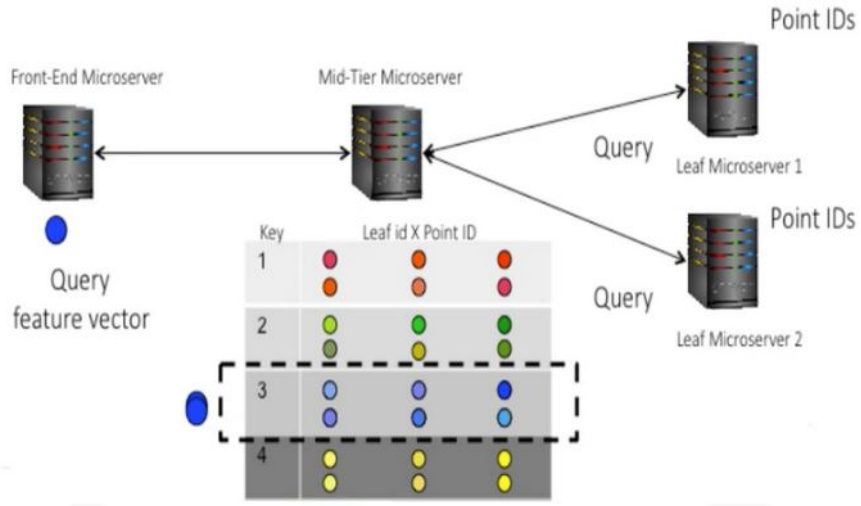


Figure 2.2: High Level Architecture of HDSearch.

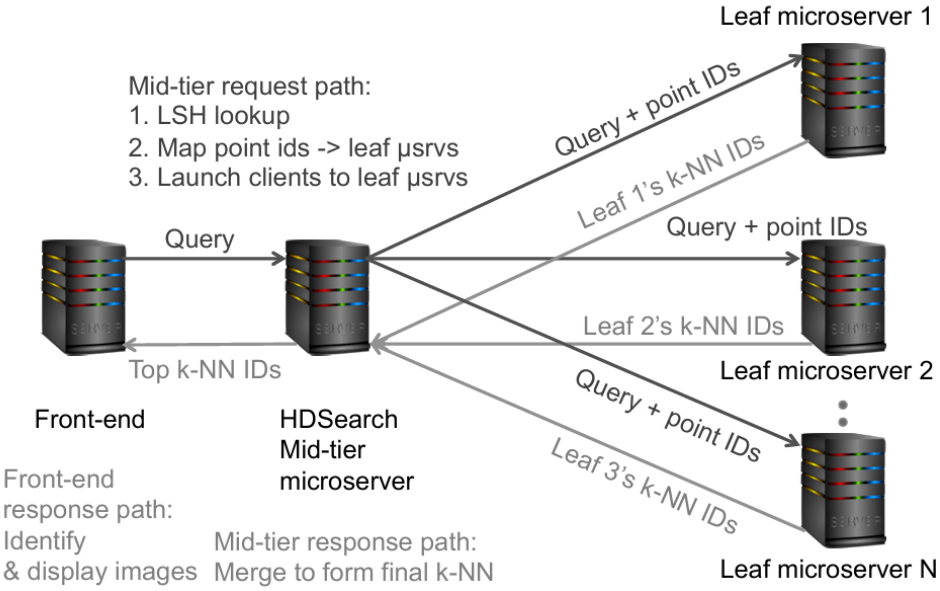


Figure 2.3: Back end request and response pipelines of HDSearch.

- **Mid-tier:** The mid-tier acts as an intermediary between the client and the leaf servers. Upon receiving a search request, it retrieves the relevant feature vectors for the queried image and utilizes Locality-Sensitive Hashing (LSH) tables to identify potential matches across the leaf servers. The mid-tier then distributes a limited number (defined by the `FIXEDCOMP` constant) of these feature vectors to each leaf server for further processing.
- **Leaf Servers (Buckets):** Each leaf server represents a microservice instance responsible for comparing the received feature vectors against its local data to identify potential matches. Finally, it transmits the results back to the mid-tier.

Advantages: The selection of HDSearch as the benchmark for this research offers several advantages:

- **Real-world Relevance:** HDSearch mimics a functionality commonly used in search engines, making the results applicable to real-world scenarios.
- **Controlled Complexity:** The benchmark offers a well-defined architecture with distinct tiers, enabling focused investigation of CPU power management's impact on each tier.
- **Open-Source Availability:** The open-source nature of HDSearch (available on GitHub [8]) fosters transparency and reproducibility of the research findings. HDSearch is an easier-to-use microsuite example, it has its own repository with step-by-step instructions, which makes it easier implementation.

2.4.2 Router

Service Description: Router's primary functionality revolves around routing key-value store requests to memcached deployments while abstracting the routing and redundancy logic from clients. This abstraction allows clients to interact with Router seamlessly, without needing to manage the complexities of memcached hosts directly. Moreover, Router incorporates replication-based protocol routing to ensure fault tolerance in memcached deployments.

One notable solution in this domain is McRouter, a memcached protocol router developed by Facebook to scale memcached deployments. McRouter employs efficient routing strategies and offers various features such as connection pooling, prefix routing, replicated pools, production traffic shadowing, and online reconfiguration. It has demonstrated the capability to handle up to 5 billion queries per second (QPS).

To address the limitations of traditional memcached systems while drawing insights from McRouter, we introduce a simplified version called Router. Router is designed to route client requests to appropriate memcached servers efficiently and provide fault tolerance for large-scale deployments.

Router's primary functionality revolves around routing key-value store requests to memcached

deployments while abstracting the routing and redundancy logic from clients. This abstraction allows clients to interact with Router seamlessly, without needing to manage the complexities of memcached hosts directly. Moreover, Router incorporates replication-based protocol routing to ensure fault tolerance in memcached deployments.

Router's functionality can be described in a series of stages. In the first stage, Router parses clients' requests and forwards them to the route computation code. This code utilizes the SpookyHash algorithm, a proven well-distributed hashing algorithm, to distribute keys from clients' requests uniformly across destination memcached servers. SpookyHash is chosen for its efficiency, low collision rate, and compatibility with various key data types.

In the final stage, Router invokes internal client code to forward the requests to specific destination memcached servers. Notably, Router maintains a single TCP connection to each destination server per Router thread, ensuring efficient communication and resource utilization.

For large-scale memcached deployments, Router employs replicated key-value store data pools to address the challenges of overwhelming client connections and ensuring high availability of critical data. By replicating data across multiple servers, Router spreads the load and provides fault tolerance, enhancing the reliability of the overall system.

Router is implemented as a suite of microservices, each serving a specific role in the system architecture:

- **Front-end Microservice**: Provides a client library that interfaces with Router through a gRPC interface. This microservice abstracts the details of communication with Router from the clients.
- **Mid-tier Microservice**: Utilizes SpookyHash to distribute keys uniformly across memcached servers and routes get or set requests accordingly. It also implements replication for fault tolerance, ensuring that data is replicated across multiple servers.
- **Leaf Microservice**: Acts as a communication wrapper around memcached server processes. It handles multiple concurrent requests from the mid-tier microservices, translating and forwarding queries to the local memcached server instances.

2.4.3 Benchmarks Objectives

The primary objective of the benchmarks is to investigate the efficacy of employing a pre-query request strategy in reducing the overall response time during data transfer processes. Specifically, we aim to determine whether sending a preliminary request before the actual data transfer can yield tangible reductions in response latency.

Through systematic experimentation and analysis, we seek to quantify the impact of pre-query

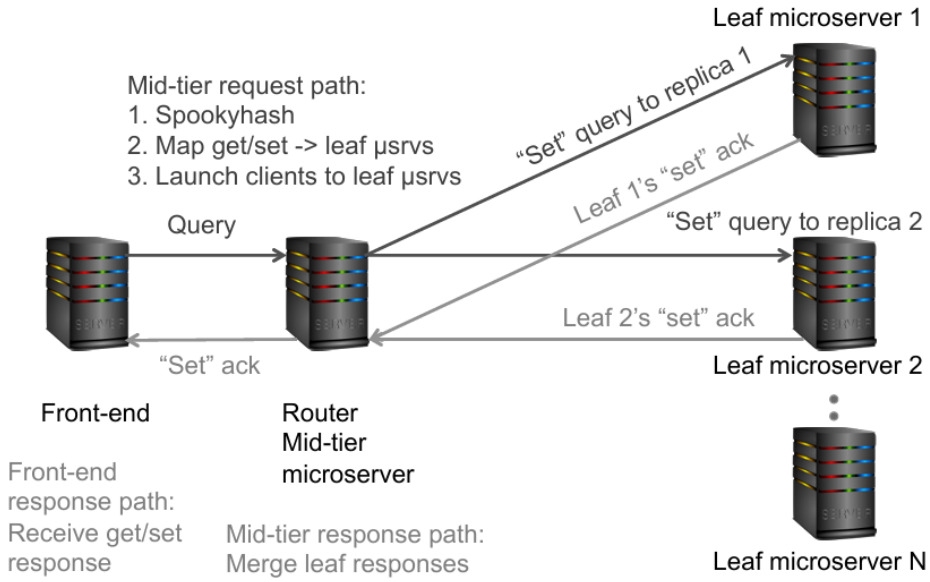


Figure 2.4: Back end request and response pipelines of Routers.

requests on response time reduction across varying system configurations and workloads. By comparing the response times of scenarios with and without pre-query requests, we aim to elucidate the potential benefits and limitations of this optimization strategy in real-world applications.

Furthermore, the benchmarks aim to contribute to a deeper understanding of the underlying mechanisms governing response time optimization. By examining the interplay between pre-query requests, interrupt handling efficiency and system responsiveness, we tried to uncover how communication protocols and system architectures work behind the scenes.

In summary, the benchmarks aim to assess the feasibility and effectiveness of integrating pre-query request strategies to enhance overall system performance and responsiveness.

2.5 Hill Climbing Algorithm

Hill Climbing [9–11] is a local search algorithm used for solving optimization problems. It starts with an initial solution and iteratively moves to a neighboring solution with better value, aiming to reach the peak (or minimum) of the solution space.

The algorithm maintains a current solution and explores neighboring solutions by making incremental changes to it. At each iteration, it selects the neighboring solution with the highest (or lowest) value and moves to it, considering only improvements over the current solution.

The process continues until no better neighboring solution can be found or a stopping criterion is

met. The algorithm may terminate at a local optimum, where no neighboring solution provides further improvement, or at a predefined termination condition.

Hill Climbing is simple yet effective for optimization tasks where the solution space is relatively smooth and continuous. However, it may get stuck in local optima and fail to find the global optimum in more complex solution spaces.

Below is the pseudocode for the Hill Climbing algorithm:

Algorithm 1 Hill Climbing Algorithm

```
Input: Initial solution s_0
Output: Optimized solution s
s ← s_0
while termination condition not met do
    s_best ← null
    for each neighbor s' ∈ neighbors(s) do
        if value(s') > value(s_best) then
            s_best ← s'
    if value(s_best) > value(s) then
        s ← s_best
    else
        break
return s
```

In this pseudocode:

- The algorithm starts with an initial solution s_0.
- It iterates through the neighboring solutions, comparing their values.
- If a better neighboring solution is found, it updates the current solution to this better neighbor.
- The process continues until no better neighboring solution is found or a predefined termination condition is met, such as an amount of iterations or time.

3 Proactive Wake-up Mechanism

In modern distributed computing environments, optimizing system responsiveness is essential for achieving efficient performance, particularly in scenarios involving microservices architectures. Our approach to enhance responsiveness involves implementing a proactive wake-up mechanism, which aims to minimize latency by preemptively waking up system components before they are needed.

The proactive wake-up mechanism operates on the principle of anticipating upcoming tasks or requests and initiating necessary preparations in advance. This proactive approach mitigates the overhead associated with latency-inducing operations, such as CPU wake-ups from deep sleep states (C-States) or network initialization.

At its core, the proactive wake-up mechanism relies on timely signaling and coordination between different components within the system. For instance, in a microservices architecture, the mechanism may involve a hierarchy of communication channels, with higher-level components signaling lower-level components to prepare for incoming requests.

The implementation of the proactive wake-up mechanism typically involves several key steps:

1. **Signaling and Coordination**: Upon detecting an impending task or request, higher-level components within the system signal lower-level components to initiate preparation steps. This signaling mechanism serves as an early notification system, allowing components to proactively wake up or allocate resources in anticipation of upcoming demands.
2. **Preparation and Wake-up Procedures**: Lower-level components respond to the signaling cues by initiating preparation procedures, such as waking up from low-power states, initializing network connections, or loading necessary resources into memory. These proactive steps aim to minimize the latency associated with task execution by reducing the time required for component activation and readiness.
3. **Optimization and Fine-tuning**: The proactive wake-up mechanism undergoes iterative refinement and optimization to ensure its effectiveness in different workload scenarios. Fine-tuning parameters enable the mechanism to adapt dynamically to changing system conditions and workload patterns. In this research, we also aimed to fine-tune the appropriate interval time between the pre-query request (wake-up call) and the original query, using a Hill Climbing algorithm.

In this research, a socket send request was utilized as part of the proactive wake-up mechanism due to its speed and lightweight nature. However, it is worth noting that there may exist alternative methods or technologies that could potentially offer even faster and/or lighter

approaches to achieving the same goal. Exploring these alternatives and evaluating their suitability for specific use cases could be a valuable direction for future research in this area.

Overall, the proactive wake-up mechanism represents a proactive approach to latency reduction in distributed computing environments. By anticipating and preparing for upcoming tasks or requests, the mechanism optimizes system responsiveness and minimizes latency, ultimately enhancing the overall efficiency and performance of the system.

3.1 Scenarios for Proactive Wake-Up Mechanism

The proactive wake-up mechanism is particularly beneficial in distributed systems with distinct stages of request processing involving multiple components. This mechanism can be effectively applied in scenarios such as:

- **Client-Midtier-Backend Architectures:** In distributed architectures where a client sends a request to a midtier component, which then forwards it to a backend server, the midtier can utilize the proactive wake-up mechanism to signal the backend server to prepare for the incoming request. This ensures that the backend server is ready to process the request promptly, minimizing latency.
- **Microservices with Dependent Services:** In microservices environments where services depend on each other to fulfill a request, a service receiving an initial request can preemptively signal dependent services to wake up and prepare for subsequent processing. This approach reduces the cumulative latency across the service chain.
- **Pre-Scheduled Tasks:** In scenarios where tasks are scheduled to run at specific intervals, components can be preemptively awakened shortly before the task execution time, ensuring that they are fully operational and ready to process tasks immediately as they arrive.

In these scenarios, the proactive wake-up mechanism leverages the knowledge of upcoming requests or tasks to initiate preparatory actions in advance, thereby reducing overall latency.

3.1.1 Application in Our Experiments

In our experiments, we utilized the proactive wake-up mechanism within a setup involving three nodes: a client, a midtier, and a bucket. The midtier is responsible for coordinating the request flow and employs the proactive wake-up mechanism to ensure that the bucket is ready to handle incoming queries efficiently.

- **Client Node:** The client node initiates requests that are directed towards the midtier.

- **Midtier Node:** Upon receiving a request from the client, the midtier sets a flag to signal the proactive wake-up mechanism. This mechanism involves sending a lightweight pre-query request to the bucket to wake it up and prepare it for the subsequent actual query.
- **Bucket Node:** The bucket node, which might be in a low-power state, receives the pre-query request and transitions to the active state, ready to process the actual query sent by the midtier.

A demonstration is shown in Figure 3.1.

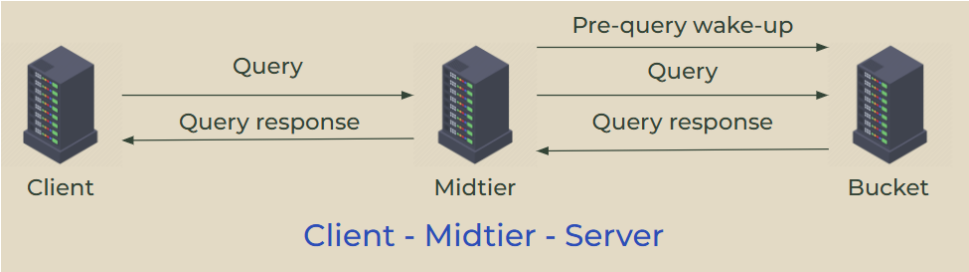


Figure 3.1: Client-Midtier-Backend Architecture.

We aimed to integrate the pre-query request strategy, utilizing socket send requests, into the HDSearch and Router experiments. In this setup, a dedicated server is deployed within the bucket, tasked with listening for the pre-query request. Both the listening server and the bucket operate on a single core. Within the midtier, a helper thread manages the transmission of pre-query requests. Upon receiving a request from the client, the midtier sets a flag indicating to the helper thread to dispatch a pre-query request to the bucket, effectively initiating its wakeup process.

The HDSearch and Router send requests from the client to the midtier following the exponential distribution. Notably, it does not matter which distribution is used, all distributions yield better results with pre-query request.

We utilize a `std::atomic<bool>` variable named `sendRequestFlag`, initialized to `false`, to facilitate communication from the main thread to the helper thread. This variable serves as a signaling mechanism, indicating to the helper thread when to send the pre-query request.

Within the helper thread, the main loop continuously waits for the `sendRequestFlag` to be set to `true` before proceeding with sending the pre-query request. This waiting mechanism is achieved through a busy-wait loop:

```
while (true)
        {
            // Wait until the flag is set to true
            while (!sendRequestFlag.load(std::memory_order_acquire))
```

```
        {
        }
        if (pre-query-request-interval > 0)
        {
            struct timeval start_time, end_time;
            gettimeofday(&end_time, NULL);
            gettimeofday(&start_time, NULL);

            // Instead of sleep(pre-query-request-interval), I actively
            // read the current timestamp and check
            // whether the <pre-query-request-interval> us have passed
            // Also, I need to check that while waiting, I still want to
            // send the pre-query request
            while (end_time.tv_usec - start_time.tv_usec <
            pre-query-request-interval &&
            sendRequestFlag.load(std::memory_order_acquire))
            {
                gettimeofday(&end_time, NULL);
            }
        }

        if (sendRequestFlag.load(std::memory_order_acquire))
        {
            // Once the pre-query-request-interval us have passed,
            // send the request
            send_request();
        }

        // Reset the flag
        sendRequestFlag.store(false, std::memory_order_release);
    }
```

After setting the flag to `true`, indicating the necessity of a pre-query request, the helper thread proceeds to send the pre-query request to the bucket. This however, comes with the cost of a greater power consumption in the midtier.

Additionally, we conducted experiments involving sending the pre-query request after a certain number of microseconds to assess potential benefits. The methodology involved the midtier setting the flag to `true` upon receiving a request from the client, triggering the helper thread to

handle the sending of the pre-query request. Notably, the helper thread executes the pre-query request transmission only if the pre-query request interval time exceeds the processing time required for the midtier to dispatch the request to the bucket. To clarify, the pre-query request interval time is the time between the pre-query request and the actual request, and NOT the time between two consecutive pre-query requests.

In other words, the midtier resets the flag to `false` once it completes its processing, signaling to the helper thread that it has finished. Meanwhile, the helper thread remains in a busy-waiting state, continually checking the flag's status. If the flag remains `true`, indicating that a pre-query request is still required, the helper thread proceeds with the sending process. This approach ensures that the pre-query request is sent only when necessary.

3.2 Optimization Using a Hill Climbing Algorithm

The critical challenge in our scenario lies in determining the optimal timing for sending the pre-query request. The midtier must find the appropriate waiting interval between receiving the query from the client and dispatching the pre-query request to the bucket. This interval must strike a delicate balance, allowing sufficient time for the bucket to awaken and prepare for the impending query processing task while minimizing any unnecessary delays due to the pre-query processing.

In other words, the challenge lies in determining the optimal waiting time for sending the pre-query request. If the request is sent too early, the bucket may be awakened unnecessarily and enter a deep C-State, leading to wasted resources. Conversely, if the request is sent too late, the bucket may not be fully awake in time to process the original query efficiently because it is still processing the pre-query request, resulting in increased latency.

To address this challenge, the Hill Climbing algorithm dynamically adjusts the waiting time for the pre-query request based on the response time of the bucket. By measuring the response time and iteratively adjusting the waiting time, the algorithm aims to minimize latency and optimize the performance of the distributed system.

The process involves iteratively adjusting the waiting time for the pre-query request and measuring the corresponding response time of the bucket. The algorithm selects neighboring solutions by incrementally changing the waiting time and evaluates their performance based on the response time. It then moves to the neighboring solution with the lowest response time, iteratively refining the waiting time until an optimal solution is found.

Therefore, we employed a hill climbing algorithm to optimize the waiting microseconds for sending the pre-query request. Initially, the waiting microseconds were set to 0 and then they were incremented by 100µs, which is the initial step size. Subsequently, the step size was

reduced to 50, then 25, and so on until reaching 3 microseconds. If the algorithm found that increasing the waiting microseconds improved performance, it would adjust the waiting time accordingly, by adding the step size to the waiting microseconds. Conversely, if decreasing the waiting microseconds yielded better results, the algorithm would adapt accordingly. This iterative process allowed us to dynamically adjust the waiting time for the pre-query request, optimizing system performance and responsiveness. The waiting microseconds were changed for each 1000 requests, by comparing the average response time of these requests.

```
// Check if 1000 iterations have passed
if (process_request_count % 1000 == 0 && send_request_time >= 0)
{
    // Calculate average time
    uint64_t average_time = calculate_average_time();

    // Adjust send_request_time parameter based on comparison with previous average
    // If the pre-query request interval increased previously, then increase it now
    // (initially it will increase)
    if (increasing)
    {
        // If the new average is less, then increase the send_request_time
        if (average_time < previous_average_time)
        {
            send_request_time += step;
        }
        else // Decrease the send request time
        {
            send_request_time -= step;
            increasing = false;
        }
    }
    else
    {
        // If the new average is less, then decrease the send_request_time
        if (average_time < previous_average_time)
        {
            send_request_time -= step;
        }
        else // Increase the send request time
```

```
        {
            send_request_time += step;
            increasing = true;
        }
    }
    // Change the step (the rate which will increase or decrease the
    // pre-query request interval)
    if (step > 5)
    {
        step /= 2;
    }
    previous_average_time = average_time;
}
```

4 Research Methodology

4.1 Tools Used

4.1.1 SocWatch

In our research, we place particular emphasis on harnessing the capabilities of Intel SocWatch to track interrupts and their associated Interrupt Request (IRQ) numbers, as well as monitoring C-States residencies and tracing their transitions.

Purpose: The primary focus of our utilization of Intel SocWatch is to conduct in-depth analysis of system interrupts and C-States behavior on Intel architecture-based platforms. By leveraging SocWatch, we aim to precisely monitor interrupt handling mechanisms and C-States residency patterns, facilitating a thorough understanding of system-level performance characteristics.

Key Features:

1. **Interrupt Tracking:** Intel SocWatch enables precise tracking and analysis of system interrupts, providing detailed insights into interrupt handling mechanisms and IRQ assignments.
2. **C-States Residency Analysis:** SocWatch facilitates comprehensive monitoring of C-States residencies, which gives us the big picture of each processor's idleness.
3. **Traceability:** With Intel SocWatch, we can trace the transitions of the processor into and out of each C-state, allowing us to track the entry and exit of the processor into/from various C-States. This traceability feature enables us to correlate C-States transitions with system events and workload characteristics, and analyze when our pre-query requests wakeup the processor and what happens after this.

Analysis and Interpretation: Utilizing the data collected by Intel SocWatch, we perform comprehensive analysis and interpretation of interrupt behavior and C-States residency patterns. We analyze interrupt frequency, distribution, and latency characteristics to identify potential optimization opportunities. Similarly, we examine C-States residency durations and transitions.

4.1.1.1 Example of C-States Transitions Trace

An example of a C-States transitions trace is as shown in Table 4.1:

In this trace, each row represents a sample captured during system monitoring. The first column denotes the sample number, the second column represents the continuous time in milliseconds

Table 4.1: C-States Transitions Example as Shown in SocWatch

Sample #	Continuous Time (ms)	C-State	Duration (ms)
416	1169.26	CC0	0.56
417	1178.80	CC6	9.54
418	1179.59	CC0	0.78

in which the processor transitioned out of the C-State, the third column indicates the C-State and the last column specifies the duration (in milliseconds) for which the processor remained in that particular C-State.

For instance, the first row indicates that the processor stayed at C0 for 0.56 ms, which is until 1169.26 ms since the beginning of the trace collection, and the second row indicates that the processor transitioned to C6 at 1169.26 ms and stayed in this state until 1178.80 ms, resulting in a duration of 9.54 ms.

4.1.1.2 Example of IRQs Trace

An example of IRQs captured during monitoring is provided below in Table 4.2:

Table 4.2: IRQ Transitions Example as Shown in SocWatch

Sample #	Continuous Time (ms)	IRQ No	Interface	Duration (ms)
209	1178.80	39	i40e-enp94s0f0-TxRx-1	10.11
210	1179.59	39	i40e-enp94s0f0-TxRx-1	0.79
211	1189.41	39	i40e-enp94s0f0-TxRx-1	9.81

In this example, each row represents an IRQ event captured during system monitoring. The columns provide the following information: the sample number, continuous time in milliseconds, IRQ number, interface description, and duration since the last IRQ event.

For instance, the first row indicates that an IRQ 39 event occurred 10.11 ms prior to 1178.80 ms since the beginning of the trace (therefore at 1168.69 ms), and the second row indicates that an IRQ 39 event occurred at 1178.80 ms, and another interrupt arrived at 1179.59 ms, resulting in a duration of 0.79 ms without another interrupt occurring.

4.1.2 Turbostat

In addition to Intel SocWatch, we also employ Turbostat as a complementary tool in our research endeavors. Turbostat offers a lightweight and straightforward solution for monitoring processor performance metrics.

Purpose: The primary purpose of integrating Turbostat into our toolkit is to gain insights into processor performance metrics, including frequency, utilization, and power consumption.

Unlike Intel SocWatch, Turbostat also provides insights into processor behavior and C-States residencies. However, it distinguishes between software hints and real C-States. Software hints refer to instructions or signals provided by the software to the processor, indicating the desired C-State it should enter into. On the other hand, real C-States represent the actual C-States that the processor was observed to be in.

Key Features:

1. **Lightweight Monitoring:** Turbostat offers a lightweight monitoring solution, making it easier to implement and less resource-intensive compared to Intel SocWatch.
2. **Processor Metrics:** Turbostat provides essential metrics such as CPU frequency, utilization, and power consumption, offering valuable insights into processor behavior.
3. **Real C-States Residencies:** Turbostat offers real-time monitoring of C-States residencies, allowing us to observe the processor's idleness and power-saving mechanisms.

Limitations: While Turbostat provides valuable insights into processor performance metrics, it lacks the traceability feature offered by Intel SocWatch. As a result, Turbostat only offers software hints and real C-States residencies, without providing detailed trace information on transitions between C-States.

Analysis and Interpretation: Despite its limitations, Turbostat serves as a valuable tool for assessing processor performance and power management strategies. By analyzing the data collected through Turbostat, we gain insights into processor behavior and power-saving mechanisms, enabling us to optimize system performance and efficiency.

Moreover, to ensure the accuracy and reliability of our findings, we cross-validate the results obtained from Turbostat with those from Intel SocWatch. This cross-validation process enhances the robustness of our analysis and interpretation, providing a comprehensive understanding of system-level performance characteristics and power management strategies.

4.2 Fixed CPU Frequency

In all our experiments, we maintained a fixed CPU frequency to eliminate its variability as a factor affecting the results. Although this factor did not significantly impact the outcomes, ensuring a consistent frequency helped maintain the stability and reproducibility of our experiments.

To achieve a fixed CPU frequency, we utilized the `intel_pstate=disable` option in the `GRUB_CMDLINE_LINUX_DEFAULT` parameter within the `/etc/default/grub` configuration file. This option disables the Intel P-state driver, allowing manual control over the CPU

frequency.

Additionally, we used the command `sudo cpupower frequency-set -f 2200MHz` to set the CPU frequency to 2200MHz. This command ensured that the CPU operated at a consistent frequency throughout our experiments, enabling accurate and reliable measurements across different test scenarios.

4.3 SMT Disabled

For our experiments, we disabled Simultaneous Multithreading (SMT), also known as Hyper-Threading, on the CPU. By doing so, we ensured that each core was dedicated to executing only one thread at a time, effectively restricting concurrent execution threads per core.

The rationale behind disabling SMT was to streamline our experimental setup and ensure clearer and more straightforward results. With SMT disabled, we aimed to reduce the potential complexity introduced by concurrent execution threads sharing resources within a single core.

To disable SMT, we utilized the following command:

```
echo "off" | sudo tee /sys/devices/system/cpu/smt/control
```

This command effectively turned off SMT, ensuring that each physical core operated independently without the simultaneous execution of multiple threads. By enforcing this configuration, we aimed to enhance the clarity and interpretability of our experimental outcomes while maintaining consistency and reliability across our test scenarios.

4.4 IRQ-Core Bind

In our experimental setup, we aimed to achieve a clearer understanding of system performance by binding each Interrupt Request (IRQ) to a dedicated CPU core. By assigning specific cores to handle individual interrupts, we sought to maintain a consistent and predictable environment across multiple experiment runs.

The motivation behind this approach was to ensure that the system's behavior remained stable and reproducible throughout the experimental process. By dedicating specific cores to handle interrupts, we minimized the likelihood of variability in interrupt handling and CPU resource allocation, thereby facilitating more consistent and interpretable results.

In Linux, CPU affinity is managed using bitmasks. A bitmask is a binary pattern where each bit represents a CPU core. By setting specific bits in the bitmask, we can control which CPU cores are allowed to handle interrupts associated with a particular IRQ [12].

For example, if we have four CPU cores and we want to assign IRQ handling exclusively to core 0, we would set the bitmask to 0001 (in binary), indicating that only core 0 is allowed to handle interrupts for that IRQ. Similarly, if we want to assign IRQ handling to cores 0 and 1, we would set the bitmask to 0011 (in binary), allowing both cores 0 and 1 to handle interrupts.

To implement IRQ-Core binding, we utilized a script that interacted with the system's /proc filesystem to access and manipulate IRQ configurations. The key command involved setting the CPU affinity for each IRQ to its corresponding core. This was achieved by writing the calculated bitmask to the smp_affinity file located in the /proc/irq/[IRQ]/ directory.

By writing the appropriate bitmask to the smp_affinity file for each IRQ, we effectively specify which CPU cores are eligible to handle interrupts associated with that IRQ. This ensures that IRQ handling is distributed evenly across the available CPU cores, optimizing system performance and resource utilization.

4.5 Utilizing CloudLab for Experimentation

This research leveraged CloudLab, a cloud computing platform designed for reproducible research experimentation. CloudLab offers a controlled environment where researchers can rent machines with various configurations, including the specific operating system chosen for this research, Ubuntu 20.04 LTS. Users can select from a wide range of node types, each offering different hardware specifications such as processors (CPUs), and memory (RAM). This flexibility allows researchers to tailor the experimental environment to their specific needs.

A significant advantage of CloudLab is the close resemblance of its nodes to real-world servers. The hardware configurations offered by CloudLab mirror those found in production environments, ensuring the generalizability of the research findings. This close simulation minimizes the risk of results being skewed by factors unique to a particular hardware setup.

Figure 4.1 demonstrates how the nodes are chosen.

By employing a modern operating system on a controlled platform like CloudLab, this research establishes a reliable foundation for investigating the impact of CPU power management, specifically C-States managed by the idle governor, on the performance of microservices. The ability to configure the hardware characteristics further strengthens the research by enabling experimentation that closely reflects real-world deployments.

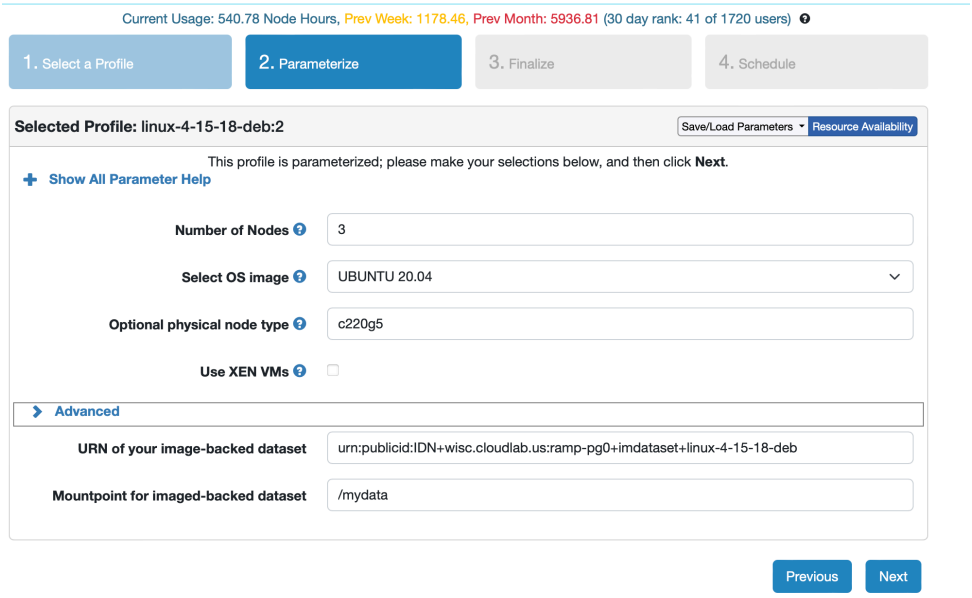


Figure 4.1: CloudLab nodes selection.

4.5.1 Limitations:

Despite the advantages offered by CloudLab, several limitations were encountered during the experimentation process. Firstly, the CloudLab nodes are available for a maximum duration of 16 hours, extendable up to 7 days and then again up to another 7 days. This limitation imposed constraints on the duration of experiments, requiring frequent manual intervention to extend the duration, leading to stress and time consumption. Additionally, after 14 days, a new experiment had to be created, potentially resulting in different node configurations and IRQ numbers, which could impact the reproducibility of results. Furthermore, the unavailability of the c220g5 nodes and the long initial node boot time posed more challenges. Moreover, setting up the experimental environment on each node, including downloading tools such as SocWatch, setting fixed frequencies, disabling SMT affinity, etc., was time-consuming and repetitive. Lastly, the nodes provided by CloudLab had limited storage capacity, which occasionally posed challenges and hindered the continuity of experiments.

4.6 Workload Distributions

In our experiments, we utilized two commonly used probability distributions: the Poisson distribution and the Exponential distribution.

4.6.1 Poisson Distribution

The Poisson distribution [13] is a discrete probability distribution that represents the number of events occurring in a fixed interval of time or space, given the average rate of occurrence. It is characterized by a single parameter, λ (lambda), which represents the average rate of occurrence of the event.

The probability mass function (PMF) of the Poisson distribution is given by:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

where:
- X is the random variable representing the number of events.
- k is the number of events that occur.
- e is the base of the natural logarithm, approximately equal to 2.71828.

4.6.2 Exponential Distribution

The Exponential distribution [14, 15] is a continuous probability distribution that represents the time between events in a Poisson process, where events occur continuously and independently at a constant average rate. It is characterized by a single parameter, λ (lambda), which represents the average rate of occurrence of the event.

The probability density function (PDF) of the Exponential distribution is given by:

$$f(x|\lambda) = \lambda e^{-\lambda x}$$

where:
- x is the time between events.
- λ is the rate parameter, representing the average rate of occurrence of events.

4.6.3 How Distributions Were Used

Example: Let's consider a scenario where we are simulating a web server that receives queries at an average rate of $\lambda = 10$ queries per second (QPS). To model the time intervals between consecutive requests with an average of 100 milliseconds (ms) between each request, we calculate the corresponding query per second (QPS), which is $QPS = \frac{1}{\text{interval time}} = \frac{1}{100\text{ ms}} = 10$.

Using the Poisson distribution with $\lambda = 10$, we can generate random numbers representing the time intervals between requests.

Note: Web service performance is typically modeled using a Poisson process for arrivals, which results in inter arrival times following the exponential distribution [16, 17].

4.7 Metrics Used

4.7.1 Average Latency

The Average Latency represents the typical or average time taken for all requests processed by a system or service. It is calculated by summing up the individual latency values for each request and then dividing the total by the number of requests.

Mathematically, the Average Latency Avg can be expressed as:

$$\text{Avg} = \frac{\sum_{i=1}^{n} \text{Latency}_i}{n}$$

Where:

- Latency_i represents the latency of the i^{th} request.
- n is the total number of requests.

The Average Latency signifies the typical duration users can expect to wait for their requests to be processed. This metric provides valuable insight into the overall performance of the system, helping to measure the level of responsiveness and efficiency in handling user requests. By monitoring the Average Latency, we can ensure that the system meets user expectations for timely responses and optimize its performance accordingly

4.7.2 99th Percentile Latency

The 99th percentile latency is a performance metric commonly used in computing and networking to quantify the responsiveness or delay experienced by a system or service. It represents the value below which 99% of measured latency values fall, indicating the latency experienced by the vast majority of requests or events.

To calculate the 99th percentile latency, all latency measurements are sorted in ascending order, and the value corresponding to the 99th percentile position is identified. In other words, if there are n latency measurements, the 99th percentile latency is the value of the latency measurement at the index $n \times 0.99$ when the measurements are arranged in ascending order.

Mathematically, the 99th percentile latency PL_{99} can be expressed as:

$$PL_{99} = \text{Latency}_{\lceil 0.99 \times n \rceil}$$

Where:

- $\text{Latency}_{\lceil 0.99 \times n \rceil}$ represents the latency measurement at the index corresponding to the 99th percentile position when the latency measurements are sorted in ascending order.
- n is the total number of latency measurements.

This metric is particularly useful for understanding the tail end of the latency distribution, capturing outliers or instances of unusually high latency that may significantly impact user experience or system performance. By focusing on the 99th percentile latency, organizations can ensure that the vast majority of users or requests experience acceptable levels of responsiveness, while also identifying and addressing potential performance bottlenecks or issues affecting a small but significant portion of users or events.

In summary, the 99th percentile latency provides valuable insights into the performance characteristics of a system, helping organizations optimize resources, improve user experience, and ensure reliable and responsive service delivery.

4.8 Applying Socket Pre-Query Request in gRPC Helloworld Experiment

In the Router gRPC Helloworld Experiment, we employed the C++ version of the gRPC Helloworld example to minimize latency and closely replicate real-world scenarios. By utilizing the C++ implementation, we aimed to achieve optimal performance and reduce overhead associated with language bindings or additional layers of abstraction.

In the gRPC Helloworld Experiment, we replicated the methodology employed in the HDSearch and Router experiments, adapting it to the gRPC framework to explore its potential benefits within this context.

In this experiment, the setup involved two distinct nodes: the client and the server. The client node was responsible for generating and transmitting a series of requests to the server node. These requests were sent utilizing the gRPC framework, allowing for efficient communication between the client and server (for further details, see section 2.2.1).

Similar to the HDSearch and Router setup, we introduced a mechanism for pre-query requests within the gRPC experiment. A dedicated server was deployed within the gRPC listener server to listen for pre-query requests from the client. Upon receiving a request from the client, the server processed the request and prepared a pre-query response, priming the server for subsequent

requests.

4.9 System Information

The cloudlab node that we used throughout my whole research was c220g5 [18]. This node has 3 sleep C-States: `C1`, `C1E` and `C6`.

The system's details are shown below:

Feature	**Details**
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
Address sizes	46 bits physical, 48 bits virtual
CPU(s)	40
On-line CPU(s) list	0-39
Thread(s) per core	2
Core(s) per socket	10
Socket(s)	2
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	85
Model name	Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
Stepping	4
Frequency boost	enabled
CPU MHz	1014.566
CPU max MHz	2201.0000
CPU min MHz	800.0000
BogoMIPS	4400.00
Virtualization	VT-x
L1d cache	640 KiB
L1i cache	640 KiB
L2 cache	20 MiB
L3 cache	27.5 MiB
NUMA node0 CPU(s)	0-9, 20-29
NUMA node1 CPU(s)	10-19, 30-39

Table 4.3: System Details

5 Results

5.1 Custom Interrupt Micro Benchmark

To evaluate the effectiveness of our pre-query request method, we developed a custom micro benchmark tool [19] inspired by socket communication. This benchmark utilizes a client-server architecture:

- **Client:** The client initiates the interaction by sending a "hello" request through a network socket.
- **Server:** The server listens for incoming socket requests. Upon receiving a "hello" request, the server responds with a "hello" message.

Our rationale for developing a custom benchmark was to create a controlled environment for studying the impact of pre-query requests on response latency. This simplified approach allowed us to isolate and analyze the specific behavior of our proposed method.

We opted to delve into the lowest level of system interaction possible. This led us to simulate queries as socket send requests, with an additional investigation into the impact of sending pre-query requests slightly before the actual queries.

Each experiment instance was conducted over 1000 queries. In this experiment, we systematically explored multiple parameters, including:

- **Sleep Interval Time:** This parameter determines the duration of sleep between each query (actual request). For experiments using the Poisson Distribution, the interval time is calculated as $1/\lambda$, where λ represents the rate of occurrence of events. For example, a sleep interval time of 100ms corresponds to a rate of 10 queries per second (QPS), which is equivalent to $\lambda = 10$.
- **C-States Enabled or Disabled:** We examined the effects of enabling or disabling C-States to discern their influence on tail latency.
- **Distribution:** We used Poisson distribution, as it is implemented in the HDSearch benchmark. However, we also changed our analysis to incorporate the Exponential distribution. This allowed us to assess the impact of real-life scenario probability distributions on query scheduling and subsequent tail latency since the distribution that is observed in most microservice scenarios is exponential [16]. Moreover, we continued to evaluate the efficacy of sending queries at fixed intervals, comparing this approach against the dynamic scheduling enabled by the Poisson and Exponential distributions.

- **Send Pre-query Request or Not:** We evaluated whether sending a pre-query request yielded tail latency improvements.
- **Pre-query Interval**: We explored various timings for sending the pre-query requests, adjusting the delay between the arrival of the actual query and the transmission of the pre-query request. This investigation aimed to identify the optimal timing strategy for preemptively waking up the processor and minimizing tail latency. Notably, the pre-query request did not alter the queries intervals. If the pre-query request interval (from the time the query arrived to the midtier until the midtier sends the pre-query request) exceeds the time needed for the query to be forwarded from the midtier to the bucket, the pre-query request will not be sent.

Firstly, we validated that our queries followed an exponential distribution in the exponential distribution implementation. Figure 5.2 illustrates the probability density function (PDF) of the request intervals, represented as a histogram. The horizontal axis represents the pre-query interval time, while the vertical axis represents the density, which sums up to 1. This histogram is compared to the PDF of an exponential distribution, demonstrating the distribution of intervals observed in the experiment. This comparison is conducted within the experiment where the average query interval time is 10000 µs ($\frac{1 \text{ second}}{100}$).

Similarly, we validated that our queries followed Poisson distribution in the Poisson distribution implementation. The PDF graph of the Poisson distribution is illustrated in Figure 5.1. While the Poisson distribution is used in HDSearch, we were not sure whether it was the best approach, because based on Cao et al. [16] "The arrival process of HTTP requests is assumed to be Poissonian and the service discipline is processor sharing", meaning that the inter-arrival times should follow an exponential distribution [17]. This is the reason we shifted to exponential distribution.

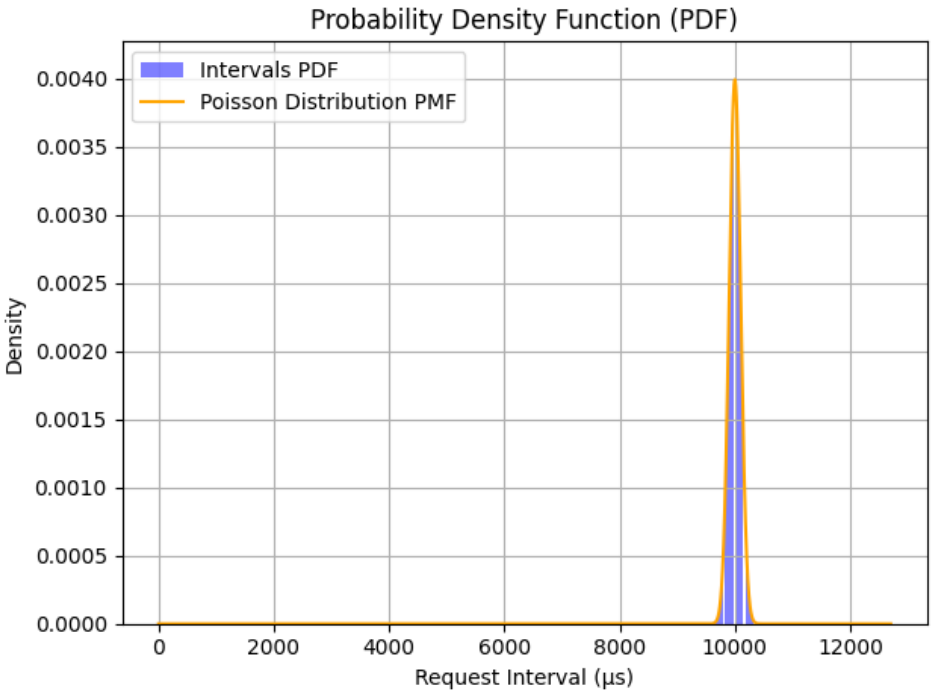


Figure 5.1: Requests Poisson Distribution with Interval Rate 1000µs.

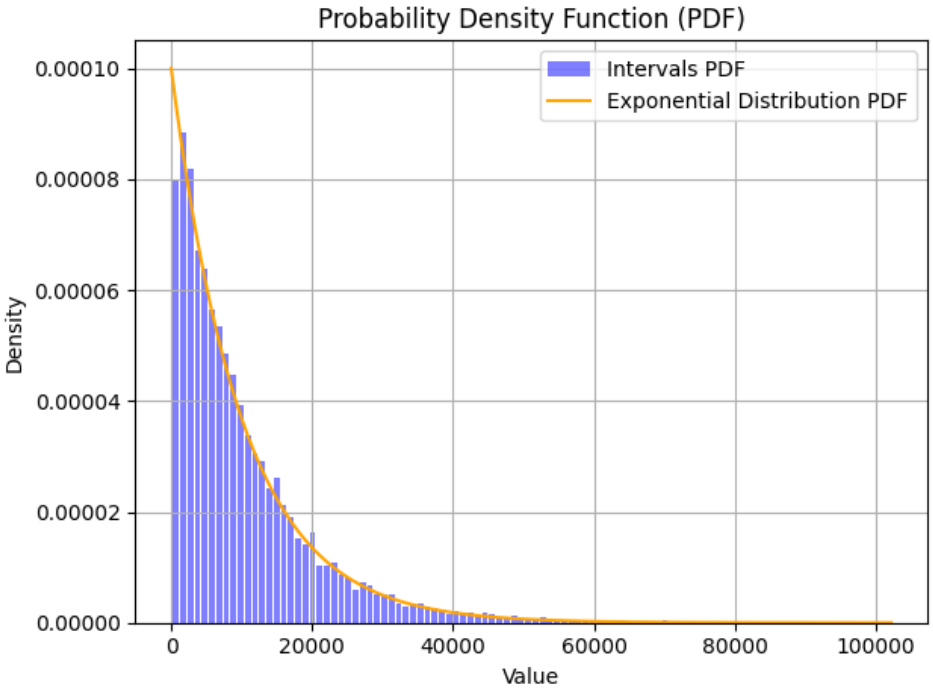


Figure 5.2: Requests Exponential Distribution Validation.

With Poisson Distribution

Table 5.1: Custom Interrupt Micro Benchmark Results with Poisson Distribution, C-States Enabled and Pre-Request Interval 10µs.

(a) Without Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	216.49	348
1	214.24	302

(b) With Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	144.68	173
1	141.70	172

Comments / Observations:

- When using a Poisson distribution with intervals ranging from 100ms to 1ms, adding a pre-request consistently results in lower average latency and 99th percentile latency compared to scenarios without pre-requests. This suggests that sending a pre-request before the actual query reduces tail latency.

With Exponential Distribution

Table 5.2: Custom Interrupt Micro Benchmark Results with Exponential Distribution, C-States Enabled and Pre-Request Interval 10 µs.

(a) Without Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	235.24	308
1	207.02	325
0.1	90.00	305

(b) With Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	199.14	258
1	163.61	243
0.1	80.02	273

Table 5.3: Custom Interrupt Micro Benchmark Results with Exponential Distribution, C-States Enabled and Pre-Request Interval 300 µs.

(a) Without Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	231.45	267
1	225.32	294
0.1	157.67	274

(b) With Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	117.65	243
1	110.02	223
0.1	103.55	385

Comments / Observations:

- When using Exponential Distribution with pre-request intervals of 10 µs and 300 µs, we observe consistent reductions in average latency and 99th percentile latency compared to scenarios without pre-requests. This indicates that incorporating pre-requests effectively improves responsiveness and reduces tail latency across varying interval durations.
- Notably, the magnitude of latency improvement varies across different interval durations. The largest difference in latency between scenarios with and without

pre-requests is observed at 1 ms intervals, indicating a substantial improvement with pre-requests. Additionally, at 0.1 ms intervals, the average latency is lower for both scenarios, but the pre-request version consistently exhibits lower both average and 99th percentile latency compared to the counterpart without pre-requests, demonstrating the continued effectiveness of pre-requests in improving overall responsiveness.

- Even with very short intervals of 10µs and longer intervals of 300µs, we observe reduced latency compared to scenarios without pre-requests. This highlights the effectiveness of pre-requests in improving responsiveness across a wide range of interval durations.

With Fixed Interval Time

These results did not follow any distribution, but the query interval time was fixed instead.

Table 5.4: Custom Interrupt Micro Benchmark Results with Fixed Interval Time, C-States Enabled and Pre-Request Interval 10 µs.

(a) Without Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	238.42	307
1	146.51	319
0.1	88.13	312

(b) With Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	204.76	255
1	139.26	246
0.1	79.55	210

Table 5.5: Custom Interrupt Micro Benchmark Results with Fixed Interval Time, C-States Enabled and Pre-Query Request Interval 300 µs.

(a) Without Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	232.02	276
1	227.06	313

(b) With Pre-Query Request.

Interval(ms)	Avg Latency(µs)	99th PL(µs)
100	114.90	130
1	115.38	207

Comments / Observations:

- Across all fixed interval timings, the inclusion of pre-requests consistently leads to reductions in both average latency and 99th percentile latency. This indicates that pre-requests effectively contribute to minimizing tail latency across various fixed interval durations.

5.1.1 Latency Comparisons With Different Parameters

We measured the average latency and the 99th percentile latency in microseconds. For further information on what 99th percentile latency is, see section 4.7.2.

The average and 99th percentile latency results are based on 1000 requests for each interval configuration.

We incorporated a phase comprising 50 warmup requests preceding the actual experiment and compared the results. The results are as shown below in graphs 5.3 and 5.4. The blue columns represent the 99th percentile tail latencies, and the orange is the average tail latencies.

Each figure contains 16 columns, divided as follows: the first 8 columns are with pre-query request enabled, and the last 8 are with pre-query request disabled. Within each group of 8 columns, they are further divided: the first 4 columns have a pre-query request time of 100 µs, and the second 4 columns have an interval of 50 µs. The pre-query request time is the time between the pre-query and the original query, and not the interval time between two consecutive pre-query requests. Additionally, within each subgroup of 4 columns, the first 2 columns have a query interval of 1000 µs, and the next 2 columns have a query interval of 100 µs. The query interval is the time between two consecutive queries. Finally, within each subgroup of 2 columns, the first column does not have a warm-up phase whereas the second is with a 50 requests warm-up phase.

We also included speedup graphs to compare each column i with its corresponding $(i + 8)$th column. Essentially, each $(i + 8)$th column represents the same experiment instance but with the pre-request disabled. These graphs illustrate the performance differences, indicating how much faster (or slower) the experiment instance with the pre-request enabled is compared to when it is disabled. The speedup comparisons are presented in Figures 5.5 and 5.6.

C-States Disabled: For experiments with C-States disabled, the average latency consistently hovered around 67 µs, with the 99th percentile latency consistently at 140 µs across all intervals tested, including 1000ms, 100ms, 10ms, 1ms and 0.1ms.

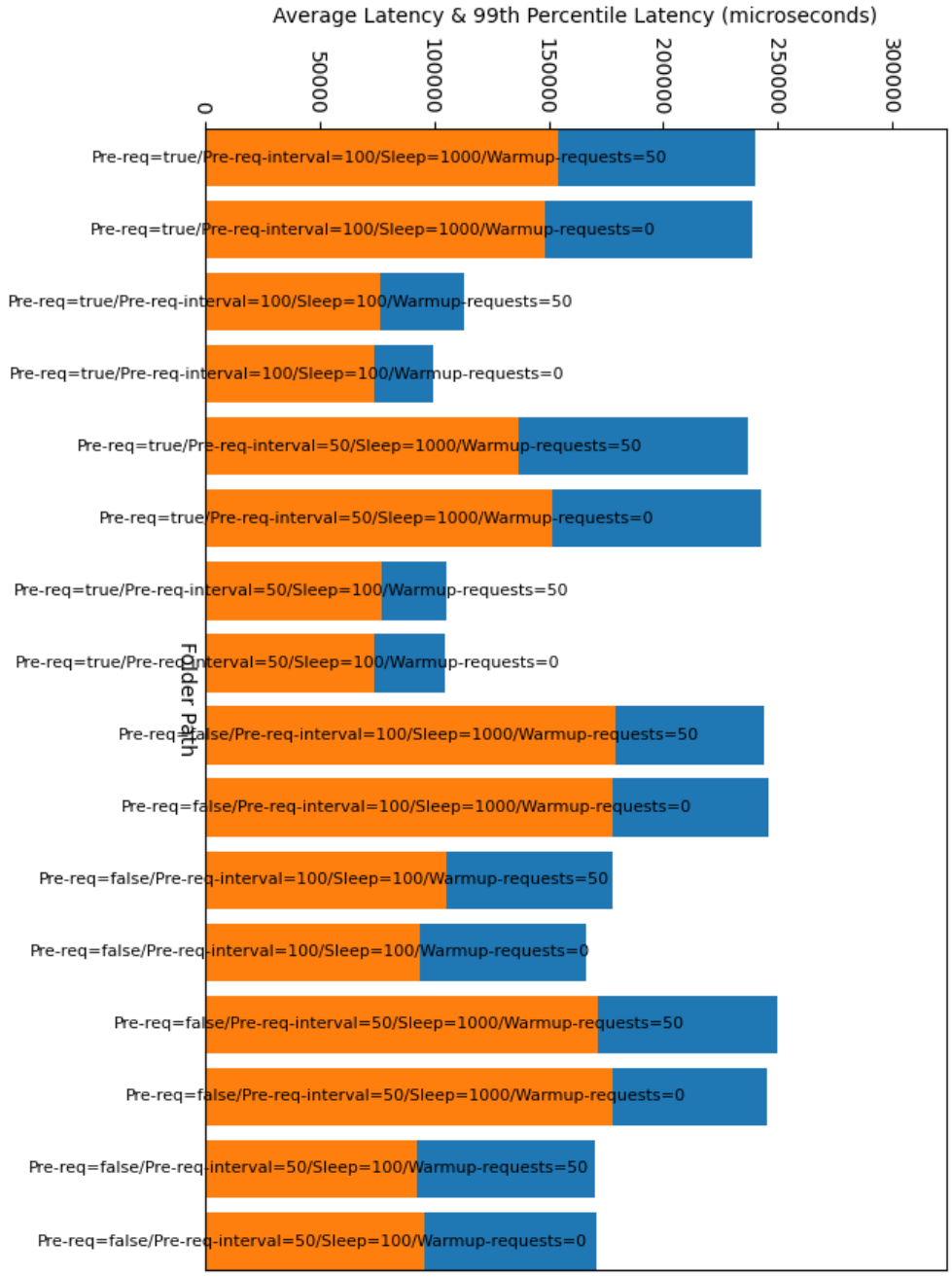


Figure 5.3: Custom Interrupt Micro Benchmark Results with Exponential Distribution and C-States Enabled.

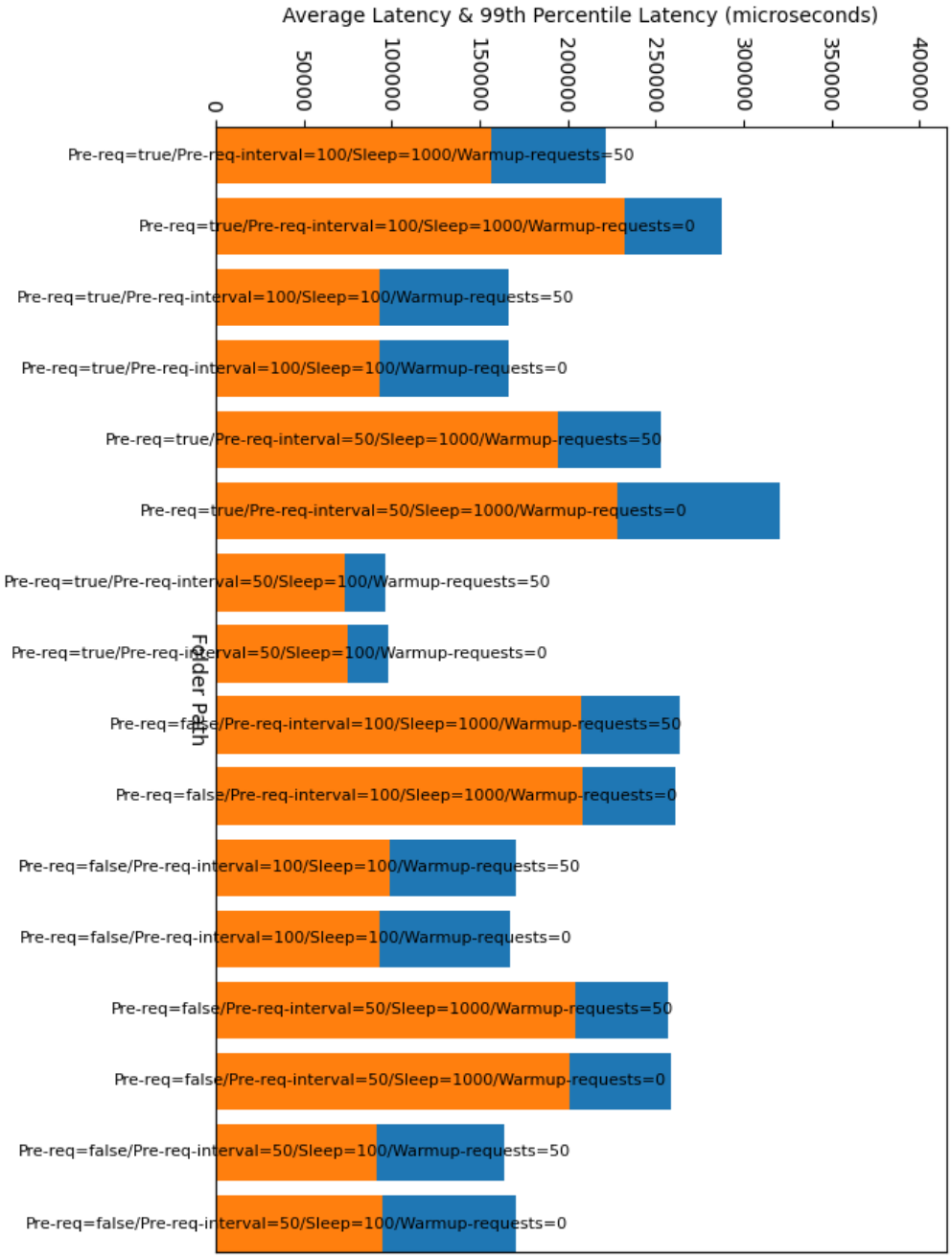


Figure 5.4: Custom Interrupt Micro Benchmark Results with Fixed Interval Time and C-States Enabled.

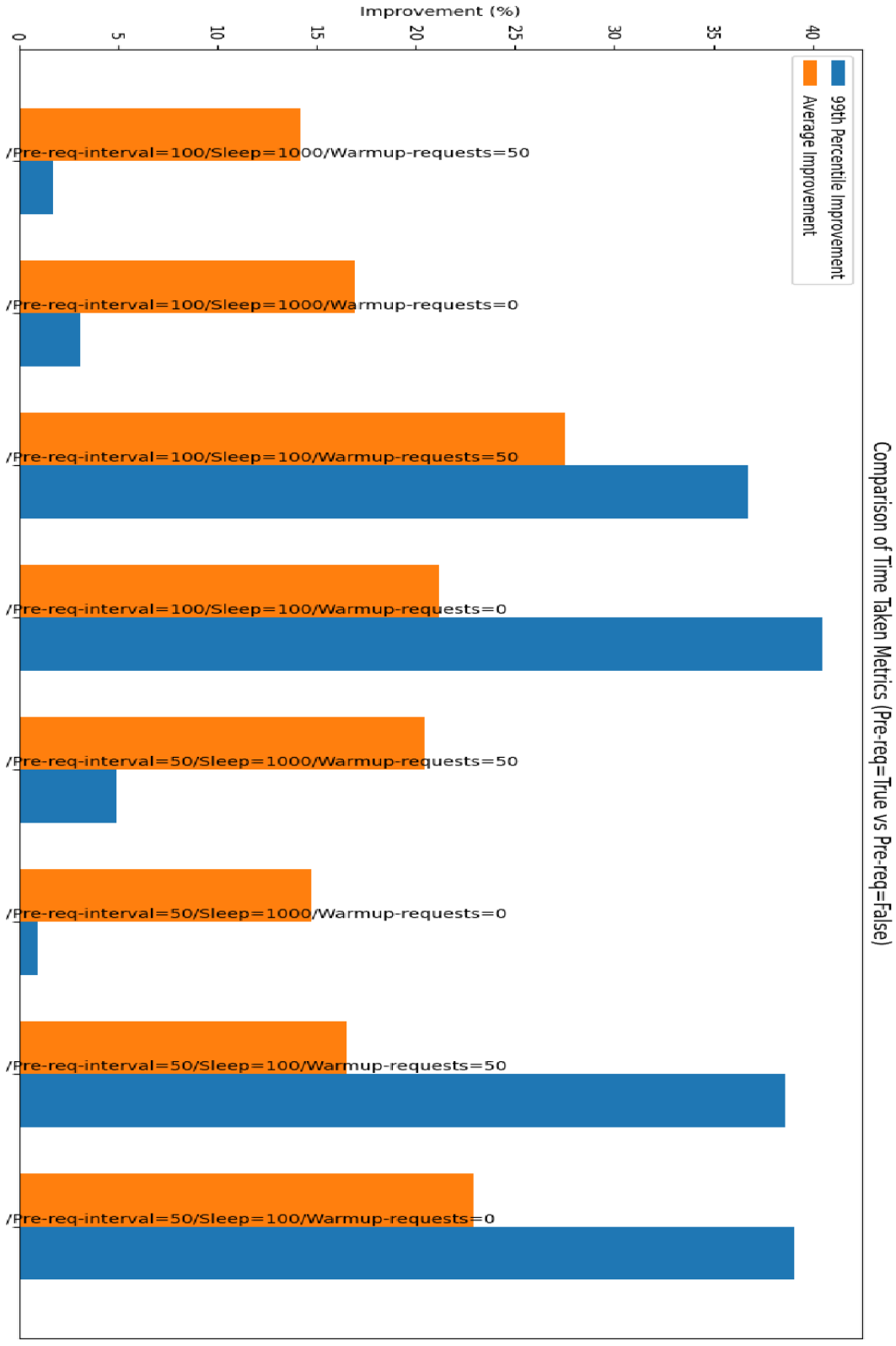


Figure 5.5: Custom Interrupt Micro Benchmark Results Comparison with Exponential Distribution and C-States Enabled.

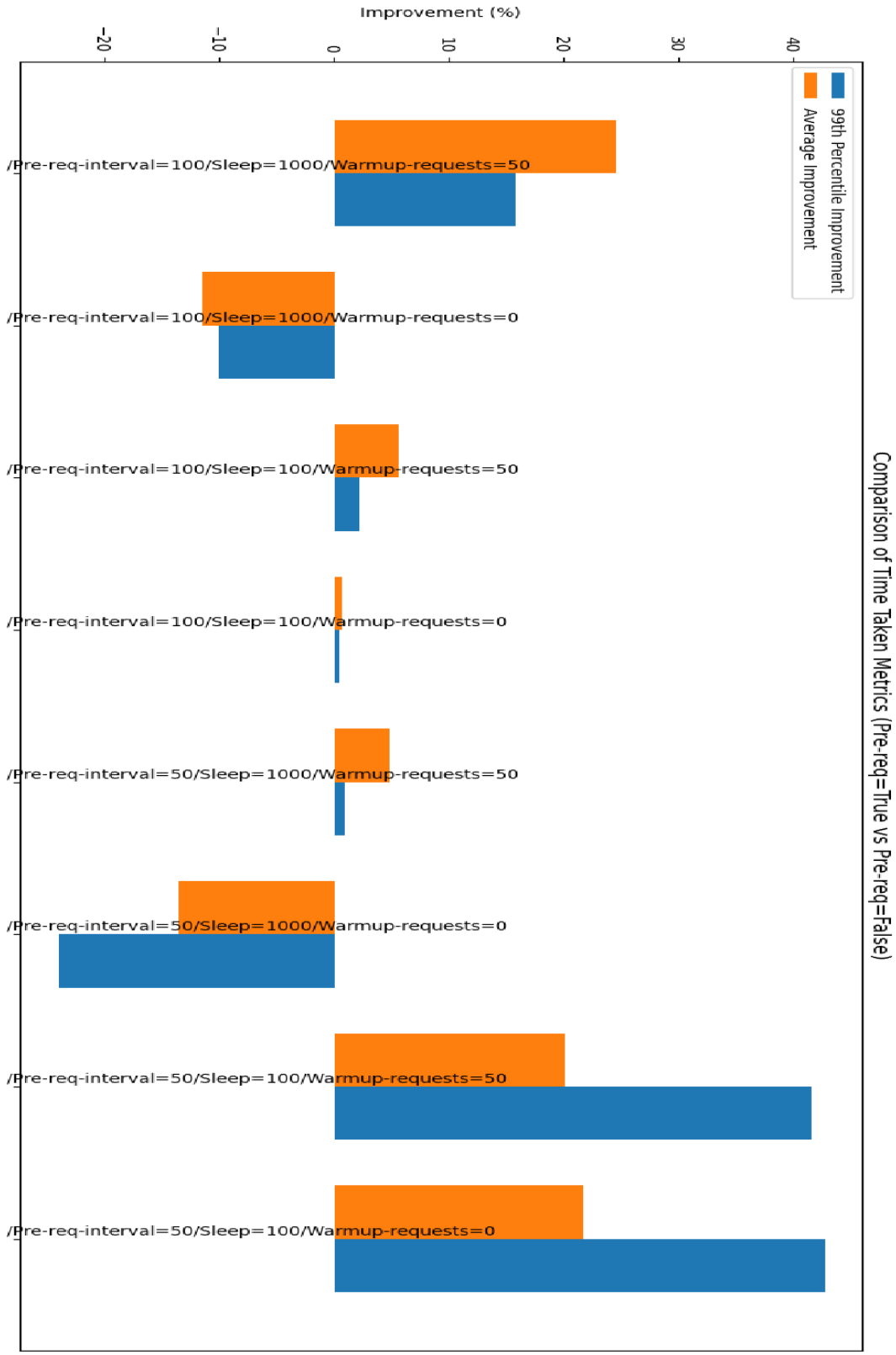


Figure 5.6: Custom Interrupt Micro Benchmark Results Comparison with Fixed Interval Time and C-States Enabled.

Comments / Observations:

- Sending a pre-request continues to reduce both the average and the 99th percentile tail latencies across most scenarios, with a more pronounced effect observed when employing fixed interval time.
- With fixed interval time, the average latency values are notably higher, and they approach the 99th percentile latency times more than exponential distribution latency values do.
- With fixed interval time, we observe a discernible distinction based on whether warmup requests are sent prior to the main experiment. In contrast, such disparity is not evident in the case of the exponential distribution, particularly concerning average latency measurements.

We utilized SocWatch to capture the C-States transitions during the experiment. Our observations revealed that after the query execution, the system entered C6, as anticipated. Subsequently, upon sending the pre-query request, the system transitioned into C1, until it receives the query. Tables 5.6 and 5.7 present a snippet of the recorded results.

Table 5.6: Micro Benchmark C-State Transitions With Query Interval 10ms and 50µs Pre-Query Request Interval

Sample #	Continuous Time (ms)	C-State	Duration (ms)
2889	7235.22	CC6	9.54
2890	7235.91	CC0	0.68
2891	7235.95	CC1	0.05
2892	7236.03	CC0	0.08
2893	7245.82	CC6	9.78
2894	7246.25	CC0	0.44
2895	7246.30	CC1	0.05
2896	7246.38	CC0	0.08
2897	7256.17	CC6	9.79

Table 5.7: Micro Benchmark C-State Transitions With Query Interval 1ms and 50µs Pre-Query Request Interval

Sample #	Continuous Time (ms)	C-State	Duration (ms)
2536	1728.75	CC0	0.68
2537	1728.77	CC1	0.02
2538	1728.83	CC0	0.07
2539	1729.16	CC6	0.33
2540	1729.76	CC0	0.60
2541	1729.94	CC1	0.17
2542	1730.04	CC0	0.11
2543	1730.07	CC1	0.02
2544	1730.13	CC0	0.07
2545	1730.96	CC6	0.83

5.2 HDSearch

We conducted the HDSearch experiment five times, each QPS set to 100 and 500, respectively. For each experiment, we measured the average latency and the 99th percentile tail latency. Subsequently, we calculated the average of the average latencies, the average of the 99th percentile tail latencies, and identified the highest 99th percentile tail latency. The summarized results are presented in Table 5.8. The summarized results with fixed interval time are presented in Table 5.9.

Table 5.8: Comparison of Latencies (ms) for Different Configurations, with Exponential Distribution. Each value set, separated by "-", represents the average latency, the average 99th percentile tail latency, and the highest 99th percentile tail latency, respectively.

Configuration	**QPS = 100 (ms)**	**QPS = 500 (ms)**
Without pre	1.88 - 2.40 - 2.411	1.54 - 2.72 - 2.741
With immediate pre	1.25 - 1.57 - 1.585	1.11 - 1.49 - 1.529
With pre 50us	1.25 - 1.56 - 1.586	1.09 - 1.46 - 1.500
With pre 100us	1.26 - 1.56 - 1.578	1.09 - 1.47 - 1.443
With pre 150us	1.26 - 1.52 - 1.541	1.09 - 1.43 - 1.438
With adaptive pre	1.25 - 1.52 - 1.538	1.09 - 1.45 - 1.457

Table 5.9: Comparison of Latencies (ms) for Different Configurations, with Fixed Interval Time. Each value set, separated by "-", represents the average latency, the average 99th percentile tail latency, and the highest 99th percentile tail latency, respectively.

Configuration	**QPS = 10 (ms)**	**QPS = 100 (ms)**
Without pre	2.18 - 2.76 - 2.781	1.96 - 2.31 - 2.34
With immediate pre	1.29 - 1.45 - 1.478	1.46 - 1.88 - 1.92
With pre 50us	1.30 - 1.47 - 1.483	1.50 - 1.90 - 1.926
With pre 100us	1.31 - 1.48 - 1.484	1.49 - 1.90 - 1.929
With pre 150us	1.28 - 1.44 - 1.471	1.46 - 1.87 - 1.909
With adaptive pre	1.28 - 1.47 - 1.478	1.46 - 1.87 - 1.909

Furthermore, we ran the same experiment again, this time without running the pre-query request server and the bucket on a specific core. The summarized results are presented in Table 5.10.

Table 5.10: Comparison of Latencies (ms) for Different Configurations Without Running on Specific Core and Using Exponential Distribution. Each value set, separated by "-", represents the average latency, the average 99th percentile tail latency, and the highest 99th percentile tail latency, respectively.

Configuration	**QPS = 100 (ms)**	**QPS = 500 (ms)**
Without pre	2.24 - 2.92 - 3.005	2.08 - 2.83 - 2.882
With immediate pre	2.29 - 2.92 - 3.022	2.02 - 2.74 - 2.811
With pre 50us	2.30 - 2.94 - 3.029	2.00 - 2.72 - 2.808
With pre 100us	2.30 - 2.93 - 3.025	2.02 - 2.73 - 2.810
With pre 150us	2.28 - 2.90 - 3.020	2.04 - 2.74 - 2.809
With adaptive pre	2.31 - 2.94 - 3.030	2.00 - 2.78 - 2.813

Additionally, we repeated the experiment with C-States disabled to investigate whether the observed advantage of pre-query requests is attributable to C-state wake-up effects. The summarized results are presented in Table 5.11.

Table 5.11: Comparison of Latencies (ms) for Different Configurations With C-States Disabled and Exponential Distribution. Each value set, separated by "-", represents the average latency, the average 99th percentile tail latency, and the highest 99th percentile tail latency, respectively.

Configuration	**QPS = 100 (ms)**	**QPS = 500 (ms)**
Without pre	1.26 - 1.40 - 1.519	1.07 - 1.40 - 1.450
With immediate pre	1.28 - 1.58 - 1.586	1.10 - 1.48 - 1.530
With pre 50us	1.30 - 1.55 - 1.587	1.08 - 1.46 - 1.501
With pre 100us	1.25 - 1.57 - 1.579	1.03 - 1.42 - 1.444
With pre 150us	1.27 - 1.47 - 1.568	1.06 - 1.42 - 1.462
With adaptive pre	1.28 - 1.42 - 1.525	1.09 - 1.47 - 1.509

Lastly, we also experimented with QPS set to 10 and 100 and we used the Cloudlab's node c220g2 to see whether we observe the same result. For each experiment, we measured the average latency and the 99th percentile tail latency. Subsequently, we calculated the average of the average latencies, the average of the 99th percentile tail latencies, and identified the highest 99th percentile tail latency. The summarized results are presented in Table 5.12. The summarized results with fixed interval time are presented in Table 5.13.

Table 5.12: Comparison of Latencies (ms) for Different Configurations, with Exponential Distribution on c220g2. Each value set, separated by "-", represents the average latency, the average 99th percentile tail latency, and the highest 99th percentile tail latency, respectively.

Configuration	QPS = 10 (ms)	QPS = 100 (ms)
Without pre	1.64 - 2.07 - 2.089	1.54 - 1.88 - 1.895
With immediate pre	1.28 - 1.66 - 1.690	1.17 - 1.39 - 1.39
With pre 50us	1.29 - 1.64 - 1.693	1.17 - 1.37 - 1.38
With pre 100us	1.27 - 1.59 - 1.642	1.17 - 1.35 - 1.363
With pre 150us	1.27 - 1.59 - 1.608	1.18 - 1.36 - 1.38
With adaptive pre	1.28 - 1.56 - 1.625	1.18 - 1.35 - 1.399

Table 5.13: Comparison of Latencies (ms) for Different Configurations, with Fixed Interval Time on c220g2. Each value set, separated by "-", represents the average latency, the average 99th percentile tail latency, and the highest 99th percentile tail latency, respectively.

Configuration	QPS = 10 (ms)	QPS = 100 (ms)
Without pre	1.71 - 2.05 - 2.073	1.53 - 1.76 - 1.771
With immediate pre	1.35 - 1.64 - 1.650	1.17 - 1.30 - 1.305
With pre 50us	1.34 - 1.63 - 1.633	1.16 - 1.31 - 1.315
With pre 100us	1.33 - 1.59 - 1.605	1.17 - 1.31 - 1.317
With pre 150us	1.31 - 1.60 - 1.617	1.15 - 1.29 - 1.294
With adaptive pre	1.31 - 1.66 - 1.621	1.15 - 1.28 - 1.320

Comments / Observations:

- Implementing both fixed interval time and Exponential Distribution methods reveals a notable decrease in average tail latency due to pre-query requests.
- When employing fixed time intervals, we observed greater latencies in comparison to those observed with exponential distribution.
- Without assigning dedicated cores for running the bucket and the pre-query request server, we did not observe any noticeable benefit from sending a pre-query request. This outcome was somewhat expected since the designated core does not wake up prior to receiving the query.
- Additionally, in scenarios where C-States are disabled, we found no observable difference in latency improvement when utilizing pre-query requests. This suggests that the effectiveness of pre-query requests rely on the CPU idleness.

5.2.1 Observations with SocWatch

We monitored IRQs and C-States traces using SocWatch under fixed interval times. Setting the interval to 10 ms, we anticipated results similar to those of the micro benchmark, as shown in Table 5.6. While we observed similarities, we also noted an intriguing pattern: occasionally, the system did not transition to C1 after the pre-query request, suggesting the possibility of no sleep occurring post pre-query. Focusing on the experiment with a pre-query request interval of 100µs, which produced optimal results, Figure 5.7 presents a segment of the C-States transitions trace for HDSearch, and Figure 5.8 presents the corresponding segment of the IRQs. For further explanation of the output, read section 4.1.1.

Figure 5.7: C-States Transitions Trace of HDSearch Experiment with Fixed Interval Time 10ms and Pre-Query Request Interval 100µs. The corresponding IRQs trace is shown in Figure 5.8.

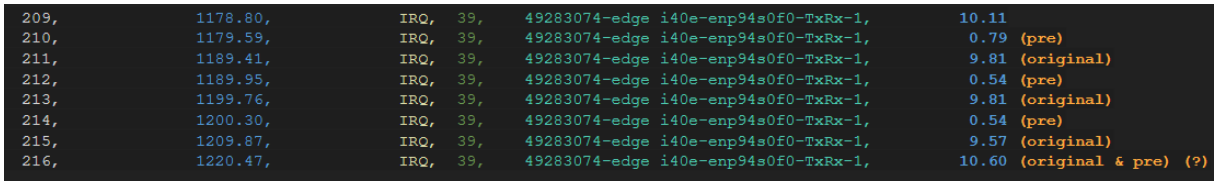


Figure 5.8: IRQs Trace of HDSearch Experiment with Fixed Interval Time 10ms and Pre-Query Request Interval 100µs.

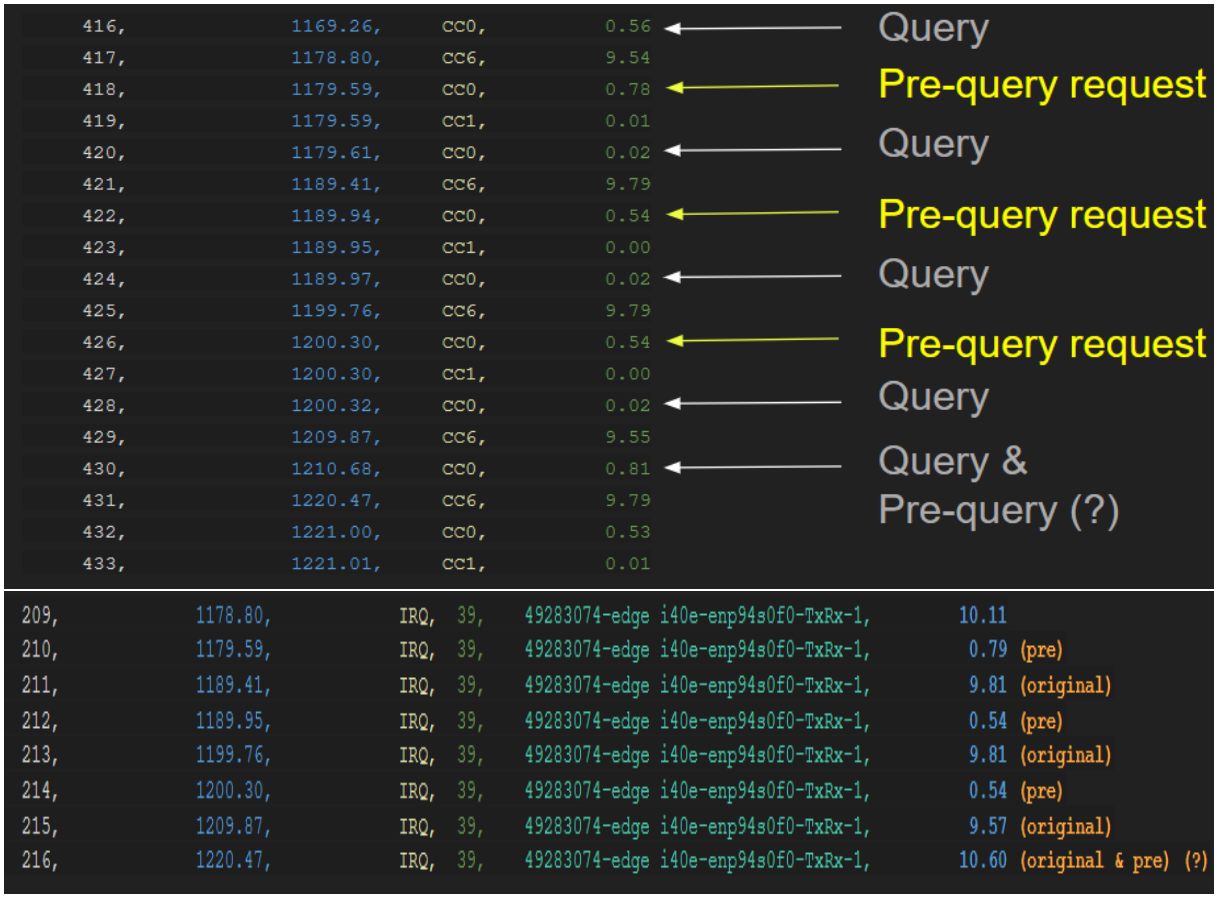


Figure 5.9: Combined Figures 5.7 and 5.8.

We also measured the residencies of the C-States, which are presented in Table 5.14 and Figure 5.10. The latency with the pre-query request is the average of sending an immediate pre-query request (0µs), 50µs, 100µs, and 150µs. As expected, without sending a pre-query request, the C0 residency time is less, while C1 and C6 residency times are greater.

Table 5.14: C-States Residencies in HDSearch in Running Core (1)

Configuration	C0 Residency (%)	C1 Residency (%)	C6 Residency (%)
Without Pre-Query	9.71	0.16	90.14
With Pre-Query	10.38	0.22	89.40

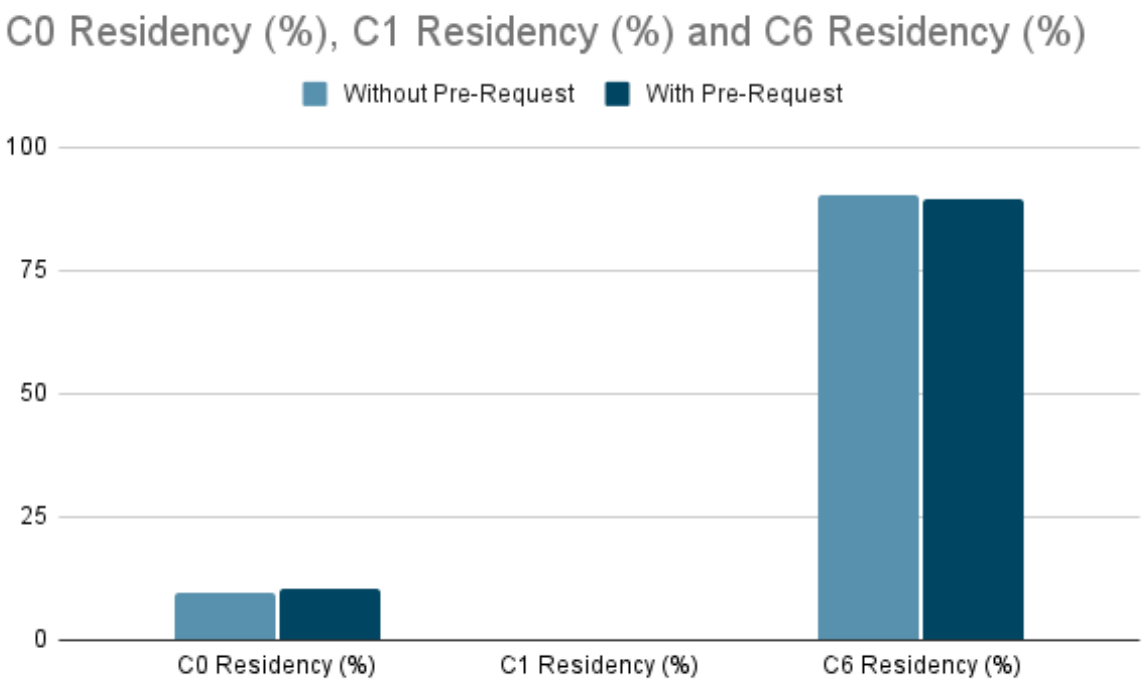


Figure 5.10: C-States Residencies in HDSearch in Running Core (1)

5.3 Router MicroSuite Benchmark

We applied the same approach for pre-query requests as described in the HDSearch experiment in the Router MicroSuite Benchmark. The results are shown in Table 5.15.

Table 5.15: Comparison of Latencies (ms) for Different Configurations in Router, with Exponential Distribution. Each value set, separated by "-", represents the average latency, the average 99th percentile tail latency, and the highest 99th percentile tail latency, respectively.

Configuration	QPS = 100 (ms)	QPS = 500 (ms)
Without pre	1.68 - 1.98 - 2.02	1.68 - 1.95 - 2.02
With immediate pre	1.49 - 1.80 - 1.91	1.49 - 1.79 - 1.92
With pre 50us	1.52 - 1.83 - 1.96	1.52 - 1.79 - 1.96
With pre 100us	1.53 - 1.84 - 1.93	1.53 - 1.80 - 1.93
With pre 150us	1.55 - 1.84 - 1.88	1.55 - 1.79 - 1.88
With adaptive pre	1.50 - 1.86 - 1.87	1.50 - 1.80 - 1.91

Comments:

- Similarly to the findings in HDSearch, we observed that sending a pre-query request leads to a reduction in tail latency in the Router Benchmark.

5.4 gRPC Helloworld Experiment

During the experiment (mentioned in section 4.8), we initiated a series of 10,000 requests from the client to the server, each with a fixed time interval of 10 milliseconds between them (Table 5.16). Additionally, we conducted a variation of the experiment by adjusting the interval time between requests. to 2 milliseconds (equivalent to 500 queries per second, Table 5.17), aiming to analyze the system's performance under different query rates.

Additionally, we incorporated intervals generated using an exponential distribution, with average numbers of 2 (Table 5.19) and 10 (Table 5.18), mirroring the fixed intervals. This approach allowed us to simulate more realistic request patterns and comprehensively assess the system's responsiveness and scalability across a range of load conditions.

Moreover, to investigate the potential advantages of pre-query requests, we experimented with sending a pre-query request to the server at various time intervals before the original query. These intervals included immediately (0 microseconds), as well as before 50 microseconds, 100 microseconds, and 200 microseconds of the original query. By examining the effects of pre-query requests at different time offsets, we aimed to identify any potential performance enhancements or efficiency gains that could be achieved through this approach.

We monitored IRQs and C-States traces using SocWatch under fixed interval times, exactly as we did in section 5.2.1. Setting the interval to 10 ms, we anticipated results similar to those of the micro benchmark, as shown in Table 5.6. While we observed similarities, we also noted an intriguing pattern: occasionally, the system did not transition to C1 after the pre-query request,

suggesting the possibility of no sleep occurring post pre-query. Focusing on the experiment with a pre-query request interval of 100µs, which produced optimal results, Figure 5.7 presents a segment of the C-States transitions trace for HDSearch, and Figure 5.8 presents the corresponding segment of the IRQs. For further explanation of the output, read section 4.1.1.

Table 5.16: Latency Measurements with Different Pre-Query Interval Times in gRPC Helloworld Example with Fixed Interval Time 10ms

Interval Time (µs)	Average Latency (µs)	99th Percentile Latency (µs)
Without pre	3338.13	3676
Immediate pre (0)	3354.43	3686
50	3339.07	3582
100	3329.62	3560
200	3314.85	3517
Adaptive pre	3325.10	3525

Table 5.17: Latency Measurements with Different Pre-Query Interval Times in gRPC Helloworld Example with Fixed Interval Time 2ms

Interval Time (µs)	Average Latency (µs)	99th Percentile Latency (µs)
Without pre	3308.46	3566
Immediate pre (0)	3315.70	3505
50	3252.08	3536
100	3324.66	3607
200	3284.50	3480
Adaptive pre	3251.43	3548

Table 5.18: Latency Measurements with Different Pre-Query Interval Times in gRPC Helloworld Example with Exponential Distribution with Average Interval Time 10ms

Interval Time (µs)	Average Latency (µs)	99th Percentile Latency (µs)
Without pre	3264.58	3517
Immediate pre (0)	3288.05	3481
50	3238.52	3455
100	3267.42	3460
200	3251.07	3455
Adaptive pre	3264.40	3452

5.5 TCP Delayed Acknowledgment

During our experiments, when the interval time was 100ms or greater, we observed using Wireshark that another message was sent after 40ms. This message was identified as the TCP

Table 5.19: Latency Measurements with Different Pre-Query Interval Times in gRPC Helloworld Example with Exponential Distribution with Average Interval Time 2ms

Interval Time (µs)	Average Latency (µs)	99th Percentile Latency (µs)
Without pre	3338.13	3676
Immediate pre (0)	3354.43	3686
50	3339.07	3582
100	3329.62	3560
200	3314.85	3517
Adaptive pre	3353.32	3564

Delayed Acknowledgment [20, 21], which, in our case, is an acknowledgment sent after 40ms, but this timing varied. The significance of this observation lies in its implications for processor wake-up cycles. When the TCP Delayed Acknowledgment is triggered, it prompts the processor to wake up after only 40ms, following which it returns to a deep sleep state (C6) once the acknowledgment is processed.

5.5.1 Explanation of TCP Delayed Acknowledgment

TCP (Transmission Control Protocol) uses acknowledgments (ACKs) to confirm the receipt of data packets. Typically, an ACK is sent immediately after receiving a data packet. However, TCP Delayed Acknowledgment is a mechanism that delays the sending of ACKs in certain situations.

The purpose of TCP Delayed Acknowledgment is to improve network efficiency by reducing the number of ACK packets sent over the network. Instead of acknowledging each received packet immediately, TCP Delayed Acknowledgment allows the receiver to wait for a short period (in our case 40ms) to see if it can piggyback the ACK on the next outgoing data packet.

By delaying ACKs and potentially piggybacking them on outgoing data packets, TCP can reduce the overall number of packets transmitted on the network, which can help alleviate network congestion and improve performance, especially in high-latency environments.

6 Related Work

6.1 Menu Governor Enhancement Approaches

Several approaches have been proposed to enhance the `menu` governor.Dial and Song [22] and Sharafzadeh et al. [23] introduced methods to address its limitations. They utilized machine learning to estimate idle periods and select appropriate idle states. While these methods improve idle state selection, they still struggle to reduce wake-up latency when transitioning from deep states.

There were also many other attempts to create new idle C-States, such as AgileWatts [24], in order to find the best trade-off for performance and energy consumption.

6.2 Previous Work on Pre-Wakeup

A recent research from Kei Fujimoto et al. [25] has addressed the challenge of maintaining high real-time performance in services while facing increased server power consumption due to the disabling of power-saving features. To mitigate the deep C-State wakeup latency issue, a pre-wakeup (PWU) system has been proposed. This system pre-wakes CPU cores before task assignment, reducing wake-up latency while allowing for transition to deeper idle states when appropriate. Evaluations conducted on an Intel Xeon processor demonstrated that the PWU system incurs minimal power-consumption overhead and can reduce recovery time from the C6 state by 84%.

7 Conclusions

7.1 Conclusion

This thesis has shed light on a potential avenue for improving the performance of microservices implementations, and potentially other systems utilizing gRPCs or remote API calls. Through the introduction of a pre-query request mechanism aimed at awakening the CPU before the arrival of the primary query, a promising benefit has been demonstrated. By sending a fast and lightweight request to the server, this mechanism effectively reduces tail latency, leading to enhanced system responsiveness.

The findings presented in this thesis underscore the significance of proactive approaches to CPU wake-up management in microservices environments. By strategically timing the pre-query request to ensure the CPU is awakened sufficiently ahead of the primary query, significant improvements in tail latency have been observed. This not only enhances the end-user experience but also contributes to the overall efficiency and effectiveness of microservices-based systems.

Moving forward, further research and experimentation could explore the broader applicability of the pre-query request mechanism across various system architectures and use cases. Additionally, investigating optimizations and refinements to the mechanism could uncover additional performance benefits, further enhancing the value proposition for its adoption in real-world deployments.

In conclusion, this thesis offers valuable insights into the potential of proactive CPU wake-up management strategies to optimize system performance in microservices environments. By reducing tail latency through the implementation of the pre-query request mechanism, a tangible improvement in system responsiveness has been demonstrated, paving the way for more efficient and effective microservices implementations in the future.

7.2 Future Work

7.2.1 Further Research and Investigation

While the current research provides valuable insights into the impact of CPU power management on microservices performance, there are avenues for further exploration and analysis.

Firstly, conducting experiments in a different environment, distinct from CloudLab, would offer valuable insights into the generalizability of the findings. Testing the results in a real-world

deployment or a different cloud computing platform could provide a broader perspective and validate the reproducibility of the observed effects.

Additionally, further investigation is warranted to understand the underlying reasons behind the suboptimal performance observed in the gRPC helloworld experiment.

7.2.2 Linux Kernel and Idle Governor Optimizations

Exploring potential enhancements to the idle governor could yield significant reductions in microservice response latency. By introducing smarter mechanisms within the governor, such as predictive wake-up strategies or intelligent handling of interrupt coalescing, the kernel could proactively awaken CPU cores in anticipation of impending requests, thus minimizing response latency.

One avenue for improvement lies in devising algorithms within the idle governor that anticipate upcoming workloads, such as gRPC requests, and preemptively adjust CPU states accordingly. By intelligently predicting workload patterns and preemptively transitioning CPU cores from idle states to active states, the kernel could effectively reduce the latency incurred during request processing.

Furthermore, optimizing interrupt handling through techniques like interrupt coalescing could also contribute to latency reduction. By intelligently aggregating and processing interrupts, the kernel can minimize the overhead associated with interrupt handling, thereby streamlining the response process for microservices.

7.2.2.1 Idle Governors with Machine Learning

Idle governors utilizing machine learning techniques hold promise for further optimizing CPU power management and reducing response latency in microservices environments. Recent research efforts [23, 26] have explored the application of machine learning algorithms within idle governors, demonstrating potential benefits in terms of performance and efficiency.

By leveraging machine learning models, idle governors can adaptively learn and adjust CPU power states based on dynamic workload patterns and system characteristics. These models can analyze historical workload data, system metrics, and environmental factors to make proactive decisions regarding CPU power management.

For example, reinforcement learning algorithms can enable idle governors to learn optimal policies for transitioning CPU cores between different power states in response to varying workload demands. Similarly, recurrent neural network (RNN) architectures can capture temporal dependencies in workload patterns, allowing for more accurate predictions and proactive adjustments.

The integration of machine learning into idle governors offers several potential benefits, including improved responsiveness, enhanced energy efficiency, and better adaptability to workload fluctuations. By dynamically optimizing CPU power states based on real-time data and predictive models, machine learning-powered idle governors have the potential to significantly enhance the performance and efficiency of microservices environments.

7.2.3 Exploring Different Linux Distributions

In our experimentation, we only utilized Ubuntu LTS 20.04 as the operating system for our research environment. However, with the release of newer versions such as Ubuntu 24.04, there is an opportunity to explore the potential impacts of updated distributions on microservices performance.

The adoption of a newer Linux distribution may introduce changes to various components of the operating system, including the idle governor. As such, experimenting with Ubuntu 24.04 or other recent releases could provide insights into how these changes affect CPU power management and response latency in microservices environments.

Furthermore, exploring alternative Linux distributions beyond Ubuntu, such as Fedora or CentOS, offers additional avenues for investigation. Each distribution may employ different configurations, package versions, and default settings, which can influence system behavior and performance characteristics.

BIBLIOGRAPHY

[1] [Online]. Available: https://medium.com/design-microservices-architecture-with-patterns/when-to-use-and-when-not-to-use-microservices-no-silver-bullet-3ae293faf6d

[2] [Online]. Available: https://grpc.io/docs/languages/cpp/quickstart/

[3] [Online]. Available: https://github.com/grpc/grpc

[4] [Online]. Available: https://docs.kernel.org/driver-api/pm/cpuidle.html#:~:text=A%20CPU%20idle%20time%20(%20CPUIdle,order%20to%20save%20some%20energy.

[5] [Online]. Available: https://docs.kernel.org/admin-guide/pm/cpufreq.html

[6] A. Sriraman and T. F. Wenisch, "μ suite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 1–12.

[7] [Online]. Available: https://github.com/cseas002/MicroSuite

[8] [Online]. Available: https://github.com/cseas002/HDSearch-Multinode

[9] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[10] [Online]. Available: https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence

[11] [Online]. Available: https://en.wikipedia.org/wiki/Hill_climbing

[12] [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-irq

[13] [Online]. Available: https://www.scribbr.com/statistics/poisson-distribution/#:~:text=A%20Poisson%20distribution%20is%20a,the%20mean%20number%20of%20events.

[14] [Online]. Available: https://courses.lumenlearning.com/introstats1/chapter/the-exponential-distribution

[15] [Online]. Available: https://en.wikipedia.org/wiki/Exponential_distribution#:~:text=In%20probability%20theory%20and%20statistics,rate%3B%20the%20distance%20parameter%20could

[16] J. Cao, M. Andersson, C. Nyberg, and M. Kihl, "Web server performance modeling using an m/g/1/k*ps queue," in *10th International Conference on Telecommunications, 2003. ICT 2003.*, vol. 2, 2003, pp. 1501–1506 vol.2.

[17] [Online]. Available: https://www.probabilitycourse.com/chapter11/11_1_2_basic_concepts_of_the_poisson_process.php

[18] [Online]. Available: https://www.wisc.cloudlab.us/portal/show-nodetype.php?type=c220g5

[19] [Online]. Available: https://github.com/cseas002/interrupts

[20] [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/reducing_the_tcp_delayed_ack_timeout

[21] [Online]. Available: https://en.wikipedia.org/wiki/TCP_delayed_acknowledgment

[22] Q. Diao and J. Song, "Prediction of cpu idle-busy activity pattern," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008, pp. 27–36.

[23] E. Sharafzadeh, S. A. S. Kohroudi, E. Asyabi, and M. Sharifi, "Yawn: A cpu idle-state governor for datacenter applications," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2019.

[24] J. H. Yahya, H. Volos, D. B. Bartolini, G. Antoniou, J. S. Kim, Z. Wang, K. Kalaitzidis, T. Rollet, Z. Chen, Y. Geng, O. Mutlu, and Y. Sazeides, "Agilewatts: An energy-efficient cpu core idle-state architecture for latency-sensitive server applications," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 835–850.

[25] K. Fujimoto, H. Harasawa, K. Natori, I. Otani, S. Saito, and A. Shiraga, "Pwu: Pre- wakeup for cpu idle to reduce latency and power consumption," in *2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2022, pp. 1–6.

[26] S. Pattanayak and B. Thangaraju, "Linux cpu-idle menu governor with online reinforcement learning and scheduler load balancing statistics," in *2019 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2019, pp. 1–6.