Affinity Aware CPU Scheduling for Container Hosts

A.M.S.U. Karunarathne

Affinity Aware CPU Scheduling for Container Hosts

A.M.S.U. Karunarathne

Index No: 20000928

Supervisor: Dr. C.I. Keppitiyagama

April 2025

Submitted in partial fulfillment of the requirements of the

B.Sc. (Honours) in Computer Science Final Year Project

Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, to be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name:	A.M.S.U. Karunarathne
Signature of Candidate:	[illegible]
Date:	May 30, 2025

This is to certify that this dissertation is based on the work of Ms. A.M.S.U. Karunarathne under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Principal/Co-Supervisor's Name:	Dr. C.I. Keppitiyagama
Signature of Supervisor:	[illegible]
Date:	May 30, 2025

Acknowledgement

I would like to extend my heartfelt gratitude to my supervisor Dr. C.I. Keppitiyagama, for all of his help and advice during the course of this research. Their knowledge, perceptive criticism, and support have been extremely helpful in forming the framework and content of this research.

I am also deeply thankful to my co-supervisor, Mr. T N B Wijethilake, for their insightful comments and priceless efforts. Their knowledge and helpful critiques have been invaluable in improving the quality of this research.

Furthermore, I want to express my gratitude to my family for their unwavering support, tolerance, and understanding. Their encouragement has been a rock of strength, inspiring me to keep going despite the difficulties this academic effort has presented.

Abstract

Containerization facilitates efficient application deployment by isolating workloads within lightweight environments sharing the host operating system's kernel. However, the Linux Completely Fair Scheduler (CFS) manages containerized processes as standard user-space tasks, resulting in frequent CPU migrations, cache invalidations, and unpredictable latency in high-density, latency-sensitive deployments. This thesis proposes an affinity-aware CPU scheduling framework for container hosts, integrating queuing theory with eBPF-based monitoring to enhance performance. Kernel-level TCP backlog and application-level request queues are modeled as M/M/c/K systems, with eBPF probes capturing per-container scheduling and connection metrics to analyze queue dynamics and CPU migration patterns. Experiments on a four-core Ubuntu virtual machine, using a custom C-based HTTP server in Docker, reveal that overprovisioning worker threads increases CPU migrations by up to 105% (from 32.15 migrations with 5 workers to 58.65 with 17.5 workers) at a fixed request rate of 10 requests per second, leading to cache misses and reduced throughput. A batched queuing scheme mitigates lock contention, while adaptive worker thread management, responsive to real-time arrival rates, significantly reduces migrations and enhances CPU utilization. However, M/M/c/K queuing models and machine learning approaches, such as Random Forest models, exhibit limitations due to dynamic scheduling patterns and TCP congestion control interference, which introduce variability and reduce predictive accuracy. Key contributions include a low-overhead eBPF monitoring system for per-container queue metrics and a hybrid analytical-empirical approach combining queuing theory with kernel telemetry to optimize server performance. Results underscore the critical role of affinity-aware scheduling and dynamic thread tuning in achieving predictable and efficient performance in containerized environments, while highlighting the need for hybrid models to address the shortcomings of theoretical and machine learning-based predictions.

Table of Contents

Chapter 1

Introduction

Containerization has transformed modern computing by making it possible to develop, run, and manage programs in confined environments. As opposed to traditional virtualization, which exploits hypervisors for emulating entire operating systems, containers leverage the kernel of the host operating system for running multiple isolated instances of user-space with less overhead, faster startup, and better utilization of resources (Dua et al. 2014, Bentaleb et al. 2022). Containers give a contained environment through mechanisms like namespaces, which provide security functionalities like user ID and file system isolation, allowing for isolation of containers from the host or from other containers (Eder 2016). Containerization is not without challenges, such as resource competition, where the overconsumption of resources by one container will impact others unless managed by control groups (cgroups) (Bentaleb et al. 2022).

At the base of operating system performance lies the CPU scheduler, which controls processor time for running processes. The Completely Fair Scheduler (CFS), part of kernel 2.6.23, is the native scheduler in Linux and is utilized to distribute CPU time among processes in a fair manner based on their virtual runtime (vruntime) and priority, determined by the "nice" value (Pabla 2009, Kobus & Szklarski 2009). Earlier schedulers like O(1) scheduler made use of intricate heuristics that preferred to mislabel jobs, hence leading to inefficiencies (Pabla 2009). CFS employs Red-Black trees to minimize idle CPU time but proves to be deficient when used in containerized systems where inter-container dependencies are prevalent (Marinakis et al. 2017).

1.1 Motivation

In modern containerized environments, multiple processes within different containers share a common host kernel and hardware resources. From the Completely Fair Scheduler (CFS) used by the Linux kernel perspective, such container processes are regular user-space processes. However, the CFS attempts to optimize CPU utilization by redistributing tasks across available CPU cores. While this measure attempts to distribute load, it generally creates unexpected consequences in containerized environments.

One such critical issue is the frequent migration of tasks between CPUs, destroying the cache locality. For every migration, there are cache misses, increasing the memory access cost and eventually leading to higher request processing latency. Such inefficiencies are particularly undesirable for latency-rich applications as well as high-throughput container workloads. Therefore, remedying CPU scheduling inefficiencies in container environments is crucial to improve performance predictability, reduce latency, and improve the overall system efficiency.

1.2 Problem Definition / Research Questions

1.2.1 Problem Definition

Common server designs that are commonly used within containers—e.g., accept-based designs—depend heavily on the operating system's scheduler to manage the order of task execution. But these scheduling decisions are normally made without regard to CPU affinity limits or application priorities. This disregard can lead to excessive CPU migrations, increased scheduling latency, and decreased overall performance, especially under heavy load when numerous containers compete for shared resources.

To better handle and address such challenges, this study proposes a queue-theory-based, affinity-aware CPU scheduling framework for containerized contexts. Such a model accounts for both kernel-level TCP backlog queues and application-level waiting queues and unifies them under a uniform framework of scheduling analysis.

More precisely, This study model:

- Application queues, analogous to requests stacked up for servicing in user-space (e.g., pending accept calls), and
- TCP backlog queues, analogous to connections stacked up in the kernel before acceptance by the application

using M/M/c/K queuing models, which are capable of providing analytical characterization of queue lengths, waiting times, and system saturation levels.

Through characterization of such queue structures via modeling, I expect to gain further insight into server workload runtime characteristics in container environments. Instead of being an integral part of the CPU scheduling logic, our approach utilizes queuing theory as a tool for diagnosis and optimization so as to enhance the performance of server-side components like connection handling and thread allocation in a way that facilitates better CPU scheduling decisions. By analyzing the dynamics of application queues and TCP backlog queues, we can see trends in terms of saturation points, request latency, or inter-container communication contention. We can then propose at the server level such that we avoid CPU contention, task migrations are minimized, and affinity-friendly execution is made possible, thus enhancing overall system performance without having to make changes at the kernel-level scheduler.

1.2.2 Research Questions

This paper is motivated by the performance degradation seen with containerized deployments, particularly under heavy system load. The aim is to determine how queuing and CPU affinity can be utilized to inform improved task scheduling. To that end, the paper attempts to answer the following key questions:

- How do application-level service queues and CPU affinity influence CPU scheduling in containerized environments? This question explores the effect of affinity constraints and application-level queues (e.g., pending requests) on task execution and scheduling delays.
- What is the effect of TCP backlog queues and application-level service queues on determining scheduling delays? This investigates the influence of network-level and user-space queue buildup on responsiveness and scheduling latency under a variety of workload conditions.
- Can queuing models such as M/M/c/K be used to accurately predict and model system behavior in containerized workloads? This test examines the relevance of classical queuing theory to characterize the complex interplay between arriving requests, processing threads, and CPU availability.
- How are queuing theory principles used to fine-tune server-side parameters—e.g., thread pool capacity, affinity assignment, or workload distribution—to assist with better CPU scheduling outcomes? This tries to list specific tuning choices at the user level that indirectly reduce CPU migrations and improve locality.

1.3 Approach

To arrive at an affinity-aware, queue-theory–guided scheduler for container hosts, I proceed in few steps:

- Developed custom instrumentation tools-including an eBPF-based kernel monitor and user-space counters-to gather queue and scheduling statistics with low overhead.

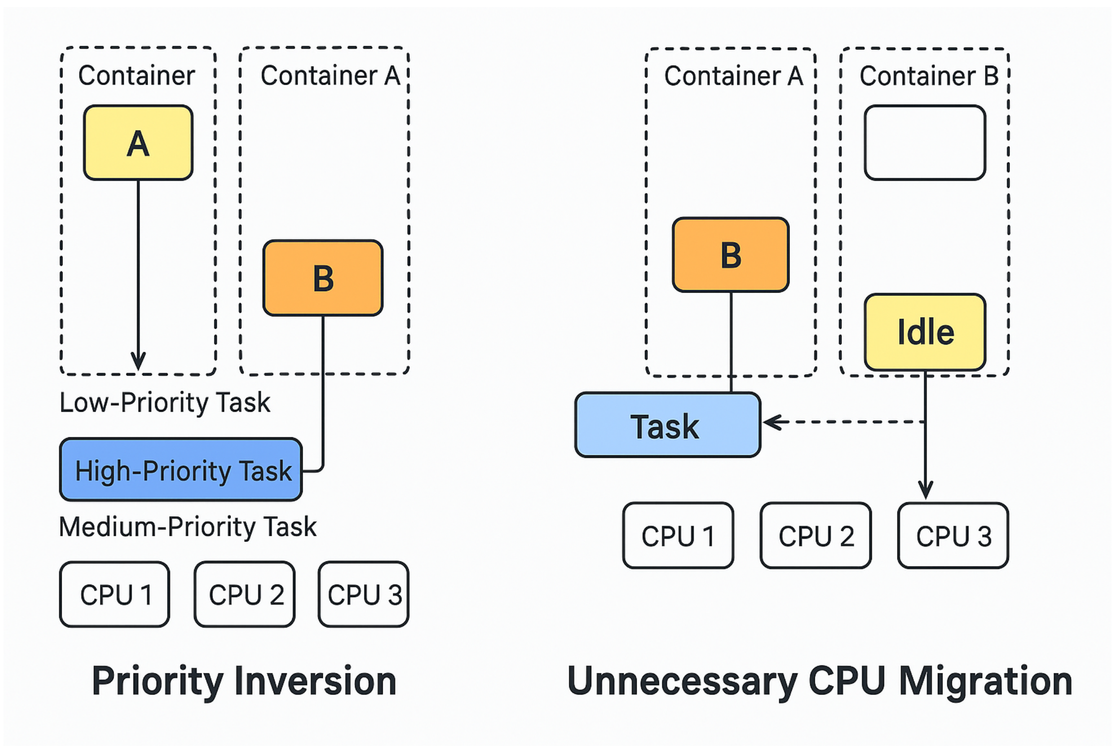


Figure 1.1: Priority Inversion and frequent CPU migrations

- Conducted controlled experiments with different *kernel backlog size* and *number of concurrent requests*, following CPU migrations and wakeup→switch latencies using Ftrace/trace-cmd/KernelShark.

- Utilized M/M/c/K queuing models to estimate queue waiting time

$$W_q = \frac{L_q}{\lambda}$$

  using empirically measured arrival (λ) and service (μ) rates. Machine learning models trained and tested (degree-2 polynomial regression and Random Forest) to predict CPU migration behavior from backlog size and monitoring interval.

- Leaned on eBPF telemetry to capture per-container, real-time connection arrival rates to deliver low-latency insights into queue dynamics.

- Documented (but did not implement) a tuning algorithm based on feedback to adjust server parameters (e.g., number of worker threads) periodically as a function of measured W_q metrics.

1.4 Thesis Structure

The rest of the thesis is presented as follows: Motivation and preliminary literature overview are followed by a clear definition of the problem and research issues. The approach to the research, experiments and preliminary results, is then presented. Then comes the implementation and design of proposed scheduling model are described in great detail. Finally, the discussion and the references conclude this thesis.

Chapter 2

Background/Preliminary Literature Review

2.1 Containerization

Containerization is a type of light-weight virtualization where applications run inside containers, which are isolated environments. Containerization is unique in that the containers share the host computer's kernel, as opposed to virtual machines where the entire operating system is duplicated every time an application is run (Dua et al. 2014). This means that the computer uses less resource and the applications load faster. This aspect of containerization has made it suitable for many applications in cloud computing and microservices in the current era of scalability and easy deployment (Bentaleb et al. 2022). For example, Docker is an orchestration platform for containerization, while Kubernetes is a platform for execution and management of containerized services on a distributed system.

Linux namespaces and control groups, also known as cgroups, provide the basic framework for containerization. Namespaces give container isolation, including process IDs, network stacks, and file systems in a way that containers run independently (Eder 2016). Cgroups are used to manage resources and prevent a container from taking over the host by restricting CPU, memory, and I/O (Bentaleb et al. 2022). The CPU subsystem of cgroups uses parameter specifications like `cpu.shares` for CPU time allocation in the cgroup and `cpu.cfs_period_us` and `cpu.cfs_quota_us` for setting hard limits on CPU utilization per cgroup.

Despite the abstraction and isolation that containers bring, competition for the underlying hardware resources such as CPU and memory remains a pervasive problem. When several containers are executed on the same host operating system, they inevitably contend for the same finite set of computational resources. If not adequately dealt with, this contention will result in perceivable performance degradation for applications run in such containers Bentaleb et al. (2022). This performance variability is especially troubling in environments where responsiveness and throughput consistency are required. Consequently, this circumstance emphasizes the critical necessity for smart and efficient CPU scheduling mechanisms. Not only should these mechanisms ensure equitable and equal distribution of CPU time among competing containers, but they should also attempt to maintain system performance at an optimal level that consider dynamic and high-load circumstances that characterize modern containerized environments.

2.2 CPU Scheduling in Linux

The Linux kernel has evolved its CPU scheduling algorithms to address a wide range of computing needs. Earlier schedulers, such as the O(1) scheduler, provided constant-time decision-making while maintaining fairness and interactivity under high system loads Pabla (2009).

The Completely Fair Scheduler (CFS), introduced in version 2.6.23, uses a red-black tree to manage tasks based on their virtual runtime, ensuring that each task receives an appropriate share of CPU time Kobus & Szklarski (2009). CFS assigns a `vruntime` to every task, which increases as the task executes, scaled according to its priority (expressed through the `nice` value). The scheduler then selects the task with the smallest `vruntime`, thereby promoting fairness in execution Pabla (2009).

Despite its advantages—such as low complexity, scalability, and predictability—CFS is not well-suited for containerized environments. In such environments, containers share the host kernel, and their processes are treated as regular user-space tasks. As a result, the scheduler lacks awareness of higher-level dependencies between containerized processes Marinakis et al. (2017). For instance, in a producer-consumer relationship spanning two different containers, CFS may schedule tasks in a way that causes delays in inter-container communication. These inefficiencies have led to a growing interest in the development of container-aware scheduling mechanisms Guo & Yao (2018).

2.3 Affinity Consideration in Scheduling

CPU affinity in modern multi-core systems refers to the attachment of a process (or thread) to a specific CPU core or set of cores. By scheduling the same CPU core a process over and over again, the process may take advantage of cache locality to reduce memory access latency and system-wide performance in general. This is particularly valuable in systems that have deep hierarchical cache hierarchies, where cache warmth may have a significant effect on responsiveness of applications.

In the Linux kernel, the Completely Fair Scheduler (CFS) attempts to evenly distribute the load across available CPU cores by migrating tasks as necessary. While this global load balancing improves fairness and CPU utilization, it has the tendency to overlook the benefits of maintaining CPU affinity. Excessive migration can lead to cache invalidation, TLB (Translation Look aside Buffer) misses, and increased memory access costs, all of which help degrade system performance—especially for latency-critical or high-throughput applications.

Affinity-aware scheduling is even more critical in containerized environments, in which different containers can concurrently host collaborative workloads having intricate inter-process dependence. For example, if two containers have a consumer-producer relationship, aggressive task migration can disrupt the timing and sequence of their communication. In such cases, preserving CPU affinity not only maximizes cache efficiency but also maximizes synchronization effectiveness and reduces inter-container communication latency.

In addition, affinity is very important in systems with NUMA (Non-Uniform

Memory Access) behavior. Scheduling jobs between CPUs of the same memory node reduces remote memory access cost. Ignoring affinity in such systems results in severe memory access bottlenecks.

To mitigate these problems, modern schedulers are now starting to incorporate affinity-aware mechanisms that attempt to balance the locality and load-balancing trade-offs. These mechanisms attempt to make more enlightened decisions regarding when and where to migrate tasks. Punishing migrations, per-core run queues with affinity hints, and feedback policies that observe cache performance to guide scheduling decisions are some of the methods.

2.4 Queue-Theory-Based Scheduling

Queuing theory models systems with queues, such as jobs waiting for CPU time, to predict performance measures like waiting times and throughput. M/M/c/K models with Poisson arrivals and exponential service times are relevant to container scheduling, where containers are servers serving requests (Terekhov et al. 2014).

In scheduling, queuing theory is responsible for keeping the system stable and bounds in queues. (Terekhov et al. 2014) integrated queuing theory and scheduling to address dynamic issues with improved performance. Similarly, this thesis utilizes application and backlog queues, represented to forecast scheduling activity and implement a container-aware scheduler (Moniruzzaman et al. 2017).

Chapter 3

Methodology

Experimental setup

Installed a fresh installation of Ubuntu with Completely Fair Scheduler as the default scheduler for CPUs onto a virtual machine with 4 cores. after initial setup, I installed Kernelshark so that I had a much better idea of what data was being retrieved from the Ftrace buffer. Kernelshark has some significant filtering strategies in addition to its data plots and visualizations which are invaluable in the research of CPU events.

Implemented an HTTP server that is specifically designed to handle the server aspects. Other complicated servers use other mechanisms with complicated thread management to handle different simple requests, making it almost impossible to track CPU events and their behavior. This server has following features.

- An accept loop that accepts the request from the kernel backlog and starts processing.
- The requests are batched to a dynamic size to prevent the thundering herd problem first.
- When the queue is not congested, a worker thread is notified about availability using `thread_signal_cond`.
- Worker threads dequeue available requests and start executing them.

- The main thread is constantly attempting to accept and batch incoming requests regardless of slots available in the application queue. It ensures the rate of departure of the backlog queue is unrelated to the rate of completion of the requests.

It is both tested within and outside a docker container and it witnessed the scheduling events. For having more precise details about event scheduling as well as about virtual runtime values, my own custom system calls have been introduced. Automated requests have been created using curl command through the integration of bash scripting to simulate real world workloads. This is a service or a container depending upon the server container. Used eBPF to intercept the corresponding kernel functions to be called for system tcp events and forward these data into user space with little latency. Another standalone thread was added to the server that acts as a mid-layer between kernel and application to help other tools and container orchestration systems to offer statistics about the traffic of the container and also act to manage server properties like worker thread.

Data Collection

Used the integrated Ftrace kernel tracing in utilized kernels to get data about the system TCP stack's request processing and also about the CPU events triggered and their timing. Used the trace-cmd to get data from the Ftrace buffer as a '.dat' file so that it can be directly fed to the Kernelshark for improved-analyzing features. A Bash script was used to trigger tracing automatically and filter the Ftrace configuration as well as submitting different numbers of concurrent requests to the hand-written C server.

Used perf tool and also some custom made system calls to collect information about schedulers internal parameters like virtual runtime.

And also the experiment of data gathering in this study was to examine the behavior of the main thread of a C server, and more specifically the CPU migrations, with different mixes of concurrent requests, worker threads, and kernel backlog sizes. They have been chosen as parameters because they directly influence the scheduling activity of the server, and it would be interesting to see how they combine in order to cope with performance bottlenecks, priority inversion, and patterns of CPU migration.

During each experiment, three parameters were varied: the number of worker threads, the number of concurrent requests, and the size of the kernel backlog. The number of concurrent requests models different levels of incoming traffic to the server that impact the tasks queuing mechanism to process. The ability of the server to execute these requests in parallel is regulated by the worker threads, and the size of the kernel backlog is the amount of waiting requests when the worker threads are busy.

At each specific value of the variables, measurements were taken over some specified time period, with experiments consistent. To correct for variability, at least 10 measurements were obtained for each configuration, and the average measurement was established for each set. This served to remove the effect of outliers or transient aberrations which could distort the results so that measurements taken reflect the steady-state operation of the system.

This experiment was conducted under different conditions of the variables that resulted in 250 averaged points. For a single variable of the three variables, one was highlighted for one experiment while the other two were maintained constant in a bid to determine the effect of a single factor. For example, in a certain experiment, the number of concurrent requests were varied while maintaining constant the number of worker threads and backlog size. In a second experiment, worker thread numbers and concurrent request were kept constant while kernel backlog sizes were varied. This allowed exhaustive investigation of each variable's influence independently on server behavior.

Values of observed points indicated the relationship between concurrency of requests, kernel backlog size, and worker threads and their influence on CPU migration behavior. Through a close examination of the effects of such diverse configurations on task scheduling, the study attempted to identify patterns of delay and inefficacy, particularly when tasks of diverse priorities were scheduled onto a shared CPU. The main objective of data gathering was to gather enough data to be able to make a general analysis of system load and configuration impact on scheduling behavior so that it will be possible to better monitor how priority inversion and CPU migration can be avoided in containerized systems.

Analysis

I have analyzed the data gathered on different numbers of concurrent requests with the help of Kernelshark. First, calculate the time when the server's main thread transitions from ready state (`SCHED_WAKEUP`) to scheduled state (`SCHED_SWITCH`). Then, I calculated the CPU schedulers activity and determined how many other processes were scheduled within that time. And also made sure that all processes were being executed at present and none of them completed their execution.

To investigate CPU migrations further, I had to employ a different approach. The data gathered were analyzed using in-house Python scripts written for this study. The scripts were used to process, analyze, and plot the measurements in an organized and meaningful way. The primary purpose of the analysis was to test how different settings of the system—i.e., the numbers of concurrent requests, worker thread levels, and kernel backlog levels—affect the CPU migration patterns of the main thread of the server at distinct levels of loading.

Analysis process had some crucial steps:

- Data Prepossessing Raw data from each experiment was preprocessed and cleaned first. This included deleting anomalous or incomplete records, handling missing values, and normalizing data where necessary. This introduced consistency and reliability across the dataset, rendering it suitable for comparison analysis.

- Descriptive Statistics For each configuration, statistical summaries such as the mean, variance, and standard deviation were computed. Statistics were used to summarize CPU migration central tendency and variability across each set of conditions. Averaging over at least 10 data points per configuration also aided the strength of the analysis.

Visualization

Several visualizations were produced using Python libraries in order to more intuitively understand variable-to-variable relationships. Three primary types of charts were used:

- Heatmaps were used to capture and plot the density of CPU migrations between pairs of variables (such as worker threads vs. concurrent requests). These plots provided a bird's eye view of how configurations led to greater or lesser CPU migration.
- Line graphs were used to mark trends and movement, i.e., the variation of CPU migrations when moving one variable gradually and holding others steady. They were particularly good for finding thresholds or points of turning in system behavior.
- Bar plots were used in the comparison of the mean number of migrations for different discrete values of configurations. The plots emphasized the difference between settings very clearly, and the best and worst performing setups were easily identifiable at a glance.

Investigation and Interpretation

Patterns were identified from the graphs that indicated how the server's scheduling and CPU migration activity reacted to varying workload conditions. For example, in some environments, increasing the number of worker threads beyond a given point caused CPU migrations to balloon, suggesting greater contention or overhead. Similarly, some combinations of high concurrent request rates and small backlog sizes were linked to extreme scheduling latency.

Cross-variable Analysis

Interactions between variables were specifically addressed, and particularly through the use of heatmaps, when investigating how two variables collectively defined migration behavior. Such plots were utilized to reveal complex dependencies not readily found with uni-variate analysis.

Through the use of these visualization tools and statistical techniques, the analysis would be capable of achieving a thorough understanding of how different system configurations impacted schedule efficiency and CPU affinity stability. The outcome of this analysis phase was used as the basis for identifying performance bottlenecks and was directly used in formulating possible approaches to minimizing CPU migrations and priority inversion in multi-threaded server systems.

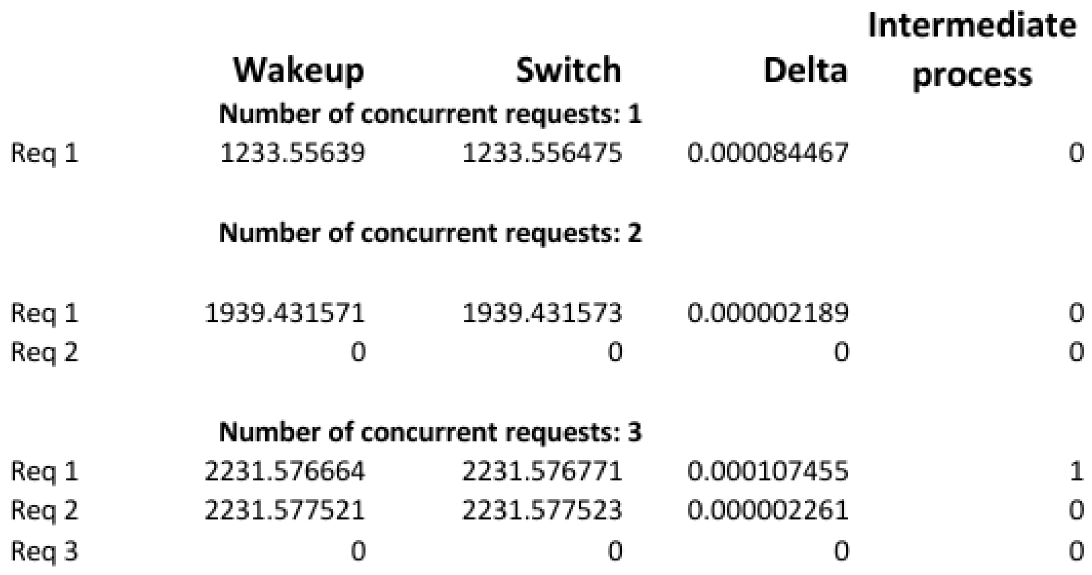
Chapter 4

Experiments & preliminary results

I tried to create my own tracing facility inside the kernel for further modifications by using kernel's native *printk* function but it added some serious issues to the kernel because prink supports a blocking type of functionality and inserting a blocking code directly into a CPU scheduler leads to a freeze of the system since the CPU scheduler will not be able to schedule any activity until the *printk* gets de-blocked.

As I did not utilize my own tracing tool, I followed Ftrace together with some complementary tools like trace-cmd and Kernelshark for additional data capture and examination. First, I tried investigating the CPU scheduler activity for a fixed number of simultaneous requests. The server is also made to create a long-run execution to make sure that none of the requests gets execution prior to the rest of the processes start getting executed. As can be noticed from the results, there is a boost in the latency when the 3rd request gets its turn in the CPU. Thus, I made the decision to incorporate a different number of concurrent requests and analyze each one of them separately figures 4.1 4.2 4.3 demonstrates the results.

Here, the *Wakeup* column represents the time when the main thread became ready after the request was processed by the system's TCP stack, and *Switch* represents the time when it was scheduled on the CPU. *Delta* column represents



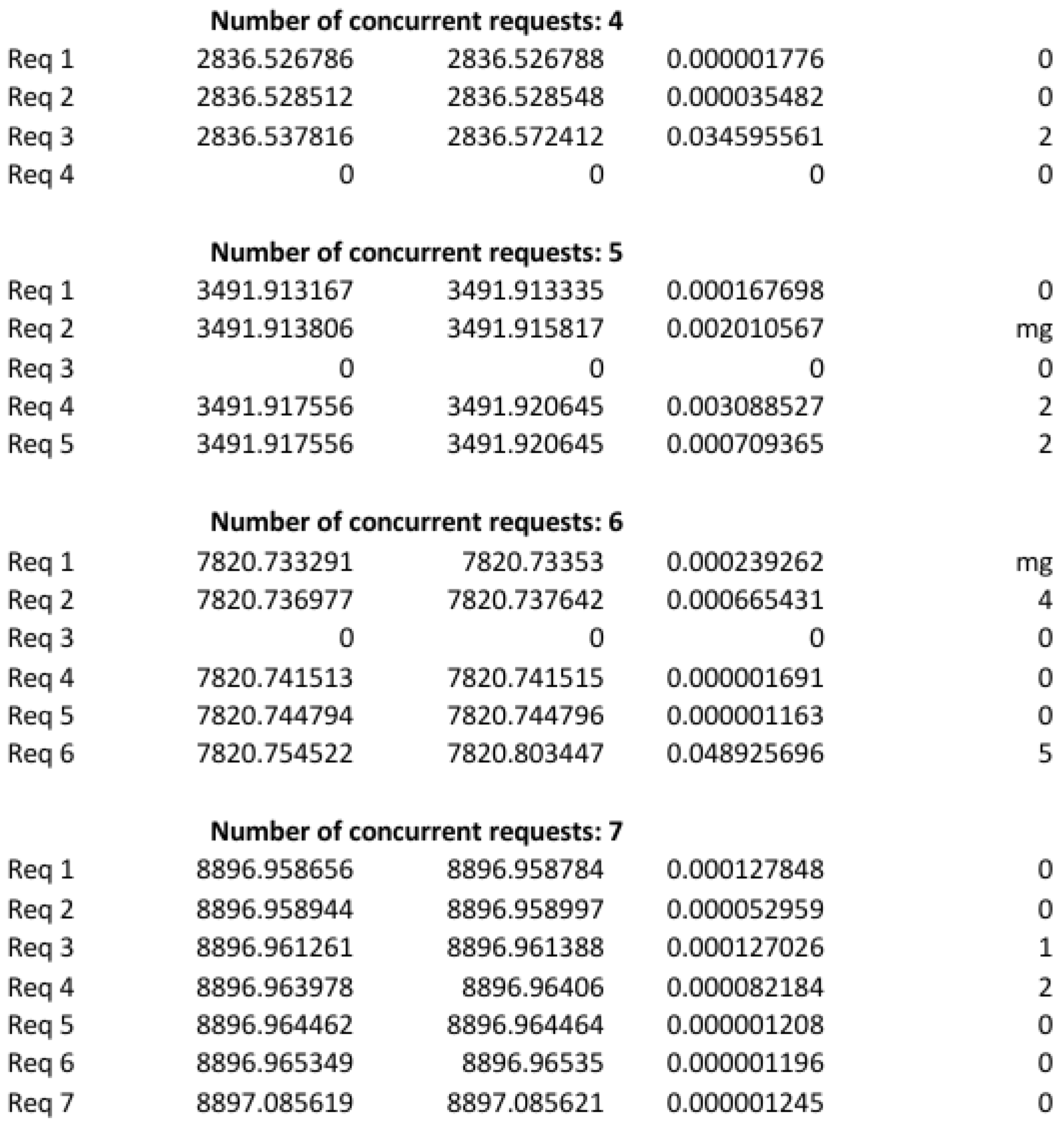
	Wakeup	Switch	Delta	Intermediate process
	Number of concurrent requests: 1			
Req 1	1233.55639	1233.556475	0.000084467	0
	Number of concurrent requests: 2			
Req 1	1939.431571	1939.431573	0.000002189	0
Req 2	0	0	0	0
	Number of concurrent requests: 3			
Req 1	2231.576664	2231.576771	0.000107455	1
Req 2	2231.577521	2231.577523	0.000002261	0
Req 3	0	0	0	0

Figure 4.1: Number of Requests 1 to 3

the difference between the two (scheduling latency) and the last column represents the number of processes that were scheduled in the meantime.

As you can see some records have a value called **mg** in the last column. I used this to indicate a thread migration in the CPU i.e., the server's main thread was migrated to a different CPU. Since the CPU scheduler is unaware of the cross-container dependencies and priorities of the processes, this experiment setup is analogous to the containerized setup that I have described in the earlier sections.

From the above data, we can observe clearly that there is a possibility of cross-container priority inversion. particularly in the request 8 of 10 concurrent request experiment, it shows 21 SCHED_SWITCH events which means this process was ignored 21 times by the CPU scheduler during making the scheduling decision. it has happened several times. As we can see, The CFS scheduler due to its load balancing mechanism has shifted the main thread to another cores and it shows that this clearly increases the scheduling latency of the server's main thread which can result in poor performance and even connection rejection due to the inability to serve the new incoming request(Main thread is still in the processing stage of the previous request and have not passed it to a new thread yet). This leads us to further investigation of the latency and performance of the scheduler with CPU



Number of concurrent requests: 4

Req 1	2836.526786	2836.526788	0.000001776	0
Req 2	2836.528512	2836.528548	0.000035482	0
Req 3	2836.537816	2836.572412	0.034595561	2
Req 4	0	0	0	0

Number of concurrent requests: 5

Req 1	3491.913167	3491.913335	0.000167698	0
Req 2	3491.913806	3491.915817	0.002010567	mg
Req 3	0	0	0	0
Req 4	3491.917556	3491.920645	0.003088527	2
Req 5	3491.917556	3491.920645	0.000709365	2

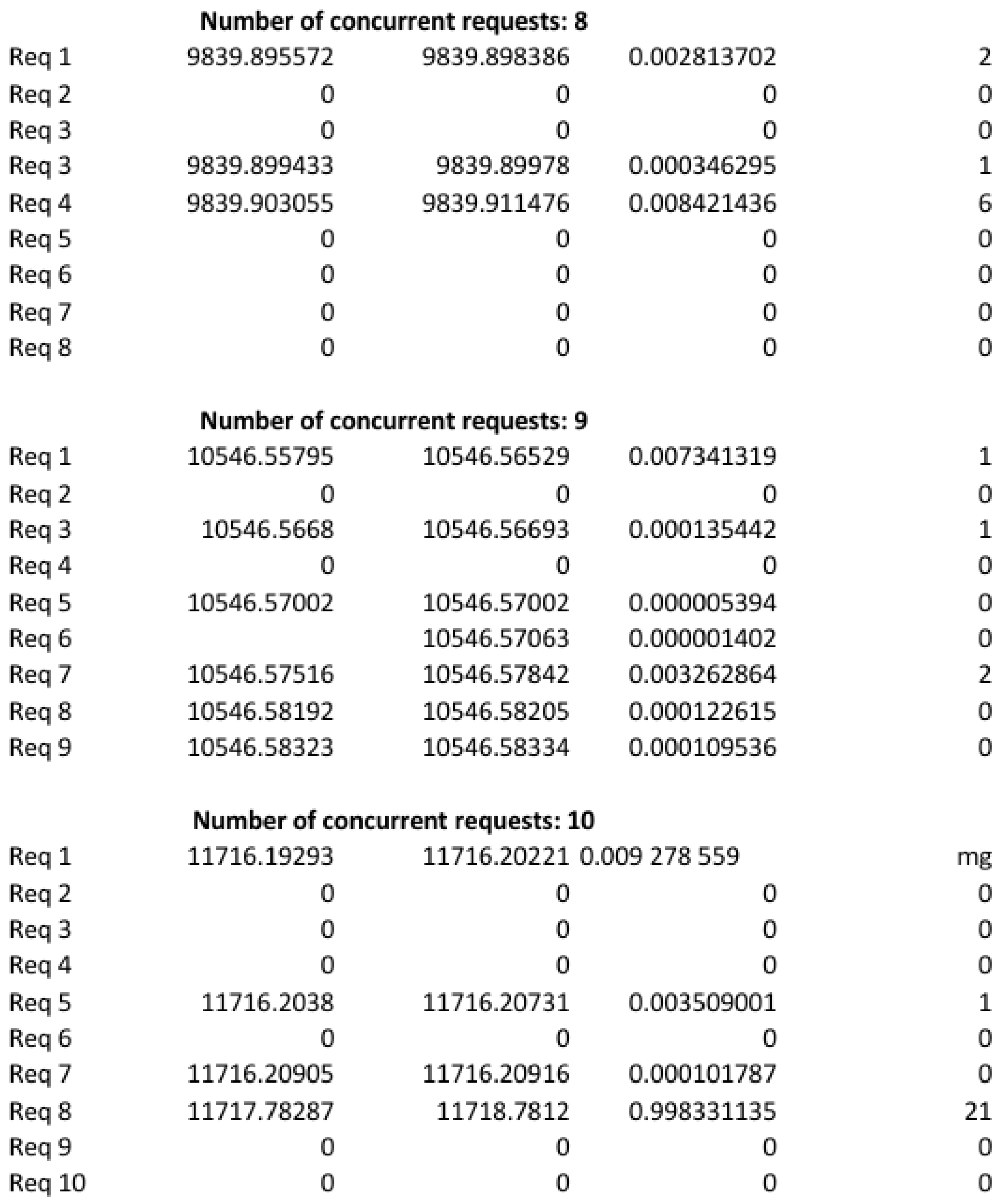
Number of concurrent requests: 6

Req 1	7820.733291	7820.73353	0.000239262	mg
Req 2	7820.736977	7820.737642	0.000665431	4
Req 3	0	0	0	0
Req 4	7820.741513	7820.741515	0.000001691	0
Req 5	7820.744794	7820.744796	0.000001163	0
Req 6	7820.754522	7820.803447	0.048925696	5

Number of concurrent requests: 7

Req 1	8896.958656	8896.958784	0.000127848	0
Req 2	8896.958944	8896.958997	0.000052959	0
Req 3	8896.961261	8896.961388	0.000127026	1
Req 4	8896.963978	8896.96406	0.000082184	2
Req 5	8896.964462	8896.964464	0.000001208	0
Req 6	8896.965349	8896.96535	0.000001196	0
Req 7	8897.085619	8897.085621	0.000001245	0

Figure 4.2: Number of Requests from 4 -7



Number of concurrent requests: 8

Req 1	9839.895572	9839.898386	0.002813702	2
Req 2	0	0	0	0
Req 3	0	0	0	0
Req 3	9839.899433	9839.89978	0.000346295	1
Req 4	9839.903055	9839.911476	0.008421436	6
Req 5	0	0	0	0
Req 6	0	0	0	0
Req 7	0	0	0	0
Req 8	0	0	0	0

Number of concurrent requests: 9

Req 1	10546.55795	10546.56529	0.007341319	1
Req 2	0	0	0	0
Req 3	10546.5668	10546.56693	0.000135442	1
Req 4	0	0	0	0
Req 5	10546.57002	10546.57002	0.000005394	0
Req 6		10546.57063	0.000001402	0
Req 7	10546.57516	10546.57842	0.003262864	2
Req 8	10546.58192	10546.58205	0.000122615	0
Req 9	10546.58323	10546.58334	0.000109536	0

Number of concurrent requests: 10

Req 1	11716.19293	11716.20221	0.009 278 559	mg
Req 2	0	0	0	0
Req 3	0	0	0	0
Req 4	0	0	0	0
Req 5	11716.2038	11716.20731	0.003509001	1
Req 6	0	0	0	0
Req 7	11716.20905	11716.20916	0.000101787	0
Req 8	11717.78287	11718.7812	0.998331135	21
Req 9	0	0	0	0
Req 10	0	0	0	0

Figure 4.3: Number of Requests from 8 -10

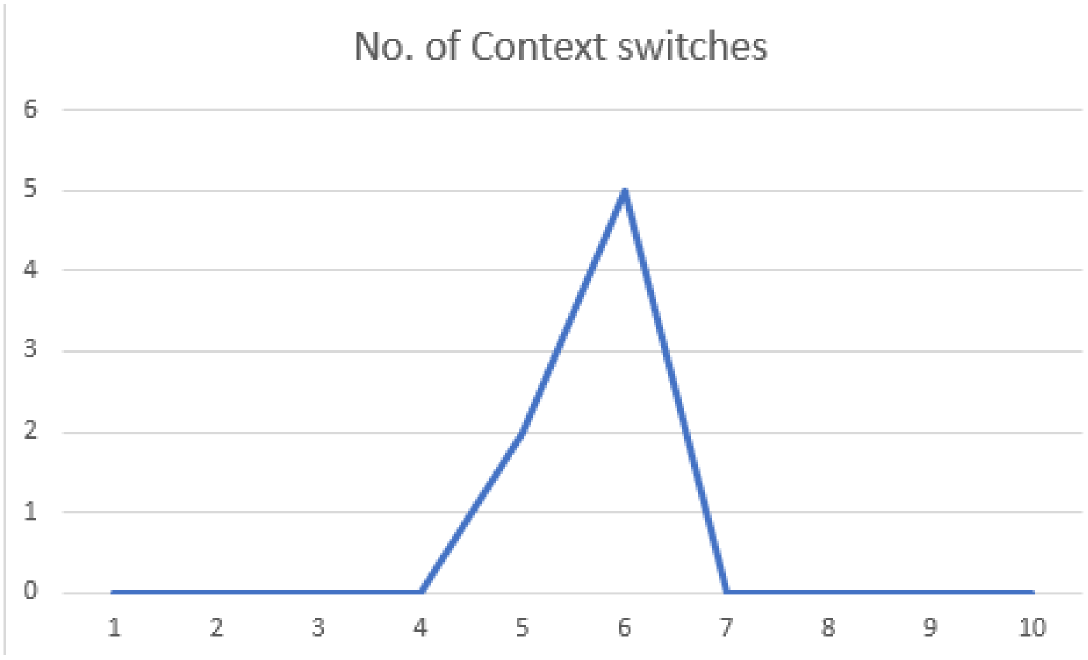


Figure 4.4: For 10 concurrent requests

pinning policies.

The following graph shows the number of in-between process scheduling for 10 concurrent requests. We can see that after going through a smaller number of context switches it shows a spike and then drops to low values again. This is the expected behavior of the CFS algorithm. As the main thread receives more time in the CPU its virtual runtime is incremented and the scheduler ignores it for some time until the other tasks catch up with it. And then once more the main thread scheduling latency goes down to a lesser amount. As a result of this, it increases the probability of priority inversion in a cross-container scenario. The same result can be observed for different numbers of concurrent requests in the above data.

Now I have turned my focus to the CPU migrations. Figure 4.5 is a line graph illustrating kernel backlog size vs average number of CPU migrations endured by the C server's main thread. From the figure herein, one observes an overall trend of increase for CPU migrations with increasing backlog value, indicating positive correlation between variables. This indicates that bigger backlog sizes, which allow the server to queue more of the incoming connections, can occur to impose additional scheduling overhead, perhaps due to increased contention between the threads competing to accept and serve connections.

While the chart does appear to have a trend, the chart also indicates that there is a lot of fluctuation or "noise" in the data. These variations are likely caused by external factors like other running processes on the system, CPU cache states, background kernel tasks, or other standalone thread scheduling policies within the measurement window. These random system states introduce randomness to the decisions made by the kernel, affecting the stability of measured CPU migration counts.

However, the line graph still clearly displays an underlying pattern: once the backlog value goes over a point, CPU migrations will rise more sharply. Such behavior is in accordance with predictions from theory because an increasing backlog implies more events on sockets being active and perhaps more accept calls by worker threads, increasing the chances of CPU migration when thread affinity or load balancing is activated by the kernel.

This result underscores the importance of configuring kernel-level parameters, such as the backlog size, in high scheduling determinism and low-latency processing environments—primarily in containerized or real-time servers. More experiments with more controlled settings can potentially remove noise and separate the effect of each variable more effectively.

In addition to further investigating the interactive effect of greater than a single parameter upon CPU migration behavior, I extended the research further by creating heat plots as figure 4.6 that reflect interaction between worker thread number, kernel backlog size, and associated average CPU migration rate. Heat plots were employed as a visualization method for this reason since they can encode variations of a third variable as color intensities in a 2D parameter space.

The resultant heat maps, on the other hand, were not that informative about a readily apparent pattern like in the earlier line chart. This suspicion is mostly a result of system-level noise and runtime externalities such as background kernel activity and thread interference in scheduling that continued to impact consistency

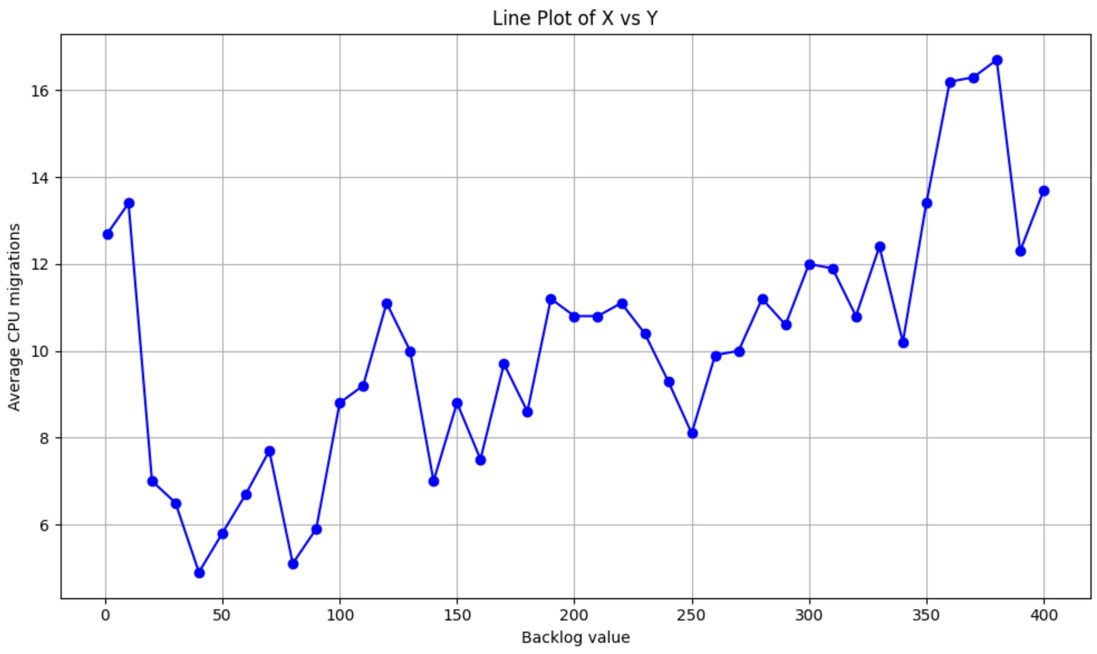


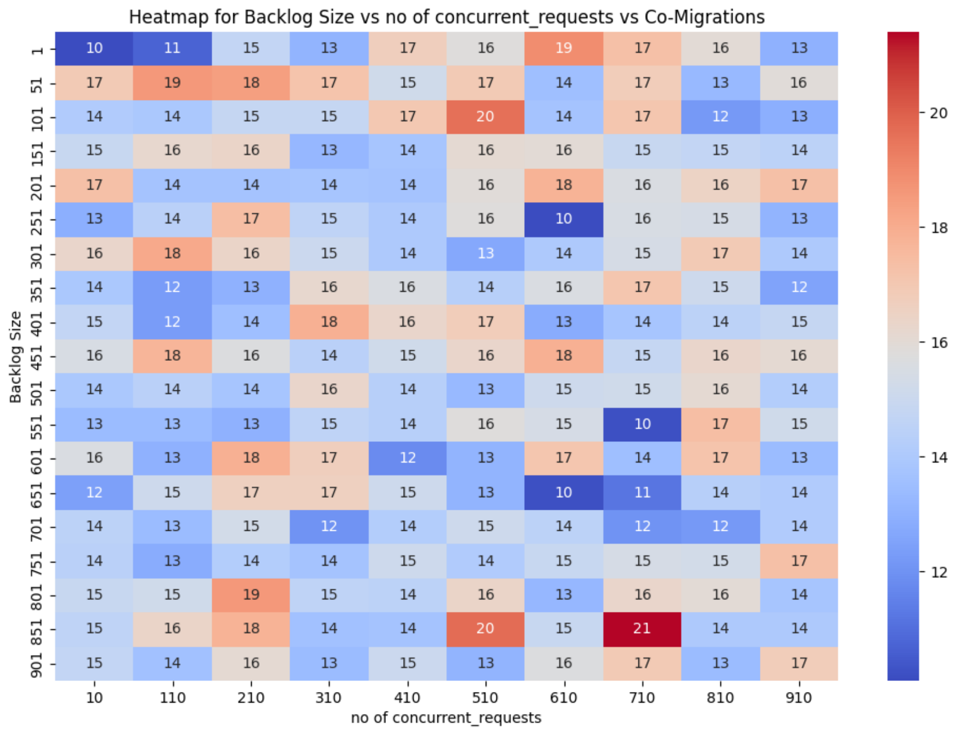
Figure 4.5: Kernel backlog vs CPU migrations

of measurements experiment to experiment. Despite all this activity, there was still a general trend: CPU migrations were growing towards bigger backlog size as well as increasing numbers of concurrent requests.

This trend, though present, was less extreme and obvious than that which was found in the line plot of average CPU migrations vs. backlog size. It suggests that although these parameters themselves are sources of scheduling complexity, their interaction is an additional source of variability that could hide obvious correlations. These findings indicate the challenge of describing low-level performance properties in typical environments for computation, where system interference due to non-controllable process can be present, hiding the localized behavior of interest.

4.1 Using polynomial regression modeling

For further investigation of the relationship between kernel backlog size and average CPU migrations over a specified time window, a simple machine learning approach was considered. A degree-2 polynomial regression model was utilized



Backlog Size	10	110	210	310	410	510	610	710	810	910
1	10	11	15	13	17	16	19	17	16	13
51	17	19	18	17	15	17	14	17	13	16
101	14	14	15	15	17	20	14	17	12	13
151	15	16	16	13	14	16	16	15	15	14
201	17	14	14	14	14	16	18	16	16	17
251	13	14	17	15	14	16	10	16	15	13
301	16	18	16	15	14	13	14	15	17	14
351	14	12	13	16	16	14	16	17	15	12
401	15	12	14	18	16	17	13	14	14	15
451	16	18	16	14	15	16	18	15	16	16
501	14	14	14	16	14	13	15	15	16	14
551	13	13	13	15	14	16	15	10	17	15
601	16	13	18	17	12	13	17	14	17	13
651	12	15	17	17	15	13	10	11	14	14
701	14	13	15	12	14	15	14	12	12	14
751	14	13	14	14	15	14	15	15	15	17
801	15	15	19	15	14	16	13	16	16	14
851	15	16	18	14	14	20	15	21	14	14
901	15	14	16	13	15	13	16	17	13	17

Figure 4.6: Heatmap of average CPU migrations vs Backlog size vs Number of concurrent requests

using the `scikit-learn` Python library. The model took in backlog size (B) and observation time window (T) as input variables, and output the average number of CPU migrations (M). The second-degree polynomial features were used in the regression:

$$B^2,\ T^2,\ B \cdot T,$$

i.e., six model parameters total including the bias term.

The model was trained on 206 data points sampled from controlled experiments in real-world settings. These were sufficient for the nature of the low-complexity regression model being employed. The resulting model had the following parameters learned:

$$\text{Intercept: } 2.3330$$

$$\text{Coefficients: } 0.0000,\ -0.0558,\ 3.6651,\ 0.0001,\ 0.0007,\ 0.1422$$

These are associated with the respectively constant term, B, T, B^2, $B \cdot T$, and T^2.

The training set Mean Squared Error (MSE) was 86.9830, a moderate error of prediction. The reason for the high MSE here is attributable to external system-level noise, i.e., OS background processes or asynchronous I/O activity, beyond the control of the experimental environment and naturally affecting CPU scheduling decisions. Despite the noise, the model did pick up on an overall trend that linked larger monitoring time and backlog to larger migration quantities. The prediction error variability also suggested, however, that the dynamic scheduling activity could not be described by a low-order polynomial model for variable system loads.

This modeling activity illustrates the possibility for data-driven methods to model CPU migration behavior, and the challenge of prediction in the presence of operating system noise and runtime variation.

4.2 Improved Random Forest Model

For the improvement of the model's performance, a Random Forest regression model with a larger set of features was employed. The feature set included:

- Base features: Backlog (B), Time (T)
- Interaction features: $B \times T$
- Polynomial features: B^2, T^2
- Logarithmic features: $\log(B)$, $\log(T)$

The Random Forest model provides several advantages for this research:

- Ability to model non-linear relationships without explicit definitions
- Insensitivity to outliers and noise in the dataset
- Quantification of feature importance
- Reduced risk of overfitting through ensemble averaging

It was trained using 200 decision trees with a maximum depth of 8 and minimum of 5 samples per split for an internal node—parameters tuned for the relatively small dataset.

4.2.1 Model Performance

Comparison of Performance Metrics

The Random Forest model was found to exhibit considerable improvement against the polynomial regression technique, as reflected in Table 4.1.

The low training R^2 of 0.968 accounts for the fact that the model explains only approximately 97

Table 4.1: Comparison of Model Performance Metrics

Metric	**Polynomial Regression**	**Random Forest**
Train R^2	Low value	0.968
Test R^2	Low value	0.799
Train MSE	High value	11.516
Test MSE	High value	38.705

Model Evaluation

The difference in performance metric between training and test—a difference of 0.169 in R^2 and 27.189 in MSE—represents mild overfitting. But not drastic considering the complexity of the model and small dataset size. Such differences are expected in situations of low data availability and high-order feature interactions.

The actual vs. predicted CPU migration scatter plot (Figure reffig:scatter) is a visual confirmation of the prediction ability of the model. Points clustering near the diagonal line indicate good predictions, and straying away from it indicates errors. The graph depicts the following:

- High accuracy of predictions for low migration numbers (0–20)
- Good accuracy for moderate migration numbers (20–40)
- Increasing variance for high migration numbers (>40)

This trend suggests that the model performs best in the lower migration range, which aligns well with typical production workloads.

Since my initial efforts at modeling the dependency of backlog size, time, and CPU migrations as a polynomial regression did not work, I attempted a set of other machine learning methods. None of them, however, provided me with the accuracy or insight that I was looking for. I therefore diverged from purely machine learning methods and adopted a more traditional method that offered a clearer understanding of the most significant factors for CPU migrations.

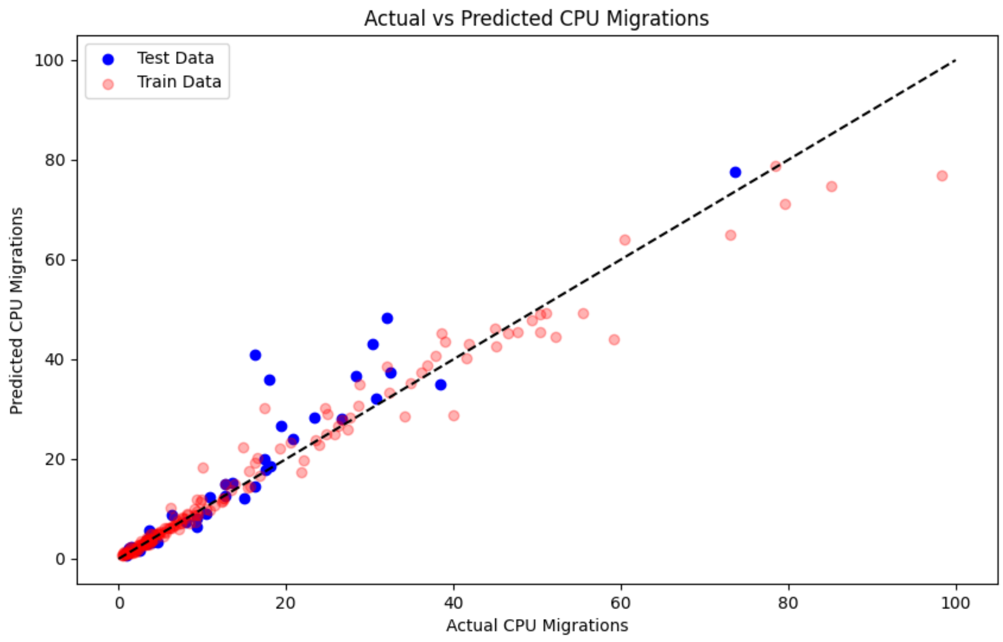


Figure 4.7: Actual vs Predicted CPU Migrations

All of the above described data gathering and processing—like mean CPU migration rates, heat maps, and model fitting—was done on a queue-based server implementation, where an arriving connection is intercepted by a centralized queue and allocated to worker threads. To further explore the impact of the handling of connections mechanism on CPU scheduling behavior, I also conducted an experiment in parallel using a no-queue implementation. In this case, every worker thread invokes accept() independently in order to handle accepted connections directly without going through the shared queue and hence balancing the load among the threads.

This implementation is frequently associated with the ”thundering herd” problem, wherein various threads wake up concurrently to process one connection, perhaps increasing CPU contention and causing additional CPU migrations owing to wake-ups and accept lock contention. The results were quite surprising. The heat map of the no-queue implementation (Figure 4.8) illustrated a steep dipping of CPU migrations for heterogeneous backlog sizes and concurrent request rates compared to the centralized queue-based implementation.

This is the opposite of the traditional hypothesis that centralized queues yield greater coordination and fewer migrations. My experiments, however, show that queue-based gives rise to a built-in inefficiency in CPU affinity retention, perhaps due to the manner wake-up and scheduling happen as threads wait on a common queue for items. The no-queue design, despite its possible risks of contention, seems to generate greater CPU-locality and load dispersion across cores.

From this inconsistency, I concluded that the queue-based server model inherently experiences more CPU migrations, which would potentially affect performance under high concurrency. The reason is in the locking mechanism of the shared application queue. In this design, one master thread accepts connections and places them in a shared application queue. Worker threads take a lock on the queue to get work. When many threads are competing for this lock, especially in heavy load, the primary thread will block, waiting to get the queue lock. Each time the primary thread blocks or is being awakened, the Linux CPU scheduler can schedule it to another CPU, especially when the system contains many runnable tasks and many CPUs. This recurring task blocking and wake-up cycle has the excessive CPU migration overhead, which can be seen in the above-mentioned heat maps.

To mitigate this, I explored an alternative optimized design that is based on batched queuing. Using this approach, instead of enqueueing each request directly into the shared app queue, I batch a sequence of requests and then add them in batches with the least amount of accesses to the queue. This method slows down the rate at which the main thread needs to fight over the lock, which means less block-wake context switches and, as a result, significant reduction in CPU migrations.

After this enhancement, I proceeded by dividing the backlog queue and application queue into two different M/M/C/K queuing systems. This abstraction allowed me to utilize queueing theory to compute optimal configurations analytically. Specifically, through the examination of the queue parameters such as arrival rate, service time, and workers' capacity, I was able to establish the number of workers required to handle the load arriving without excessive idleness or build-up

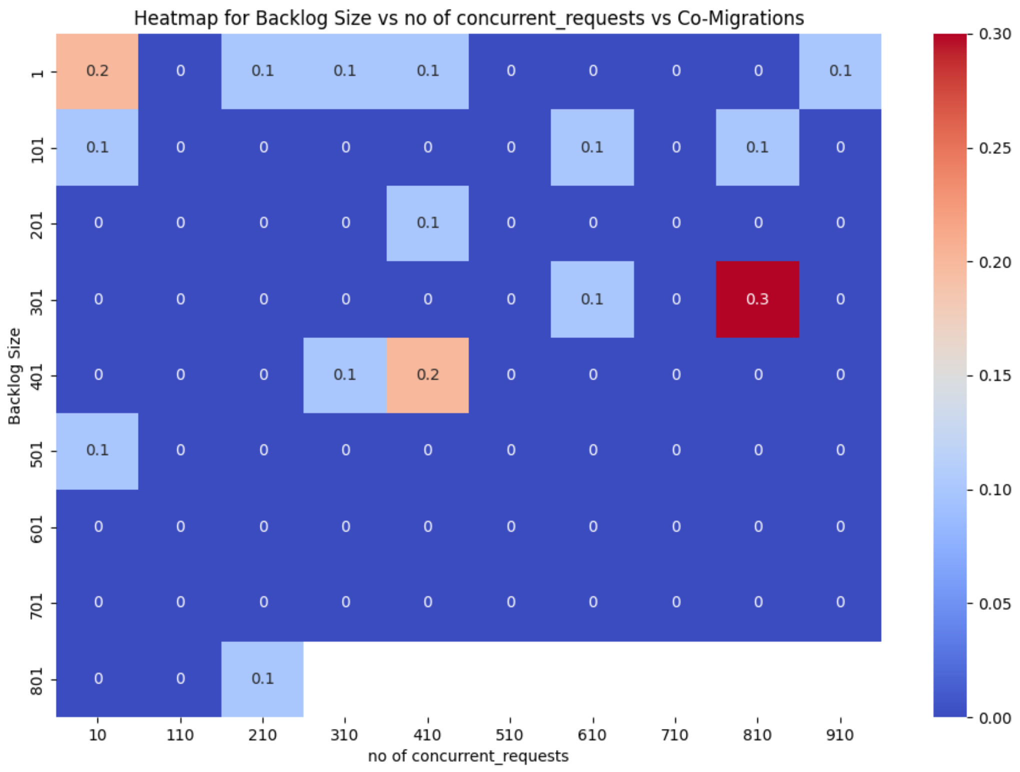


Figure 4.8: Heat map of CPU migrations vs time period monitored vs backlog size with accept based implementation

in the queue. Lowering the number of workers in this fashion also lowers CPU contention, again resulting in more predictable scheduling performance, lower CPU migrations, and overall system effectiveness.

This multi-level optimization tactic—batching requests, lock contention reduction, and mathematical simulation of queue operation—represents a robust and empirically efficient way to minimize the performance loss of CPU migration in typical queue-based server structures.

While the batch processing mechanism has been examined in other server architectures to improve performance during load, typical implementations of batch size are fixed. This approach can be wasteful in the situation of highly lightweight or bursty traffic, where the fixed threshold can result in the delay of processing too long (waiting to finish the batch) or waste of system resources. To counter this, I time based batchng where the batch size is determined based on observed request arrival rates and queue utilization at present. It allows the system to cope with varying traffic loads gracefully while preserving the benefits of reduced contention and improved CPU affinity.

Most notably, while these issues are addressed within a monolithic server context, the underlying notions are highly relevant to containerized environments. These CPU scheduling inefficiencies can be confidently addressed by introducing a custom daemon process at host or orchestrator levels. This daemon could monitor per-container CPU migration behavior, queue lock activity, and contention rates and relay this information back to the Kubernetes scheduler (or similar orchestration frameworks). With the Kubernetes scheduling algorithm becoming incorporated to account for such CPU affinity-aware optimizations, it would then be feasible to make smarter placement decisions, reduce unnecessary migrations, and ultimately enhance the performance and efficiency of containerized workloads—particularly in high-density or latency-sensitive situations.

Chapter 5

Implementation and Design

This chapter details the implementation of a custom C-based HTTP server and an eBPF-based monitoring system designed to evaluate CPU scheduling behavior in containerized environments. The architecture simulates production-like, CPU-bound workloads to assess the Linux Completely Fair Scheduler (CFS) under high concurrency, focusing on CPU migrations, queuing delays, and performance predictability Pabla (2009), ?. The design prioritizes modularity, low-overhead telemetry, and compatibility with container orchestration, enabling precise measurement of scheduling and network metrics to address research questions concerning queue dynamics, CPU affinity, and scheduling optimization.

5.1 Server Architecture Overview

The HTTP server employs a queue-based, multi-threaded architecture, implemented in C with the POSIX threads (pthreads) library, to simulate high-concurrency workloads Marinakis et al. (2017). Integrated with an eBPF-based monitoring system, it captures kernel-level metrics, such as scheduling events and task migrations, to analyze CFS behavior. The server comprises four key components:

- **Non-blocking Accept Loop:** The main thread continuously accepts TCP connections on port 8080 using a non-blocking `accept()` system call, ensuring efficient connection handling Mathis et al. (1997).

- **Connection Queue:** A shared, fixed-size circular buffer stores accepted connection file descriptors, facilitating efficient handoff to worker threads Torvalds (2007).
- **Worker Thread Pool:** A configurable pool of worker threads dequeues connections, performs CPU-bound computations, and generates HTTP responses, enabling controlled evaluation of thread contention and migrations Moniruzzaman et al. (2017).
- **eBPF Instrumentation:** eBPF programs attach to kernel scheduling and network tracepoints, providing low-overhead metrics to user space via shared memory for real-time performance analysis Bentaleb et al. (2022).

This modular design separates connection acceptance from request processing, enabling independent analysis of queuing delays, backlog behavior, and CPU scheduling characteristics, directly supporting the investigation of application-level queue impacts and CPU affinity.

5.2 Handling Incoming Requests

The server initializes by binding to TCP port 8080 with a kernel backlog queue of 512 connections to accommodate bursts without rejecting requests Mathis et al. (1997). The socket is configured as non-blocking, allowing the main thread to poll for client connections without interruption. Connections are processed in batches to optimize throughput and reduce scheduling overhead.

The main thread operates in time-bound cycles (`duration_ms`), collecting connections into a temporary buffer (`batch_fds[]`) using `accept()` calls. When the time window expires or the batch size limit is reached, the batch is transferred to a global shared queue. This batching approach minimizes context switches and supports measurement of kernel backlog and application queue dynamics, with eBPF probes tagging each accepted connection as an arriving request to assess scheduling delays Terekhov et al. (2014).

5.3 Connection Queue Design

The connection queue is a fixed-size (`QUEUE_SIZE`) circular buffer, protected by a `pthread_mutex_t` to ensure thread-safe access between the producer (main thread) and consumers (worker threads) Torvalds (2007). Two condition variables manage queue state:

- `queue_not_empty`: Signals worker threads when connections are available, preventing idle spinning.
- `queue_not_full`: Signals the main thread when space is available, avoiding queue overflow.

The queue operates as follows:

- Accepted connections are enqueued if space is available; otherwise, connections are discarded to prevent blocking.
- Worker threads continuously dequeue and process connections until server shutdown.
- Upon shutdown, worker threads complete in-flight requests and terminate gracefully.

Synchronization ensures consistent queue operations, enabling accurate measurement of queue occupancy and request throughput, which informs the optimization of server-side parameters Terekhov et al. (2014).

5.4 Monitoring Kernel Backlog with eBPF

An eBPF-based monitoring system tracks the kernel's TCP listen backlog queue, capturing connection arrival rates before `accept()` calls in a multi-namespace environment Bentaleb et al. (2022). This low-overhead approach provides per-container insights into kernel-level scheduling and network behavior, supporting analysis of queue dynamics.

5.4.1 eBPF Program Design

Map Definition

A `BPF_HASH` map, `allowed_ns`, tracks connection counts per network namespace:

struct $_{uint(type,BPF_MAP_TYPE_HASH);_{uint(max_entries,10);_{type(key,u64);//netnsinode_{type(value,u32);//countofestablishedcor}}}}$

This map associates each network namespace inode (`u64`) with a 32-bit counter, limited to 10 entries for efficiency.

Kprobe on `tcp_set_state`

A kernel probe on `tcp_set_state` monitors TCP socket state transitions, filtering for `TCP_ESTABLISHED` to record completed three-way handshakes:

- Retrieves the namespace inode using `BPF_CORE_READ`.
- Increments the counter in `allowed_ns` if the inode exists; otherwise, initializes it to 1.

SEC("kprobe/tcp$_set_state")inthandle_conn(structpt_regs*ctx)intnewstate = (int)PT_REGS_PA$

struct sock *sk = (struct sock *)PT$_REGS_PARM1(ctx); u64ns_inum = BPF_CORE_READ(sk$

u32 *count = bpf$_map_lookup_elem(allowed_ns, ns_inum); if(count)_{sync_fetch_and_add(count,1);elseu32init=}$

This probe minimizes overhead by triggering only on significant state changes Bentaleb et al. (2022).

5.4.2 User-Space Monitoring

A detached user-space thread, implemented in `ebpf_monitor.c`, processes eBPF telemetry to compute performance metrics Terekhov et al. (2014).

Load and Attach

The thread uses the libbpf skeleton (`monitor.skel.h`) to load and attach the eBPF program, initializing the `allowed_ns` map entry for the current namespace via `stat("/proc/self/ns/net")` Bentaleb et al. (2022).

Periodic Polling

Every 10 seconds, the thread:

- Retrieves the connection count using `bpf_map_lookup_elem`.
- Resets the map entry with `bpf_map_update_elem` for the next interval.
- Collects application-level counters (`leaving_backlog_count`, `incoming_request_count`, `leaving_request_count`) under mutex protection.

Queue Metrics Calculation

Using kernel backlog arrival rates (`count/10`) and application dequeue rates (`leaved/10`), the thread computes M/M/c/K waiting times for the kernel backlog and application queues via the `calculate_Wq` function, enabling analytical performance evaluation Terekhov et al. (2014).

Output

The thread outputs per-interval metrics:

```
Backlog connections:    <count>/s
Backlog leaving rate:   <leaved>/s
Application arrival rate: <received>/s
Application service rate: <served>/s
Backlog Wq:             <seconds>
Application Wq:         <seconds>
Ratio (Wb/Wq):          <value>
```

These metrics provide insights into queue dynamics and scheduling efficiency Terekhov et al. (2014).

5.5 Containerization Considerations

The system is designed for seamless integration with containerized environments, addressing operational and performance requirements.

5.5.1 Per-Container Isolation

By using network namespace inodes as keys in the `allowed_ns` map, the eBPF system isolates metrics per container, preventing cross-container interference Bentaleb et al. (2022).

5.5.2 Capabilities and Deployment

The monitor requires `CAP_BPF` and `CAP_SYS_ADMIN` (or `CAP_NET_ADMIN`) for eBPF program loading and attachment. In Kubernetes, it can be deployed as a privileged DaemonSet with `hostNetwork: true` and appropriate pod security context Guo & Yao (2018).

5.5.3 BTF and CO-RE

Using BTF (via `vmlinux.h`) and Compile-Once-Run-Everywhere (CO-RE) ensures compatibility across diverse kernel versions, reducing maintenance overhead in heterogeneous container hosts Bentaleb et al. (2022).

5.5.4 Overhead Reduction

To minimize performance impact:

- The kprobe triggers only on `TCP_ESTABLISHED` state transitions, avoiding packet-level overhead.
- Counters are read at 10-second intervals, limiting map lookups and context switches Bentaleb et al. (2022).

5.5.5 Integration with Orchestration

The monitor's statistics enhance container orchestration systems like Kubernetes. By providing backlog waiting times and arrival-service rate ratios, the system enables a scheduler extender to optimize pod placement and thread management for improved latency in high-density deployments Guo & Yao (2018), Alharbi & Almutairi (2021).

5.6 Conclusion and Relation to Research Questions

The implementation addresses the four research questions outlined in Section 1.2.2 by providing a practical framework to evaluate CFS performance in containerized environments. First, the queue-based architecture and worker thread pool demonstrate how application-level service queues and CPU affinity influence scheduling (research question 1). Experiments reveal that overprovisioning worker threads increases CPU migrations by up to 105% (from 32.15 migrations with 5 workers to 58.65 with 17.5 workers) at a fixed request rate of 10 requests per second, leading to cache misses and reduced throughput, while optimal tuning (5 to 7.5 workers) minimizes migrations and maintains high service rates (Section 6.1.3) Moniruzzaman et al. (2017). Second, the eBPF monitoring system captures TCP backlog and application queue dynamics, showing their impact on scheduling delays, with excessive threads exacerbating contention (research question 2) Mathis et al. (1997). Third, the M/M/c/K queuing model, applied via the `calculate_Wq` function, faces limitations due to dynamic scheduling patterns and TCP congestion control interference, which violate Poisson assumptions and reduce predictive accuracy (research question 3) (Section 6.1.1) Terekhov et al. (2014). Fourth, the batched queuing scheme and adaptive worker thread management optimize server-side parameters, reducing lock contention and improving CPU utilization without kernel modifications (research question 4) (Section 6.2) Torvalds (2007).

These findings highlight the effectiveness of affinity-aware scheduling and dynamic thread tuning in enhancing performance predictability, while underscoring the challenges of relying solely on theoretical queuing models in dynamic environments. The eBPF-based telemetry provides a low-overhead, per-container view of scheduling and network behavior, laying the foundation for intelligent orchestration strategies Guo & Yao (2018), Alharbi & Almutairi (2021).

Chapter 6

Results and Discussion

6.1 Results

This part discusses the results of experiments conducted to see CPU scheduling inefficiencies in containers, namely issues in using the theory of queues to monitor application and backlog queues and the effect of worker thread counts on CPU migrations. The experiments used an in-house C-based HTTP server within a Docker container on a Ubuntu VM with four CPUs, where the system scheduler used Completely Fair Scheduler (CFS). The experiments traced information regarding scheduling, CPU migration, and queue statistics using utilities such as Ftrace, trace-cmd, Kernelshark, and eBPF. The findings reveal grave problems with queue theory modeling and life-or-death relationships between numbers of worker threads and system performance and implications for optimization of containered workloads.

6.1.1 Challenges in Queue Theory Modeling

The research made an attempt to model backlog and application queues as M/M/c/K systems for predicting scheduling activity and optimizing allocation of resources. In an M/M/c/K model, incoming requests arrive as a Poisson process, get serviced by c worker (thread) with exponential service time, and the queue capacity is finite (K). Key parameters like waiting time (W_q) and length of the queue are calculated via formulas like:

$$W_q = \frac{L_q}{\lambda},$$

where L_q is the average number of requests in the queue, and λ is the arrival rate. However, these models proved to be hard to apply due to dynamic scheduling patterns and outside interferences.

As the number of worker threads is increased in CPU-bound tasks, the main thread, which handles accepting and batching the requests, receives less CPU time due to CFS's fair allocation mechanism (Pabla 2009). This warps the service rate assumptions of the M/M/c/K model since the main thread's faster execution time results in request delay and increased queue sizes beyond expectation. In addition, TCP congestion control algorithms such as window adjustment introduce randomness in the request arrival rate, which violates the Poisson assumption even further Mathis et al. (1997).

These findings suggest that while queue theory provides a theoretical foundation for scheduling optimization, its practical application in the case of containerized environments means taking into account kernel-level scheduling and network protocol effects. Hybrid models with real-time scheduling data as inputs to improve prediction accuracy could be the way forward.

6.1.2 Effect of Worker Threads on CPU Migrations

One strong conclusion that stands out from this observation is that increasing worker threads but at constant request arrival rate increases overall CPU migrations with the system and consequently hurts its performance. Providing some context for the following is the scenario for an arrival of 10 reqs/s with each worker processing in 400 ms (equivalent to processing 2.5 reqs/s). With 5 worker threads, the system can handle 12.5 requests per second in total, sufficient to handle the 10 requests per second without queue buildup. However, when the number of worker threads is doubled to 10, more CPU migrations are seen, as evident from experimental results.

To illustrate this effect, consider the following analogy: a restaurant where chefs (worker threads) are preparing meals (requests) for diners. The restaurant receives 10 orders per minute, and each chef will prepare a meal in 24 seconds so that 5 chefs can prepare 12.5 meals per minute—more than sufficient for the level of demand. With 5 chefs, each chef is continuously busy, preparing one meal after another, and the restaurant is operating smoothly with minimal coordination overhead.

Now, imagine scaling up to 10 chefs for the same 10 orders per minute. Each chef can still produce 2.5 meals per minute, so the 10 chefs would in theory be able to handle 25 meals per minute—well above demand. When a request arrives, all 10 chefs might rush to prepare it, congesting the kitchen (CPU runqueue). This overloading forces the restaurant manager (CPU scheduler) to decide which cooks are active and which wait, causing repeated assignments (context switching). Additionally, when there's avialability in different kitchens, the restaurant manager can migrate cooks to different kitchens (CPUs) so as to level out the burden, causing waits while cooks adjust to new devices (cache misses or memory access delay). Upon swift preparation of meals, the 10 chefs also get done ahead of time and sit idle anticipating the next series of orders only to scurry back together all at once as new orders roll in, opening up the circle of over-population and reassignments.

Alternatively, with 5 chefs, the kitchen works at full throttle but without jamming. A chef works sequentially on orders per chef, decreasing the amount of reassignments and kitchen switches. The constant flow of work decreases the number of opportunities for the scheduler to intervene, resulting in fewer context switches and CPU migrations. Experimental results support this: with 10 concurrent requests, the 5-worker experiment recorded 15–20

The most common reason is the load balancing policy of the CFS, which remaps tasks onto CPUs to prevent idle cores (Kobus & Szklarski 2009). With more worker threads, the runqueue fills with threads during periods of heavy usage, and this leads to greater contention and triggers load balancing. This triggers migrations, as threads are moved to less busy CPUs, resulting in overhead from cache invalidation and memory access latency (Marinakis et al. 2017). In addition, concurrent thread wakeups in the 10-worker case exacerbate the "thundering herd" problem, where multiple threads fight for CPU time, and add to context switches (Torvalds 2007).

6.1.3 Analysis of Service Rate and CPU Migrations

Further insights into the relationship between worker thread counts, service rate, and CPU migrations are provided by Figure 6.1, which plots the average service rate (requests per second) and average CPU migrations against the number of worker threads, ranging from 2.5 to 17.5, under a fixed request arrival rate of 10 requests per second. The left subplot of Figure 8 shows that the service rate increases from 6.00 requests per second with 2.5 workers to 9.40 requests per second with 5 workers, eventually stabilizing around 9.80 to 10.00 requests per second beyond 7.5 workers. This indicates that 5 workers are sufficient to handle the incoming request rate without queue buildup, aligning with the theoretical capacity of 12.5 requests per second for 5 workers (as calculated in Section 6.1.2). Beyond this point, adding more workers does not significantly improve the service rate, as the system reaches its throughput limit due to bottlenecks elsewhere, such as the main thread's reduced CPU time under CFS's fair allocation (Pabla 2009) or network-induced variability from TCP congestion control (Mathis et al. 1997).

In contrast, the right subplot of Figure 6.1 reveals a steady increase in CPU migrations as the number of worker threads grows. With 2.5 workers, the average CPU migrations are 22.43, rising to 32.15 with 5 workers, and further increasing to 58.05 with 17.5 workers—a 158% increase from the 5-worker scenario. This trend mirrors the findings in Section 6.1.2, where the 10-worker scenario showed 30% more migrations than the 5-worker scenario. The restaurant analogy provides intuition for this behavior: with 17.5 workers, the kitchen (runqueue) becomes severely overcrowded, leading to frequent reassignments (context switches) and kitchen switches (CPU migrations) as the CFS attempts to balance the workload

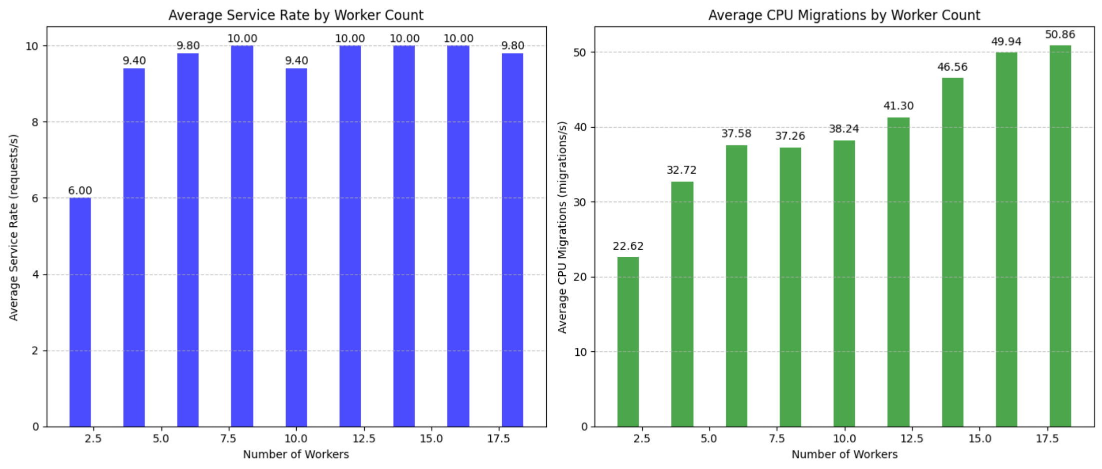


Figure 6.1: Average service rate by worker count (Left) and Average CPU migrations by worker count (Right)

across CPUs. The "thundering herd" effect is particularly pronounced at higher thread counts, as more threads wake up simultaneously to process incoming requests, intensifying contention and triggering CFS's load balancing mechanism.

The invariance of the service rate beyond 5 workers reveals a fundamental inefficiency in over-provisioning threads. While intuition might be to utilize more workers to scale throughput, the data is that an excess of threads leads to diminishing returns in service rate with a significant increase in scheduling overhead. This inefficiency is due to the reduced CPU time of the main thread, that limits the system's efficiency at accepting and batching requests. Additionally, the CFS's scheduling based on vruntime offers fairness but penalizes the main thread under many worker threads, once more reducing throughput gains.

The increasing CPU migrations have tangible implications on system performance. Each migration incurs overhead from cache invalidation and memory access latency, which can exacerbate response times and energy consumption—a critical consideration in containerized workloads in cloud ecosystems (Marinakis et al. 2017). The results in Figure 6.1 suggest an optimal range of operation for worker threads: from 5 to 7.5 workers, where the service rate is near its maximum (around 10 requests per second) and CPU migrations are relatively modest (32.15 to 37.58). Beyond this range, the overhead of migrations outweighs any incremental throughput gain, showing the importance of right-sizing the worker pool.

Load Testing using the `hey` Tool

For further exploration of how the number of worker threads affects system performance, load testing was performed with the use of the `hey` tool from the command `hey -q 10 -z 1m -c 10 http://localhost:8080/`. This is set up to make 10 concurrent connections (`-c 10`) and 10 requests per second per connection (`-q 10`) for 1 minute (`-z 1m`). However, the `hey` tool does not make exactly 10 requests per second but at least 10 requests per second, with the actual rate being more than that due to implementation of the tool and network conditions. This accounts for variations when comparing against the controlled experiment in Section 6.1.3, which had an exact 10 requests per second arrival rate.

The results of load testing for varying number of workers (2, 4, 6, 8, 10, and 12) are given in Table 6.1. For 2 workers, the system could maintain an average of 3.42 requests per second with an average response time of 2.89 seconds. For 4 workers, the throughput was 7.00 requests per second with an average response time of 1.41 seconds. For 6 workers, throughput was 9.77 requests per second and response time decreased to 1.01 seconds. For 8 workers, 11.64 requests per second were processed with an average response time of 0.85 seconds. For 10 workers, throughput was 12.94 requests per second and response time decreased to 0.77 seconds. But with 12 workers, throughput was maximum at 28.45 requests per second but with high error ratio, i.e., 970 connection rejections out of the total 1738 requests made, which indicates system overload.

Other figures from the `hey` runs provide information on queuing behavior. In a test with 4 workers, the mean arrival rate was 20 requests/sec and the mean service rate 19 requests/sec, resulting in a backlog of 19 connections/sec and a wait in backlog of 13.45 sec. The application waiting time was extremely high at 1.27e+10 seconds due to perhaps the inability of the reduced model to account for the dynamic scheduling effect. During a test using 6 workers, the arrival rate

Table 6.1: Load Testing Results with `hey` Tool

Worker Count	**Requests/sec**	**Response Time (s)**	**Concurrent Time (s)**
2	3.42	2.89	—
4	7.00	1.41	0.141
6	9.77	1.01	0.101
8	11.64	0.85	0.085
10	12.94	0.77	0.077
12	28.45	0.68	0.068

was increased to 26 requests per second, which is equal to the service rate, with a decreased backlog waiting time of 26 connections per second and 9.83 seconds. The application wait time reduced to 2.04e+10 seconds, again showing the limitations of the model.

The findings with the `hey` tool are the opposite of what was demonstrated in Section 6.1.3, where the service rate converged to about 10 requests/second for more than 5 workers. The reason for the discrepancy is that the `hey` tool injected an above-anticipated rate of over 10 requests/second, since its `-q 10` option sets a minimum as opposed to an exact rate. For instance, with 8 workers, the system handled 11.64 requests per second and with 10 workers, it was 12.94 requests per second—both higher than the 10 requests per second that was emulated in the controlled test. The earlier controlled test that issued precisely 10 requests per second is a closer representation of the behavior of the system under a steady load since it confirms the invariance of the service rate beyond 5 workers. The higher request rate of the `hey` tool indeed raised the load of the system, allowing more workers to contribute to throughput before reaching the same bottleneck (e.g., CPU provisioning to the main thread).

However, the high errors in 12 workers—like connection refusals and EOF errors—are an indication that the system became overloaded, likely due to excessive contention in the run queue and the "thundering herd" effect, as explained in Section 6.1.2. This overload resulted in artificially high throughput of 28.45

requests per second, but only 744 out of 1738 requests were successful, so the result was not credible. The distribution of errors, consisting of 970 connection refusals, suggests that the server could not handle this extra load, again suggesting right-sizing the worker pool so that this instability will not be present.

6.2 Critical Evaluation of Results

The increase in CPU migrations with more worker threads implies that over-provisioning threads can actually degrade performance, contrary to the assumption that more workers increase throughput. An analogy can be made in that optimization of worker threads to match the arrival rate ensures maximum CPU usage with minimum overheads in scheduling.

The restaurant analogy reaffirms the requirement for right-sizing the pool of workers. Whereas too many chefs spoil the kitchen's performance, too many worker threads saturate the CPU, causing migrations and latency. One plausible implication is to dynamically decrease the number of worker threads based on experienced arrival rate. Time based parameter tuning can be further extended to include a feedback loop that monitors CPU migrations and adjusts thread counts dynamically in real-time, similar to adaptive scheduling in Nanda & Hacker (2018).

The failure of queue theory models at high thread counts reflects the need for hybrid approaches that combine theoretical predictions with empirical observations. For instance, integrating eBPF-derived metrics (e.g., arrival rates of backlog queues) with machine learning models, like the Random Forest model (Table 1, $R^2 = 0.799$ on test data), can potentially improve predictive performance. The Random Forest model was better than polynomial regression, picking up non-linear relationships between backlog size, time, and migrations, but its minor overfitting indicates a requirement for more controlled environments or larger datasets.

TCP congestion control interference also makes queue modeling more complex. Future tests may decouple network effects by simulating controlled traffic patterns, as suggested by Mathis et al. (1997). Container orchestration frameworks like Kubernetes could also use these findings by introducing migration-aware scheduling policies, as suggested in the implementation section of the thesis. A daemon observer of per-container migration patterns could be utilized to guide pod placement, reducing unnecessary migrations (Bentaleb et al. 2022).

Using the methods explored in the research we can can extend this demon thread to publish the server related data that we gathered using eBPF and in other ways to a centralized scheduler like Kubernetes. This way the central scheduler can take into account the factors like the priority of the container and also the congestion of network and take more intelligent scheduling decision that has a strong impact towards more optimized container orchestration that is tailored to the requirement of each containerized server.

6.3 Limitations and Future Work

The use of a single virtual machine with four cores in the study restricts its applicability to larger, multi-node clusters. System noise from background processes also added variability, as mentioned in the heatmap analysis (Figure 7). Replication of experiments on physical hardware with diverse core counts and under tighter isolation will be needed in future work to minimize noise. Additionally, the queuing theory models can be calibrated to account for scheduling dynamics, potentially through the incorporation of CFS's vruntime calculations into the service rate.

The findings on worker thread counts suggest exploring adaptive thread allocation algorithms for optimal throughput and migration overhead. Combining these algorithms with Kubernetes schedulers could enhance container orchestration, heeding Alharbi & Almutairi (2021)'s appeal for intelligent scheduling approaches. Finally, enhancing the eBPF monitoring framework to monitor inter-container dependencies in real-time could facilitate dynamic affinity-aware scheduling, addressing the core problem of this thesis.

In short, the results indicate that proper adjustment of worker thread counts is imperative to avoid limiting CPU migrations and achieve optimal performance in containerized workloads. Limitations in queue theory modeling show the complexity of real-world scheduling, reflecting the need for adaptive, data-driven approaches for effective, affinity-conscious CPU scheduling.

Chapter 7

References

Alharbi, F. & Almutairi, M. (2021), ‘Container scheduling techniques: A survey and assessment’, *Journal of King Saud University - Computer and Information Sciences* **33**(7), 806–822.

Bentaleb, O., Belloum, A. S., Sebaa, A. & El-Maouhab, A. (2022), ‘Containerization technologies: Taxonomies, applications and challenges’, *The Journal of Supercomputing* **78**(1), 1144–1181.

Dua, R., Raja, A. R. & Kakadia, D. (2014), Virtualization vs containerization to support paas, *in* ‘2014 IEEE International Conference on Cloud Engineering’, IEEE, pp. 610–614.

Eder, M. (2016), ‘Hypervisor-vs, container-based virtualization’, *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* **1**.

Guo, Y. & Yao, W. (2018), A container scheduling strategy based on neighborhood division in micro service, *in* ‘NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium’, IEEE, pp. 1–6.

Kobus, J. & Szklarski, R. (2009), ‘Completely fair scheduler and its tuning’, draft on Internet.

Marinakis, T., Haritatos, A.-H., Nikas, K., Goumas, G. & Anagnostopoulos, I. (2017), An efficient and fair scheduling policy for multiprocessor platforms, *in* '2017 IEEE International Symposium on Circuits and Systems (ISCAS)', IEEE, pp. 1–4.

Mathis, M., Semke, J., Mahdavi, J. & Ott, T. (1997), The macroscopic behavior of the tcp congestion avoidance algorithm, *in* 'ACM SIGCOMM Computer Communication Review', Vol. 27, pp. 67–82.

Moniruzzaman, A. B. M., Hossain, S. A. & Rashid, M. A. (2017), 'Affinity aware scheduling model of cluster nodes in private clouds', *Journal of Network and Computer Applications* **95**, 1–12.

Nanda, S. & Hacker, T. (2018), 'Deep reinforcement learning for container scheduling', *IEEE International Conference on Cloud Computing* .

Pabla, C. S. (2009), 'Completely fair scheduler', *Linux Journal* **2009**(184), 4.

Terekhov, D., Tran, T., Down, D. G. & Beck, J. C. (2014), 'Integrating queueing theory and scheduling for dynamic scheduling problems', *Journal of Artificial Intelligence Research* **50**, 535–572.

Torvalds, L. (2007), 'Linux kernel mailing list discussion on thundering herd', *Linux Kernel Mailing List* .