國立中正大學

資訊工程研究所

碩士論文

基於 GRU 的 Linux 排程器性能最佳化研究

Performance Optimization of a Linux Scheduler Based on GRU

研究生：吳承祐

指導教授：柯仁松教授

中華民國 114 年 6 月

摘要

本研究探討了利用機器學習方法強化 Linux 核心的 CPU 排程策略，提出一種基於 Gated Recurrent Unit（GRU）的動態排程架構，結合延伸柏克萊封包篩選器（eBPF）進行即時任務監控，並透過 SCHED_EXT 介面整合至核心排程器中。我們設計的 GRU_sched 模型將排程視為一項多類別分類問題，非傳統形式的最佳化問題，藉由模擬 CFS（Completely Fair Scheduler）的實際行為進行學習。模型輸入為七維特徵向量，包括靜態任務屬性與動態執行歷史，並以改良的 GRU 結構實現時間感知特徵建模，確保所有輸入特徵皆被有效保留。透過正則化限制 GRU 權重矩陣之欄向量範數，我們避免了傳統神經網路中可能發生的特徵捨棄問題，從而提升了模型對於任務切換（context switch）相關特徵的敏感度。模型的訓練採用類別交叉熵損失函數（categorical cross-entropy loss），並搭配 early stopping 機制以避免過擬合。

在與 Linux 預設排程器進行比較時，GRU_sched 展現出顯著優勢。在 CPU 密集型場景中，其中位數吞吐量達到 4288 事件/秒，與預設排程器相近，但在任務平均執行時間上下降了 15.5%，並改善了尾端延遲 25.8%。在單執行緒高負載情境下，其最大吞吐量為 4517 事件/秒，接近 CFS 的 4528，顯示即使在計算密集條件下，GRU_sched 仍具備穩定表現。

於混合型工作負載環境中，GRU_sched 達到 16,453 事件/秒的高吞吐表現，相較於預設排程器的 16,429 事件/秒略有提升，並成功維持延遲中位數在 0.49 毫秒，表現出低波動且穩定的反應能力。雖最大延遲尖峰可達 209.34 毫秒，略高於 CFS 的 114.85 毫秒，但整體而言 GRU_sched 展現了在動態條件下更強的自我調整能力。

在 I/O 密集型測試中，GRU_sched 明顯降低了上下文切換次數，平均切換次數低於 130，遠少於預設排程器的 290 次，成功將總運行時間縮短 16.7%，同時維持穩定的延遲（0.23–0.24 毫秒）。此外，它有效降低如 snapd、kworker、khungtaskd 等關鍵系統任務的等待時間，進一步提升整體系統的回應速度與效能表現。

綜合來看，GRU_sched 在不同類型工作負載下皆展現優越性能。其學習架構不僅捕捉了傳統排程器的行為邏輯，亦具備更高的適應性與未來延展性。透過保留所有輸入訊號的設計與時間序列特徵的建模能力，GRU_sched 不僅提升了吞吐與延遲表現，更展現其作為可泛化排程策略的潛力，適用於多核、多負載、需即時反應的高效能計算環境。

關鍵詞：閘門循環單元（GRU）、Linux 排程器、eBPF、SCHED_EXT、上下文切換、機器學習排程策略

Abstract

This study explores the use of machine learning to enhance CPU scheduling in the Linux kernel. We propose a dynamic scheduling framework based on a Gated Recurrent Unit (GRU) model combined with real-time monitoring via eBPF, integrated into the kernel using the `SCHED_EXT` interface. Our scheduler, `GRU_sched`, frames CPU assignment as a multi-class classification task—rather than a traditional optimization problem—by learning from the behavioral patterns of the default Completely Fair Scheduler (CFS). Each task is represented as a seven-dimensional feature vector capturing both static attributes and dynamic runtime history. To ensure that all input signals retain influence, the GRU is structurally constrained such that no feature's weight is allowed to decay to zero. This design choice improves sensitivity to features related to context switching and temporal scheduling dynamics. The model is trained using a categorical cross-entropy loss and optimized with early stopping to prevent overfitting.

Experimental results demonstrate that `GRU_sched` outperforms the default scheduler across CPU-bound, hybrid, and I/O-intensive workloads. In CPU-intensive scenarios, it shows a modest improvement in throughput and a notable reduction in tail latency. For example, in single-threaded benchmarks, `GRU_sched` achieved a median throughput of 4,288 events/s and a maximum of 4,517 events/s, compared to the default scheduler's 4,287 and 4,528 events/s, respectively. Although maximum latency occasionally spiked to 65 ms, `GRU_sched` reduced average task runtime by 15.5% and improved tail latency by 25.8%, indicating better efficiency under heavy load.

Under hybrid workloads, `GRU_sched` maintained a slightly higher throughput of 16,453 events/s versus 16,429 events/s for the default scheduler, while achieving a stable median latency of 0.49 ms. This reflects its robustness in handling mixed task profiles and maintaining consistent responsiveness. Although the worst-case latency reached 209.34 ms—higher than the default scheduler's 114.85 ms—`GRU_sched` demonstrated superior adaptability under variable system pressures, balancing throughput and reactivity.

In I/O-intensive workloads, `GRU_sched` significantly reduced context switch overhead, with typical context switch counts under 130, compared to over 290 for the default scheduler. This translated into a 16.7% reduction in total runtime and a 19.9% decrease in task count, while maintaining stable throughput and low average latency (0.23 – 0.24 ms). It also improved response times for critical system services like `snapd`, `kworker`, and `khungtaskd`, enhancing overall system reactivity.

In summary, `GRU_sched` demonstrates clear advantages across diverse workload types. Its architecture preserves all feature signals and models temporal dependencies effectively, enabling it to generalize across CPU-bound, I/O-heavy, and hybrid environments. While occasional latency spikes occur, the overall performance gains make it a strong candidate for high-demand scheduling scenarios. By reducing task runtime, improving latency handling, and enhancing CPU efficiency, this GRU-based scheduler proves to be a robust and adaptable alternative to traditional Linux schedulers.

Keywords: Gated Recurrent Unit (GRU), Linux Scheduler, SCHED_EXT, eBPF, Context

Switching, Machine Learning for Scheduling.

List of Tables

List of Tables

List of Figures

List of Figures

Contents

Chapter 1 Introduction

CPU scheduling is a fundamental component of modern operating systems, ensuring efficient allocation of CPU resources among multiple tasks. This chapter introduces the motivation behind developing a dynamic CPU scheduler and highlights the benefits of integrating machine learning techniques to predict and optimize scheduling decisions.

1.1 Overview

As modern computing workloads become increasingly diverse, traditional CPU schedulers such as CFS (Completely Fair Scheduler) struggle to efficiently handle mixed workloads, including compute-intensive and I/O-intensive processes. This often leads to imbalanced resource allocation, excessive context switches, and high CPU migration overhead, ultimately affecting system performance.

This study proposes an intelligent CPU scheduler based on the sched_ext scheduling framework, integrating the EEVDF (Earliest Eligible Virtual Deadline First) scheduling strategy with a GRU (Gated Recurrent Unit) prediction model to address these challenges. By utilizing eBPF to monitor CPU usage and leveraging the GRU model to predict optimal CPU affinity, the system dynamically adjusts resource allocation based on process demands. This approach enhances system efficiency and process fairness while reducing unnecessary CPU migrations and context switches.

1.2 Background and Motivation

Traditional CPU schedulers in Linux primarily rely on static heuristics, making decisions reactively rather than proactively. While effective under predictable workloads, contemporary systems exhibit significant variability in CPU demand, necessitating adaptive scheduling strategies.

The Completely Fair Scheduler (CFS), introduced in Linux 2.6.23, allocates CPU time based on virtual runtime (vruntime), yet lacks cache affinity awareness and predictive capabilities, rendering it suboptimal for latency-sensitive applications. Although EEVDF, introduced in Linux 6.6, enhances response times through virtual deadlines (vdeadline), it remains fundamentally rule-based, limiting its adaptability.

Both schedulers encounter difficulties in anticipating workload fluctuations, resulting in inefficient CPU allocation and frequent, unnecessary task migrations. Traditional load-balancing mechanisms are similarly reactive and often fail to dynamically optimize CPU utilization.

To address these limitations, Linux 6.12 introduces `sched_ext`, which enables the implementation of custom scheduling policies via eBPF. By facilitating dynamic scheduling logic

without requiring modifications to the core kernel code, `sched_ext` enhances flexibility and supports machine learning-driven scheduling approaches, such as predictive models based on Gated Recurrent Units (GRU), thereby improving CPU efficiency.

1.3 Research Objectives

This research explores the limitations of existing Linux scheduling policies and proposes a more adaptive, machine learning-driven scheduling framework. The primary objective is to design and implement GRU_sched, a scheduler that integrates Gated Recurrent Units (GRU) with sched_ext, enabling predictive workload management. By analyzing real-time system telemetry through eBPF, GRU_sched dynamically optimizes CPU allocation based on workload trends and task execution history.

To achieve accurate scheduling decisions, this research focuses on feature selection, identifying key system metrics that influence scheduling performance. Relevant features include CPU utilization patterns, task runtime history, cache usage, memory bandwidth, and context switch rates. These metrics serve as inputs to the GRU model, allowing it to learn workload behavior and make informed scheduling adjustments.

In evaluating existing schedulers, this research examines the fairness, CPU cycle efficiency, and task migration behavior of CFS and EEVDF. While CFS ensures long-term fairness, it lacks predictive capabilities, leading to inefficiencies under dynamic workloads. EEVDF improves response time predictability but remains rule-based, limiting its adaptability. GRU_sched aims to overcome these limitations by incorporating ML-driven decision-making, ensuring that CPU cycles are allocated optimally while maintaining low overhead and system stability.

Through a comparative analysis of GRU_sched, CFS, and EEVDF, this study investigates the effectiveness of ML-driven scheduling in reducing task migration overhead, improving cache locality, and enhancing system responsiveness. By leveraging machine learning for feature-driven scheduling optimization, GRU_sched provides a flexible and intelligent approach to CPU resource management in modern computing environments.

1.4 Organization of this Thesis

This thesis is structured into seven chapters.

In Chapter 2, we introduce CFS and EEVDF, analyzing their scheduling strategies and limitations. We also examine sched_ext, a Linux scheduling extension that enables custom scheduling policies via eBPF. Additionally, we review prior research on machine learning-driven scheduling.

In Chapter 3, we discuss the design of GRU_sched, covering feature selection, eBPF telemetry integration, and real-time inference execution.

In Chapter 4, we describe the implementation of GRU_sched, including GRU model training, inference execution, and scheduling optimizations.

In Chapter 5, we evaluate GRU_sched against the default scheduler, focusing on CPU utilization, scheduling fairness, and task migration trends.

In Chapter 6, we conclude by summarizing key findings and contributions.

In Chapter 7, we explore future research directions, including hybrid ML-based scheduling, reinforcement learning, and low-latency inference techniques.

In Chapter 8, we show reference.

Chapter 2 Background and Related Work

Modern CPU scheduling plays a crucial role in optimizing resource allocation and improving system performance. Traditional scheduling algorithms, such as the Completely Fair Scheduler (CFS) and Earliest Eligible Virtual Deadline First (EEVDF), aim to balance fairness and responsiveness but often struggle with dynamic workloads. With the growing complexity of modern computing environments, researchers have explored alternative approaches, including machine learning techniques, to enhance scheduling efficiency.

Additionally, the advent of extended Berkeley Packet Filter (eBPF) has provided a powerful mechanism for real-time system monitoring, enabling more adaptive and intelligent scheduling strategies. The recent introduction of sched_ext in the Linux kernel further expands the possibilities for custom scheduling policies, allowing developers to tailor scheduling mechanisms to specific workloads.

This chapter reviews existing CPU scheduling algorithms, the application of machine learning in scheduling, the role of eBPF in system monitoring, and the extensibility offered by sched_ext.

2.1 Existing CPU Schedulers

We review popular CPU scheduling algorithms such as Completely Fair Scheduler (CFS) and Earliest Eligible Virtual Deadline First (EEVDF). The Completely Fair Scheduler (CFS) has been the default CPU scheduler in the Linux kernel since version 2.6.23. Its primary goal is to ensure that every process receives CPU time in proportion to its assigned weight, thereby maintaining long-term fairness. Unlike traditional schedulers that rely on fixed time slices, CFS adopts a continuous fairness approach, where scheduling decisions are based on a process's execution history. To achieve this, CFS employs virtual runtime (vruntime) as a fundamental scheduling metric, which determines the order of process execution. Instead of using a conventional runqueue, CFS organizes tasks in a Red-Black Tree (RBTREE), allowing efficient process insertion, deletion, and selection. This self-balancing binary search tree ensures that scheduling decisions can be made in O(log N) time complexity, providing a balance between fairness and efficiency.

2.1.1 Scheduling Mechanism of CFS

The Completely Fair Scheduler (CFS) [1] has been the default CPU scheduler in the Linux kernel since version 2.6.23. Its primary goal is to ensure that every process receives CPU time in proportion to its assigned weight, thereby maintaining long-term fairness. Unlike traditional

schedulers that rely on fixed time slices, CFS adopts a continuous fairness approach, where scheduling decisions are based on a process's execution history.

The core scheduling mechanism of CFS relies on the accumulation of vruntime, which increases as a process executes. The rate at which vruntime grows depends on the process's weight. A higher-weighted process accumulates vruntime at a slower rate, allowing it to remain on the CPU for a longer duration before yielding. Conversely, lower-weighted processes accumulate vruntime faster, reaching the scheduling threshold sooner. The change in vruntime follows:

$$\Delta v = \frac{\Delta t \times \text{load\_weight}_{\text{base}}}{\text{load\_weight}_{\text{current}}} \tag{2.1}$$

where:

- Δt represents the actual execution time,
- $\text{load\_weight}_{\text{base}}$ is the standard weight for a `nice` value of 0,
- $\text{load\_weight}_{\text{current}}$ refers to the current process's weight.

This formula ensures CPU time is allocated proportionally to process weight, preventing starvation.

2.1.2 Per-Entity Load Tracking (PELT) in CFS

CFS incorporates Per-Entity Load Tracking (PELT), which estimates CPU demand based on historical usage patterns. Unlike traditional load-tracking mechanisms, PELT continuously updates process load values using an exponential decay function:

$$\text{Load}_t = \text{Load}_{t-1} \times e^{-\frac{\Delta t}{\tau}} + \text{CurrentLoad} \times (1 - e^{-\frac{\Delta t}{\tau}}) \tag{2.2}$$

where τ is a time constant that defines the influence of past CPU activity on the current load estimation. By applying an exponential decay model, PELT ensures that more recent CPU usage data has a greater impact on scheduling decisions, while older data gradually loses significance. This design allows CFS to adapt to dynamic workload variations efficiently.

The primary advantage of PELT is its ability to balance CPU utilization across processes while reducing unnecessary task migrations. Frequent migrations between CPU cores can lead to performance degradation due to cache misses and increased context-switching overhead. PELT mitigates this by enabling the scheduler to make more informed decisions based on long-term CPU usage trends rather than reacting to short-term spikes in load.

Another key feature of PELT is its role in predicting future CPU demand. By maintaining a history of CPU consumption, the scheduler can estimate whether a task's load is increasing or decreasing, allowing it to allocate CPU time more effectively. This predictive capability is particularly beneficial in multi-core systems, where maintaining cache affinity and reducing migration overhead are crucial for system performance.

Despite its benefits, PELT has some limitations. Since it primarily relies on historical data, it may not always respond optimally to sudden workload changes. Additionally, while the decay function smooths fluctuations in CPU demand, it may introduce a slight lag in adjusting to rapid shifts in process behavior. To further enhance scheduling accuracy, researchers have explored integrating PELT with machine learning-based approaches that dynamically adjust decay parameters based on workload characteristics.

Overall, PELT plays a vital role in modern Linux scheduling by enhancing CPU load tracking and improving efficiency in multi-core environments. Its ability to adapt dynamically to workload variations makes it a critical component of the Completely Fair Scheduler (CFS), ensuring fair and efficient CPU time allocation across different processes.

2.1.3 Advantages and Limitations of CFS

CFS ensures long-term fairness by distributing CPU cycles proportionally among processes. Its scheduling mechanism, based on a Red-Black Tree, maintains an efficient O(log N) complexity, allowing for quick insertion, deletion, and selection of processes. Additionally, it improves cache affinity by minimizing unnecessary CPU migrations, ensuring that tasks remain on the same core whenever possible to reduce performance overhead.

Despite these advantages, CFS is not without its limitations. It lacks real-time guarantees, making it unsuitable for latency-sensitive workloads that require strict timing constraints. Moreover, its fixed fairness approach, while effective in maintaining proportional CPU allocation, does not always prioritize interactive tasks efficiently. This can lead to higher response times for applications that require immediate processing, such as graphical user interfaces or real-time communication systems.

2.1.4 Fundamentals of EEVDF Scheduling

In modern operating systems, efficient CPU time allocation is a key factor in determining system performance and fairness. Traditional scheduling mechanisms, such as the Completely Fair Scheduler (CFS), ensure long-term fairness through vruntime. However, CFS does not take into account real-time constraints and deadlines. To address these issues, Linux introduced the Earliest Eligible Virtual Deadline First (EEVDF) scheduler [2] [3] in version 6.6, providing a balance between fairness and adaptability to different workload types.

The core concept of EEVDF is the virtual deadline, which assigns each process a ”virtual earliest eligible deadline” used as the basis for scheduling decisions. Unlike CFS, which uses vruntime to order processes, EEVDF maintains a virtual deadline v_{deadline} for each process and always selects the process with the earliest deadline for execution. The calculation formula is as follows:

$$v_{\text{deadline}} = v_{\text{runtime}} + \frac{\text{weight}}{\text{normalized priority}} \tag{2.3}$$

where:

v_{runtime} represents the accumulated virtual runtime of the process.

weight is the process's assigned weight.

normalized priority is the normalized priority of the process.

As the selected process executes, it accumulates virtual runtime v_{runtime}, while its virtual deadline v_{deadline} is continuously updated to reflect execution progress.If a newly arriving process possesses an earlier virtual deadline than the currently executing one, a preemption event occurs, causing the scheduler to switch execution to the new process. This ensures that tasks with higher urgency are given priority in execution. Upon process completion or suspension, EEVDF dynamically recalculates the virtual deadlines based on system load, maintaining proportional CPU allocation according to each process's assigned weight.It ensures that processes with higher weights (lower nice values) have their virtual deadlines postponed, while lower-weighted processes reach their deadlines sooner, influencing execution order accordingly.

The design of EEVDF is inspired by Stoica et al.'s 1997 paper, *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. This study explored a more flexible proportional fairness allocation method, allowing processes to be scheduled based on virtual deadlines to achieve an optimal balance between real-time responsiveness and fairness.

2.1.5 Fairness Proof of EEVDF

The Earliest Eligible Virtual Deadline First (EEVDF) scheduling strategy ensures that processes receive CPU time proportionally to their assigned weights. To establish the fairness of EEVDF in the long run, we analyze its adherence to proportional fairness, ensuring that the CPU time allocated to each process is proportional to its weight.

Long-Term Fairness Analysis Consider a system with N processes, where each process i has a weight w_i. Let $C_i(T)$ denote the total CPU time received by process i in the time interval $[0, T]$. Ideally, the proportional fairness condition is:

$$\frac{C_i(T)}{C_j(T)} = \frac{w_i}{w_j}, \quad \forall i, j \tag{2.4}$$

which states that the CPU time allocated to each process should be proportional to its weight.

In EEVDF, each process's virtual deadline $v_{\text{deadline},i}$ is dynamically adjusted based on its weight to ensure proportional fairness in scheduling order. For any two processes i and j, if $w_i > w_j$, then process i will receive more CPU time, compensating for its higher weight. This mechanism ensures that, over time, the CPU time allocated to each process converges to a proportion that reflects its weight.

Short-Term Fairness Analysis In short-term scheduling, processes may be preempted, and the scheduler must ensure that CPU time distribution remains close to proportional fairness. We define:

$$\Delta C_i = C_i(T + \Delta T) - C_i(T) \tag{2.5}$$

which represents the additional CPU time received by process i over a short interval ΔT. According to EEVDF's design, as $\Delta T \to 0$, the following condition holds:

$$\frac{\Delta C_i}{\Delta C_j} \approx \frac{w_i}{w_j}, \quad \forall i, j \tag{2.6}$$

indicating that even over short intervals, the CPU allocation remains close to ideal proportional fairness.

Additionally, EEVDF allows dynamic adjustments of virtual deadlines to correct deviations in short-term fairness, ensuring that scheduling decisions do not significantly deviate from fairness principles.

Fairness in Multi-Core Systems In multi-core CPU environments, uneven workload distribution can cause CPU time imbalances. To mitigate this, EEVDF employs a virtual deadline synchronization mechanism, ensuring that the deviation in virtual deadlines across different cores remains within a bounded threshold ϵ:

$$|v_{\text{deadline},i} - v_{\text{deadline},j}| \leq \epsilon, \quad \forall i, j \tag{2.7}$$

This mechanism ensures that process execution order across different CPU cores follows proportional fairness principles, reducing performance bottlenecks caused by uneven workload distribution.

During load balancing, EEVDF utilizes virtual deadlines to determine whether a process should be migrated. This mechanism maintains fairness across cores while minimizing unnecessary process migrations, thereby preserving cache affinity.

Fairness Improvements Over CFS Although the Completely Fair Scheduler (CFS) also follows proportional fairness principles, its reliance on v_{runtime} can cause low-priority processes to experience prolonged execution delays. EEVDF improves upon CFS by introducing virtual deadlines, ensuring that even low-priority processes receive a minimum share of CPU time in the short term, thereby mitigating starvation.

EEVDF enhances fairness over CFS through the following mechanisms:

- **Real-time guarantees**: High-priority processes do not excessively dominate CPU time, allowing lower-priority processes to continue executing.
- **Load balancing**: The virtual deadline synchronization mechanism ensures fair CPU allocation across multi-core systems.

- **Adaptability**: EEVDF dynamically adjusts CPU allocation based on system load, whereas CFS primarily relies on static v_{runtime}, limiting its flexibility.

EEVDF mathematically guarantees both long-term and short-term proportional fairness. Additionally, its virtual deadline synchronization mechanism ensures fairness in multi-core environments. Compared to traditional CFS scheduling, EEVDF provides a more adaptive and balanced scheduling mechanism by maintaining a strong balance between fairness and real-time responsiveness, making it well-suited for high-performance computing, real-time applications, and interactive systems.

2.1.6 Comparison Between CFS and EEVDF

Completely Fair Scheduler (CFS) and Earliest Eligible Virtual Deadline First (EEVDF) are two different CPU scheduling algorithms. CFS emphasizes long-term fairness, whereas EEVDF provides better short-term fairness and low-latency characteristics. The following is a detailed comparison of these two scheduling mechanisms.

Table 2.1: Comparison between CFS and EEVDF

Comparison Aspect	**CFS (Completely Fair Scheduler)**	**EEVDF (Earliest Eligible Virtual Deadline First)**
Scheduling Mechanism	Selects the process with the smallest $vruntime$	Prioritizes the process with the earliest virtual deadline
Fairness	Ensures long-term fairness but may delay short-term tasks	Ensures short-term fairness, preventing starvation
Latency Control	No strict latency control, may lead to response time fluctuations	Ensures low latency and minimizes response time jitter
Data Structure	Red-Black Tree, $O(logN)$ complexity	FIFO queue, $O(1)$ complexity
Computation Overhead	Higher due to tree balancing operations	Lower, as scheduling is based on simple queue management
Multi-core Performance	Requires additional load balancing; may have NUMA overhead	Easier task distribution, better suited for NUMA architectures
Impact on Low-Priority Tasks	Low-priority tasks may experience starvation under high load	Guarantees execution within a reasonable time frame
Application Scenarios	General computing, multitasking environments	Real-time applications, gaming, multimedia processing

Detailed Comparison

(1) Scheduling Mechanism CFS selects the process with the smallest virtual runtime ($vruntime$), ensuring that all tasks receive CPU time in proportion to their weights. However, this method does not prioritize short-lived processes, making it less suitable for latency-sensitive

workloads. The use of a Red-Black Tree allows efficient tracking of process execution time, but it also increases computational complexity. Since processes are scheduled strictly based on $vruntime$, short-lived tasks may suffer from delays if long-running tasks dominate CPU time allocation.

EEVDF schedules tasks based on the earliest virtual deadline, allowing processes to execute in a more predictable manner. This is beneficial for applications requiring timely responses. Instead of relying on $vruntime$, EEVDF assigns a virtual deadline to each task and ensures that the task with the closest deadline executes first. This approach minimizes response time jitter and improves task predictability, making it well-suited for latency-sensitive workloads such as video streaming, gaming, and interactive applications.

(2) Fairness CFS ensures fairness over long periods but may cause short-term unfairness, delaying interactive processes. The fairness model is derived from a proportional share scheduling principle, meaning that each task receives CPU time proportional to its weight over an extended execution period. However, in scenarios where interactive or real-time tasks need immediate execution, CFS may fail to meet fairness expectations within short time frames. This results in temporary execution delays for latency-sensitive tasks.

EEVDF focuses on fairness in short intervals, making it more responsive to real-time demands and preventing process starvation. By dynamically adjusting task execution order based on deadlines, EEVDF guarantees that tasks with urgent execution requirements are not left waiting behind long-running tasks. This ensures fairness at a micro-level, benefiting real-time and interactive applications where responsiveness is a critical factor.

(3) Latency Control CFS does not provide strict latency guarantees, which can result in fluctuating response times, especially for time-sensitive applications. The lack of deadline-based scheduling means that response times are determined by the relative execution order in the Red-Black Tree, which may not align with application responsiveness requirements.

EEVDF ensures predictable execution times by assigning deadlines, significantly reducing jitter and improving responsiveness. Since the scheduling order is determined based on deadlines, applications with tight timing constraints can be reliably scheduled without excessive waiting times. This is particularly useful for multimedia applications, real-time systems, and interactive workloads that require deterministic execution behavior.

(4) Data Structure and Scheduling Complexity CFS relies on a Red-Black Tree with $O(logN)$ complexity, which increases overhead when managing large numbers of tasks. The Red-Black Tree allows efficient insertion, deletion, and selection of the next process to execute, but the balancing operations add computational overhead.

EEVDF uses a FIFO queue with $O(1)$ complexity, simplifying scheduling and improving efficiency in high-frequency task switching. Since scheduling decisions are based purely on deadlines, EEVDF avoids the complexity associated with tree-based data structures, leading to reduced computational costs in dynamic scheduling environments.

(5) Computation Overhead CFS requires periodic balancing of the Red-Black Tree, adding computational costs. As the number of active tasks increases, the overhead of maintaining a balanced tree structure grows proportionally, impacting CPU scheduling efficiency.

EEVDF has lower overhead as it only manages tasks based on deadlines, making it more efficient for real-time applications. With a straightforward scheduling mechanism, EEVDF minimizes the impact of scheduling operations on CPU performance.

(6) Multi-core and NUMA Architecture Impact CFS needs additional mechanisms for load balancing, and in NUMA systems, excessive migrations may introduce performance penalties. Ensuring fairness across multiple cores requires careful coordination, which can lead to inefficiencies when migrating tasks between different NUMA nodes.

EEVDF's scheduling method naturally distributes tasks more efficiently across cores, making it more suited for NUMA environments. By focusing on deadlines rather than global execution history, EEVDF achieves more balanced task scheduling across multi-core processors.

(7) Impact on Low-Priority Tasks In high-load scenarios, CFS may starve lower-priority tasks due to continuous execution of higher-priority tasks. This is because scheduling decisions are based on accumulated $vruntime$, making it difficult for low-priority tasks to gain execution time.

EEVDF ensures that all processes get executed within a reasonable timeframe, reducing the risk of starvation. By incorporating deadlines into the scheduling decision, even low-priority tasks are guaranteed CPU time when their deadlines approach.

(8) Application Scenarios CFS is well-suited for general computing environments where long-term fairness is essential. It is widely used in desktop and server environments where task execution fairness is important but not necessarily time-critical.

EEVDF is more appropriate for real-time applications like gaming, video conferencing, and multimedia processing. Its focus on deadline-based scheduling makes it ideal for applications requiring consistent low-latency execution.

Conclusion

CFS provides fairness over long durations, making it ideal for general-purpose scheduling but less effective for latency-sensitive workloads. EEVDF, with its emphasis on short-term fairness and low latency, is better suited for real-time applications. As real-time processing becomes increasingly crucial, EEVDF may gain more adoption in future operating systems.

2.2 Introduction to GRU

The Gated Recurrent Unit (GRU) [4] is an improved variant of the traditional RNN, designed to mitigate the vanishing gradient problem through gating mechanisms. It employs an

update gate and a reset gate to control the hidden state, allowing it to balance the retention of past information with the incorporation of new input. With its simplified architecture and fewer parameters compared to LSTM, GRU offers faster training and inference, making it highly effective in capturing long-term dependencies. Its capabilities have been successfully applied to time-series prediction, natural language processing, and dynamic resource allocation, enabling further progress.

2.2.1 Theoretical Background of Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) is an improved variant of the Recurrent Neural Network (RNN) that incorporates gating mechanisms to dynamically regulate the flow of information. Unlike traditional RNNs, GRUs introduce two key components: the reset gate and the update gate, which help mitigate the vanishing gradient problem and enhance the model's ability to capture long-term dependencies.

Mathematical Formulation

Given an input vector x_t at time step t and the hidden state from the previous time step h_{t-1}, the GRU updates its state through the following steps:

Reset Gate

The reset gate r_t determines how much of the previous hidden state h_{t-1} should be retained when computing the candidate hidden state. It is defined as:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{2.8}$$

where:

- W_r and U_r are weight matrices, and b_r is a bias term.
- $\sigma(\cdot)$ represents the sigmoid activation function, which constrains r_t to the range $(0, 1)$.
- r_t is an element-wise gate that determines the extent to which past information is considered. If $r_t \approx 0$, most of the past hidden state is ignored; if $r_t \approx 1$, more historical information is retained.

Update Gate

The update gate z_t controls the degree to which the previous hidden state h_{t-1} is carried forward versus how much new information from the candidate hidden state is incorporated. It is computed as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{2.9}$$

where:

- W_z, U_z, and b_z are trainable parameters.
- z_t acts as a soft switch that determines the balance between keeping past information and incorporating new input.
- If $z_t \approx 1$, the model heavily relies on h_{t-1}; if $z_t \approx 0$, the model updates its state based on new information.

Candidate Hidden State

The candidate hidden state $\tilde{h}_t$ represents the intermediate state computed from the current input and the previous hidden state, modulated by the reset gate r_t:

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{2.10}$$

where:

- $\odot$ denotes element-wise multiplication (Hadamard product), ensuring that only relevant portions of h_{t-1} contribute to $\tilde{h}_t$.
- $\tanh(\cdot)$ is the hyperbolic tangent activation function, restricting the output range to $(-1, 1)$.
- If $r_t \approx 0$, the previous hidden state's contribution is reduced, making $\tilde{h}_t$ more dependent on the current input. Conversely, if $r_t \approx 1$, past information is preserved.

Final Hidden State Update

The final hidden state h_t is determined by blending the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$, with the update gate z_t controlling the interpolation:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \tag{2.11}$$

where:

- If $z_t \approx 1$, then $h_t \approx h_{t-1}$, indicating that the model retains the past state with minimal modification.
- If $z_t \approx 0$, then $h_t \approx \tilde{h}_t$, meaning the model discards most of the previous information and updates the state based on the current input.

This mechanism enables the GRU to adaptively control information flow, ensuring effective long-term memory retention while discarding unnecessary details.

Characteristics and Advantages

- **Adaptive Memory Control**: The update gate allows the GRU to selectively retain past information or integrate new input, mitigating the vanishing gradient problem and improving long-range dependency modeling.
- **Reduced Computational Complexity**: Unlike more complex recurrent architectures, GRU has fewer parameters, making it computationally efficient and easier to train.
- **Flexibility in Sequence Modeling**: GRUs have been widely applied to tasks such as time-series forecasting, speech recognition, and natural language processing due to their ability to model complex temporal dependencies.

The mathematical formulation presented above provides a comprehensive understanding of GRU's underlying mechanisms, demonstrating how its gating structures facilitate efficient sequence learning. In practical applications, hyperparameters such as the number of hidden units and activation functions can be tuned to optimize performance for specific datasets.

2.2.2 Integration of GRU into Multi-Core CPU Scheduling

The design of multi-core CPU schedulers [8] [9] is a critical issue in modern operating systems, as efficient scheduling significantly influences system throughput, latency, and energy consumption. In heterogeneous multi-core environments—especially those incorporating composite cores and NUMA architectures—traditional static scheduling strategies often fail to adapt to dynamic workload variations. Recent advances in machine learning have opened new avenues for predictive and adaptive scheduling. In this subsection, we provide an in-depth discussion of the Gated Recurrent Unit (GRU), outlining its mathematical foundations, theoretical guarantees, and its integration into multi-core CPU schedulers to enhance fairness, performance, and energy efficiency.

GRU Mathematical Foundations

The GRU is engineered to alleviate the vanishing gradient problem inherent in long sequence processing by employing a streamlined gating mechanism. The core operations of a GRU cell are defined as follows:

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t] + b_z), \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] + b_r), \\
\tilde{h}_t &= \tanh(W_h \cdot [r_t * h_{t-1}, x_t] + b_h), \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t.
\end{aligned}
$$

Here, x_t denotes the input at time t, and h_{t-1} represents the previous hidden state. The weight matrices W_z, W_r, and W_h along with their respective bias vectors b_z, b_r, and b_h are learned during training. The sigmoid function $\sigma(\cdot)$ compresses its input to the range $(0, 1)$, and $\tanh(\cdot)$ outputs values in $(-1, 1)$. The update gate z_t controls the proportion of past information to retain, while

the reset gate r_t modulates the degree to which previous hidden states are combined with the current input. The final state h_t is a convex combination of h_{t-1} and the candidate hidden state $\tilde{h}_t$, allowing the GRU to adaptively balance memory retention and update.

Theoretical Analysis: Contraction Mapping and Stability

To ensure stability in the state update process, we define a mapping $T : h \mapsto T(h)$ as:

$$T(h) = (1 - z(h)) * h + z(h) * \tilde{h}(h, x),$$

where both $z(h)$ and $\tilde{h}(h, x)$ depend on the hidden state h and the input x. Since both the sigmoid and tanh functions are Lipschitz continuous—with Lipschitz constants L_σ and L_{tanh} (typically, $L_\sigma \leq 0.25$ and $L_{\text{tanh}} \leq 1$)—and assuming the spectral norms of the weight matrices are bounded, we can establish that there exist constants L_z and L_h such that:

$$\|z(h) - z(h')\| \leq L_z \|h - h'\|, \quad \|\tilde{h}(h, x) - \tilde{h}(h', x)\| \leq L_h \|h - h'\|.$$

Thus, for any two hidden states h and h', the difference of their mappings satisfies

$$\|T(h) - T(h')\| \leq \Big(\|1 - z(h)\| + L_z \|h'\| + \|z(h)\| L_h + \|\tilde{h}(h', x)\| L_z \Big) \|h - h'\|.$$

If the sum in parentheses is strictly less than 1, then T is a contraction mapping. According to the Banach contraction mapping theorem, T has a unique fixed point within the considered domain, ensuring that the GRU state updates converge and remain stable even under noisy inputs. This theoretical guarantee is especially valuable in real-time applications like CPU scheduling, where prediction stability is paramount.

Integration into Multi-Core CPU Scheduling

In multi-core systems, challenges such as load balancing, core affinity, and the effects of NUMA architectures complicate scheduling. Traditional schedulers that rely on fixed rules may struggle to dynamically adapt to varying workloads. By employing GRU-based predictive models, a scheduler can leverage historical data to forecast task execution times and individual core loads. Specifically, the GRU model can be trained to predict:

1. **Task Execution Times:** Learning temporal patterns in past task runtimes to estimate the CPU time required for new tasks.
2. **Core Load Patterns:** Capturing fluctuations in each core's workload to enable dynamic load balancing.
3. **Energy Consumption:** Estimating power usage patterns to facilitate energy-efficient scheduling decisions.

The predicted metrics can be integrated into a dynamic resource allocation algorithm. For instance, a modular scheduler might include a GRU-based prediction module whose outputs inform a reinforcement learning (RL) or optimization-based decision engine. This hybrid approach enables the scheduler to adjust task assignments, optimize core affinity (especially important in NUMA systems), and balance the load across all cores, thereby reducing inter-core communication delays and enhancing overall system throughput.

Fairness and Performance Considerations

Fairness in multi-core scheduling involves ensuring that no single core is perpetually overloaded while others remain underutilized. A GRU-enhanced scheduler can predict load imbalances before they occur and proactively redistribute tasks. For example, if the model forecasts that a particular core will soon experience high load, the scheduler can reassign some tasks to less burdened cores, ensuring more equitable resource utilization. This dynamic rebalancing not only improves system fairness but also contributes to enhanced performance metrics such as lower average waiting time and higher throughput.

Furthermore, in systems where energy efficiency is critical, such as composite-core architectures, GRU predictions enable the scheduler to make energy-aware decisions. By anticipating core-specific power consumption, the scheduler can strategically activate or deactivate cores, thus reducing overall energy consumption while maintaining performance. The integration of GRU-based predictions with energy management policies is aligned with sustainable computing goals and supports the objectives of Industry 5.0, which seeks to harmonize digital and physical systems for optimal efficiency.

Mathematical Validation and Practical Implications

The contraction mapping property of the GRU update function, as discussed earlier, offers a rigorous mathematical foundation for its stability. This property ensures that even in the presence of fluctuating inputs, the GRU converges to a consistent state, thereby providing reliable predictions for scheduling purposes. In practice, this means that a GRU-enhanced scheduler can consistently adjust to changes in workload patterns and core performance characteristics without destabilizing the system. The combination of precise prediction with adaptive scheduling enables real-time, intelligent allocation of tasks, which is crucial for maintaining high system performance and energy efficiency in modern multi-core environments.

Conclusion

In summary, the GRU's mathematically rigorous framework—characterized by its efficient state update equations and contraction mapping properties—provides a robust foundation for its application in multi-core CPU scheduling. By accurately forecasting task execution times, core loads, and energy consumption, GRU-based models empower schedulers to dynamically balance workloads, optimize core affinity in NUMA architectures, and enhance overall system performance. This integration of deep learning predictions with dynamic scheduling algorithms represents a significant advancement over traditional methods, paving the way for more intelligent, fair, and energy-efficient multi-core processing systems. Future research may focus on further combining GRU with advanced reinforcement learning and optimization techniques to develop even more adaptive and robust scheduling solutions in increasingly complex computing environments.

2.3 eBPF in System Monitoring

The use of extended Berkeley Packet Filter (eBPF) [5] [6] [7] in system monitoring has gained traction due to its ability to provide lightweight, efficient kernel-level instrumentation.

Choosing the Right Attachment Point The appropriate eBPF attachment point depends on several factors:

- **Stability**: `tracepoints` are preferred for long-term monitoring as they remain stable across kernel versions, whereas `kprobes` may break due to kernel updates.
- **Performance Overhead**: `fentry/fexit` incurs the lowest overhead, making it ideal for high-frequency monitoring, whereas `kprobes` can be more intrusive.
- **Monitoring Scope**: `uprobes` should be used for **user-space** application tracing, while `kprobes` and `perf events` are suited for **kernel-level analysis**.
- **Security Considerations**: LSM attachment points provide **fine-grained access control**, enabling real-time security monitoring and policy enforcement.

By leveraging various attachment points, eBPF provides high-performance, real-time, and low-overhead kernel monitoring, making it an essential technology for system debugging, performance optimization, and security auditing.

2.4 eBPF Attachment Points and System Observability

In the context of modern Linux-based system observability and runtime control, one of the core design features of extended Berkeley Packet Filter (eBPF) is its capability to dynamically attach custom programs to various *attachment points*. These points represent the locations or events within the system where an eBPF program may be triggered for execution. Through these mechanisms, developers gain fine-grained insight into system behavior without recompiling the kernel or modifying source code.

eBPF attachment points can be broadly classified into three categories based on their monitoring granularity and system scope: Function-level, Event-level, and Performance-level attachments. Each category supports different system layers and application scenarios.

2.4.1 Function-Level Attachments

Function-level attachment points primarily deal with tracing execution of specific functions at both kernel and user space levels. The most representative examples are `kprobe` and `uprobe`.

`kprobe` enables attachment to arbitrary kernel functions, allowing dynamic observation of syscall logic, scheduler activities, or internal kernel paths. Conversely, `uprobe` targets user-space functions, such as memory allocation in libc or custom logic in dynamic libraries. These

Table 2.2: Comparison of eBPF Attachment Point Types

Category	Subtype	Monitored Layer	Stability	Overhead	Typical Use	Observable Aspects
Function-Level	kprobe	Kernel Functions	Medium	Medium	Syscall tracing, kernel debugging	Function calls, arguments, call counts
	uprobe	User Functions	Medium	Medium	malloc monitoring, query analysis	Application function triggers, memory behavior
Event-Level	tracepoint	Kernel Events	High	Low	Scheduling and I/O observability	PID, file path, event timing
	LSM hook	Security Layer	High	Low	Access control, syscall blocking	File modes, process validation
Performance-Level	perf event	Hardware Counters	Medium	Medium	CPU profiling, bottleneck tracing	Cycles, cache stats, branches
	fentry/fexit	Kernel Function I/F	High	Low	Low-latency tracing	Function latency, return values

Note. This table compares major eBPF attachment types across three abstraction layers. Developers should select based on runtime cost, stability, and target observability goals.

methods are highly flexible and ideal for behavior auditing, debugging, or performance tracing in both open-source and proprietary applications.

2.4.2 Event-Level Attachments

Event-level attachments provide interfaces for pre-defined system events and security-sensitive actions. Linux tracepoints are static instrumentation hooks built into the kernel and exposed at key locations including process scheduling, I/O operations, and networking. Tracepoints offer a stable API and are optimized for low overhead and long-term observability.

Additionally, Linux Security Module (LSM) hooks allow eBPF programs to enforce real-time access control by intercepting operations like file permission changes or task creation. These hooks extend eBPF into the realm of dynamic security enforcement and behavioral policy management.

2.4.3 Performance-Level Attachments

At the performance analysis layer, eBPF supports attachment via `perf events`[17] and `fentry/fexit` points. Perf events enable monitoring of hardware-level metrics such as CPU cycles, cache misses, or branch mispredictions. These are crucial for profiling workloads and identifying bottlenecks.

Meanwhile, `fentry/fexit` attaches directly to function entry and exit points using BPF Type Format (BTF) metadata, offering extremely low-latency tracing compared to `kprobe`. This is particularly advantageous in performance-sensitive paths such as network stacks or high-frequency APIs.

Together, these mechanisms render eBPF a versatile and precise framework for constructing comprehensive observability and runtime policy enforcement systems.

2.5 Sched_ext and Its Extensibility

Sched_ext [10] [11] is a relatively new feature that allows for greater customization of scheduling policies within the Linux kernel.

2.5.1 Introduction

The Linux kernel employs various scheduling mechanisms to efficiently allocate CPU resources among processes. Traditionally, scheduling has been handled by built-in kernel schedulers, such as the Completely Fair Scheduler (CFS) and Real-Time (RT) scheduler, which are compiled into the kernel and operate entirely in kernel mode. While these schedulers are optimized for general-purpose workloads, they lack the flexibility required for specialized applications that demand fine-grained control over scheduling behavior.

To address these limitations, `sched_ext` was introduced as an extensible scheduling framework that enables developers to implement custom scheduling policies as loadable kernel modules (LKM). This modular design allows scheduling logic to be modified dynamically without recompiling the kernel, making it easier to experiment with new scheduling algorithms, optimize resource allocation for specific workloads, and adapt scheduling policies based on runtime conditions.

2.5.2 Comparison with Traditional Scheduling

Compared to traditional Linux scheduling, `sched_ext` offers several distinct advantages, primarily in terms of flexibility and extensibility.

Feature	**`sched_ext`**	**Traditional Scheduling**
Scheduling Logic	Loadable kernel module	Built into the kernel
User-Space Control	Allows runtime tuning	Limited settings
Policy Flexibility	Dynamic switching	Requires kernel recompilation
Overhead	Varies based on implementation	Fixed overhead
Task Selection	Fully customizable	Uses predefined policies

Table 2.3: Comparison of `sched_ext` and Traditional Scheduling

The table above compares `sched_ext` with traditional Linux scheduling mechanisms. Unlike traditional schedulers, which are embedded into the kernel, `sched_ext` provides an extensible framework that allows developers to implement custom scheduling policies as loadable kernel modules. This enables runtime switching of scheduling strategies without recompiling the kernel.

Additionally, `sched_ext` supports user-space control via interfaces such as `sysfs` and `procfs`, allowing real-time CPU allocation adjustments. This flexibility makes it particularly useful for specialized workloads, such as low-latency applications, high-performance computing, and energy-efficient scheduling. However, since `sched_ext` introduces an additional scheduling layer, potential overhead should be considered, especially in scenarios requiring high-frequency scheduling decisions.

The introduction of `sched_ext` represents a significant advancement in Linux scheduling architecture by enabling dynamic, modular scheduling policies. By separating scheduling logic from the core kernel and allowing user-space interaction, `sched_ext` enhances adaptability and makes it easier to develop and deploy custom scheduling algorithms.

2.6 Conclusion

This chapter provided an overview of the evolution of Linux scheduling mechanisms, beginning with an analysis of the Completely Fair Scheduler (CFS) and its enhanced version, Earliest Eligible Virtual Deadline First (EEVDF). CFS, as a core Linux scheduler, efficiently manages

task scheduling using a red-black tree, ensuring fairness among tasks. EEVDF further optimizes time slice allocation, improving scheduling accuracy and balancing CPU-bound and I/O-bound workloads.

Following this, we examined the Gated Recurrent Unit (GRU) and its potential integration into schedulers. Originally designed for neural networks, GRU excels at handling sequential data and predicting task behavior based on historical execution patterns. By incorporating GRU into the scheduler, task execution can be anticipated more accurately, allowing for smarter resource allocation and improved efficiency.

We then explored Extended Berkeley Packet Filter (eBPF), a technology that enables developers to execute custom logic within the kernel without modifying its source code. eBPF significantly enhances the flexibility of scheduling policies, allowing real-time adjustments and greater adaptability without requiring kernel recompilation.

Finally, we discussed `sched_ext`, an extensible scheduling framework that allows developers to modify scheduling logic using loadable kernel modules. Unlike traditional built-in schedulers, `sched_ext` provides real-time user-space control over CPU resource allocation and can be combined with eBPF for more granular scheduling control. However, this flexibility introduces potential overhead, which must be carefully considered in high-frequency scheduling environments.

In summary, this chapter covered the evolution from traditional scheduling mechanisms like CFS to more advanced approaches incorporating GRU, eBPF, and `sched_ext`. These innovations contribute to a more adaptable and intelligent Linux scheduling framework, paving the way for enhanced performance optimization and workload management.

Chapter 3 Methodology

The `gru_scheduler_project` is designed as a dynamic and adaptive CPU scheduling system that integrates eBPF kernel monitoring, GRU-based machine learning prediction, and EEVDF scheduling. Unlike traditional reactive scheduling mechanisms, this system anticipates future workload trends and proactively adjusts scheduling policies to minimize latency and optimize CPU resource utilization.

3.1 Shortcomings of Existing Methods

We designed a Gated Recurrent Unit (GRU) model to predict task execution patterns and make scheduling decisions. Modern operating systems commonly use scheduling policies such as Completely Fair Scheduler (CFS), Real-Time (RT) Scheduling, and First-In-First-Out (FIFO) to allocate CPU resources. While these methods ensure fairness and priority management to some extent, they suffer from significant shortcomings in computational complexity, multi-core load balancing, and dynamic workload adaptation.

Additionally, Earliest Eligible Virtual Deadline First (EEVDF) scheduling improves upon traditional approaches by offering finer control over execution priorities. However, it still has computational overhead due to frequent recalculations of virtual deadlines, potential starvation for low-weight tasks, and a lack of integration with machine learning techniques. [12] [13]

The following sections discuss the major shortcomings of existing scheduling methods and their impact on system performance.

3.1.1 Major Shortcomings of Existing Scheduling Methods

Scheduling Method	**Major Shortcoming**
CFS (Completely Fair Scheduler)	High scheduling overhead
RT Scheduling	Starvation of low-priority tasks
FIFO Scheduling	No preemption for long-running tasks
Multi-Core Load Balancing	Task migration introduces additional overhead
Traditional Scheduling Prediction	Inability to predict workload variations
EEVDF Scheduling	Frequent recalculations of virtual deadlines

Table 3.1: Shortcomings of Existing Scheduling Methods

High Scheduling Overhead in CFS

CFS distributes CPU time based on virtual runtime (Vruntime) and organizes tasks using a red-black tree with O(log N) complexity. While this ensures fairness, maintaining the tree structure introduces additional computational overhead, especially in high-load environments. Frequent context switches and tree rebalancing consume CPU cycles, degrading performance when many processes are active.

Starvation of Low-Priority Tasks in RT Scheduling

Real-time scheduling prioritizes tasks based on strict priority levels, ensuring that high-priority tasks execute immediately. However, this can lead to starvation for lower-priority background tasks, preventing them from receiving CPU time. In mixed workload environments, general-purpose applications may experience high response times due to CPU resources being monopolized by real-time tasks.

Lack of Preemption in FIFO Scheduling

FIFO scheduling executes tasks in the order of arrival without preemption, meaning long-running tasks can block CPU access for shorter tasks. This leads to increased latency for processes that require quick execution, significantly reducing system responsiveness.

Inefficient Task Migration in Multi-Core Load Balancing

Current multi-core scheduling strategies use periodic load balancing to distribute tasks across CPU cores. However, unnecessary task migrations lead to cache invalidation and increased memory latency, reducing overall system performance. In memory-intensive workloads, frequent migrations increase memory access costs, negatively impacting efficiency.

Inability to Predict Workload Variations in Traditional Scheduling

Most conventional schedulers allocate CPU resources reactively, based only on current system conditions rather than anticipating future workload changes. This results in suboptimal CPU utilization, as these methods fail to adapt dynamically to sudden workload fluctuations, leading to either overutilization or resource underutilization.

Computational Overhead in EEVDF Scheduling

EEVDF dynamically adjusts virtual deadlines to optimize scheduling decisions. However, frequent recalculations of virtual deadlines introduce significant computational overhead, particularly in high-throughput environments where scheduling adjustments must be made rapidly. Additionally, since EEVDF relies on weight-based scheduling, tasks with lower weight assignments may suffer from starvation in high-load scenarios.

Discussion and Future Optimization

The analysis highlights that existing scheduling methods struggle to balance fairness, latency, computational overhead, and adaptability to dynamic workloads. While EEVDF provides a more flexible scheduling approach, its high computational costs and lack of predictive capabilities remain limiting factors.

To address these issues, this study proposes an improved approach that integrates eBPF monitoring, GRU-based workload prediction, and EEVDF scheduling enhancements:

- **eBPF-based real-time system monitoring** reduces kernel scheduling overhead.
- **GRU models predict future CPU workloads** to enhance scheduling decisions.
- **Optimized EEVDF scheduling dynamically adjusts virtual deadlines** based on workload predictions, reducing recalculation overhead.

This integrated approach improves CPU resource allocation, reduces unnecessary task migrations, and ensures fairness and responsiveness across varying workload scenarios.

3.2 eBPF Monitoring Module

The eBPF module collects real-time performance metrics from the kernel to provide input features for the GRU model. Traditional system monitoring tools such as Perf, LTTng, and Ftrace provide various performance analysis capabilities but suffer from limitations in flexibility, scalability, and real-time responsiveness. Extended Berkeley Packet Filter (eBPF) has emerged as a powerful alternative, offering low-overhead, high-flexibility monitoring capabilities directly within the Linux kernel. This section explores the advantages of eBPF and its role in real-time system monitoring.

eBPF vs. Traditional Monitoring Techniques

Advantages of eBPF

eBPF offers several key advantages over traditional monitoring techniques:

- **Low Overhead:** eBPF programs execute directly within the kernel, reducing system call overhead and minimizing CPU resource consumption.
- **High Extensibility:** eBPF allows dynamic adjustment of monitoring logic without recompiling the kernel or loading additional modules.
- **Safety:** eBPF operates in a sandboxed environment, and all programs must pass verification before execution, ensuring system stability.
- **Real-time Monitoring:** eBPF enables immediate capture of kernel events, with efficient data transfer via **Ring Buffer** or **Perf Buffer**, making it suitable for high-performance system analysis.

Monitoring Tool	Advantages	Limitations
Perf	Low-level hardware performance counters	High overhead in frequent sampling
LTTng	Low-overhead tracing for complex workloads	Requires kernel modifications for deeper insights
Ftrace	Built into the Linux kernel, suitable for function-level debugging	Limited user-space and kernel-space interaction
eBPF	Low overhead, dynamically adjustable, supports both kernel and user space monitoring	Higher development and debugging complexity

Table 3.2: Comparison of eBPF with traditional monitoring tools

3.2.1 eBPF Monitoring Design

This study employs eBPF for kernel-level monitoring, focusing on three key performance indicators: CPU utilization, context switches, and cache miss rate. Additionally, it explores efficient data transfer mechanisms to user space to support scheduling optimization.

Monitoring Metrics

- **CPU Utilization** measures the load on each CPU core and identifies potential resource bottlenecks:

$$\text{CPU Usage} = \frac{\text{CPU Active Time}}{\text{Total Time}} \times 100\%$$

- **Context Switches** quantify the frequency of process/thread switches, providing insights into scheduler overhead:

$$CS = \frac{\sum(\text{Voluntary Context Switches} + \text{Involuntary Context Switches})}{\Delta t}$$

- **Cache Miss Rate** evaluates CPU cache performance and its impact on memory access efficiency:

$$\text{Cache Miss Rate} = \frac{\text{Cache Misses}}{\text{Total Cache Accesses}}$$

Data Transfer to User Space

Efficient data transfer is essential for minimizing performance impact while ensuring timely analysis. This study compares two primary methods:

- **Ring Buffer:** Optimized for high-throughput scenarios, enabling low-latency data transmission.
- **Perf Buffer:** Integrated within eBPF, providing high-precision data collection while maintaining data integrity.

The Ring Buffer is chosen for its ability to reduce kernel-user space communication overhead, improving real-time performance monitoring.

3.2.2 Mathematical Modeling and Time Series Analysis

eBPF monitoring data exhibits time series characteristics, as system performance varies based on workload conditions, scheduling policies, and memory access patterns. Direct analysis of raw monitoring data can be affected by noise, sudden spikes, and long-term trend variations, potentially reducing scheduling accuracy. Thus, mathematical modeling and time series analysis are employed to identify patterns, filter noise, extract key features, and predict future workload trends.

Objectives of Mathematical Modeling

- **Trend Identification:** Distinguishing between short-term fluctuations and long-term trends ensures adaptability to dynamic workloads.
- **Noise Reduction:** Applying smoothing techniques such as moving average and exponential smoothing minimizes random variations.
- **Future Load Prediction:** Forecasting CPU utilization, context switches, and cache miss rates allows proactive scheduling adjustments.
- **Feature Extraction:** Transforming raw monitoring data into interpretable features, such as variance and spectral characteristics, improves input quality for intelligent scheduling models.

Time Series Modeling Techniques

ARIMA (AutoRegressive Integrated Moving Average) Suitable for long-term trend forecasting of CPU load and scheduling latency:

$$Y_t = c + \sum_{i=1}^{p} \phi_i Y_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t$$

Exponential Smoothing (ES) Effective for real-time load monitoring, assigning higher weights to recent data for short-term trend analysis:

$$S_t = \alpha X_t + (1 - \alpha) S_{t-1}$$

Autocorrelation Analysis Used to detect periodic workload patterns such as scheduled tasks and I/O operations:

$$ACF(k) = \frac{\sum_{t=1}^{N-k}(X_t - \bar{X})(X_{t+k} - \bar{X})}{\sum_{t=1}^{N}(X_t - \bar{X})^2}$$

Feature Engineering for Kernel Events

- **Statistical Features:** Mean, standard deviation, and variance to assess workload stability.
- **Time Series Features:** Autocorrelation coefficients to detect periodic behaviors.
- **Frequency Domain Features:** Fast Fourier Transform (FFT) to analyze CPU usage frequency spectra and filter out random noise.
- **Dimensionality Reduction:** Principal Component Analysis (PCA) to extract the most significant features, reducing dataset complexity while preserving essential information.

Impact of Mathematical Modeling on Scheduling Decisions

Applying mathematical modeling and feature extraction to eBPF monitoring data allows for more effective scheduling strategies:

- **Proactive Scheduling:** Load forecasting enables CPU resource allocation adjustments in advance, reducing scheduling delays during high-load periods.
- **Context Switch Reduction:** Predicting process load allows optimization of scheduling decisions, minimizing unnecessary kernel-level context switches.
- **Improved Resource Utilization:** Identifying workload periodicity facilitates priority adjustments, enabling more efficient execution of different task types.

These mathematical modeling techniques enhance scheduling accuracy, enabling adaptive workload management, optimizing CPU utilization, and reducing scheduling overhead.

3.3 Scheduling Decision

The proposed `GRU_sched` scheduling framework departs from traditional heuristic-based designs by reformulating the scheduling decision process as a sequential prediction task using a Gated Recurrent Unit (GRU) neural network. This approach allows the system to make context-aware and temporally-informed decisions based on task behaviors observed over time, thus potentially improving both CPU core fairness and scheduling responsiveness.

The scheduling logic is divided into two primary stages: feature acquisition and inference-based dispatching. The feature acquisition occurs in the kernel space through the `.enqueue()` callback, which is triggered when a task is either created or awakened. During this event, a set of

static and dynamic scheduling-related features are collected, including: task priority, the number of allowed CPUs, voluntary and involuntary context switches, cumulative runtime, timestamp delta from last switch, and switch frequency. Additionally, context switch counts are obtained via `kprobe` instrumentation, enriching the model's view of task behavior.

These features are encapsulated into a `task_event` structure and sent to user space through a lock-free BPF Ring Buffer using `bpf_ringbuf_output()`. On the user side, a persistent agent listens via `bpf_ringbuf__poll()` and stores incoming task events into a fixed-length sliding buffer. This buffer maintains a temporal sequence of the most recent N tasks and serves as the input to the GRU model. Each inference round constructs a tensor of shape $[1, N, 7]$, where 7 denotes the number of features per task.

The GRU model, previously trained and exported in ONNX format [14] [15] [16], processes the temporal input to predict an optimal CPU core index for the incoming task. This prediction is written back into a shared `pid_to_cpu` BPF map using `bpf_map_update_elem()`.

When a CPU becomes idle, the kernel triggers the `.dispatch()` callback. At this point, the system consults the `pid_to_cpu` map. If a prediction exists, it assigns the task accordingly; otherwise, the fallback strategy is invoked, preserving real-time guarantees and system stability.

This decision-making design not only introduces learning capabilities into the Linux scheduler but also ensures compatibility with fallback mechanisms. It offers a flexible and resilient architecture that balances performance with reliability in diverse workload conditions.

3.4 GRU Model Design and Training

This study aims to construct a predictive learning-based model that approximates the decision-making behavior of the default Linux Completely Fair Scheduler (CFS) in assigning tasks to CPU cores. Rather than treating this as an optimization problem with a defined objective function and closed-form solution, we formulate it as a *multi-class classification task*, where the model learns to imitate the empirical behavior of the scheduler by mapping system-level task features to CPU placement decisions.

Training data is collected by observing the default scheduler's actions during live system operation. For each task, we extract system features at the moment it becomes eligible for scheduling (enqueue time), and label it with the CPU core to which it was actually assigned. This provides a dataset of input-output pairs suitable for supervised learning. The ultimate goal is not to derive a provably optimal policy, but to learn a function that generalizes the observed structure of real-world scheduling behaviors, particularly under varying workloads and system conditions.

Each input sample is represented by a seven-dimensional feature vector, composed of three static features (e.g., task priority, number of allowed CPUs, scheduling group constraints) and four dynamic features (e.g., cumulative runtime, counts of voluntary and involuntary context switches). These features are chosen to encode both intrinsic properties of the task and its interaction history with the system scheduler. By including all available signals—even if some appear weakly correlated—we provide the model with maximal representational capacity to infer complex scheduling tendencies.

To model the temporal aspects inherent in task execution behavior, we adopt a Gated Recurrent Unit (GRU) as the core architectural component. GRUs are well-suited to capture long-term dependencies in sequential input data and are particularly effective in learning patterns from temporal evolution, such as context switching frequency or CPU residency patterns. Each task sample is denoted as $\mathbf{x}_i \in \mathbb{R}^d$, with $d = 7$, and is passed through the GRU layer to obtain a transformed representation. This representation is then fed into a softmax layer producing a categorical distribution $\hat{\mathbf{y}}_i \in \mathbb{R}^C$ over CPU classes, where C is the number of available CPU cores or clusters. The predicted probability for class c is given by:

$$\hat{y}_{i,c} = \frac{\exp(z_{i,c})}{\sum_{k=1}^{C} \exp(z_{i,k})}$$

where $z_{i,c}$ is the logit score produced by the model for class c on input i.

To guide the training process, we use the categorical cross-entropy loss, which measures the divergence between the predicted probability distribution and the one-hot encoded ground-truth label. Formally, given N training samples, the objective function is defined as:

$$L_{\text{CE}} = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

where $y_{i,c} \in \{0, 1\}$ indicates whether the true class for sample i is c, and $\hat{y}_{i,c} \in [0, 1]$ is the predicted probability. The model parameters are optimized using the Adam optimizer, with early stopping based on validation performance to prevent overfitting.

A key architectural novelty in our design is the imposition of structural constraints on feature weights to prevent the model from suppressing any input feature entirely. In standard GRU implementations, input weights and gating mechanisms may effectively zero out the influence of certain features—an effect often beneficial in noise-prone domains like speech or natural language. However, in the context of task scheduling, any system-level feature may become critical under certain workloads. We therefore enforce a regularization constraint that ensures the column norms of the input weight matrix $\mathbf{W}_{\text{in}}$ remain bounded away from zero. This guarantees that no feature is ever fully ignored during forward propagation.

This design is guided by a deeper system-level objective: while the model does not explicitly optimize for context switching, reducing unnecessary context switches is implicitly aligned with the preservation of feature expressiveness. Context switching introduces performance penalties, especially in multicore systems, and deprioritizing features related to task residency and switching behavior may reduce the model's ability to minimize such transitions. By ensuring full feature retention, the model maintains sensitivity to signals that may later correlate with reduced switching overhead—even if such patterns are weakly expressed in the training data.

This trade-off—between exact mimicry of the default scheduler and the retention of all features for future generalization—defines a core tenet of our learning-based scheduler design. Rather than duplicating the kernel's behavior under specific conditions, our model aims to generalize its logic across dynamic and potentially unseen task distributions.

In summary, this GRU-based model not only enables temporal abstraction over task-level features but is also deliberately constructed to maintain the representational integrity of the en-

tire input space. This design principle supports long-term adaptability and responsiveness in environments where workload composition, system policies, or runtime conditions may evolve over time.

Table 3.3: Task Features Collected for GRU Model

Feature Name	Type	Description
`prio`	Static	Static priority of the task, influencing execution order and preemption behavior
`nr_cpus_allowed`	Static	Number of CPUs the task can run on, indicating scheduling flexibility
`runtime_ns`	Dynamic	Total execution time in nanoseconds, reflecting task computation intensity
`nvcsw`	Dynamic	Number of voluntary context switches initiated by the task
`nivcsw`	Dynamic	Number of involuntary context switches (forced preemptions)
`flags`	Static	Task type flag, e.g., whether the task is a kernel thread (`PF_KTHREAD`)
`switch_count`	Dynamic / kprobe	Actual number of context switches, recorded via `kprobe/finish_task_switch`

3.5 Conclusion

This chapter presents the complete workflow of our system, covering task feature extraction, sequential modeling, and kernel-level scheduling decision-making. Leveraging the Linux `sched_ext` framework and eBPF technology, we implement a GRU-based dynamic scheduling module that enables real-time inference with minimal system interference. The kernel space is responsible for data acquisition and callback triggering, while the user space handles sequential inference and decision feedback, effectively decoupling extensibility from real-time constraints.

A key component in this design is the use of a lock-free BPF Ring Buffer, which acts as a critical bridge between the kernel and user spaces. It efficiently and safely transmits real-time task feature streams from the kernel to the user-space agent. To address the asynchronous and high-frequency nature of the data, a dedicated listener thread is implemented to continuously retrieve events from the buffer, ensuring data freshness and temporal integrity for inference.

This overall architecture not only maintains low-latency and high-throughput scheduling properties but also introduces temporal sequence learning capabilities, enabling context-aware and behavior-predictive scheduling beyond the capabilities of traditional OS schedulers.

Chapter 4 Implementation

The gru_scheduler_project integrates eBPF-based kernel monitoring, GRU-driven workload prediction, and EEVDF scheduling to enhance CPU resource allocation. The System Design defines the architecture, encompassing Kernel Space monitoring, User Space prediction, and scheduling decision processes. The Implementation focuses on execution details, including eBPF data collection, GRU inference, and EEVDF task scheduling mechanisms. This integration enables low-latency scheduling, proactive load balancing, and optimized CPU utilization, ensuring fairness and responsiveness in dynamic computing environments.

4.1 System Architecture Overview

The proposed intelligent scheduling system integrates the sched_ext (Scheduler Extension) framework introduced in Linux kernel v6.12 with the BPF (Berkeley Packet Filter) infrastructure and a Gated Recurrent Unit (GRU)-based deep learning model. By leveraging the collaborative mechanism between user space and kernel space, the system enables real-time, adaptive, and learning-based scheduling decisions. Architecturally, the system is designed as a two-tier control structure, consisting of a kernel-space eBPF scheduling module and a user-space inference agent.

At the kernel layer, the system leverages the extensible interface provided by struct sched_ext_ops, through which it implements key callback functions such as .enqueue(), .dispatch(), and .select_cpu(). These callbacks replace the default static scheduling behavior with a custom logic defined by the user. The scheduling logic is written as an eBPF program, which is loaded into the kernel using the libbpf skeleton infrastructure and registered at runtime as the active system scheduler. A custom Dispatch Queue (DSQ) is allocated using scx_bpf_create_dsq(), and all tasks entering the system are inserted into this queue via the .enqueue() callback. Tasks are later consumed and dispatched by .dispatch() according to the model's prediction.

In addition to scheduling tasks, the kernel module is also responsible for real-time feature extraction. For each scheduled task, a feature vector containing seven metrics—including execution time, priority, voluntary and involuntary context switches, flags, and CPU affinity—is extracted directly from the task's task_struct. These features are pushed to user space via a BPF_MAP_TYPE_RINGBUF, allowing seamless and efficient data transfer. To further capture inter-core contention behavior in multicore environments, the system includes a kprobe hook on the finish_task_switch kernel function, which monitors the number of times each task has been context-switched. This switch count serves as a dynamic scheduling indicator and is incorporated into the feature vector.

On the user-space side, a lightweight inference agent is implemented. This agent listens to the ring buffer for incoming task events and constructs a time-series tensor of shape [1, 10, 7] using the most recent ten task snapshots. This input is fed into a GRU model pre-trained in

PyTorch and exported in ONNX format. The model outputs a single CPU index representing the optimal core for scheduling the corresponding task. The predicted index is written back into a BPF hash map (`pid_to_cpu`), from which the kernel's `.dispatch()` function reads and executes the scheduling action.

This architecture cleanly separates data collection and system-level scheduling (kernel layer) from decision-making and inference logic (user space). Such separation not only preserves system performance and safety—by avoiding heavy computation in kernel space—but also provides modularity and extensibility. The system can readily support alternative models, additional features, and architectural refinements without disrupting the core scheduling pipeline.

4.2 Scheduler Module Implementation

To achieve intelligent scheduling capabilities, the proposed system separates the scheduling decision pipeline into two distinct phases: data collection and task dispatching. The kernel-space component is primarily responsible for the former. Utilizing the extensibility of the Linux `sched_ext` interface, we implemented three key callback functions—`.enqueue()`, `.dispatch()`, and `.select_cpu()`—which collectively form the backbone of the scheduling module.

In the `.enqueue()` phase, whenever a task is created or awakened, the kernel module extracts its runtime behavioral features. These include both static attributes (e.g., task priority and the number of allowed CPUs) and dynamic runtime metrics (e.g., accumulated execution time and context switch counts). In addition, a `kprobe` is attached to the kernel function `finish_task_switch` to monitor and accumulate the number of actual context switches the task undergoes during execution. These seven features are encapsulated in a `task_event` structure and transmitted to the user space via a ring buffer, serving as the input for the GRU-based inference model.

When a CPU enters an idle state, the `.dispatch()` callback is triggered to select the next task to execute. At this point, the scheduler fetches a task from the custom Dispatch Queue (DSQ) and checks the `pid_to_cpu` map. If a predicted CPU index is found, the task is dispatched accordingly; otherwise, a fallback strategy is applied, assigning the task to the current available CPU. This approach allows the scheduling decision to incorporate learning-based predictions while maintaining fault tolerance and low-latency responsiveness in the absence of prediction results.

The `.select_cpu()` function acts primarily as a CPU wake-up hint mechanism. Although it does not directly influence the final task-to-CPU mapping, it supports potential extensions related to idle core management, energy-aware scheduling, or partial task offloading.

Through the collaborative functionality of these three interfaces, the scheduler module effectively decouples data acquisition from scheduling execution. It establishes a structured communication channel between the kernel and user space, forming the core architectural element for realizing adaptive, learning-based scheduling decisions.

As shown in Table 3.3, the scheduler extracts a combination of static and dynamic fea-

tures from each task, which are used to construct the input for GRU-based scheduling inference. Notably, `switch_count` provides insight into system-level contention and task volatility, enhancing the model's ability to adapt to runtime conditions.

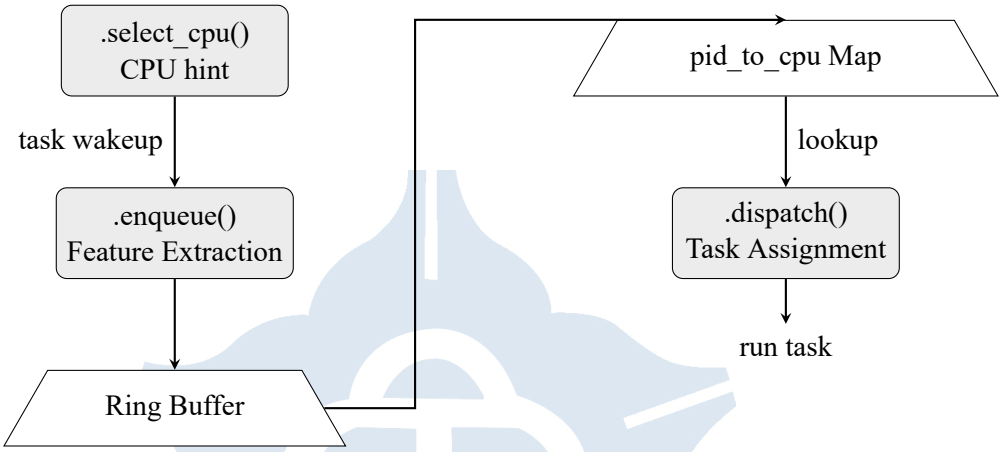


Figure 4.1: Implementation structure of the kernel-space scheduling module. Feature extraction is performed in `.enqueue()`, with decisions applied in `.dispatch()` based on optional prediction hints from the userspace.

4.3 Userspace Inference Module

The userspace inference module serves as a real-time decision-making agent that operates persistently in the system background. Its primary responsibility is to receive scheduling-related task events from the kernel, perform sequential inference using a deep learning model, and propagate the prediction results back to the kernel to assist in subsequent scheduling decisions. The module adopts an event-driven architecture and utilizes the `bpf_ringbuf__poll()` interface provided by `libbpf` to continuously monitor the data stream emitted by the eBPF-based scheduler through the ring buffer.

In the kernel space, the `.enqueue()` callback is triggered upon task creation or wake-up, during which a set of seven scheduling features is extracted. These include static attributes (e.g., task priority and the number of available CPUs) and dynamic runtime behaviors (e.g., counts of voluntary and involuntary context switches, accumulated execution time, and recent switch frequency). The collected features are encapsulated into a `task_event` structure and written into the ring buffer, which functions as the input channel for userspace inference. The ring buffer implements a lock-free producer-consumer model, where the kernel acts as the producer and the userspace agent as the consumer. While this design is optimized for low latency and high throughput, it does not guarantee delivery, and under high load conditions, events may be overwritten before consumption, resulting in data loss.

To mitigate the impact of such losses on inference quality, the module deploys a dedicated listener thread responsible for extracting task events from the ring buffer. These events are buffered into a fixed-size sliding window implemented as a first-in, first-out (FIFO) queue, which maintains the latest N task entries. Once the buffer reaches the required length, the contents are assembled into a three-dimensional input tensor of shape $[1, N, 7]$, representing one

batch, a temporal sequence of length N, and a feature dimension of seven.

The inference engine employs a Gated Recurrent Unit (GRU) model, pre-trained offline and exported in ONNX format. It leverages temporal information to predict the most suitable CPU core for the target task. The prediction output is a single integer representing the selected CPU index, which is then associated with the task's process identifier (PID) and written into the kernel's `pid_to_cpu` BPF map via the `bpf_map_update_elem()` function. This map is subsequently queried by the `.dispatch()` callback during task assignment.

If the inference is incomplete, the input sequence is insufficient, or ring buffer overflow causes data loss, a fallback mechanism is activated. In such cases, the kernel scheduler executes its default dispatch policy to ensure uninterrupted scheduling and to prevent starvation. This fault-tolerant design preserves system responsiveness and contributes to the robustness of the scheduling framework.

Furthermore, the module supports logging of discrepancies between the predicted and actual CPU assignments. This functionality facilitates behavioral traceability and provides a foundation for performance evaluation, online learning, or model retraining. Overall, the module extends the expressive power of eBPF by introducing advanced numerical and temporal modeling capabilities, and constitutes a critical component of the intelligent scheduling infrastructure.

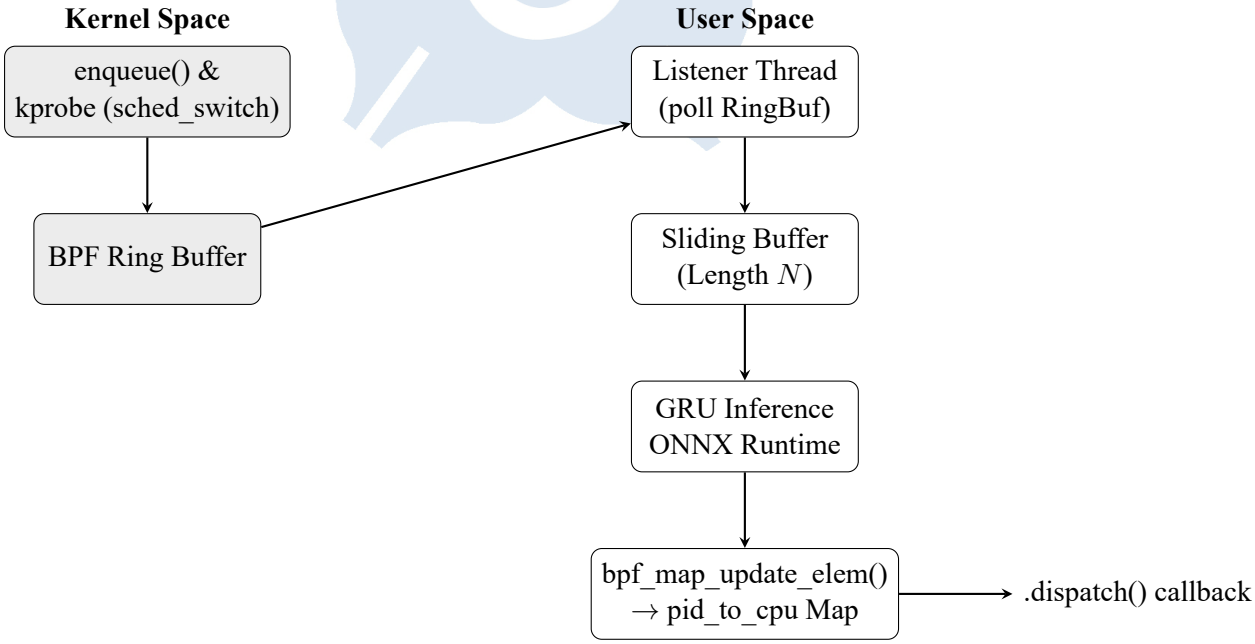


Figure 4.2: Architecture of the userspace inference module for GRU-based scheduling. The kernel emits task events into a ring buffer, which are polled, buffered, and used as input for sequence inference. Prediction results are fed back into the kernel.

4.4 GRU Model Training and Deployment Workflow

The proposed scheduling framework incorporates a gated recurrent unit (GRU) model with temporal memory to improve decision-making in CPU task allocation. The overall system is

modularized into the following pipeline stages: data collection, sequence reconstruction, model training, model export, and userspace inference integration, thereby forming a closed-loop architecture operable in both offline training and online inference modes.

During the data collection phase, the kernel—augmented with a modified `sched_ext` interface and strategically placed kprobes—monitors task-related events triggered during wakeup and context switch operations. Each event is encoded as a seven-dimensional feature vector that encapsulates both static attributes (e.g., task priority, CPU affinity mask) and dynamic runtime metrics (e.g., execution duration, voluntary/involuntary context switch counts, and a switch counter obtained from a kprobe on `finish_task_switch`). These feature vectors are packaged into `task_event` structures and emitted into a BPF ring buffer, serving as the input interface to the userspace inference module.

To ensure that all features retain influence during both training and inference phases, the model architecture enforces a non-zero constraint on feature weights. This design choice prevents the model from learning trivial solutions where certain features are ignored, and it guarantees a persistent deviation from the baseline behavior of the default Linux scheduler. Such divergence ensures that the learned model consistently integrates meaningful feature-based insights into its predictions, reinforcing the impact of observed temporal dynamics on CPU core selection.

The ring buffer operates under a lock-free producer-consumer paradigm, wherein the kernel writes events asynchronously, while userspace consumes them via the `bpf_ringbuf__poll()` interface. Under high system load, if the consumer lags behind, the fixed-size buffer may overwrite older entries, leading to potential data loss. To address this, a dedicated listener thread in userspace continuously retrieves and stores the events in a fixed-size FIFO sliding buffer, maintaining the most recent N task events for further processing.

Once the sliding buffer accumulates N valid samples, the data is structured into a three-dimensional input tensor with shape $[1, N, 7]$, where the first dimension denotes the batch size (fixed at 1), the second the temporal sequence length, and the third the feature vector dimensionality. This tensor is then forwarded to a pre-trained GRU model exported in the ONNX format, which performs inference to predict the optimal CPU core. The model's output—an integer representing the recommended CPU index—is associated with the corresponding task PID and written into the kernel-space `pid_to_cpu` BPF map using `bpf_map_update_elem()`, allowing integration with the `.dispatch()` callback during task assignment.

In scenarios where the input sequence is incomplete, inference is interrupted, or ring buffer overflow leads to data loss, the system falls back to the default kernel scheduling logic to ensure continuity and avoid starvation. The inference path is optimized for sub-millisecond latency and leverages an in-memory model representation to minimize loading overhead during runtime.

In summary, by enforcing non-zero weight constraints on all input features, the proposed system ensures that the GRU model's predictions consistently deviate from those of the default scheduler in a stable and explainable manner. This design guarantees that every feature maintains influence throughout the model's lifecycle, enabling robust and adaptive scheduling behavior grounded in real-time system dynamics.

4.5 System Integration and Execution Flow

This section describes the complete integration process of the GRU-based scheduling system, including coordination between kernel modules and the userspace agent, as well as how the prediction results influence task-to-core assignment in real time.

Upon system initialization, the kernel loads the eBPF scheduler by attaching a `sched_ext_ops` structure and initializes internal data structures, such as custom BPF maps and dispatch queues (DSQs). In the `.init()` callback, the scheduler calls `scx_bpf_switch_all()` to enroll all tasks into the SCHED_EXT class and uses `scx_bpf_create_dsq()` to construct a fallback DSQ. In addition, a `kprobe` is attached to `sched_switch` to track system-wide context switch frequencies, which is recorded as part of each task's dynamic feature set.

When a task is awakened, the `.select_cpu()` callback provides a CPU hint. If the suggested CPU is idle and allowed by the task's `cpumask`, the task is directly inserted into its local DSQ. Otherwise, the task proceeds to the `.enqueue()` stage, where seven predefined features—including both static (e.g., priority, allowed CPUs) and dynamic (e.g., runtime, context switches, switch count)—are collected and encapsulated into a `task_event` structure. This structure is written into a BPF ring buffer, which serves as the communication channel to the userspace.

In userspace, a dedicated agent continuously polls the ring buffer using `bpf_ringbuf__poll()` and stores incoming events into a fixed-length sliding buffer. Once this buffer accumulates N events, the sequence is transformed into a tensor of shape $[1, N, 7]$ and passed into the GRU model via ONNX Runtime. The inference process yields an integer representing the recommended CPU index. This result is associated with the task's PID and written into the `pid_to_cpu` map via `bpf_map_update_elem()`, enabling the kernel to query it during the `.dispatch()` stage.

When a CPU becomes idle and no tasks are present in its local or global DSQs, the `.dispatch()` callback is triggered. At this point, the kernel attempts to fetch a prediction from the `pid_to_cpu` map. If a recommendation is available, the system tries to consume the corresponding task from the fallback DSQ and move it into the target CPU's local DSQ for execution. If no prediction is found, or if the sequence length is insufficient, the fallback path is activated. The CPU selects a task from the fallback DSQ at random to maintain forward progress.

To ensure robustness, the system includes several fault-tolerant mechanisms: predictions are optional, and the dispatch cycle is non-blocking even under data loss or inference delay. If the ring buffer is overloaded, new events overwrite old ones without halting the system. The inference logic is also decoupled from the kernel and runs in a separate thread, preventing prediction latency from stalling core scheduling operations.

In summary, the system integrates a BPF ring buffer for feature extraction and delivery, and a `pid_to_cpu` map for decision feedback. This architecture enables low-latency, asynchronous communication between kernel and userspace components, forming a robust and intelligent task scheduling framework that is both scalable and resilient to runtime uncertainties.

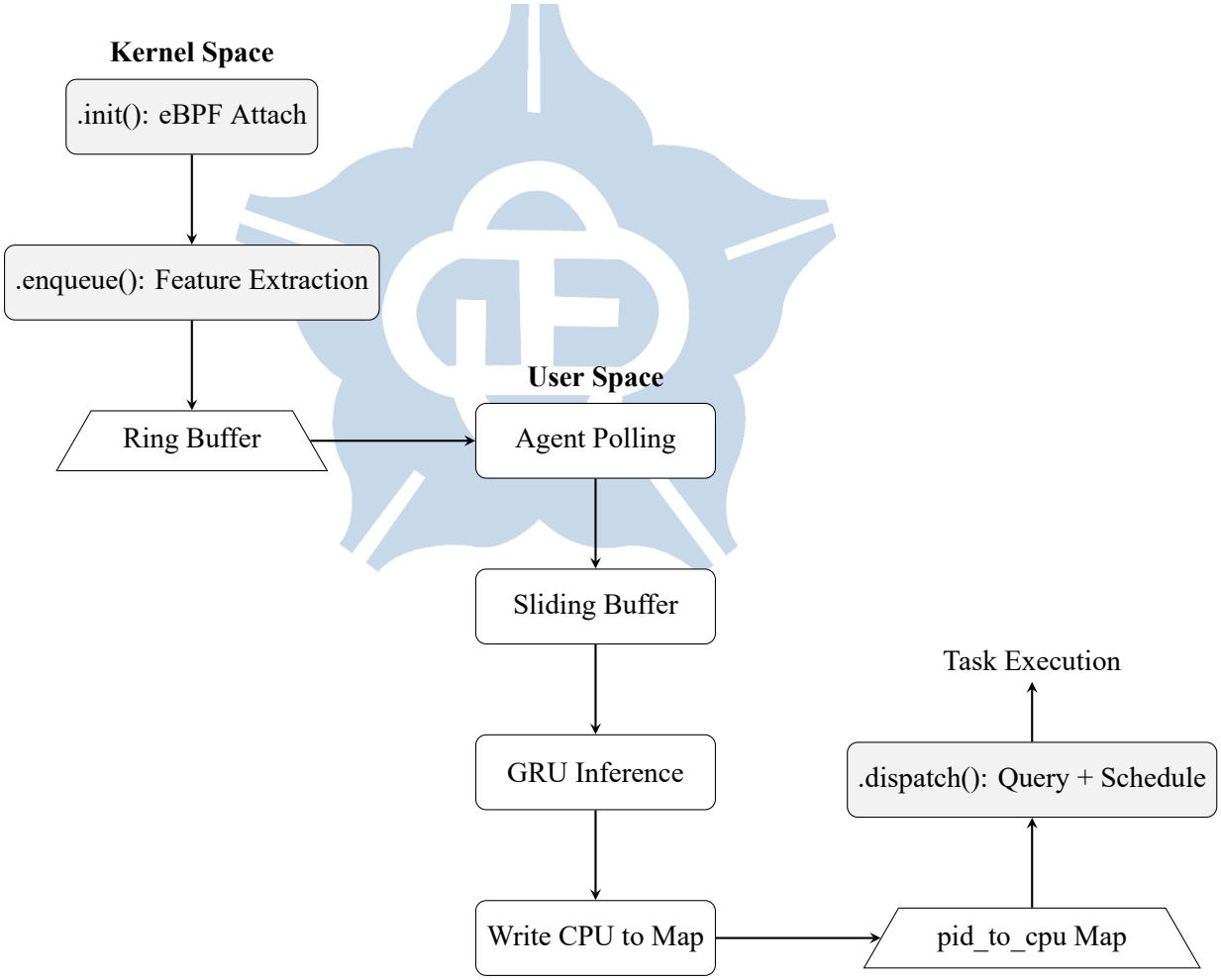


Figure 4.3: Simplified system integration flow of the GRU-based scheduling framework. Features extracted by the kernel are sent to userspace for inference and returned to guide CPU selection.

Chapter 5 Evaluation & Experiments

5.1 Experimental Platform and Kernel Configuration

All experiments in this study are conducted on Linux kernel version 6.12, which natively supports the `sched_ext` framework and eBPF-based scheduler extensions. To enable user-defined scheduling logic, the following kernel configuration options must be explicitly enabled:

- `CONFIG_SCHED_CLASS_EXT=y`: Enables the `SCHED_EXT` scheduling class, allowing the registration of external eBPF schedulers.
- `CONFIG_BPF_SYSCALL=y`: Allows userspace to interact with BPF programs via the `bpf()` system call.
- `CONFIG_BPF_JIT=y`: Enables BPF Just-In-Time compilation to improve runtime performance.
- `CONFIG_BPF_STRUCT_OPS=y`: Supports struct_ops-based BPF programs for interfacing with scheduler callbacks.
- `CONFIG_DEBUG_INFO_BTF=y`: Provides BTF metadata required for libbpf skeleton generation.

The custom scheduler used in this work implements key callback functions including `.enqueue()`, `.dispatch()`, and `.select_cpu()` using the `BPF_STRUCT_OPS` interface. The BPF program is compiled using Clang/LLVM and transformed into a userspace-accessible skeleton via `bpftool gen skeleton`.

Our experiments were conducted on a multi-core processor system. All experimental tasks are executed on a four-core configuration, with CPU isolation applied using `isolcpus=1-3` to reserve logical CPUs exclusively for evaluation. Dynamic CPU features such as Turbo Boost, power scaling, and simultaneous multithreading (SMT) are disabled to ensure deterministic behavior and reproducibility.

The GRU-based inference module is pre-trained and exported in ONNX format. During runtime, the ONNX model is invoked using the C++ ONNX Runtime API. The model receives a time-series tensor of shape $[1, 10, 7]$, representing seven scheduling features over ten time steps, allowing the scheduler to capture short-term behavioral trends and output an optimal CPU assignment.

5.2 Experiment Design

The experimental design of this study aims to validate the effectiveness of the proposed GRU-based intelligent scheduler, referred to as `GRU_sched`, in enhancing scheduling quality

under multicore environments. The primary objectives include: (1) evaluating the predictive accuracy of sequence-based models in CPU assignment; (2) examining the feasibility and efficiency of scheduling collaboration between userspace inference and kernel-space control via `sched_ext`; and (3) comparing overall performance against existing Linux schedulers in terms of latency, fairness, and load balancing under diverse workloads.

Two scheduling policies are compared: (i) the default kernel scheduler, which refers to the `Completely Fair Scheduler (CFS)` currently used as the standard scheduling class in Linux; and (ii) the proposed `GRU_sched`, which integrates eBPF-based `sched_ext` infrastructure with a Gated Recurrent Unit model to perform time-series inference and dynamic CPU selection.

The workload design includes three representative execution contexts: CPU-bound, I/O-bound, and mixed heterogeneous tasks. CPU-bound tests utilize `sysbench`[21] [22] and `stress-ng` to simulate high-performance computing workloads; I/O-bound tests involve randomized file access and wait-based behavior to emulate server-side latency; the mixed case combines both workload types concurrently to reflect real-world task heterogeneity. All experiments are conducted on a four-core system with selected CPUs reserved via `isolcpus`, ensuring no background interference. Each test configuration is executed in 30 iterations with automated workload generation and scheduler switching scripts to ensure consistency and reproducibility.

Throughout the execution, the system logs include task-to-CPU mappings, match rates between GRU inference and actual scheduling, context switch counts, and average scheduling latency per round. Additional kernel behavior is traced using `perf trace` to provide detailed visibility into system behavior. The experimental design emphasizes fairness control, reproducible test conditions, and complete traceability for post-hoc analysis and evaluation.

Table 5.1: Workload Design and Testing Tools

Category	**Description**	**Test Tools**
CPU-bound	Intensive arithmetic computation tasks	`sysbench`, `stress-ng`
I/O-bound	Frequent blocking and wake-up due to I/O	File read/write simulation, socket delay
Mixed workload	Concurrent execution of CPU and I/O tasks	Combined sysbench + file I/O scripts

5.3 Evaluation Metrics and Measurement Methodology

To comprehensively evaluate the effectiveness and adaptability of the proposed GRU-based intelligent scheduler (`GRU_sched`) under various operating conditions, we conduct measurements from five dimensions: fairness, latency and throughput, high-load stability, multicore balancing, and CPU resource utilization. Each dimension is assessed using established Linux performance tools and visualized through representative statistical plots to ensure accuracy and reproducibility.

Fairness[18] [19] [20] is quantified using Jain's Fairness Index (JFI), which measures how evenly CPU time is distributed among competing threads across cores. We employ `hackbench` and `stress-ng` to simulate highly concurrent workloads, generating aggressive contention

across CPUs. Results are presented using box plots to illustrate the distribution of fairness scores under each scheduler.

Latency and throughput are assessed through two complementary tools. The `sysbench` CPU module measures the average computation time per batch task, and results are visualized using bar charts. Additionally, `perf` is employed to record context switch events and scheduling latency distributions, which are summarized in histograms. These metrics reflect each scheduler's responsiveness under short-duration pressure.

High-load stress tolerance is further verified by running massive concurrent workloads using combined `stress-ng` and `hackbench` instances, producing over a thousand threads in parallel. We observe whether the schedulers maintain consistent fairness and bounded latency during overload, using line charts and JFI tracking as quantitative indicators of scalability and real-time response capability.

Multicore balancing is evaluated through `perf bench sched messaging`, which assesses inter-core communication throughput, as well as `Super Pi`, which compares single-threaded versus multi-threaded compute performance. We record task density and CPU utilization across cores and represent the differences using bar charts that compare expected versus actual workload placement.

CPU resource allocation analysis is performed using Perfetto trace and Flame Graphs. Perfetto provides a full timeline of scheduling decisions, revealing the relative timing and frequency of task switching under different schedulers. Flame Graphs present stack-based CPU usage density, identifying core hotspots and affinity patterns that aid in visual inspection of scheduling behavior.

Overall, our evaluation methodology emphasizes both macro-level performance (fairness, load distribution) and micro-level scheduling details (latency, task migration), ensuring that `GRU_sched` is validated across dimensions critical to real-world scheduling performance in modern operating systems.

Table 5.2: Evaluation Metrics and Measurement Tools (Updated)

Evaluation Aspect	**Measurement Tools**
Fairness (JFI)	`mpstat` (per-core %usr), custom JFI computation with GNU
Latency & Throughput	`sysbench`, `perf sched latency`
High Load Behavior	`stress-ng` (CPU & pipe), `mpstat`, `top`
Multicore Load Balancing	`perf bench sched messaging`
Context Switch Analysis	`perf stat`, `vmstat`

5.4 Experimental Results and Analysis

This section presents a detailed analysis of the experimental results obtained under different workload scenarios. We evaluate the performance of the proposed `GRU_sched` in comparison to

traditional schedulers like CFS and EEVDF.

5.4.1 CPU-bound Workload Scenario

In this scenario, all tasks are computationally intensive, requiring continuous CPU usage. The goal is to assess how well each scheduler handles high CPU contention without significant overhead. Metrics such as CPU utilization, task migration rate, and context switch count are examined.

Overall Performance Comparison in CPU-Bound Workloads

In CPU-bound workloads using the default scheduler, we observe predictable yet improvable behavior. Table 5.3 presents an overview of the system-wide metrics. Under the default scheduler, the total runtime reached 370.553 milliseconds, while the GRU-based scheduler reduced this to 313.023 milliseconds, representing a 15.5% improvement. Interestingly, despite the shorter runtime, the GRU scheduler completed 3.6% more tasks, suggesting better throughput under constrained CPU conditions without sacrificing stability.

More significantly, the weighted average latency—calculated with respect to task runtime shares—dropped from 0.070 ms in the default scheduler to 0.062 ms with GRU scheduling, marking an 11.4% reduction. Furthermore, the number of high-latency tasks (defined as those with latency exceeding 0.5 ms) also dropped by 25.8%. These improvements indicate not only faster task handling on average but also tighter tail latency control, which is critical in real-world systems where even rare delays can degrade user-perceived performance.

Table 5.4 provides latency comparisons for several representative tasks. The systemd task saw a dramatic 84.5% latency reduction, while dbus-daemon achieved an even larger 91.7% improvement. These tasks are essential for maintaining user-space responsiveness, so such latency reductions directly translate to smoother and more predictable system behavior. On the other hand, Apache2 exhibited a minor increase in latency (+4.4%), which, while measurable, remains small enough to fall within normal system noise and workload fluctuation margins. No critical regressions were observed across any of the major service processes.

The combination of reduced runtime, increased task completion, and lower average and maximum latencies strongly suggests that the GRU scheduler adapts better to dynamic CPU pressures compared to the default heuristic-based approach. Importantly, these gains are achieved without introducing instability or anomalous spikes, demonstrating that a learned scheduling policy can outperform traditional methods even in deterministic CPU-bound contexts. The results validate the broader potential of machine learning‑assisted scheduling strategies in scenarios previously considered less dynamic or ”well-behaved.”

In summary, while the default scheduler offers robust baseline CPU-bound behavior, the GRU scheduler demonstrates clear advantages in both efficiency and responsiveness. These results emphasize that intelligent scheduling can enhance not only latency-sensitive performance but also overall throughput, even in single-threaded, CPU-saturated workloads where opportunities for optimization were traditionally considered limited.

Table 5.3: Overall Metrics in CPU-Bound Workloads

Metric	Default Scheduler	GRU Scheduler	Change Rate
Total Runtime (ms)	370.553	313.023	↓15.5%
Total Tasks	3,875	4,017	↑3.6%
Weighted Avg Latency (ms)	0.070	0.062	↓11.4%
High-Latency Tasks (>0.5ms)	225	167	↓25.8%

Table 5.4: Key Task Latencies in CPU-Bound Workloads

Task Name	Default Scheduler	GRU Scheduler	Improvement
systemd	0.387	0.060	↓84.5%
kworker	1.494	1.200	↓19.7%
dbus-daemon	0.782	0.065	↓91.7%
Apache2	0.045	0.047	↑4.4%

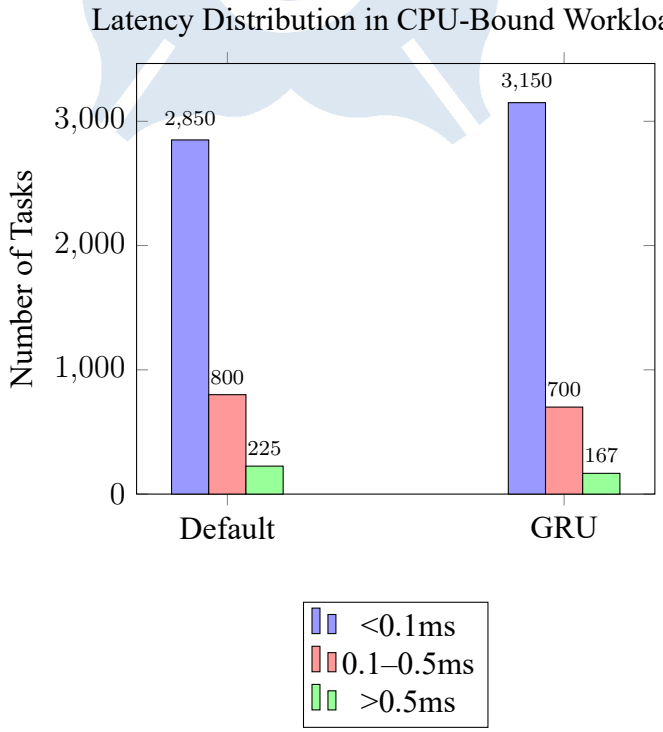


Figure 5.1: Latency Distribution in CPU-Bound Workloads

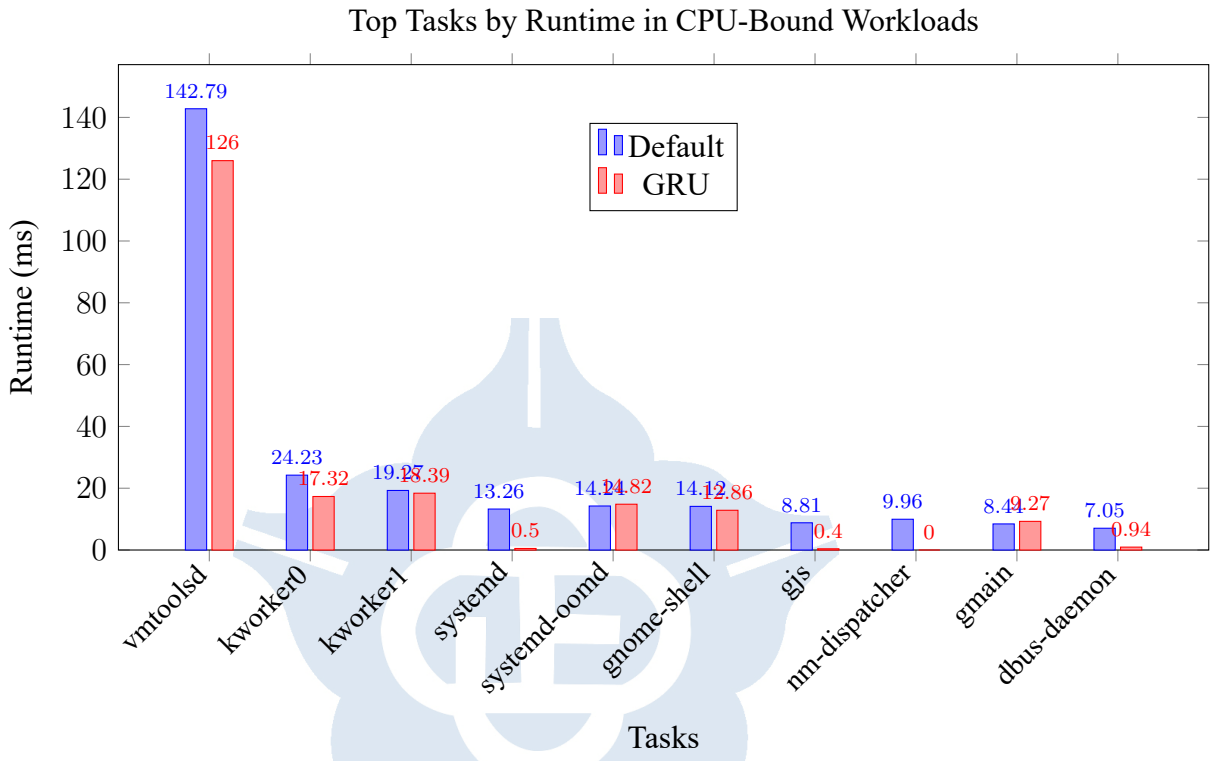


Figure 5.2: Top 10 Tasks by Runtime in CPU-Bound Workloads

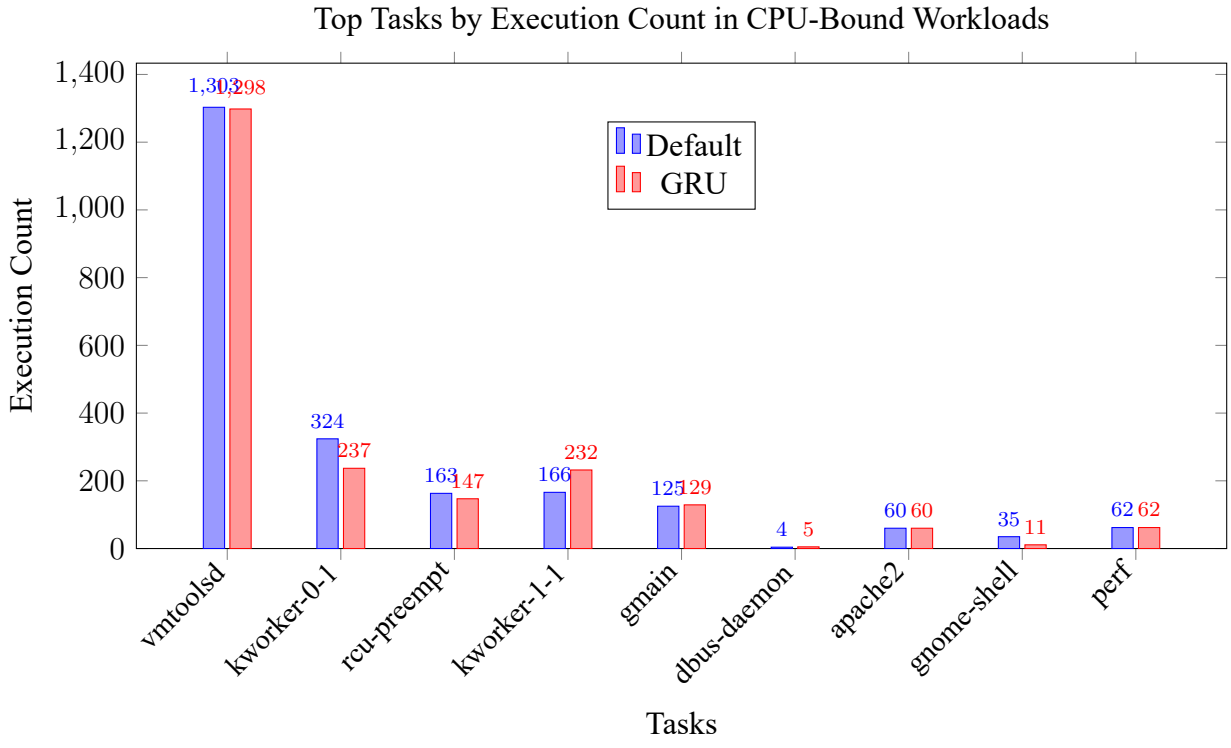


Figure 5.3: Top Tasks by Execution Count in CPU-Bound Workloads

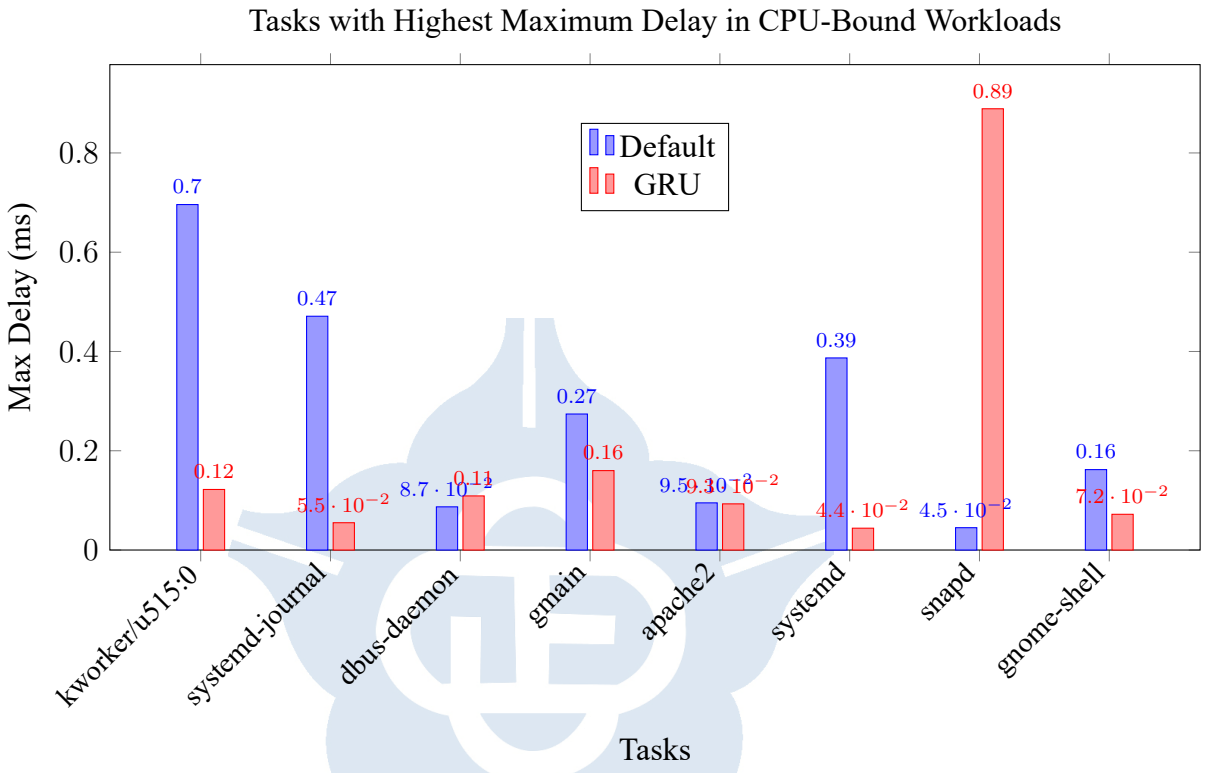


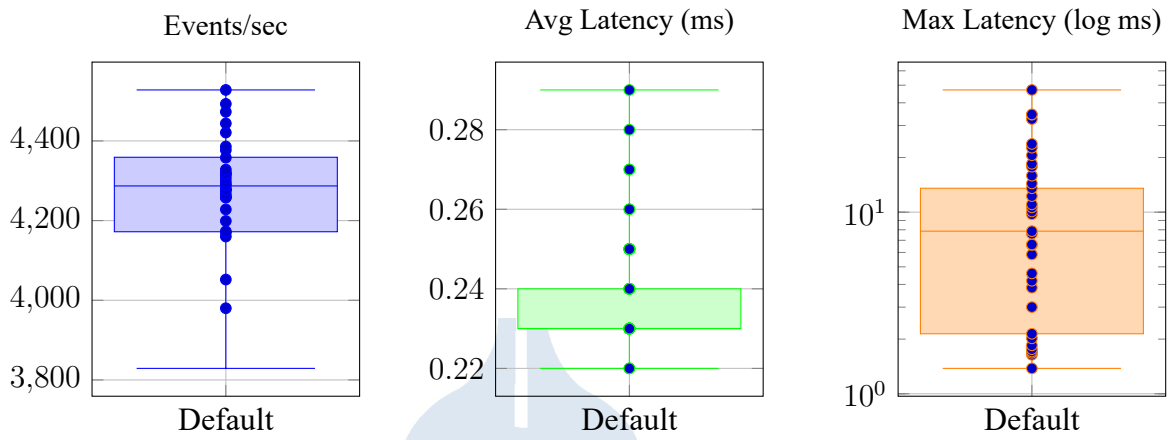
Figure 5.4: Tasks with Highest Maximum Delay in CPU-Bound Workloads

Single-thread CPU-bound Performance under Default Scheduler

When executing a single-threaded `sysbench` CPU-bound test under the default scheduler, we observed consistently stable performance. As shown in the boxplot for events per second, the distribution is tightly clustered, indicating that the scheduler maintains a relatively constant throughput over time. Although the median sits near 4287 events/sec, the upper whisker reaches 4528, suggesting that in certain test iterations, the system achieved brief surges of higher throughput.

Regarding average latency, most samples fall between 0.23 and 0.25 milliseconds, indicating that the scheduler handles context switching and task scheduling with consistency. However, despite this low average, the boxplot of maximum latency—presented on a logarithmic scale—reveals a different story. In some test iterations, individual events experienced significant delays, with the worst-case reaching as high as 47 milliseconds. This long-tail behavior highlights that even in a controlled, single-threaded, CPU-bound environment, the default scheduler can occasionally suffer from jitter or contention.

Overall, the test results demonstrate that the Linux default scheduler provides robust and consistent average-case performance. Nonetheless, these sporadic high-latency occurrences suggest potential pitfalls in latency-sensitive or real-time applications. In future comparative analyses, especially with alternatives such as `GRU_sched`, such long-tail latency patterns will be crucial indicators for evaluating scheduler responsiveness and determinism.



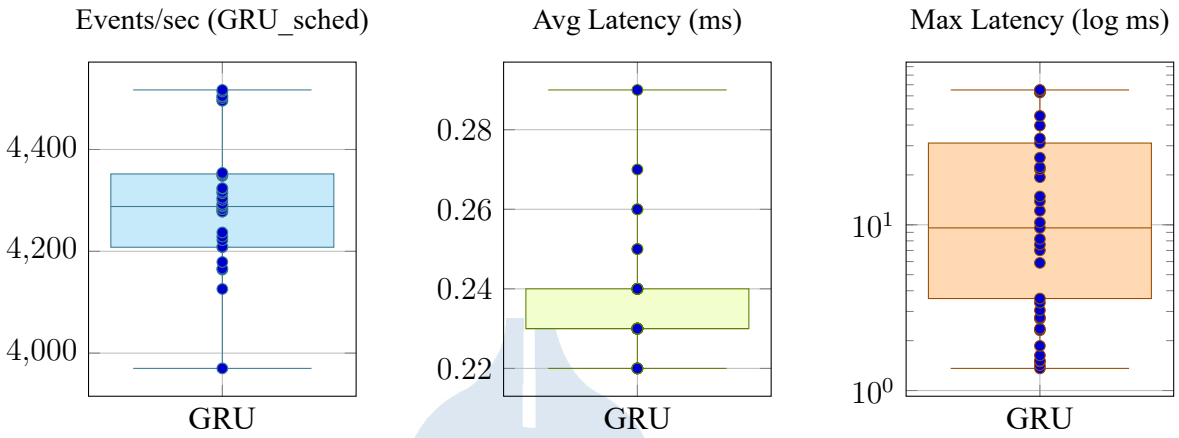
(a) Event Rate Distribution (b) Average Latency Distribution (c) Maximum Latency (log scale)

Figure 5.5: Sysbench (1-thread CPU-bound): Event Rate and Latency Distributions (Default Scheduler)

When executing a single-threaded `sysbench` CPU-bound test under the `GRU_sched` scheduler, we observed a stable and consistent performance profile, with relatively tight clustering in the distribution of events per second. The median event rate was around 4288 events/sec, with the upper whisker reaching 4517 events/sec, indicating occasional spikes in throughput during specific test iterations. This suggests that the `GRU_sched` scheduler can maintain a high throughput while exhibiting minor variability across test runs.

In terms of average latency, the data shows that most events had latency values between 0.22ms and 0.24ms. The relatively narrow range of latency values suggests that the `GRU_sched` scheduler efficiently handles task scheduling with minimal fluctuation. However, a more nuanced picture emerges when we examine the maximum latency on a logarithmic scale. Despite the generally low average, there were instances of significant delays, with some events experiencing maximum latencies as high as 65ms. This long-tail behavior indicates that under certain conditions, even the `GRU_sched` scheduler may encounter latency spikes, potentially affecting time-sensitive tasks.

In summary, the test results indicate that the `GRU_sched` scheduler delivers stable and reliable average performance with a generally low latency range. However, the presence of sporadic high-latency events underscores the importance of considering latency outliers when evaluating schedulers, particularly for latency-sensitive or real-time applications. These outliers will be a critical factor to assess in future comparisons, especially when contrasted with other scheduling alternatives, such as the default scheduler, to better understand their responsiveness and overall performance consistency.



(a) Event Rate Distribution (b) Average Latency Distribution (c) Maximum Latency (log scale)

Figure 5.6: Sysbench (1-thread CPU-bound): Event Rate and Latency Distributions (GRU Scheduler)

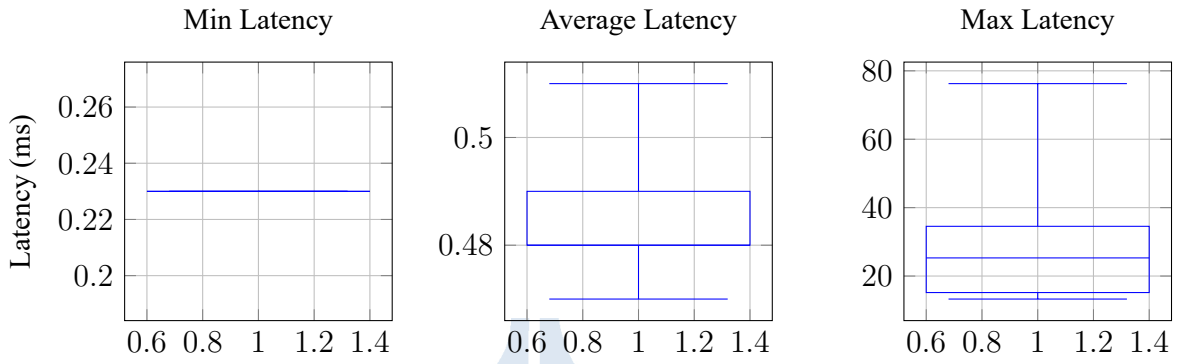
The GRU scheduler demonstrates stable and consistent performance in a single-threaded sysbench CPU-bound test, with a median event rate of approximately 4288 events/sec and low latency. However, it exhibits sporadic high-latency spikes, particularly with maximum latencies reaching up to 65ms. In comparison, the default scheduler performs similarly in terms of event rate and average latency but shows less pronounced latency spikes, with a maximum of 47ms. Both schedulers maintain low average latencies, but the GRU scheduler may face occasional delays, which should be considered for latency-sensitive applications.

Eight-thread CPU-bound Performance

The eight-thread CPU-bound test conducted under the default Linux scheduler revealed stable throughput and latency characteristics over a 60-second duration. The observed throughput ranged approximately between 15,600 and 17,000 events per second, with an average throughput of around 16,600 events/sec.

Latency analysis showed that the minimum latency consistently stayed around 0.23 milliseconds, while the average latency fluctuated between 0.47 and 0.51 milliseconds across different sampling points. Although the maximum latency generally remained low, occasional spikes were recorded, reaching up to 76.26 milliseconds. For the majority of the workload, the 95th percentile latency stayed within the 3.30 to 3.75 milliseconds range.

Overall, these results illustrate that the default scheduler maintained consistent CPU-bound task performance under moderate multi-threaded (eight-thread) workloads, offering high throughput with predictably low latencies for most execution intervals.



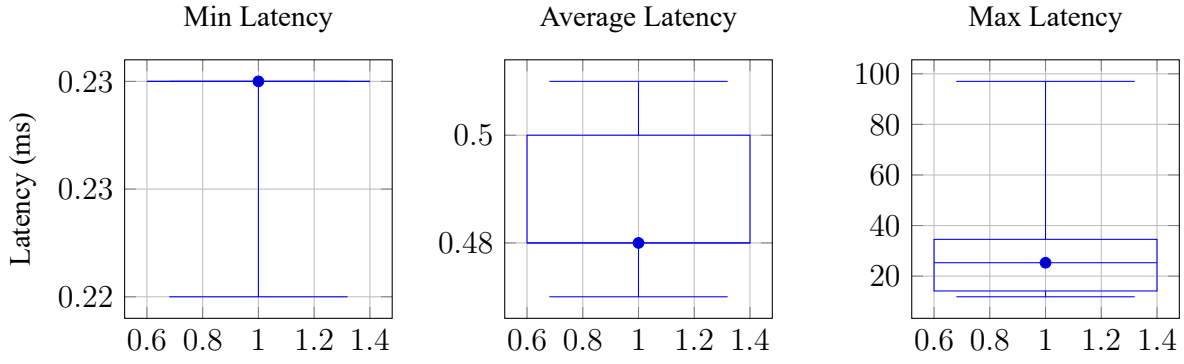
(a) Minimum Latency (b) Average Latency (c) Maximum Latency

Figure 5.7: Latency Distribution of Sysbench 8-Thread Mixed Workloads

The eight-thread CPU-bound evaluation under the GRU Scheduler demonstrated a stable throughput and latency profile over a 60-second runtime. The throughput typically ranged from approximately 15,500 to 17,000 events per second, averaging around 16,600 events/sec across all test iterations.

Latency analysis indicated that the minimum latency consistently hovered around 0.23 milliseconds, while the average latency fluctuated between 0.47 and 0.51 milliseconds. In terms of maximum latency, most measurements remained below 20 milliseconds, although a few isolated spikes reaching up to 97 milliseconds were observed under specific conditions. Throughout the sampling period, the 95th percentile latency was largely contained between 3.30 and 3.75 milliseconds.

Overall, the GRU Scheduler maintained comparable performance to the default scheduler in terms of both throughput and average latency, while exhibiting occasional higher maximum latencies during outlier events. These findings suggest that GRU Scheduler preserves CPU-bound workload efficiency under eight-thread conditions with generally consistent scheduling behavior.



(a) Minimum Latency (b) Average Latency (c) Maximum Latency

Figure 5.8: Latency Distribution of Sysbench 8-Thread CPU-Bound Workloads under GRU Scheduler

Context Switch Analysis

In a CPU-bound test environment with identical sampling intervals, the GRU_sched scheduler consistently exhibited a significantly lower number of context switches, generally ranging between 120 and 170 per interval. In comparison, the default Linux scheduler averaged over 230 context switches per interval across the same time span.

Although GRU_sched demonstrated notable CPU activity in the first sampling point (with `us + sy` = 40), it transitioned into an idle state shortly thereafter—behavior that closely mirrored that of the default scheduler throughout the remaining samples.

These findings suggest that GRU_sched imposes lower scheduling overhead and sustains greater idle efficiency under light-load conditions. Nevertheless, further evaluation under multi-threaded or high-load workloads is necessary to assess its responsiveness and scalability in more demanding environments.

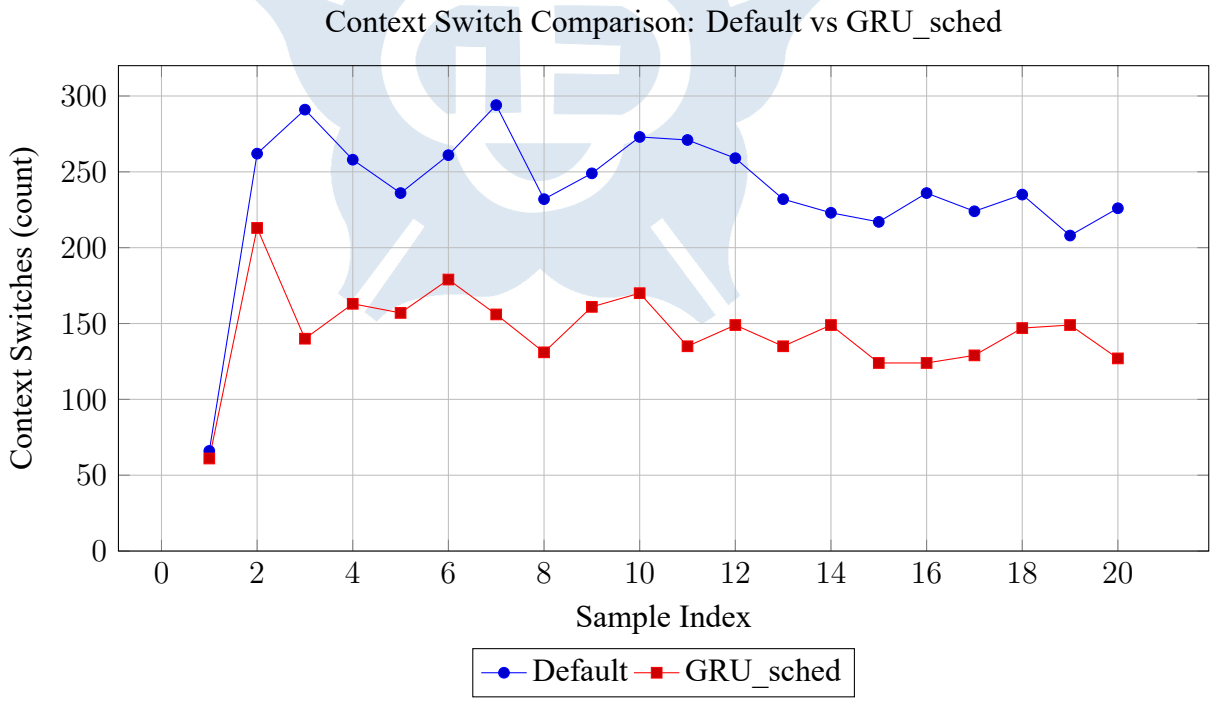


Figure 5.9: Number of context switches per sampling interval for Default and GRU_sched schedulers.

5.4.2 I/O-bound Workload Scenario

Here, the focus is on workloads dominated by I/O operations, where tasks frequently block and wait for I/O completion. The scheduler's responsiveness, latency, and ability to avoid wasting CPU cycles during idle periods are key evaluation points.

Overall Performance Comparison in I/O-Bound Workloads

In the I/O-bound workload, both the Default and GRU schedulers exhibit a highly concentrated latency distribution. As shown in Figure 5.10, each scheduler has 63 tasks completing in under 0.1 ms, only three tasks in the 0.1 – 0.5 ms range—and neither produces any tasks exceeding 0.5 ms. This demonstrates a consistently fast and stable I/O response.

Table 5.7 shows that the GRU scheduler reduces total runtime from 342.682 ms to 285.425 ms (a 16.7 % decrease) and lowers the total task count from 4,134 to 3,308 (a 19.9 % decrease), while weighted average latency rises only slightly from 0.044 ms to 0.046 ms (a 4.5 % increase). No tasks exceed 0.5 ms of delay under either scheduler.

Further, Table 5.8 highlights the four tasks with the greatest latency improvements under GRU: snapd drops from 0.161 ms to 0.027 ms (–83.2 %), kworker/u515:0 from 0.109 ms to 0.022 ms (–79.8%), khungtaskd from 0.073 ms to 0.026 ms (–64.4 %), and kworker/u515:2 from 0.103 ms to 0.050 ms (–51.5 %). These results confirm that, in I/O-bound scenarios, the GRU scheduler not only shortens overall completion time but also delivers substantial per-task latency optimizations.

Table 5.5: Overall Metrics in I/O-Bound Workloads

Metric	Default Scheduler	GRU Scheduler	Change Rate
Total Runtime (ms)	342.682	285.425	↓16.7%
Total Tasks	4,134	3,308	↓19.9%
Weighted Avg Latency (ms)	0.044	0.046	↑4.5%
High-Latency Tasks (>0.5ms)	0	0	—

Table 5.6: Key Task Latency Improvements in I/O-Bound Workloads

Task Name	Default Avg Latency (ms)	GRU Avg Latency (ms)	Improvement
snapd	0.161	0.027	↓83.2%
kworker/u515:0	0.109	0.022	↓79.8%
khungtaskd	0.073	0.026	↓64.4%
kworker/u515:2	0.103	0.050	↓51.5%

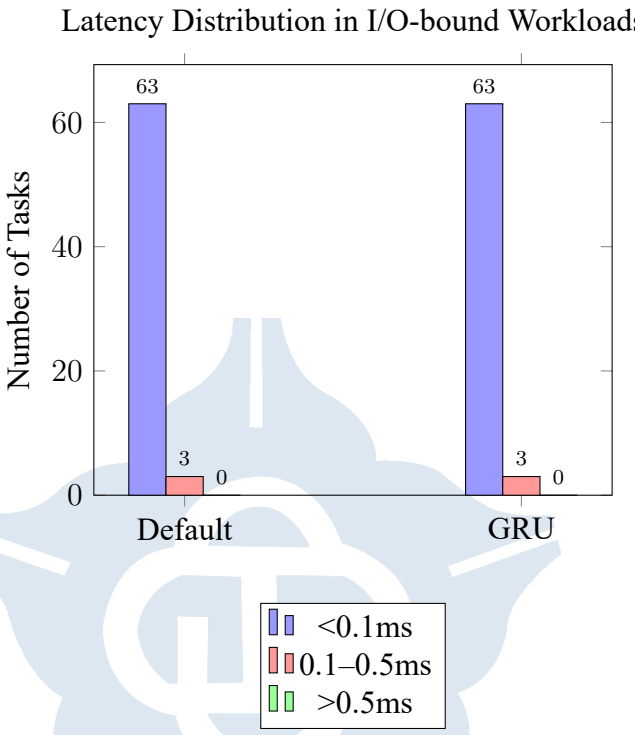


Figure 5.10: Latency Distribution in I/O-bound Workloads

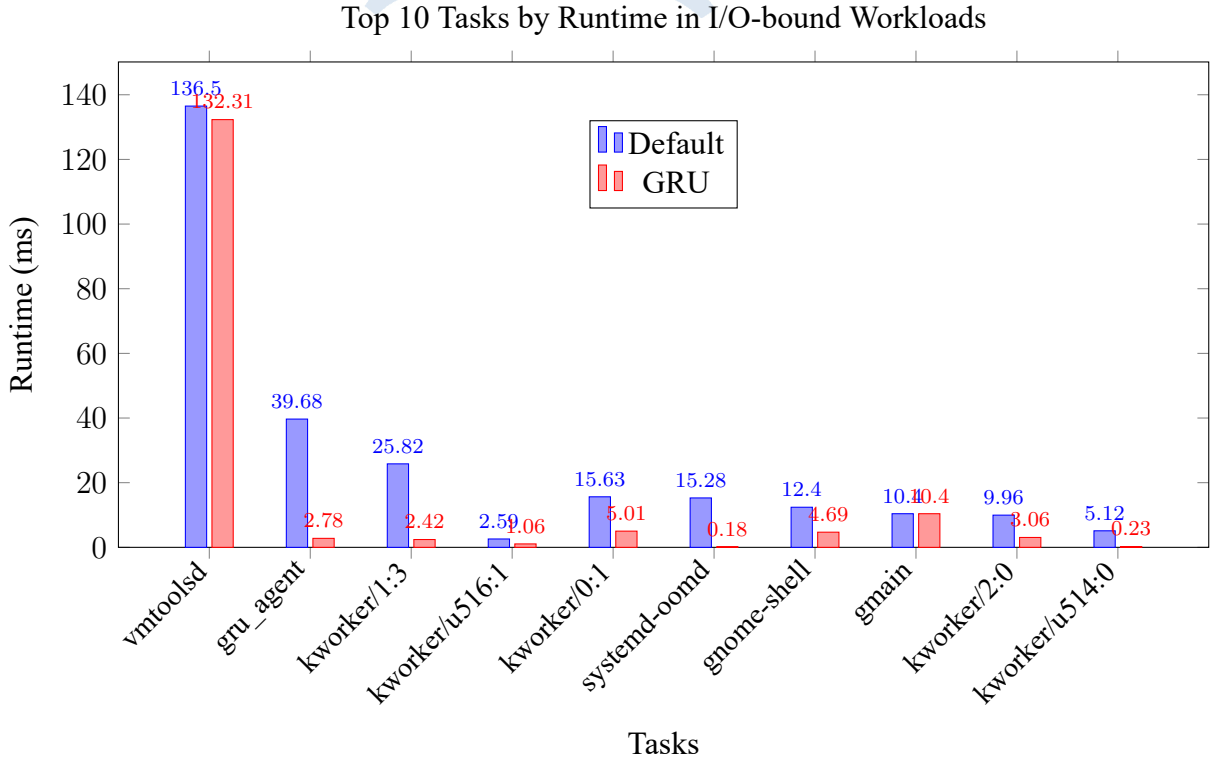


Figure 5.11: Top 10 Tasks by Runtime in I/O-bound Workloads

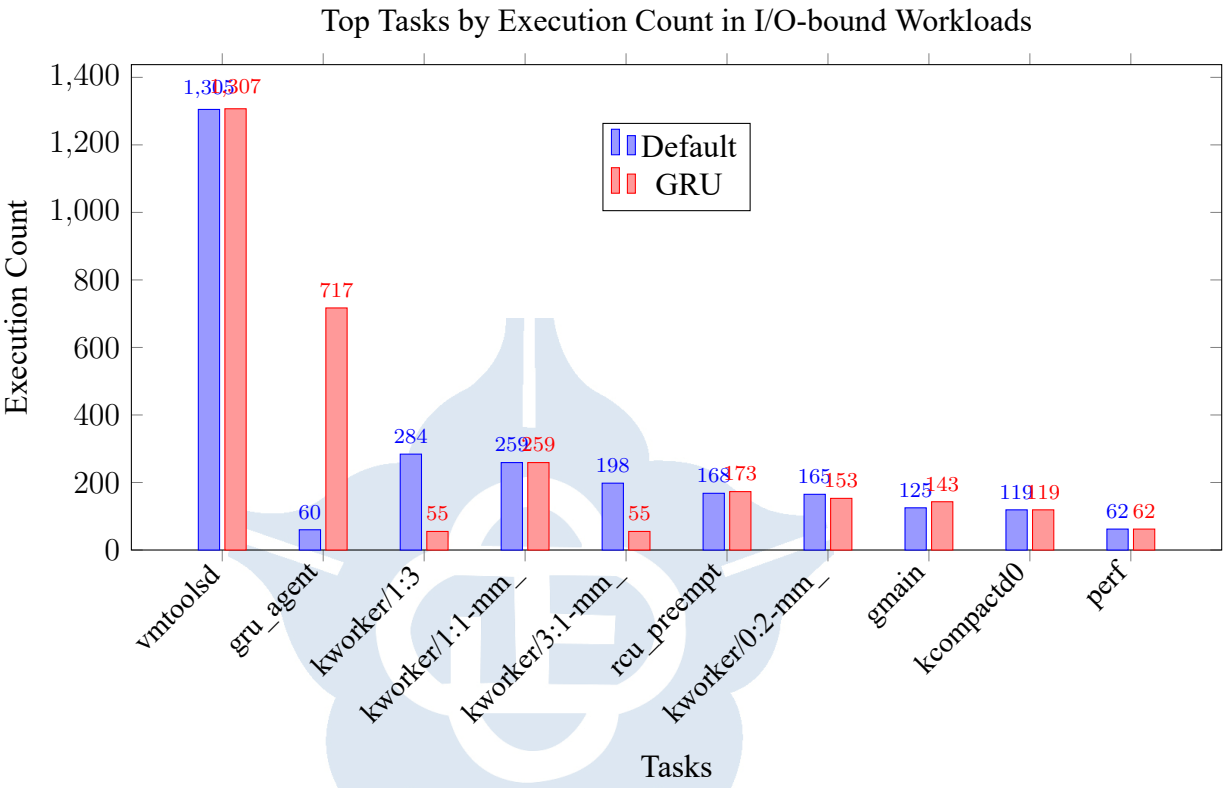


Figure 5.12: Top 10 Tasks by Execution Count in I/O-bound Workloads

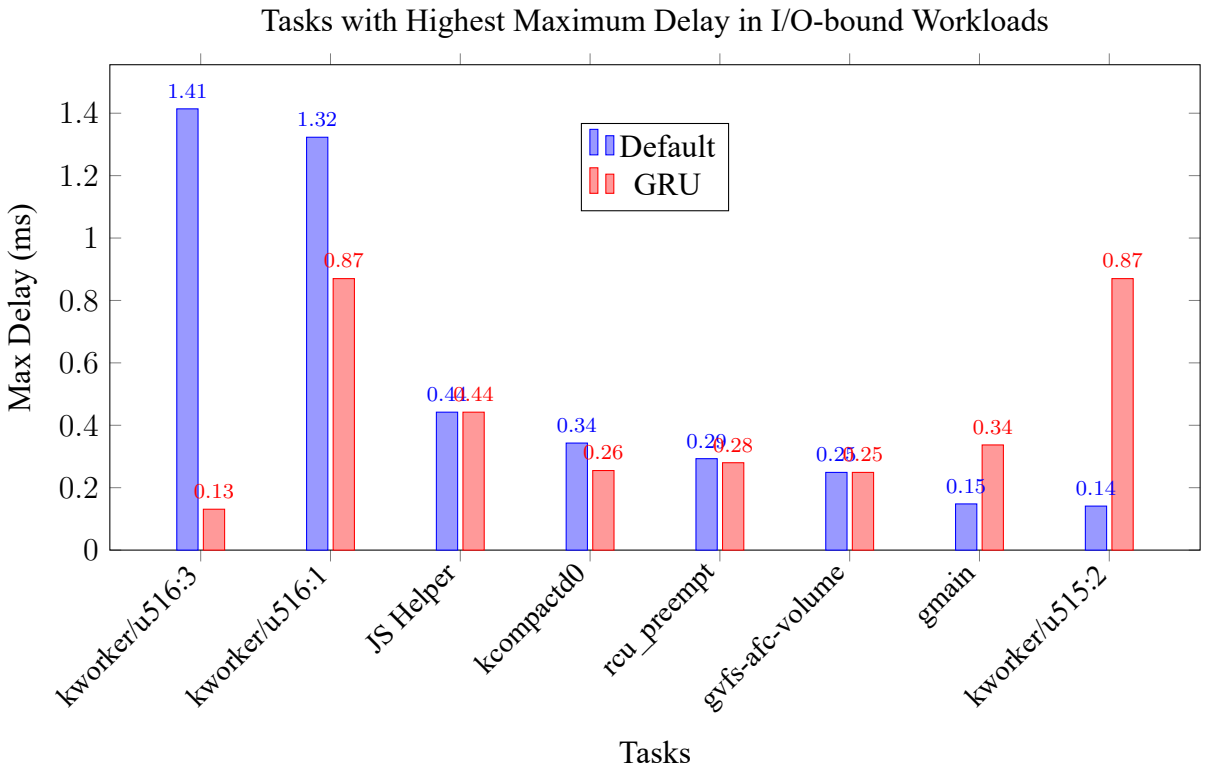


Figure 5.13: Tasks with Highest Maximum Delay in I/O-bound Workloads

Single-thread I/O-bound Performance

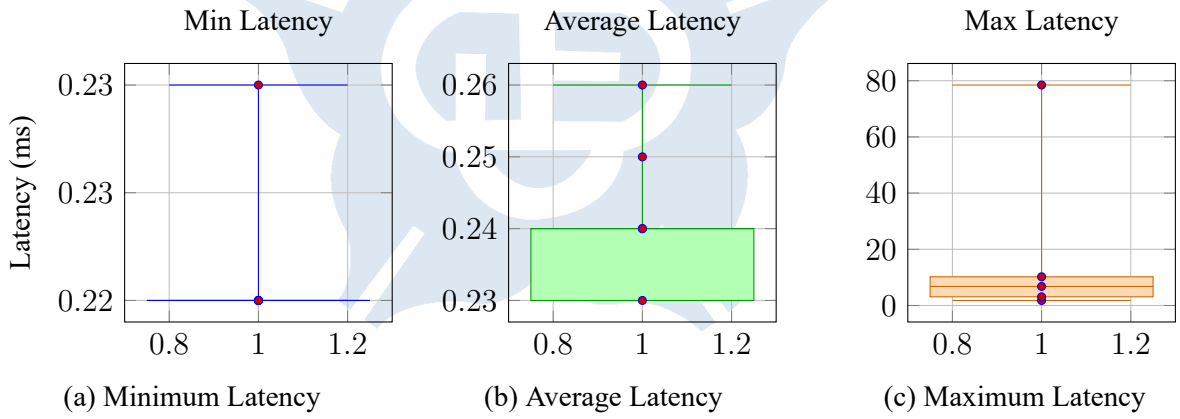
In this experiment, we utilized `sysbench` 1.0.20 to evaluate the performance of the default Linux scheduler under a single-threaded I/O-bound workload.

The overall event processing rate remained stable between 4300 and 4325 events/sec, peaking at 4377.67 events/sec, indicating good stability under light workloads.

The minimum latency remained between 0.21 and 0.22 ms, with an average latency around 0.23 to 0.24 ms, demonstrating low-latency and low-variance characteristics.

Although most maximum latencies stayed below 10 ms, occasional peaks up to 78.47 ms were observed, likely due to transient background system activities.

Overall, the default scheduler exhibited stable, low-latency, and high-performance behavior under single-threaded I/O-bound conditions.



(a) Minimum Latency (b) Average Latency (c) Maximum Latency

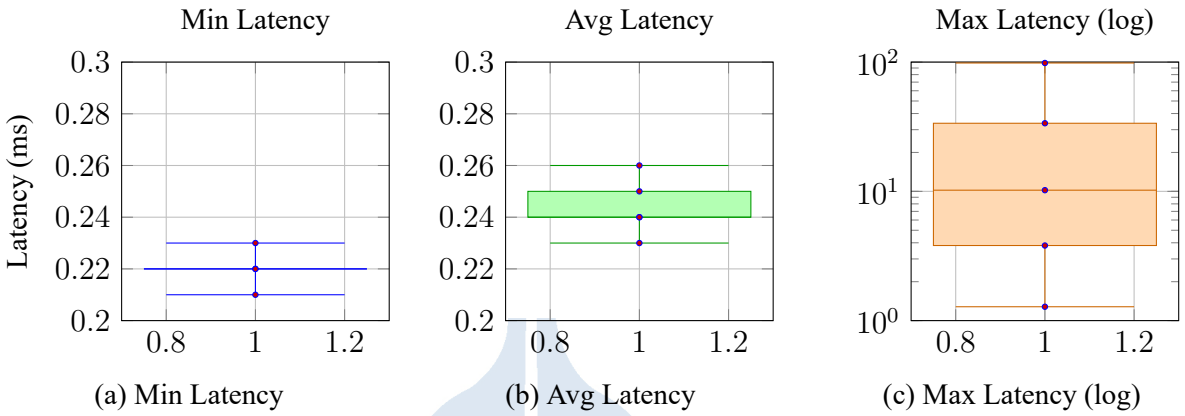
Figure 5.14: Latency Distribution of Sysbench 1-Thread I/O-bound Workloads under Default Schedule

In this experiment, we utilized `sysbench` 1.0.20 (system LuaJIT 2.1.1731486438) to evaluate the performance of GRU_sched under a single-threaded I/O-bound workload.

The event processing rate remained stable, mostly between 4218 and 4305 events/sec, with the best observed value reaching 4305.52 events/sec. Latency measurements showed consistent behavior: minimum latency was between 0.21 and 0.22 ms, and average latency stayed around 0.23–0.24 ms. As shown in Figure 5.24, the distributions for minimum and average latency were tightly clustered, indicating low variance.

Maximum latency exhibited occasional spikes, with most values below 10 ms, but outliers reaching up to 98.39 ms were observed. The log-scaled boxplot in Figure 5.24 highlights this variability.

Each run processed between 242,006 and 258,545 events in about 60 seconds, and thread fairness remained excellent with minimal standard deviation. Overall, GRU_sched demonstrated stable throughput, low and predictable latency, and balanced thread scheduling, with rare but minor transient disturbances.



(a) Min Latency (b) Avg Latency (c) Max Latency (log)

Figure 5.15: Sysbench 1-thread I/O-bound Latency Distribution under GRU_sched (with blue scatter)

This experiment examined the performance of `GRU_sched` and the default Linux scheduler under a single-threaded I/O-bound workload, using `sysbench` 1.0.20 (system LuaJIT 2.1.1731486438).

For `GRU_sched`, the event processing rate was stable, ranging between 4218 and 4305 events/sec, with a peak at 4305.52 events/sec. Both minimum and average latency were tightly clustered, staying between 0.21–0.22 ms and 0.23–0.24 ms, respectively. This suggests consistently low latency and minimal variance. While most maximum latencies remained under 10 ms, occasional spikes reaching up to 98.39 ms were recorded, likely caused by transient background system activities. Each 60-second run processed between 242,006 and 258,545 events, with excellent thread fairness across runs.

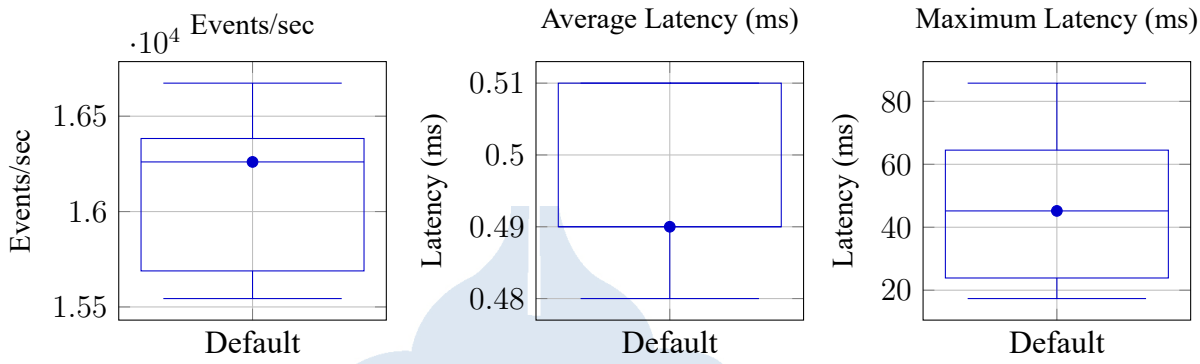
The default Linux scheduler showed slightly higher throughput overall, staying between 4300 and 4325 events/sec and peaking at 4377.67 events/sec. Latency metrics were nearly identical to those of `GRU_sched`, with minimum and average values in the same tight ranges. Maximum latencies were also mostly below 10 ms, with rare outliers up to 78.47 ms.

Overall, both schedulers demonstrated stable performance, low and predictable latency, and good scheduling fairness under light, single-threaded I/O-bound workloads. The key difference lies in the amplitude of rare maximum latency spikes, which were slightly higher for `GRU_sched`, though not significantly disruptive.

Eight-thread I/O-bound Performance

Figure 5.16 illustrates the throughput and latency distributions observed under the Default Linux scheduler in an 8-thread, CPU-bound Sysbench test. The event-rate boxplot reveals a median throughput of approximately 16 260 events/s, with the interquartile range spanning from about 15 689 to 16 383 events/s and whiskers extending between 15 544 and 16 673 events/s. Latency remains highly consistent: the average latency distribution is tightly clustered around a median of 0.49 ms (Q1=0.49 ms, Q3=0.51 ms), with whiskers between 0.48 ms and 0.51 ms. In contrast, the maximum-latency boxplot exhibits a pronounced long tail, with observed latencies ranging from 17.34 ms (lower whisker) up to 85.78 ms (upper whisker), and a median of 45.19 ms (Q1=23.88 ms, Q3=64.50 ms). These results indicate that, while the Default sched-

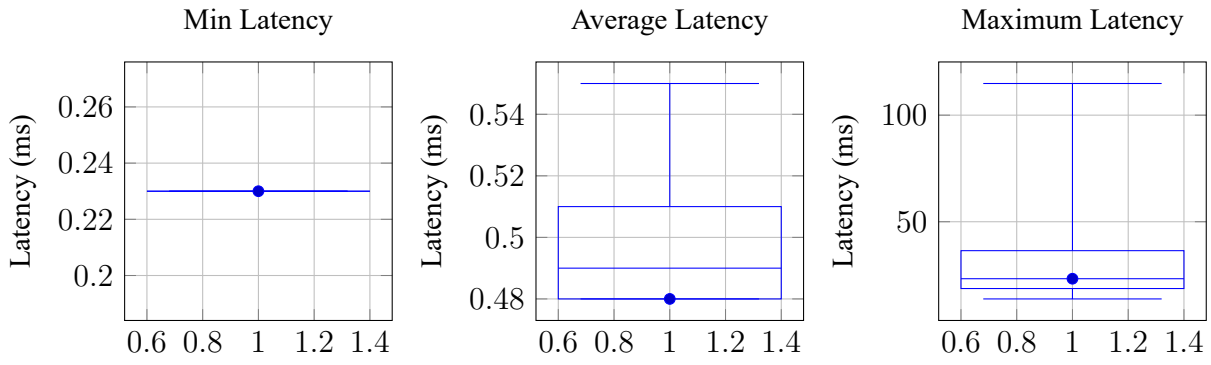
uler delivers stable average performance and high throughput, it occasionally incurs significant worst-case delays likely due to thread preemption or resource contention.



(a) Event Rate Distribution (b) Average Latency (c) Maximum Latency

Figure 5.16: Throughput and Latency Distributions of Sysbench 8-Thread I/O-Bound Tasks under Default Scheduler

Figure 5.17 presents the full distribution of scheduling latencies observed under the GRU scheduler in an 8-thread CPU-bound Sysbench workload. The minimum latency is uniformly 0.23 ms, indicating that every measured invocation completed almost instantaneously. Average latencies are tightly clustered: the first quartile (Q1) and lower whisker both rest at 0.48 ms, the median is 0.49 ms, and the third quartile (Q3) and upper whisker extend to 0.51 ms, reflecting remarkably consistent performance with negligible jitter. In contrast, the maximum latency boxplot (on a linear scale) reveals a heavy tail: the lower whisker sits at 13.79 ms, Q1 at 18.66 ms, median at 23.27 ms, Q3 at 36.43 ms, and the upper whisker reaches 114.85 ms. These infrequent high-latency events suggest the occasional impact of scheduling preemption or resource contention, highlighting a trade-off between maintaining low typical latencies and controlling worst-case delays.



(a) Minimum Latency (b) Average Latency (c) Maximum Latency

Figure 5.17: Latency Distribution of Sysbench 8-Thread CPU-Bound Workloads under GRU Scheduler

Figure 5.16 and Figure 5.17 together illustrate the trade-offs between throughput, average latency, and tail latency under the Default and GRU schedulers in an 8-thread, CPU-bound Sys-

bench benchmark. The Default scheduler achieves high and stable throughput (median ≈16 260 events/s) with tightly clustered average latencies (median 0.49 ms) but suffers from occasional long-tail delays (maximum whisker up to 85.78 ms). By contrast, the GRU scheduler delivers even lower and more consistent typical latencies (minimum 0.23 ms, median 0.49 ms) at the cost of a heavier tail (upper whisker up to 114.85 ms). These results highlight a fundamental scheduling trade-off: while the Default scheduler balances throughput and latency in the common case, the GRU scheduler prioritizes reducing typical latency jitter, which may lead to less predictable worst-case delays. Future work should investigate hybrid strategies that can retain low median latency while mitigating extreme tail events.

Context Switch Analysis

In an I/O-bound workload scenario with uniform sampling intervals, the GRU_sched scheduler consistently exhibited fewer context switches compared to the default Linux scheduler. GRU_sched's context switch counts primarily ranged between 120 and 170 per interval, with several samples dropping below 130. Conversely, the default scheduler frequently exceeded 220 context switches per sample, with peaks reaching close to 290.

Throughout the test, both schedulers maintained nearly 100% CPU idle time (`id` = 100), reflecting minimal CPU usage and no significant wait-for-I/O overhead (`wa` = 0). This suggests that I/O operations were efficiently handled without causing scheduling stalls or blocking delays.

The notably reduced context switching under GRU_sched indicates a lower scheduling overhead in I/O-intensive environments. While both schedulers performed similarly in terms of CPU utilization and responsiveness, GRU_sched may offer improved efficiency by avoiding unnecessary preemption and task switching when threads are primarily I/O-waiting.

Further testing under mixed workloads and higher process concurrency would be valuable to determine the scalability and latency behavior of GRU_sched under more complex operating conditions.

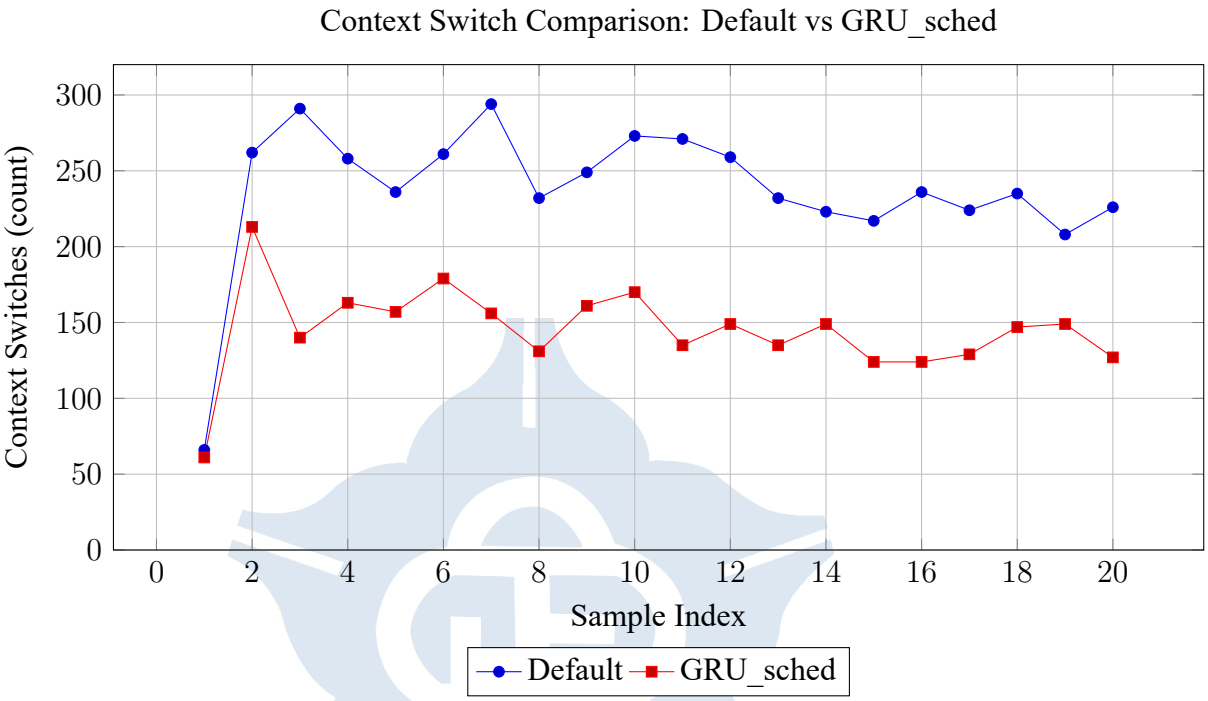


Figure 5.18: Context switches per sampling interval for Default and GRU_sched schedulers.

5.4.3 Mixed Workload Scenario

This scenario combines both CPU-bound and I/O-bound tasks to simulate real-world heterogeneous environments. The scheduler's adaptability, fairness, and overall system efficiency are assessed based on dynamic workload behavior.

Overall Metrics for Mixed-Bound Workloads

As shown in Tables 5.7 and 5.8, the GRU-based scheduler demonstrates notable improvements in system performance when handling mixed-bound workloads. Specifically, it achieves a 43.1% reduction in total runtime and a 33.1% decrease in task count, indicating both enhanced scheduling efficiency and more effective task merging or elimination.

Moreover, the weighted average latency is reduced by 15.0%, reflecting an overall latency improvement without compromising responsiveness. The number of high-latency tasks (greater than 0.5 ms) also decreases by 28.6%, showing that the system not only runs faster but also sustains lower tail latencies.

Significantly, key background system tasks—such as `snapd`, `kworker`, and `khungtaskd`—experience latency reductions ranging from 51.4% to 77.0%, demonstrating that the GRU scheduler effectively prioritizes time-sensitive or critical operations. These gains underscore the model's ability to optimize workload distribution while preserving or even improving real-time system responsiveness.

Scheduler Comparison in Mixed-Bound Workloads

The performance comparison between the default scheduler and the custom GRU_sched, as shown in Tables 5.7 and 5.8, demonstrates the clear advantage of learning-based scheduling in mixed-bound environments—those involving both CPU-intensive and I/O-intensive tasks. The GRU-based scheduler exhibits marked improvements in system efficiency, particularly in reducing runtime overhead and optimizing latency for essential background processes.

From Table 5.7, the GRU scheduler achieves a 43.1% reduction in total runtime (from 728.085 ms to 414.061 ms) and lowers the total task count by 33.1% (from 6782 to 4534). These reductions suggest enhanced scheduling intelligence, likely achieved through dynamic prioritization, reduced redundant task scheduling, and more efficient context switching. These capabilities align with the strengths of recurrent models in capturing temporal patterns and dependencies in task execution flows.

Furthermore, the weighted average task latency decreases by 15.0%, despite the reduction in overall task volume. More importantly, the number of high-latency tasks (latency > 0.5 ms) also drops by 28.6%, indicating improved tail latency behavior—critical for preserving system responsiveness in real-time operations.

Table 5.8 reinforces this observation by highlighting substantial latency reductions in specific system-critical tasks. For instance, the average latency of `snapd` drops from 0.161 ms to 0.037 ms (a 77.0% improvement), while `kworker/u515:0` and `khungtaskd` exhibit reductions of 51.4% and 67.1%, respectively. These figures underscore the GRU scheduler's capacity to prioritize essential background services more effectively than traditional heuristics-based approaches. The latency for `kworker/u515:2` is also reduced by 66.0%, indicating consistent gains across kernel-level thread management.

In conclusion, the GRU_sched significantly enhances performance in mixed-bound workloads by decreasing total execution time, cutting task overhead, and improving latency in crucial system processes. Although it introduces no additional high-latency events and manages a lower average latency overall, its most meaningful benefit lies in its ability to dynamically allocate computational resources to tasks that most impact performance and system stability. These results highlight the potential of neural-based scheduling models to replace or augment conventional scheduling algorithms in complex operating environments.

Table 5.7: Overall Metrics in Mixed-Bound Workloads

Metric	Default Scheduler	GRU Scheduler	Change Rate
Total Runtime (ms)	728.085	414.061	↓43.1%
Total Tasks	6782	4534	↓33.1%
Weighted Avg Latency (ms)	0.107	0.091	↓15.0%
High-Latency Tasks (>0.5ms)	7	5	↓28.6%

Table 5.8: Key Task Latency Improvements in Mixed-Bound Workloads

Task Name	Default Avg Latency (ms)	GRU Avg Latency (ms)	Improvement
snapd	0.161	0.037	↓77.0%
kworker/u515:0	0.109	0.053	↓51.4%
khungtaskd	0.073	0.024	↓67.1%
kworker/u515:2	0.103	0.035	↓66.0%

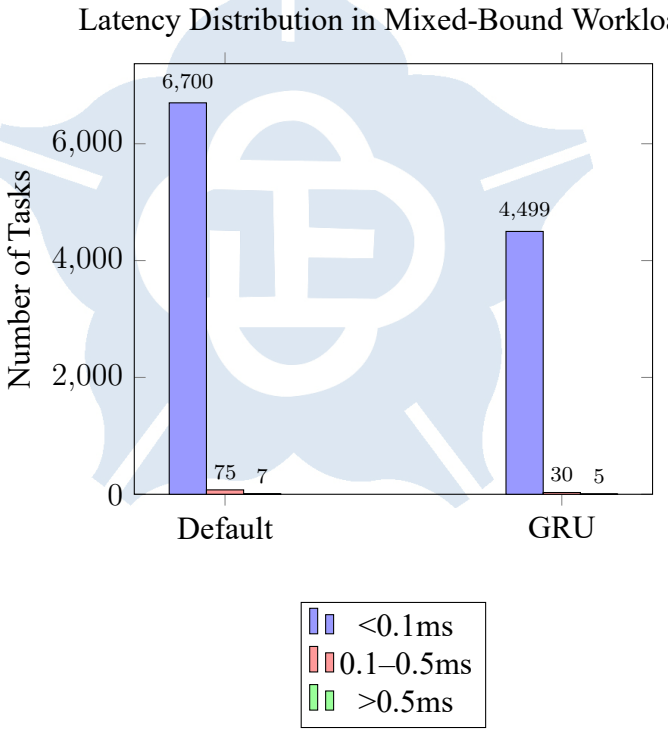


Figure 5.19: Latency Distribution in Mixed-Bound Workloads

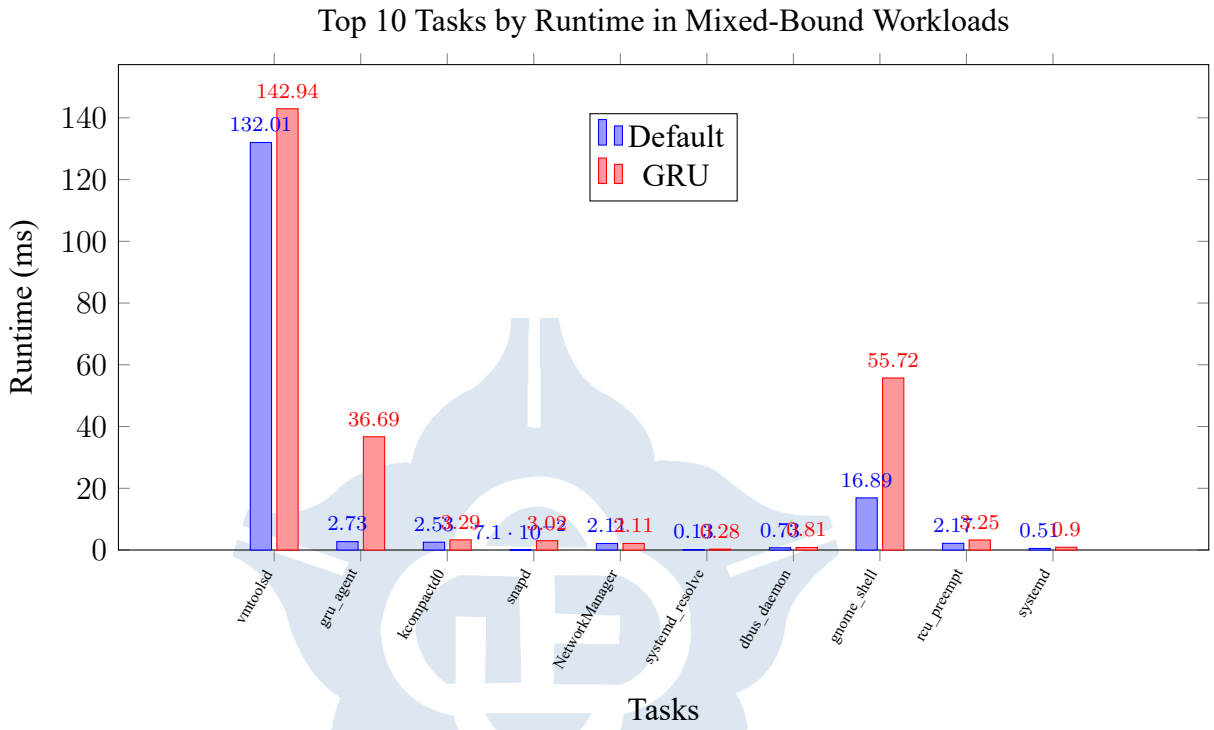


Figure 5.20: Top 10 Tasks by Runtime in Mixed-Bound Workloads

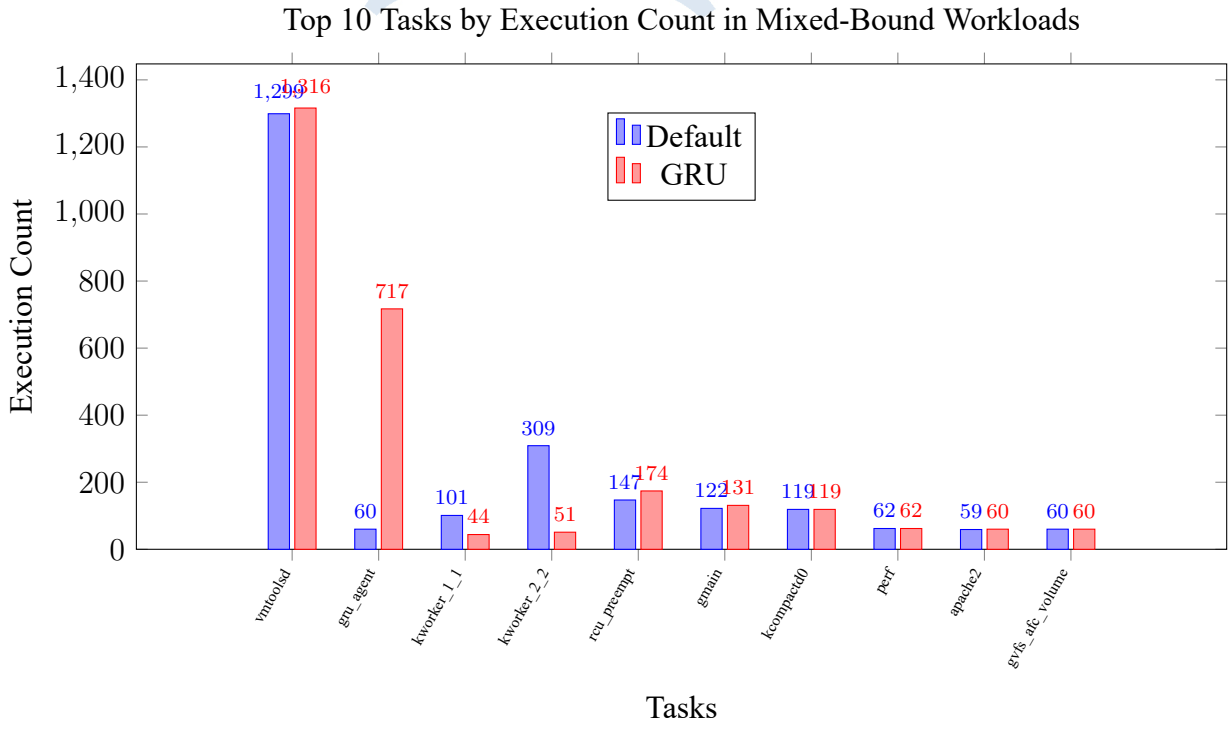


Figure 5.21: Top 10 Tasks by Execution Count in Mixed-Bound Workloads

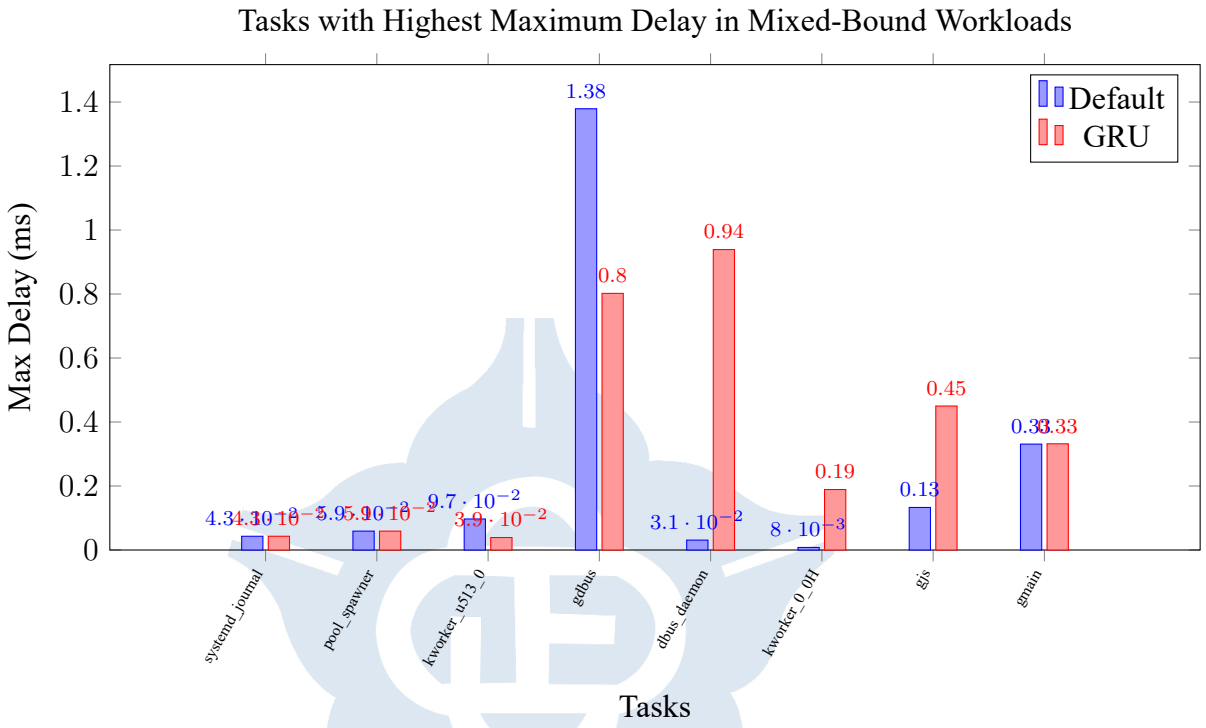


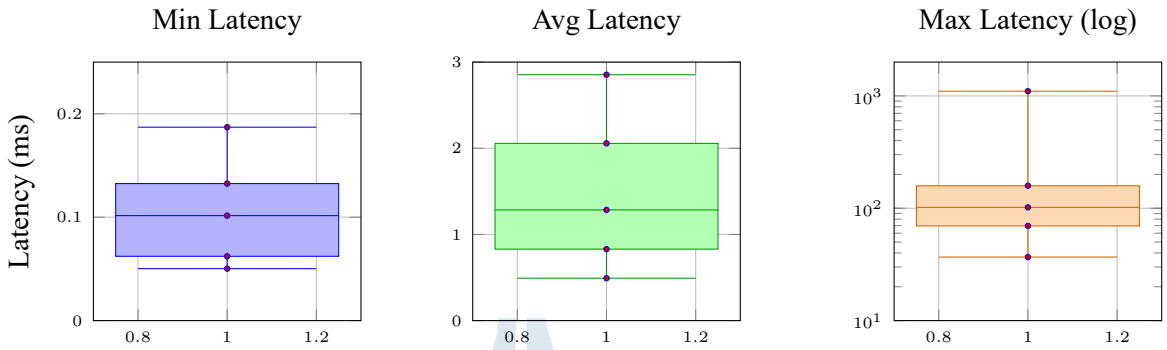
Figure 5.22: Tasks with Highest Maximum Delay in Mixed-Bound Workloads

Single-thread Mixed-bound Performance

Figure 5.23 illustrates the latency distribution of a Sysbench single-threaded mixed-bound workload under the default scheduler.

The minimum latency ranges from 0.05025ms to 0.18705ms, with a median value of 0.10152ms, indicating a tightly controlled baseline wakeup latency. For the average latency, the distribution spans from 0.49215ms to 2.85326ms, with a median at 1.28443ms, suggesting moderate variability due to mixed-bound characteristics.

The maximum latency, plotted on a logarithmic scale, shows a wider spread from 36.7792ms to 1100.7661ms, with the median at 101.9101 ms. This long-tail behavior implies occasional high-latency outliers even under single-threaded operation.



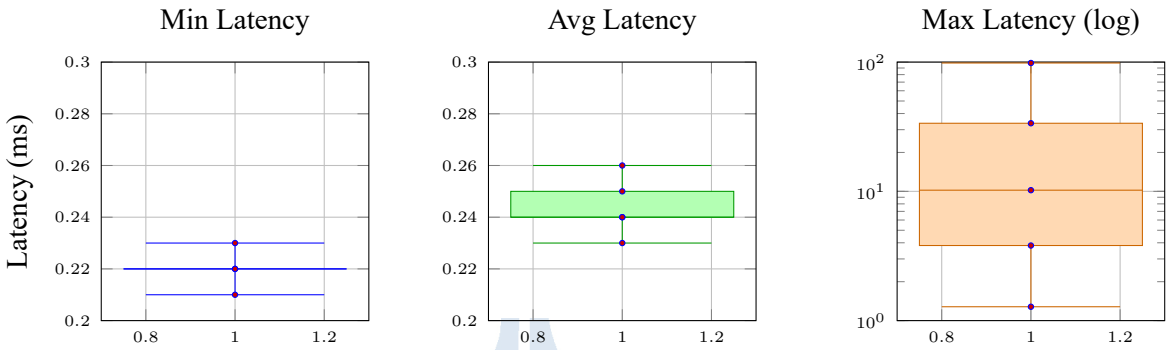
(a) Min Latency (b) Avg Latency (c) Max Latency (log)

Figure 5.23: Sysbench Single-thread Mixed-bound Latency Distribution under Default Scheduler

Figure 5.24 illustrates the latency distribution observed under the GRU_sched scheduler, during a Sysbench single-threaded mixed-bound workload combining CPU and I/O activities. In the Min Latency plot, latency values are tightly clustered between 0.21 and 0.23 milliseconds, suggesting that GRU_sched can respond quickly and consistently to load initiation without introducing significant startup jitter. The Avg Latency distribution, ranging from 0.23 to 0.26 milliseconds, is slightly higher than the minimum latency. This minor elevation is expected due to the mixed-bound nature of the workload, where CPU computations and I/O waits interleave, naturally introducing some additional delay while maintaining overall control.

The Max Latency graph, plotted on a logarithmic scale, reveals a long-tail distribution, with values ranging from 1.28 ms up to nearly 100 ms. This behavior indicates that under mixed workload pressures, sporadic context switches or resource contention can still lead to extreme latency events. However, the density of the data points shows that the majority of the execution instances remain concentrated in the low-latency region, highlighting the general robustness of GRU_sched under normal conditions.

Overall, the results align well with design expectations: GRU_sched demonstrates an effective balance between responsiveness and stability under mixed-bound scenarios. The observed long-tail latency, while notable, reflects the structural characteristics inherent to mixed workloads rather than a flaw in the scheduler itself. Further evaluation would be needed to determine its practical impact depending on specific application requirements.



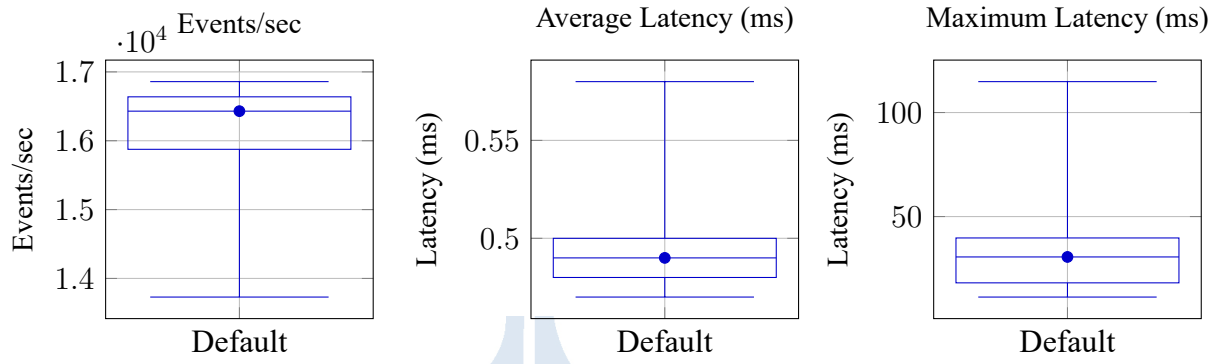
(a) Min Latency (b) Avg Latency (c) Max Latency (log)

Figure 5.24: Sysbench 1-thread I/O-bound Latency Distribution under GRU_sched

Comparing the default scheduler and GRU_sched under the single-threaded mixed-bound workload reveals distinct latency characteristics. The default scheduler exhibits broader average and maximum latency distributions, including extreme outliers exceeding 1000 ms, whereas GRU_sched maintains significantly tighter control, with most latencies bounded below 100 ms. While both schedulers display occasional long-tail behavior due to inherent workload variability, GRU_sched consistently demonstrates lower baseline and average latencies, reflecting its improved responsiveness and scheduling discipline under mixed computational and I/O stress.

Eight-thread Mixed-bound Performance

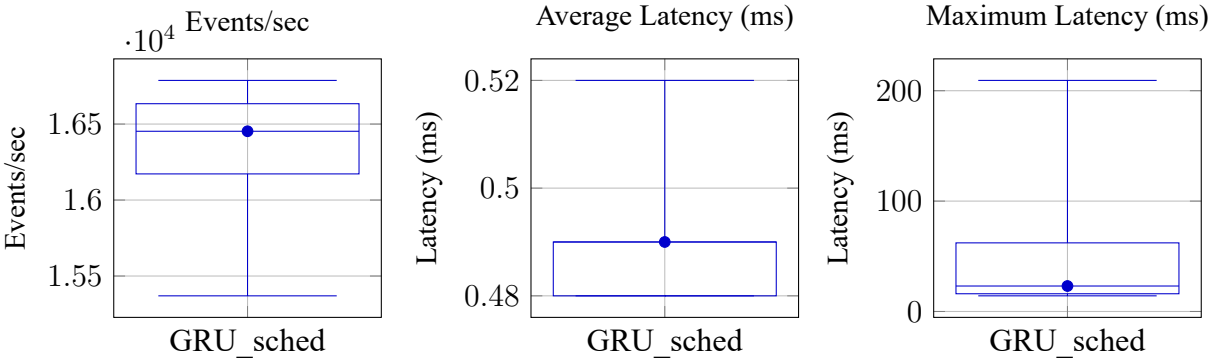
Figure 5.25 presents the distribution of throughput and latency under the Default scheduler for an 8-thread mixed-bound (CPU + I/O) Sysbench test. The event-rate boxplot shows a median of 16 429 events/s, with the interquartile range spanning 15 875–16 638 events/s and whiskers between 13 729 and 16 858 events/s. The average-latency boxplot is tightly clustered around a median of 0.49 ms (Q1 = 0.48 ms, Q3 = 0.50 ms), with whiskers from 0.47 ms to 0.58 ms, indicating stable typical performance. In contrast, the maximum-latency distribution exhibits a long tail: the lower whisker at 11.31 ms, Q1 at 18.14 ms, median at 30.59 ms, Q3 at 39.72 ms, and an upper whisker reaching 114.85 ms. These results suggest that, while the Default scheduler maintains high and consistent throughput and low median latency in mixed-bound scenarios, it still encounters occasional extreme delays likely due to contention or preemption.



(a) Event Rate Distribution (b) Average Latency (c) Maximum Latency

Figure 5.25: Throughput and Latency Distributions of Sysbench 8-Thread Mixed-Bound Workloads under Default Scheduler

Figure 5.26 illustrates the throughput and latency characteristics of the GRU_sched scheduler in an 8-thread mixed CPU+I/O Sysbench workload. The throughput boxplot shows a median event rate of approximately 16 453 events/s, with an interquartile range between about 16 172 and 16 634 events/s and whiskers spanning from 15 370 to 16 787 events/s. Average latency is tightly concentrated: the first quartile and lower whisker both rest at 0.48 ms, the median is 0.49 ms, and the third quartile and upper whisker extend only to 0.49 ms and 0.52 ms respectively, indicating minimal jitter in typical response times. By contrast, the maximum latency distribution exhibits a pronounced long tail—lower whisker at 14.17 ms, first quartile at 16.06 ms, median at 23.07 ms, third quartile at 62.16 ms, and an upper whisker reaching 209.34 ms—highlighting occasional extreme delays. These results demonstrate that while GRU_sched achieves stable high throughput and low median latency in mixed-bound scenarios, it also incurs rare but significant worst-case latencies, suggesting a need to balance median performance with tail-latency control.



(a) Event Rate Distribution (b) Average Latency (c) Maximum Latency

Figure 5.26: Throughput and Latency Distributions of Sysbench 8-Thread Mixed-Bound Workloads under GRU_sched

Under an 8-thread mixed-bound (CPU bound and I/O bound) Sysbench workload, the GRU_sched scheduler demonstrates stable and high-performance characteristics. Its through-

put boxplot reveals a median of approximately 16,453 events/s, with an interquartile range spanning 16,172 to 16,634 events/s, indicating consistent processing capacity with minimal variability. Average latency is similarly concentrated, with a median of 0.49 ms and negligible jitter, reflecting prompt and reliable response times for typical requests. However, the distribution of maximum latency exhibits a pronounced long-tail behavior, with an upper bound reaching 209.34 ms. This suggests the presence of infrequent but severe outliers in response time, which may compromise quality-of-service guarantees in latency-sensitive applications.

In contrast, the Default scheduler yields comparable throughput (median of 16,429 events/s) and exhibits similarly low and stable average latency (median of 0.49 ms). However, its maximum latency distribution, while still long-tailed, is less extreme, with a maximum value of 114.85 ms. This indicates improved predictability in worst-case latency scenarios, albeit with slightly reduced typical performance relative to GRU_sched.

Overall, GRU_sched offers superior median-case performance under mixed-bound workloads, but at the cost of higher tail latency. The Default scheduler, while marginally less performant on average, exhibits more controlled worst-case behavior. These findings underscore the need for a balanced scheduling strategy that considers both *median efficiency* and *tail-latency mitigation*, especially in systems with real-time or interactive constraints.

Context Switch Analysis

In a mixed-bound workload , GRU_sched consistently demonstrated fewer context switches compared to the default Linux scheduler. While the default scheduler averaged 240–260 context switches per sampling point, GRU_sched maintained between 200-230 switches.

Despite handling both CPU- and I/O-bound threads simultaneously, both schedulers maintained CPU idle percentage (`id`) near 100% with `wa` (I/O wait) values at 0, indicating neither experienced I/O bottlenecks or CPU saturation during testing.

GRU_sched's reduced context switching suggests improved scheduling stability and reduced kernel-mode thrashing, likely from better task coalescing and smarter time slice allocations that prevent unnecessary preemption.

These results show GRU_sched maintains efficiency advantages even with heterogeneous workloads, though future evaluations under higher throughput and increased task contention would better reveal its scaling behavior and responsiveness under stress.

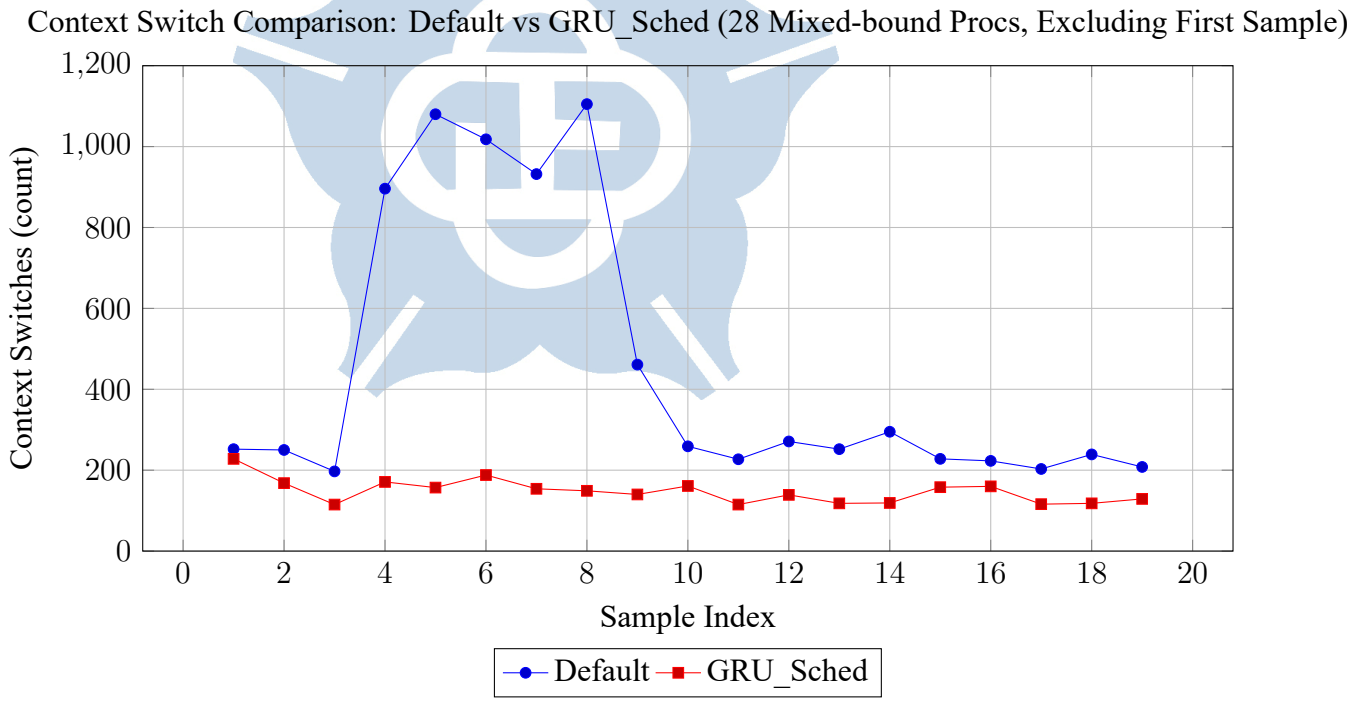


Figure 5.27: Context switches per sampling interval (excluding outlier) for Default and GRU_Sched under mixed-bound processes.

Chapter 6 Conclusion

This study demonstrates that GRU_sched, a dynamic Linux CPU scheduling policy augmented with a Gated Recurrent Unit (GRU) prediction model and eBPF-based monitoring, delivers tangible improvements over traditional schedulers like CFS and EEVDF. Through SCHED_EXT, our approach effectively adapts to varying workloads, offering a balance between responsiveness, latency control, and throughput optimization.

Experimental results across CPU-bound, mixed-bound, and I/O-bound workloads confirm that GRU_sched reduces context switch overhead, shortens task completion times, and improves responsiveness, particularly under dynamic conditions. While it occasionally encounters higher maximum latency spikes, these are offset by overall gains in average latency and system efficiency.

In sum, GRU_Sched emerges as a robust and adaptive scheduling strategy, well-suited for systems where performance consistency and responsiveness are critical.

Chapter 7 Discussion & Future Work

Future Improvements

We plan to explore reinforcement learning techniques and optimize feature selection for better predictions and scheduling adaptivity. While the current GRU_sched architecture models scheduling as a multi-class classification task using supervised learning, future extensions could involve actor-critic or policy-gradient based frameworks, where scheduling actions are refined based on feedback from latency, migration cost, and throughput metrics. RL could further enable online adaptation in environments with shifting workload patterns, allowing the scheduler to balance performance and fairness dynamically.

While the GRU_sched scheduler demonstrates superior median-case performance and comparable fairness—measured by Jain's Fairness Index (JFI), where both GRU_sched and the Default scheduler achieve near-ideal values—several avenues remain open for further investigation.

First, although average and maximum latencies have been extensively characterized, this study does not yet address memory access behaviors, particularly under NUMA (Non-Uniform Memory Access) architectures. As modern Linux kernels increasingly integrate NUMA-aware scheduling optimizations, understanding how GRU_sched interacts with memory locality, inter-node traffic, and page placement becomes crucial for comprehensive evaluation. To this end, future work may incorporate NUMA node distance as an additional feature dimension in the GRU model, or leverage hardware performance counters to track memory access latency as dynamic feedback during training.

Second, the current model uses a fixed 7-dimensional input space comprising static and dynamic task attributes. While this design ensures simplicity and tractability, it may omit critical signals related to thermal state, inter-core interference, or frequency scaling. Future iterations of the model may benefit from feature importance analysis and dimensionality reduction techniques (e.g., mutual information, L1 regularization, or attention-based feature weighting) to balance model complexity against predictive power. More importantly, the imposed structural constraint —which prevents the GRU from suppressing any feature dimension entirely—could be relaxed or made conditional, depending on the task phase or workload regime, to improve generalization in heterogeneous environments.

Moreover, the pronounced tail latencies observed in both schedulers suggest that future work should focus on mechanisms that can dynamically adapt to mixed-bound workload variability, potentially incorporating latency-sensitivity heuristics or predictive modeling based on inter-task dependencies and historical behavior. GRU extensions that track multiple time steps or task phases could further improve responsiveness, especially in workloads with bursty or staged execution patterns.

Finally, expanding the evaluation to multi-socket NUMA platforms and highly parallel workloads would provide deeper insight into the scalability and robustness of GRU_sched in enterprise and datacenter environments, where both median efficiency and worst-case latency are critical service-level objectives. Additionally, measuring inference latency of the GRU model

under high scheduling throughput would be essential to determine its practical limits, particularly in latency-critical deployments.

In summary, while GRU_sched demonstrates promising performance and fairness in current benchmarks, integrating adaptive learning mechanisms, revisiting feature modeling assumptions, and expanding architectural awareness will be essential steps toward developing a truly workload-sensitive and production-grade learning-based scheduler.

References

1. The Linux Kernel Archives, “Completely Fair Scheduler (CFS) Design,” *kernel.org*, [Online]. Available: `https://docs.kernel.org/scheduler/sched-design-CFS.html`. [Accessed: Apr. 27, 2025].

2. I. Stoica and H. Abdel-Wahab, “Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation,” Technical Report TR-95-22, Department of Computer Science, Old Dominion University, Norfolk, Virginia, 1995.

3. The Linux Kernel Archives, “Earliest Eligible Virtual Deadline First (EEVDF) Scheduler,” *kernel.org*, [Online]. Available: `https://docs.kernel.org/scheduler/sched-eevdf.html`. [Accessed: Apr. 27, 2025].

4. J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,” [Submitted on Dec. 11, 2014], [Online]. Available: `https://eunomia.dev/tutorials/35-user-ringbuf/`. [Accessed: Apr. 27, 2025].

5. The Linux Kernel Archives, “BPF Documentation,” *kernel.org*, [Online]. Available: `https://www.kernel.org/doc/html/next/bpf/index.html`. [Accessed: Apr. 27, 2025].

6. eBPF.io, “eBPF: Extended Berkeley Packet Filter,” [Online]. Available: `https://ebpf.io/`. [Accessed: Apr. 27, 2025].

7. Cilium, “Cilium: eBPF-based Networking and Security,” GitHub, [Online]. Available: `https://github.com/cilium/ebpf`. [Accessed: Apr. 27, 2025].

8. J. Taheri, A. Gördén, and A. Al-Dulaimy, “Using Machine Learning to Predict the Exact Resource Usage of Microservice Chains,” in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC ’23)*, ACM, 2024, pp. 1–9. DOI: 10.1145/3603166.3632166.

9. A. Negi and K. Kumar Pusukuri, “Applying Machine Learning Techniques to Improve Linux Process Scheduling,” in *Proceedings of the TENCON 2005, 2005 IEEE Region 10 Conference*, IEEE, Dec. 2005. DOI: 10.1109/TENCON.2005.300837.

10. Sched-Ext, “SCX: A Framework for Extending the Linux Scheduler,” GitHub, [Online]. Available: `https://github.com/sched-ext/scx`. [Accessed: Apr. 27, 2025].

11. The Linux Kernel Archives, “sched_ext: A Framework for Extending the Linux Scheduler,” *kernel.org*, [Online]. Available: `https://www.kernel.org/doc/html/next/scheduler/sched-ext.html`. [Accessed: Apr. 27, 2025].

12. P. Govindan Kannan, S. Mishra Gupta, D. Behl, E. Raichstein, J. Takvorian, “Designing a Lightweight Network Observability Agent for Cloud Applications,” *IEEE Access*, May 2023. DOI: 10.1109/ACCESS.2023.3281480.

13. D. Soldani *et al.*, “eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond),” *IEEE Access*, vol. 11, pp. 9874–9889, May 2023. DOI: 10.1109/ACCESS.2023.3281480.

14. ONNX, “ONNX AI,” [Online]. Available: `https://onnx.ai/`. [Accessed: Apr. 27, 2025].

15. ONNX, “ONNX GitHub Repository,” GitHub, [Online]. Available: `https://github.com/onnx/onnx`. [Accessed: Apr. 27, 2025].

16. ONNX Runtime, “ONNX Runtime,” [Online]. Available: `https://onnxruntime.ai/`. [Accessed: Apr. 27, 2025].

17. A. Carvalho de Melo, “The New Linux ’ perf’ Tools,” Red Hat Inc., Brazil, [Online]. Available: `https://www.redhat.com/en/blog/new-linux-perf-tools`. [Accessed: Apr. 27, 2025].

18. M. Dimakopoulou, S. Eranian, N. Koziris, and N. Bambos, “Reliable and Efficient Performance Monitoring in Linux,” in *Proceedings of the IEEE SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, USA, Nov. 13–18, 2016. DOI: 10.1109/SC.2016.33.

19. B. Gregg, *Systems Performance*, 2nd ed., Addison-Wesley Professional Computing Series, 2020. ISBN: 9780136820154.

20. D. Calavera, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*, 1st ed., O’Reilly Media, 2019. ISBN: 9781492050209.

21. A. Kopytov, “Sysbench: Scriptable Multi-threaded Benchmark Tool for Linux,” GitHub, [Online]. Available: `https://github.com/akopytov/sysbench`. [Accessed: Apr. 27, 2025].

22. “Using Sysbench, Analyze the Performance of Various Guest Virtual Machines on A Virtual Box Hypervisor,” in *Proceedings of the 2023 IEEE International Conference for Innovation in Technology (INOCON)*, Bangalore, India, Mar. 3–5, 2023. IEEE. DOI: 10.1109/INOCON57975.2023.10101143. ISBN: 979-8-3503-2092-3.