RETRACTED ARTICLE: Kernel-Level Dynamic Priority Scheduling for Containers

Thivya Rajkumar[1], Nishanth D.[2], Prabu M.[3], Sakthi Gobika S.[4], Vinothini A.[5]

Department of Artificial Intelligence and Data Science, Velalar College of Engineering and Technology, Erode, India.

E-mail: [1]thivyarajkumarvcet@gmail.com, [2]rsnishanthoz866@gmail.com, [3]msprabu12345@gmail.com, [4]sakthigobika4780@gmail.com, [5]vinoanbu2005@gmail.com

Abstract

The use of containerized cloud platforms has now become inevitable for latency-sensitive applications; but resource contention between different workloads often creates the problem of noisy neighbors, which causes higher response latency and poor service quality. We present SCX_MUS (Scheduler Extensions for Multi-User Scheduling), a dynamic priority scheduling framework at the kernel level that enhances the CPU allocation to latency-sensitive containers running in Kubernetes environments. The proposed framework uses Linux sched_ext architecture along with the extension of eBPF to support dynamic cgroup-based scheduling through runtime awareness. A lightweight user space component continuously observes the workload of Kubernetes and adjusts the scheduling priorities of the containers without any need for recompiling the kernel or restarting containers. The experimental evaluation of the proposed framework was done with the help of Redis and stress-ng workloads in the same benchmarking condition compared to the native Linux Completely Fair Scheduler (CFS). The experimental results reveal a significant improvement in terms of latency reduction, enhanced throughput, efficient utilization of CPU resources, and stable performance of scheduling under resource contention.

Keywords: Cloud Computing, Kernel-Level CPU Scheduling, sched_ext, Extended Berkeley Packet Filter (eBPF), Kubernetes Container Orchestration, Dynamic Priority Scheduling, Latency-Sensitive Workloads.

Journal of Soft Computing Paradigm,, September 2026, Volume 8, Issue 3, Pages 296-309
DOI: https://doi.org/10.36548/jscp.2026.3.008
Extended version of article received from **International Conference on Smart Innovations in Engineering Technology (ICSIET 2026)**
Received: 25.06.2026, received in revised form: 27.07.2026, accepted: 13.08.2026, published: 24.08.2026
© 2026 Inventive Research Organization. This is an open access article under the Creative Commons Attribution-NonCommercial International (CC BY-NC 4.0) License

1. Introduction

Containerization technology has been widely used for running cloud-native applications due to its efficient resource utilization. Kubernetes has been utilized for the orchestration of containerized applications within the shared computing environment. The combination of latency-sensitive applications and high-resource consumption usually leads to the noisy neighbor effect, which causes poor performance of sensitive applications due to heavy CPU competition. Classical Linux scheduling with CFS focuses on fair distribution of CPU resources rather than prioritizing applications, and static resource management approaches, such as CPU pinning and cgroups quotas, lack flexibility for dynamic workloads. Thus, an adaptive kernel-based scheduling approach should be used.

Several container scheduling techniques that have been used to enhance container efficiency, such as programmable container schedulers, eBPF kernel extensions, container isolation technologies, adaptive container schedulers, real-time kernel techniques, and QoS-aware resource management techniques. The above-mentioned techniques enhance scheduling and resource usage efficiency based on certain conditions. Nevertheless, majority of the current solutions depend on static resource allocation, user space scheduling, or complex optimizations. As a result, dynamic kernel-level scheduling with direct container awareness remains insufficiently explored.

In order to overcome this issue, this research work introduces SCX_MUS for dynamically scheduling priorities at the kernel level for Kubernetes platforms. The proposed framework combines the capabilities of sched_ext and eBPF to implement container prioritization dynamically without changing the code and without stopping and restarting the containers. This approach helps in reducing the scheduling interference and allocating the CPU efficiently for performing latency-sensitive tasks.

2. Related Works

Container-based cloud architectures have emerged as a potential solution for application deployment. However, these architectures also face challenges in terms of scalability and efficient resource utilization. Shared computing architectures are prone to resource contention issues, resulting in the noisy neighbor issue. With the advent of the sched_ext module, programmers

can write advanced scheduling policies for the Linux kernel that are more flexible than CFS. In addition to the sched_ext module, the eBPF technology offers a safe and lightweight way of writing kernel scheduling policies [1], [2]. Modular scheduling policy updating mechanisms help schedule policies be updated at runtime without compiling and rebooting the kernel [3].

Workload isolation efficiency continues to be an important requirement for ensuring the predictable performance of containers. Improved isolation techniques in the kernel improve the security of containers and reduce interference between colocated workloads [4]. From the perspective of orchestration layer, Kubernetes is considered as a powerful tool for managing the resources of containerized applications but the traditional approaches focus on balanced resource distribution and fail to meet performance needs of latency-sensitive workloads [5].

Several adaptive scheduling solutions are designed to optimize the container resource scheduling process. Reinforcement learning based scheduling optimizes the scheduling of workloads depending on resource availability; however, it results in higher computational overhead and needs to be trained [6]. On the other hand, PREEMPT_RT Linux kernel helps in deterministic scheduling, and lowers execution latency; however, it doesn't have any priority management for containers in Kubernetes environments [7].Scheduling strategies specific to applications have also been explored to boost the efficiency of clusters. The speculative scheduling method increases the efficiency of processing deep learning tasks based on predictions of future resource availability prior to scheduling [8]. Frameworks related to resource management for serverless computing focus on the aspect of dynamic allocation of resources in order to ensure scalability and quality of service.

Although considerable advancements have been made in programmable scheduling, container isolation, Kubernetes resource management, adaptive scheduling, and QoS optimization, existing solutions mostly rely on static resource allocation, user-space based mechanisms, or complex optimization techniques. As a result, a scheduling mechanism at the kernel level which will enable dynamic prioritization of latency-critical containers through sched_ext and eBPF is still required.

3. Proposed Methodology

The SCX_MUS framework enhances the implementation of latency-critical containerized applications in Kubernetes environments under resource constraints. The proposed solution

enables runtime priority-based scheduling by integrating Kubernetes-awareness into the existing sched_ext infrastructure of Linux and extended eBPF. Such an approach provides a combination of user-space controller and programmable kernel scheduler to detect important containers, change their scheduling priority, and assign CPU resources based on needs of applications. To enable direct communication between Kubernetes and the kernel scheduler using pinned BPF maps, scheduling logic is implemented without any changes in application containers, restarts of pods, and recompilation of the kernel. The flowchart of the proposed solution is depicted in Figure 1, which shows the interaction between Kubernetes control plane, user-space runtime controller, kernel scheduling module, and CPU dispatching processes.

3.1 SCX_MUS Framework Overview

The proposed SCX_MUS model comprises of four essential components that run one after the other to deliver the runtime-aware scheduling of CPUs. The entire process of operation of the proposed framework is illustrated in Figure 1. Firstly, the application containers are deployed inside a Kubernetes cluster in which latency critical services and computation intensive applications are running. The user-space runtime controller keeps on monitoring the Kubernetes system through the use of client-go API and finds those containers which need prioritized scheduling. Once a target container is found, its process identifier (PID) is determined and then linked to the relevant cgroup v2 identifier.

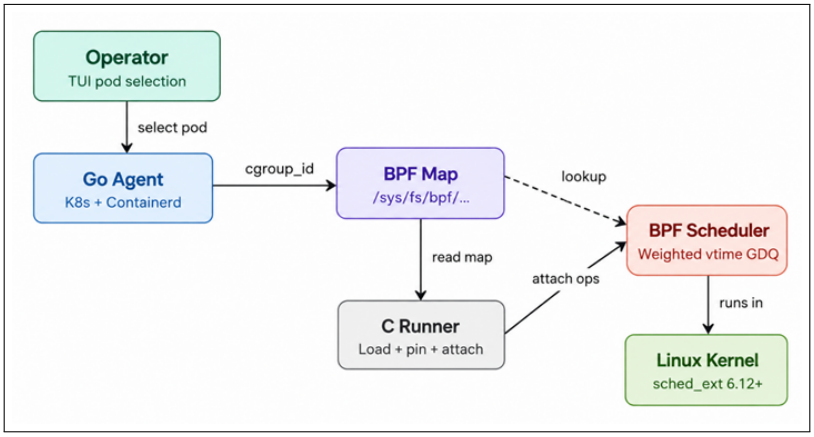


Figure 1. SCX_MUS framework architecture

During task scheduling, the sched_ext custom scheduler runs an eBPF program which checks the BPF map for checking if the arriving task is from a prioritized container. Containers with cgroup IDs registered by the users will have high scheduling priorities and will be scheduled

into the GDQ following the WVT-based scheduling algorithm proposed. Hence, delay-sensitive containers will have access to the CPU resources without any delays, and other background tasks will run as usual without needing any static partitioning of the CPUs among different tasks.

3.2 User-Space Runtime Controller

The user-space runtime controller is used as an intermediary between the Kubernetes orchestration engine and the kernel scheduler, as it constantly tracks the state of application execution. The lightweight controller that is implemented in Golang is used together with the client-go library of Kubernetes to track the lifecycle events of pods and recognize latency-sensitive containers that should be scheduled with priorities. In contrast to the traditional way of scheduling using fixed resource allocation, the new controller allows for adjusting the schedule depending on the state of the cluster.

Once a target container is identified, the controller resolves the corresponding host Process Identifier (PID) through the Container runtime interface. The associated cgroup v2 path is then obtained by reading the /proc/<pid>/cgroup file, and the kernel cgroup identifier is extracted using the stat () system call. This identifier uniquely represents the container within the Linux kernel and serves as the reference for scheduling decisions. The extracted cgroup ID is subsequently inserted into a pinned BPF hash map using the cilium/ebpf library, thereby establishing an efficient communication channel between the user-space controller and the kernel scheduler.

The BPF map stays pinned and persists throughout the scheduler operation, allowing the map to be updated at runtime without stopping container operation. As soon as a prioritized pod is created, migrated, or stopped, the BPF entry in the map gets updated right away by the controller, and thus the scheduling data will always reflect the present state of the cluster. This way of synchronization at runtime does not require the restarting of pods, recompilation of the kernel, or any manual updates of the configuration data.

3.3 Kernel-Level Dynamic Priority Scheduler

Following the runtime update of the BPF map pinned, the suggested SCX_MUS scheduler schedules CPUs through the Linux sched_ext module. In contrast to the default Completely Fair Scheduler (CFS), which schedules CPU time by considering the fairness of the tasks, the suggested scheduler determines the value of cgroup identifier for each new task

and dynamically defines the scheduling priority by looking at the records of the BPF map. At the task enqueueing stage, the eBPF scheduling program checks whether the task is part of any registered latency sensitive container. If the condition holds true for a particular task, the higher scheduling weight is provided, and the task is scheduled in the GDQ for the priority-based dispatching of the CPUs.

To differentiate critical and non-critical workloads, a scheduling weight is assigned to each task according to its cgroup membership. Let T_i denote a task associated with container C_K, and let M_{crit} represent the set of prioritized cgroup identifiers maintained in the pinned BPF map. The scheduling weight is defined as

$$W(T_i) = \begin{cases} 4096, & \text{if cgroup}(T_i) \in M_{\text{crit}}, \\ 1024, & \text{otherwise.} \end{cases} \tag{1}$$

An increase in the weight of the scheduler implies that the increment of the virtual execution time becomes slower for the task and hence, there will be more chances for it to get some allocation from the processors. After each execution interval, the virtual execution time of the running task is updated as

$$W(T_i) = vtime(T_i) + \delta_{\text{exec}} \times 1024 \tag{2}$$

where δ_{exec} denotes the execution, time consumed by task T_i, and $W(T_i)$ represents its assigned scheduling weight. As prioritized containers receive a greater weight during scheduling, their virtual run time will increase at a slower rate compared to regular containers. This means that these containers will continue being nearer to the top of the Global Dispatch Queue, and thus will be selected earlier for execution by the CPU in future scheduling instances.

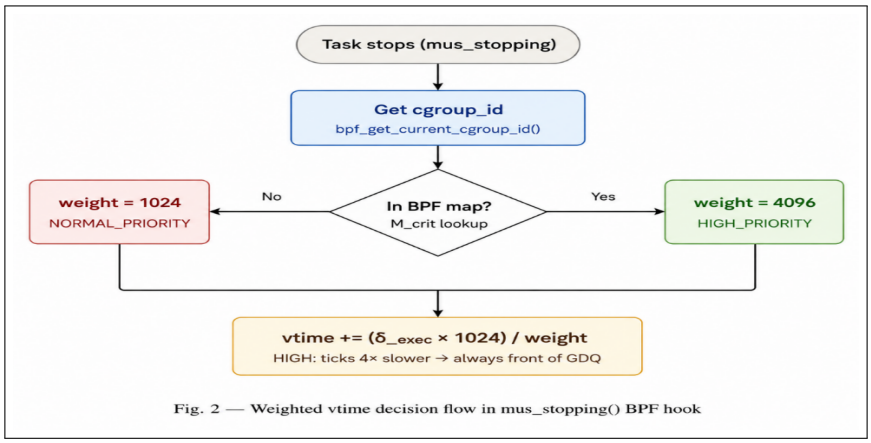


Figure 2. Weighted virtual time-based scheduling process in SCX_MUS

3.4 Runtime Dynamic Priority Management

Dynamic priority management helps SCX_MUS to adapt to any modifications in the workload executions without interrupting the current execution of containers. Since the states of containers keep on changing due to pod creation, migration, scaling, and deletion, dynamic priority management is required for providing preferential CPU scheduling for latency-sensitive workloads. The user-space runtime controller continuously tracks these events through Kubernetes API and ensures the synchronization of scheduling information maintained in the pinned BPF hash map.

Every time when there is an addition or deletion of a priority-based container, the cgroup id associated with it is added or removed from the BPF map without having to recompile the kernel or reload the scheduler or pod. As a result, the kernel scheduler runs at all times with the latest data at runtime and with the continued operation of latency-sensitive and non-latency-sensitive tasks.

The scheduling objective of the proposed framework is to minimize the waiting time experienced by prioritized containers while maintaining negligible scheduler overhead. This objective can be expressed as

$$\min L_{\text{tail}} = W_q + O_{\text{sched}} \tag{3}$$

where L_{tail} represents the tail, latency experienced by latency-sensitive workloads, W_q denotes the queue waiting time before CPU allocation, and O_{sched} represents the scheduling overhead introduced by the kernel scheduler. As the priority values are updated using lightweight BPF map calls, the scheduling strategy can be modified dynamically without incurring much computation

cost, thus allowing efficient CPU assignment even under varying workloads.

3.5 Implementation Environment

The proposed SCX_MUS was implemented using software components and experimental setup described in Table 1 below. The implementation was performed on Linux Kernel v6.12, which already includes the sched_ext framework that supports programmable kernel-level scheduling via eBPF. The scheduling mechanism was programmed using the C language, whereas the user space runtime controller was written using Golang programming language with use of client-go and cilium/ebpf libraries for Kubernetes and kernel interactions, respectively. The experimental setup was realized on a Kubernetes v1.29+ cluster running on Kind and included use of Redis as the latency-sensitive application and stress-ng to generate CPU-heavy background workloads, simulating noisy neighbors. The performance evaluation was carried out by using redis-benchmark with 50 clients performing 200K operations and evaluated using latency, throughput, CPU share distribution and scheduling overhead.

Table 1. Experimental configuration

Parameter	**Specification**
Operating Platform	Linux Kernel v6.12 with `sched_ext` support
Development Environment	eBPF (C), Golang, Clang/LLVM
Container Orchestration	Kubernetes v1.29+ (Kind)
Runtime Controller	`client-go`, Containerd, `cilium/ebpf`
Benchmark Workloads	Redis (latency-sensitive) and `stress-ng` (CPU-intensive)
Benchmark Configuration	`redis-benchmark` with 50 concurrent clients and 200,000 operations
Performance Metrics	P50, P95, P99 latency, average latency, throughput, CPU share, and scheduling overhead

4. Results and Discussion

The proposed SCX_MUS algorithm was tested with the help of CPU intensive container workloads to explore whether it is capable of enhancing the scheduling efficiency of latency sensitive applications running on the Kubernetes platform. For testing the performance, the

experimental system includes a Redis container running latency-sensitive tasks and some stress-ng containers producing continuous CPU contention, thus simulating the noisy neighbor effect. In order to analyze system efficiency, the latency, throughput, CPU resources management, scheduling efficiency, and the stability of execution were taken into consideration. Comparison experiments have been carried out by using Linux CFS as a baseline under the same conditions to test the effectiveness of kernel-based dynamic priority scheduling. The achieved results show the possibility of improving CPU scheduling.

4.1 Tail Latency Analysis

The tail latency is one of the key performance metrics which is needed to determine the responsiveness of latency sensitive applications in case of contention for CPU resources. Table 2 illustrates the comparative results of the default scheduler, which remains as the baseline comparison [1], and the SCX_MUS scheduling framework under the same workload conditions. Figure 3 shows the latency distribution for these cases. As we can see from the table, there were serious latency degradation issues on the default scheduler because of equal CPU sharing by contending workloads, which led to large delays in the Redis application.

Table 2. Performance comparison of CFS and SCX_MUS

Metric	**CFS**	**SCX_MUS**	**Improvement**
P50 Latency (ms)	1.815	0.863	52%
P95 Latency (ms)	5.071	1.463	71%
P99 Latency (ms)	860.67	2.27	99.7%
Avg Latency (ms)	14.04	1.41	89.9%
Throughput (ops/s)	3,455	28,571	+8.3×
Max Latency (ms)	2,030	951	53%

The most significant improvement was seen in the case of the P99 latency, which suggests that the proposed scheduling approach succeeded in dealing with extreme latencies due to noisy neighbor effect. The same pattern can be noticed in the case of P50 and P95 latencies, thus demonstrating that the proposed solution can not only to decrease worst-case execution time delays but also make the process of scheduling more responsive.

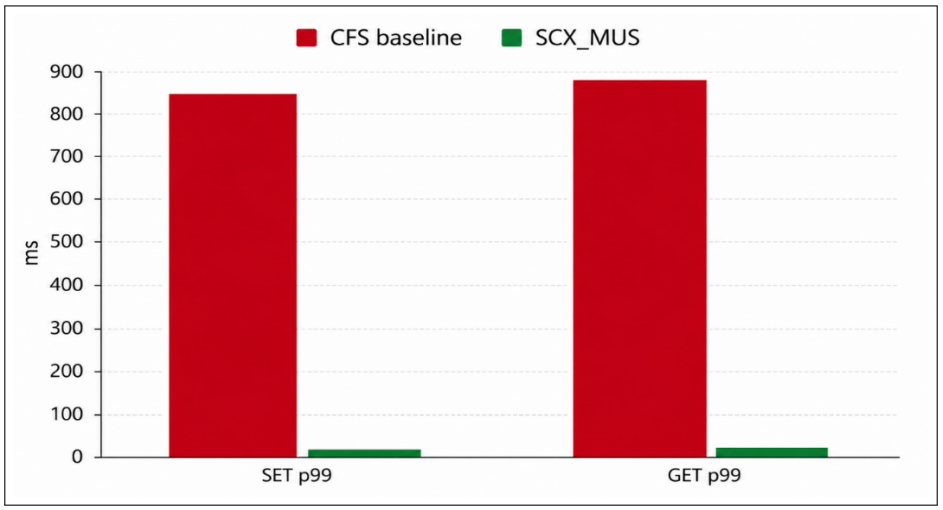


Figure 3. Redis tail latency comparison between CFS and SCX_MUS

4.2 Throughput Analysis

Throughput determines the ability of a scheduler to handle application requests in terms of resource conflicts. Figure 4 illustrates the throughput generated by the default Linux CFS and the new SCX_MUS framework when running the Redis benchmark test. The default scheduler experienced CPU conflicts as other competing workloads contended for CPU resources, which made the Redis application to process the requests at a lower efficiency. However, SCX_MUS framework maintained a higher throughput with the timely allocation of latency-sensitive requests.

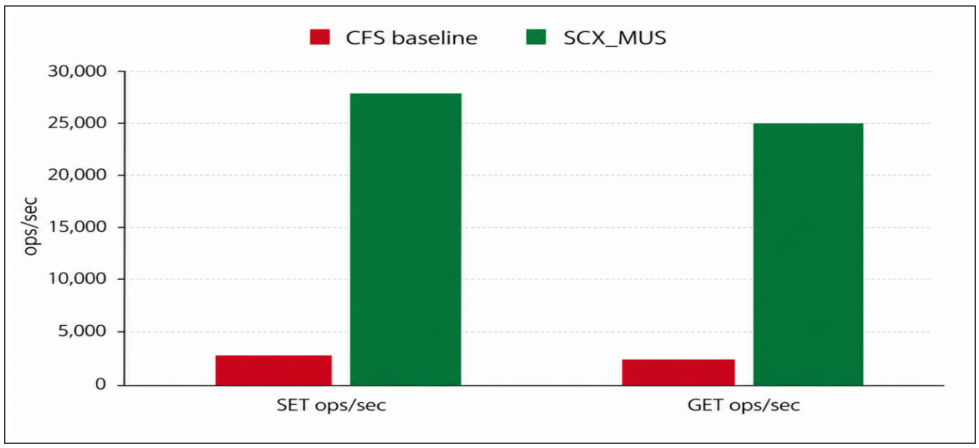


Figure 4. Throughput comparison between CFS and the proposed SCX_MUS scheduler

This increase in efficiency implies that the dynamic priority scheduling is effective in reducing the amount of waiting time for the processor to process critical tasks, making it possible to process more requests during the same runtime period. As the scheduling of tasks is done

by the kernel itself, there is an efficient utilization of processor resources without adding any scheduling overhead.

4.3 Runtime Scheduling Efficiency Analysis

Scheduling efficiency is critical to determine how well a CPU scheduler manages the allocation of processor resources with reduced runtime overhead without compromising application performance.

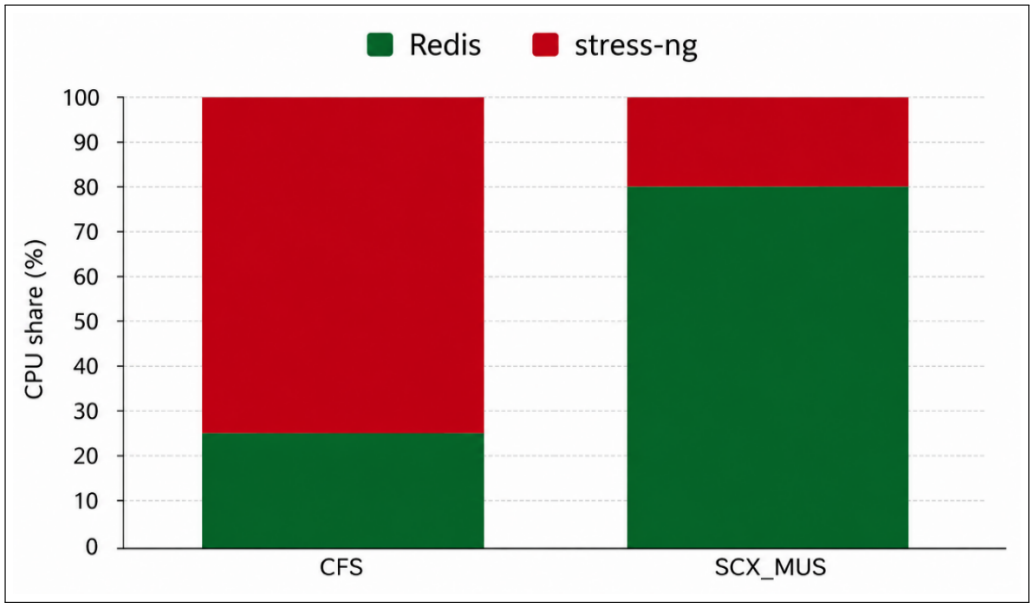


Figure 5. Scheduling efficiency comparison between CFS and SCX_MUS scheduler

The comparison presented in Figure 5 demonstrates that kernel-level dynamic priority scheduling provides efficient processor allocation while preserving low scheduling overhead under varying workload conditions. The lightweight communication between the runtime controller in user-space and the kernel-level scheduler ensures reduced overhead cost in the process of adjusting the priorities of the containers. Scheduling therefore remains dynamic despite the changes in the nature of workloads resulting from creation or termination of pods and competition for shared resources. Unlike traditional methods, which rely on static allocation of resources and reconfiguration of the scheduler, the proposed framework ensures continuous scheduling through runtime priority adjustment.

4.4 CPU Resource Distribution Analysis

Resource allocation among CPUs is important in ensuring the performance of latency-sensitive applications in shared container systems.

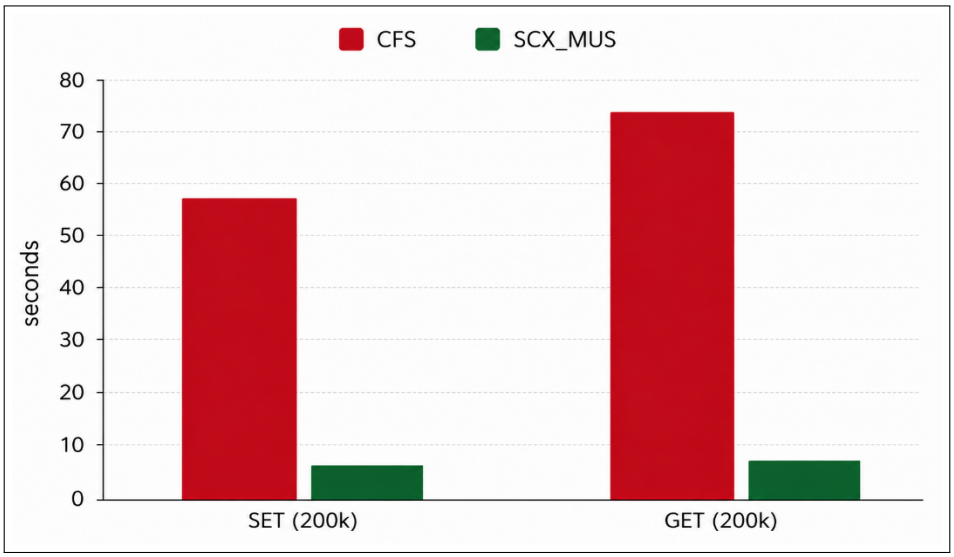


Figure 6. CPU resource distribution under the proposed SCX_MUS framework

Figure 6 depicts the CPU usage by Redis and stress-ng containers in the proposed SCX_MUS model when executing CPU-bound workloads. From the figure, it can be noted that resources were dynamically allocated based on priorities, allowing the Redis container to perform continuously without being dominated by competing workloads.

4.5 Runtime Stability Analysis

Execution consistency is another key feature of a scheduling framework due to the fact that consistent performance during several executions denotes that the framework is dependable amid changing workload scenarios.

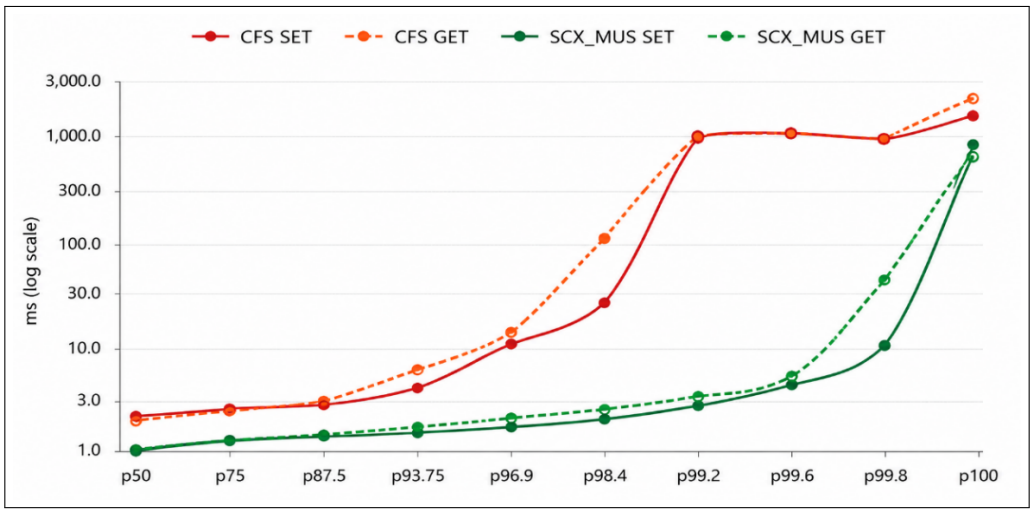


Figure 7. Runtime latency stability across multiple benchmark executions

The latency behavior of the proposed SCX_MUS scheduling framework has been depicted in Figure 7 in terms of several benchmark executions. The results obtained reveal that the scheduler executed with consistent behavior during each execution with little difference among them. As the scheduling priorities are adjusted without stopping the running containers, the proposed system is able to maintain a consistent allocation of processors irrespective of the varying workload levels.

5. Conclusion and Future Work

This research study proposed a kernel level dynamic priority scheduling framework named as SCX_MUS to enhance CPU scheduling in the container environment based on Kubernetes platform. This framework offers a scheduling approach that allows run-time aware dynamic priority-based scheduling where latency sensitive containers are identified and scheduled according to cgroups. The lightweight user-space controller and kernel-based scheduler work in collaboration to offer adaptive scheduling without necessitating any recompilation of the kernel, restarting of containers, and modification of applications. Evaluation of the proposed framework revealed that the framework successfully reduced latency, increased throughput, facilitated optimal CPU resource allocation, and offered stable scheduling performance under CPU intensive workloads when compared to the standard Linux CFS. This reveals that the SCX_MUS framework offers an efficient scheduling approach for latency sensitive cloud native applications. The future direction of this work would involve extending the proposed framework to cater for multi-node Kubernetes environments, adapting priorities according to workload, QoS integration, and heterogeneous platform support.

References

[1] Koneru, Sai Prabhath, Michkath Omanda Bouraima, and Rashedur Rahman. "Adaptive User-Space Scheduling in Linux: A Performance Study of sched_ext for Real-Time Systems." 10th International Conference on Communication and Electronics Systems (ICCES), IEEE 2025: 920–924.

[2] Vieira, Marcos AM, Matheus S. Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. "Fast Packet Processing with EPDF and XDP: Concepts, Code, Challenges, and Applications." ACM Computing Surveys (CSUR) 2026, vol. 53, no. 1: 1–36.

[3] Ma, Teng, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. "Efficient Scheduler Live Update for Linux Kernel with Modularization." In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems 2023, vol. 3: 194–207.

[4] Huang, Hang, Honglei Wang, Jia Rao, Song Wu, Hao Fan, Chen Yu, Hai Jin, Kun Suo, and Lisong Pan. "vkernel: Enhancing Container Isolation via Private Code and Data." IEEE Transactions on Computers 2024, vol. 73, no. 7: 1711–1723.

[5] Ferreira, Arnaldo Pereira, and Richard Sinnott. "A Performance Evaluation of Containers Running on Managed Kubernetes Services." IEEE International Conference on Cloud Computing Technology and Science (CloudCom) 2019: 199–208.

[6] Cao, Ronghui, Peng Zhang, Yiming Wu, Jun Liu, and Haibin Su. "Adaptive Container Scheduling based on Reinforcement Learning in Kubernetes," CCF Transactions on High Performance Computing 2025, vol. 7, no. 3: 291–304.

[7] Reghenzani, Federico, Giuseppe Massari, and William Fornaciari. "The Real-Time Linux Kernel: A Survey on Preempt_rt." ACM Computing Surveys (CSUR) 2019, vol. 52, no. 1: 1–36.

[8] Mao, Ying, Yuqi Fu, Wenjia Zheng, Long Cheng, Qingzhi Liu, and Dingwen Tao. "Speculative Container Scheduling for Deep Learning Applications in a Kubernetes Cluster." IEEE Systems Journal 2021, vol. 16, no. 3: 3770–3781.

[9] Mampage, Anupama, Shanika Karunasekera, and Rajkumar Buyya. "A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions." ACM Computing Surveys (CSUR) 2022, vol. 54, no. 11s: 1–36.

[10] Chen, Shuang, Christina Delimitrou, and José F. Martínez. "Parties: Qos-Aware Resource Partitioning for Multiple Interactive Services." In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems 2019: 107–120.