STUN: Reinforcement-Learning-Based Optimization of Kernel Scheduler Parameters for Static Workload Performance

Hyeonmyeong Lee, Sungmin Jung and Heeseung Jo *

Department of Computer Science, Chungbuk National University, Chungju 28644, Korea; myeong58@cbnu.ac.kr (H.L.); jjsm9595@cbnu.ac.kr (S.J.)
* Correspondence: heesn@cbnu.ac.kr

Abstract: Modern Linux operating systems are being used in a wide range of fields, from small IoT embedded devices to supercomputers. However, most machines use the default Linux scheduler parameters implemented for general-purpose environments. The problem is that the Linux scheduler cannot utilize the features of the various hardware and software environments, and it is therefore, difficult to achieve optimal performance in the machines. In this paper, we propose STUN, an automatic scheduler optimization framework. STUN modifies the five scheduling policies of the Linux kernel and 10 parameters automatically to optimize for each workload environment. STUN decreases the training time and enhances the efficiency through a filtering mechanism and training reward algorithms. Using STUN, users can optimize the performance of their machines at the OS scheduler level without manual control of the scheduler. STUN showed an execution time and improved FPS of 18.3% and 22.4% on a face detection workload, respectively. In addition, STUN showed 26.97%, 54.42%, and 256.13% performance improvements for microbenchmarks with 4, 44, and 120 cores for each.

Keywords: operating systems; performance; scheduler; tuning; reinforcement learning

1. Introduction

Modern Linux servers are used in various fields, from small-scale IoT devices to large-scale servers. Most IoT devices, such as Raspberry Pi [1] and high performance servers used in data centers, are based on the Linux operating system. Summit [2] is a supercomputer developed by IBM that has 2,414,592 CPU cores and 2,801,664 GB of memory and runs on the Red Hat Enterprise Linux operating system, which is a Linux distribution. Each device or server has a variety of hardware environments, from CPU cores to memory sizes, and their workloads also vary widely. Computational throughput is crucial for Summit to simulate artificial intelligence and machine learning models using millions of threads however, the responsiveness can be critical for other server machines.

CPU scheduling is a technique that determines the process to be executed next when there are multiple processes that can be executed. The optimization of the CPU scheduler used for a particular environment is an important issue for improving performance and reducing costs. It is well known that the performances of the workloads and machines can be largely dependent on the configuration of the scheduler [3]. However, most machines use the default scheduler configuration, which considers the generic hardware and software environment of Linux. Efficient scheduling has a significant performance impact on the entire Linux server, and therefore, a significant amount of research has been conducted to improve it. Most scheduling studies focused on reducing the scheduling overhead or modifying the priority operations to favor jobs under a particular situation.

Optimizing the scheduler is a difficult problem, requiring extensive expertise across the operating system. In addition, many factors affect the scheduler performance, such as the hardware, workload, and network, and how users operate a system. However, it
is difficult to understand all of the correlations between these many factors. Even if such a scheduler algorithm is implemented, scheduler measures include errors. It is therefore difficult to confirm whether the scheduler performance actually improves.

In this paper, we propose STUN, which automatically finds the optimal scheduler parameter values for the performance improvement of static workloads, whose workload patterns are rarely changed. Static workloads, such as batch-style or iterative jobs, are very common in servers. Genome analysis software, big data analysis system, and pattern recognition artificial intelligence are reasonable candidates. STUN is designed to improve the performances of workloads by adjusting the scheduling policy and parameters. In the Linux kernel, 5 scheduling policies and 14 scheduler parameters are defined. The performance of the scheduler can be improved by appropriately modifying these policies and parameters according to the hardware environment and the characteristics of the workloads. However, a large cost is consumed for individual optimization because the relationship between each parameter should be considered, and professional workload information and hardware expertise is required.

This paper makes the following contributions:

- **Transparency.** STUN is transparent for user applications because it tries to find the optimal parameter values of the Linux kernel scheduler. Therefore, users do not need to modify their conventional applications, and the process of STUN is performed automatically.
- **Reinforcement learning.** Differently from the previous studies, STUN works based on reinforcement learning. After adjusting the parameters of the scheduler, the performance is received as a reward. By using reinforcement learning, STUN can exquisitely optimize the scheduler policies and parameters.
- **Efficiency.** For an effective optimization process, STUN does not search for all scheduler parameters and can significantly shorten the search time by pre-filtering the parameters that affect the workload.

The structure of this paper is as follows. Section 2 examines and analyzes research cases for scheduler and parameter optimizations utilizing machine learning, and Section 3 clarifies the background technology for reinforcement learning used in STUN. Section 4 then describes the structure, operation, and characteristics of STUN. Section 5 describes the evaluation results of STUN performance. Section 6 provides some concluding remarks and areas of future study.

2. Related Work

2.1. Operating System Scheduler

Currently, the default Linux scheduler, i.e., Completely Fair Scheduler (CFS) [4], uses the concept of virtual runtime, aiming for an ideal and precise multitasking CPU, such that all tasks use the same CPU time. By contrast, the ULE [5] scheduler, the default scheduler of the FreeBSD operating system, is designed for a symmetric multiprocessing (SMP) environment in which two or more processors use one shared memory and allow multiple independent execution threads. It aims to improve the performance of the simultaneous multi-threading (SMT) environment. To this end, ULE has shown high performance in interactive scheduling by adjusting the interaction capability, priority, and slice size of each task independently. Bouron [6] ported the FreeBSD ULE scheduler to Linux and compared its performance with that of the CFS. As a result, it was confirmed that the two schedulers showed similar performance in most workloads, but ULE showed better performance for workloads with many interactive tasks.

Kolivas [7] argued that heuristics and tunable parameters used to improve the performance under specific environments degrade the performance. To reduce this scheduling overhead, they implemented a simple scheduling algorithm, the Brain Fuck Scheduler (BFS), which eliminates all computations for complex prioritization. BFS has improved responsiveness on Linux desktops with less than 16 CPU cores and is used as the default
scheduler for several Linux distributions, including PCLinuxOS 2010, Zenwalk 6.4, and GalliumOS 2.1.

2.2. Optimization of Parameters Using Machine Learning

Lama [8] proposed and developed AROMA, a system that automates the parameter configuration of Hadoop, a platform for big data analysis, to improve the service quality and reduce the cost. AROMA uses a support vector machine (SVM), a type of machine learning model, to optimize the parameters, allowing cloud services to be effectively used without inefficient parameter tuning.

Wang [9] automatically optimized the parameters of Apache Spark based on machine learning to improve its performance. There are over 180 parameters inside Spark, and 14 parameters that significantly affect the performance were optimized using a decision tree model algorithm. As a result of the performance evaluation, the average performance improvement of the initial setting was 36%.

In addition, studies have been addressed to improve the performance of Android scheduler. In [10], Learning EAS, a policy gradient reinforcement learning method for the Android smartphone scheduler EAS, was proposed. Learning EAS is applied on the characteristics of the running task, and adjusts the TARGET_LOAD and sched_migration_cost to improve the scheduler performance. The evaluation results on an LG G8 ThinQ show that Learning EAS reduces the power consumption by a maximum of 5.7% and improves the Hackbench performance by a maximum of 25.5%, compared with the default EAS.

This paper differs from the previous research in that we propose a novel reinforcement-learning-based internal parameter optimization of the Linux kernel scheduler. We use the Q-learning algorithm, which is lighter than deep learning and consumes less memory.

3. Background

3.1. Q-Learning

Reinforcement learning is a type of machine learning algorithm that establishes a policy to determine the behavior in which the sum of the total compensation values is at maximum in the current state. It is composed of the environment and agent, as shown in Figure 1. The agent, which is the subject of determining the next action to perform, interacts with the environment through actions, and the environment responds with the status and rewards. The agent repeatedly interacts with the environment to establish an optimal policy.

![Figure 1. Reinforcement learning structure.](image)

Reinforcement learning is suitable for solving problems that have trade-off relationships. In particular, the reward value of short- and long-term behaviors is clear. Thus, it is used in various fields, such as robot control [11] and gaming. It is well known that Atari [12] and DOTA 2 [13] are well trained through deep reinforcement learning.

There are many algorithms used to implement reinforcement learning, among which, we apply the Q-learning algorithm. The agent of Q-learning has a Q-table that records the Q-value, the value of all actions that can be taken in each state, to solve the problem. The Q-learning algorithm works as follows. When the agent starts Learning for the first time,
all values of Q-table are initialized to 0. If the Q-value is 0 in each state, the agent randomly chooses an action and updates the Q-value in the Q-table. If the Q-value is not 0, then the agent selects the action with the largest Q-value in order to maximize the reward. The agent repeats these steps until finding the best policy.

Q-learning is a type of model-free reinforcement learning which uses a Bayesian equation [14] to find the action with the highest total reward. In reinforcement learning, a model predicts the state change and compensation of the environment. Model-free reinforcement learning has the advantage of being easy to implement or tune.

**3.2. OpenAI Gym**

OpenAI Gym is a toolkit for developing and testing reinforcement learning algorithms developed by OpenAI, a non-profit AI research company [15]. OpenAI Gym provides an integrated environment interface called Env for reinforcement learning. The following functions are available in Env:

- **Render (self)** This function renders a frame within the environment and shows it as a GUI. In this study, it was not used because there was no need for GUI.
- **Reset (self)** This function returns the environment to its initial state. This function is typically applied at the end of each learning phase to proceed to the next step of learning.
- **Step (self, action)** This function executes an action one step at a time and returns the following three variables.
  - **Observation (object)**: This value is an environment-specific object representing an observation of the environment.
  - **Reward (float)**: This value represents the amount of compensation obtained through the previous action. The last goal in reinforcement learning is to maximize the sum of the total reward values.
  - **Done (boolean)**: This is a flag used to decide whether to reset the environment or finish learning.

OpenAI provides the interfaces for a variety of environments, ranging from simple algorithms to Atari games and 2D and 3D robot simulations. Our implementation is based on OpenAI Gym.

**3.3. Optimization Variable**

This section describes the scheduler policy and parameters, which are variables for optimization through reinforcement learning in STUN.

**3.3.1. Scheduler Policy**

Inside the Linux kernel, there are five scheduler policies currently defined: NORMAL(CFS), FIFO, RR, BATCH, and IDLE [16]. STUN can change these policies using schedtool, a tool provided by Linux, without having to reboot the server.

- **SCHED_NORMAL (CFS)** This is the default scheduler policy of the Linux kernel. The purpose of CFS is maximizing the overall CPU utilization and providing fair CPU resources for all tasks. CFS is based on each CPU run queue, the tasks of which run in order of the virtual runtime and are kept sorted by red–black trees.
- **SCHED_FIFO** As a fixed-priority scheduling policy, each task is executed with a priority value of 1 to 99, and it is a policy that preempts and executes CPUs in the order of high priority.
- **SCHED_RR** This is basically the same operation as SCHED_FIFO, although each task has a time quantum value, which is the maximum time of execution, and when the time quantum is expired, the task is switched to the next task in a round robin manner.
- **SCHED_BATCH** This policy is suitable for batch jobs. By avoiding preemption by other tasks, we can run a task longer and leverage the hardware cache better than other policies; however, this works poorly for interactive tasks.
• **SCHED_IDLE** This policy operates in a non-interactive manner, similarly to SCHED_BATCH. However, differing from SCHED_BATCH, SCHED_IDLE can be performed when other processes are idle.

3.3.2. Linux Kernel Scheduler Parameter

The Linux kernel provides 14 scheduler parameters for optimization. In this study, we applied an optimization on nine of the parameters, excluding five parameters that do not affect the performance. Table 1 shows the parameter value ranges that can change and the default values of the Linux kernel. They can change without a machine reboot using the command `sysctl` provided by Linux. The meaning of each parameter is as follows.

- **sched_latency_ns**: Targeted preemption latency for CPU bound tasks. Increasing this parameter increases the timeslice of a CPU bound task.

- **sched_migration_cost_ns**: This is the amount of time after the last execution that a task considered a hot cache in a migration decision. A hot task is less likely to be migrated to another CPU, and thus, increasing this variable reduces the task migrations. If the CPU idle time is higher than expected when there are runnable processes, it is recommended to reduce this value. If tasks bounce between CPUs or nodes too often, it might be better to increase it.

- **sched_min_granularity_ns**: Minimal preemption granularity for CPU bound tasks. This parameter is tightly related to sched_latency_ns.

- **sched_nr_migrate**: This controls how many tasks can migrate across processors for load-balancing purposes. As load-balancing iterates the runqueue with disabled interrupts (softirq), it can incur irq-latency penalties for real-time tasks. Therefore, increasing this value may give a performance boost to large SCHED_OTHER threads at the expense of increased irq-latencies for real-time tasks.

- **sched_rr_timeslice_ms**: This parameter can adjust quantum(timeslice) in a SCHED_RR policy.

- **sched_rt_runtime_us**: This is a quantum allocated to real-time tasks during sched_rt_period_us. Setting the value to −1 disables RT bandwidth enforcement. By default, RT tasks may consume 95% of the CPU resources per seconds, thereby leaving 5%, or 0.05 s, to be used by SCHED_OTHER tasks.

- **sched_rt_period_us**: This is the period over which real-time task bandwidth enforcement is measured.

- **sched_cfs_bandwidth_slice_us**: When CFS bandwidth control is in use, this parameter controls the amount of run-time (bandwidth) transferred to a run queue from the control group bandwidth pool of the task. Small values allow the global bandwidth to be shared in a fine-grained manner among the tasks, whereas larger values reduce the transfer overhead.

- **sched_wakeup_granularity_ns wakeup**: This is a wake-up preemption granularity. Increasing this variable reduces the wake-up preemption, reducing the disturbance of computing bound tasks. Lowering it improves the wake-up latency and throughput for latency critical tasks, particularly when a short duty cycle load component must compete with CPU bound components.
Table 1. Initial settings of 9 scheduler parameters and modifiable ranges.

| Parameter                | Default Value          | Range                        |
|--------------------------|------------------------|------------------------------|
| latency_ns               | 24,000,000             | 100,000~1,000,000,000 (1 s)  |
| migration_cost_ns        | 500,000                | 0~1,000,000,000              |
| min_granularity_ns       | 3,000,000              | 100,000~1,000,000,000 (1 s)  |
| nr_migrate              | 32                     | 0~                           |
| rr_timeslice_ms          | 100                    | 0~1000                       |
| rt_period_us             | 1,000,000              | 900,000~1,000,000            |
| rt_runtime_us            | 950,000                | 0~1,000,000                  |
| cfs_bandwidth_slice_us   | 5000                   | 0~1,000,000                  |
| wakeup_granularity_ns    | 4,000,000              | 0~1,000,000,000              |

4. Design of STUN

In this section, we describe the structure, operation, and characteristics of STUN in detail. The features of STUN can be summarized as a reward algorithm and parameter filtering.

4.1. Overview

STUN is designed to use the Q-learning algorithm. For reinforcement learning, we tried the deep Q-network (DQN), which is most widely used for reinforcement learning. However, we decided not to use DQN because it consumes a lot of computational power and memory, and if it uses a CPU only without a GPU, takes too long for learning [17]. Actually, it cannot search the optimal parameters of a kernel scheduler within a reasonable time. The Q-learning algorithm is lighter than deep learning and consumes less memory. Therefore, STUN adopts the Q-learning algorithm, and this is a better choice considering the implementation of STUN within the Linux kernel later.

Figure 2 shows the structure of STUN. STUN consists of an environment module and an agent module. The environment module defines the environment for kernel parameter optimization, and the agent module defines the Q-learning algorithm and executes the agent. The reset and step functions defined in the environment module are as follows.

- **Reset** (self, f = True) This function initializes the parameters to optimize. The reset is used to prevent the excessive accumulation of rewards for a specific parameter value during learning. It initializes the value randomly up to half of the total number of learning times and the rest of them at the values showing the best performance.
- **Step** (self, action): STUN returns a state, reward, and done after conducting a test workload with the changed parameter values according to an action value. The meaning of each variable is as follows.
  - **Action**: This is a value that determines the increase or decrease in each parameter. When optimizing n parameters, it has n + 1 values from 0 to n. STUN changes the optimized values by adding or subtracting a predefined value $\alpha$. 

![Figure 2. Structure of STUN.](image-url)
- **State**: This is an element used to indicate the value of the parameter to be learned for optimization. Each parameter is divided into equally 50 within the range that can be modified, and each state has a value of 0 to 49 and is expressed as \((s_1, s_2 \ldots s_n)\).

- **Reward**: As a result of the test workload, this is used to determine whether better performance is achieved than under the previous state.

- **Done**: This is an element used to finish this episode of learning for efficiency. The “done” flag is initialized to true. If the result of the test workload is less than 20% of the default performance, it returns false. If more than a certain number of false values consecutively occur, the episode ends.

The agent module defines the Q-table, which records the value of the action when a certain action is conducted under each state. STUN updates the Q-table by using the Q-learning model and the value function \(Q(s_t, a_t)\). The value obtained from the value function is the expected value that can be obtained in the future, rather than judging the good or poor values at the present moment, unlike the reward value. The Q-learning algorithm updates the expected value of the state in the Q-table each time it acts, and then selects the action with the largest expected value. The formula for updating the Q-value is as follows.

\[
Q_{t+1}(s_t, a_t) = Q(s_t, a_t) + \alpha_t \cdot (R_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t))
\] (1)

In Formula (1) above, \(s_t\) and \(a_t\), respectively, mean the current state and action; \(R_{t+1}\) is the reward value obtained by acting in the current state; and \(\max_a Q_t(s_{t+1}, a)\) is the expected value obtained when the most rewarding action is assumed in the future. Here, \(\gamma\) is the discount rate reflecting that the value of the present reward is higher than that of the future reward. Since the future reward has a lower value than the present reward, it is reflected by multiplying it by \(\gamma\), a value between 0 and 1.

In the agent module, the number of learning iterations is determined by the number of episode executions (N) and the number of step function executions (T) for the agent. The parameter values are changed and learned by executing the step function as much as T within a single episode. Therefore, the agent has up to \(N \times T\) learning iterations.

### 4.2. Parameter Filtering

Not all of the parameters affect the performance, even if the values change according to the scheduler policy or the workload. For example, Figure 3 increases the sched_rr_timeslice_ms value by 10 within the range of the FIFO policy and records the resulting value by executing Sysbench. The x-axis represents the parameter value and the y-axis represents the total number of events resulting from Sysbench. In the FIFO policy, CPUs are occupied until termination in the order in which all jobs come in. In addition, sched_rr_timeslice_ms is a parameter that determines the CPU usage time for each task in the RR policy. As shown in Figure 3, Sysbench represents a constant value of between 2500 and 3000, regardless of the parameter value, which can be from 0 to 1000.

The filtering process shortens the training time by minimizing unnecessary training for performance improvement, and reduces memory usage to perform optimization more efficiently. Using all five scheduler policies and nine parameters in the Q-learning algorithm wastes a large amount of memory and increases the learning time. It is therefore necessary to find a parameter that significantly affects the performance. To achieve an efficient learning, STUN selects the scheduler policy and parameters for optimization through a filtering process.

STUN changes the individual parameter values to the minimum, maximum, and default values one by one for each policy and records the performance changes in the test workload. At this time, the other parameters are fixed as the default values. The filtering process removes the parameters that do not affect the overall performance of the application. To identify the parameters, the filtering process uses a threshold value as 20% due to the measurement error. As there may be an error range of 10–15% depending on the system
situation, we decided the performance difference of 20% is meaningful, and the parameter is tested for the optimal value.

Figure 3. Performance change according to the increase in sched_rr_timeslice_ms based on the FIFO policy.

Throughout the filtering process, the relationships between these policies and parameters are known, and thus only the parameters that most affect the performance as learning variables are used. As a result, the filtering process shortens the learning time and reduces the memory usage, allowing the optimization to be achieved more efficiently.

4.3. Reward Algorithm

A reward is a value that indicates whether the workload performance is improving. By subdividing and applying a reward, STUN can update the Q-table more effectively, and the learning time can be reduced.

Some ideas of the reward and punishment rules in a STUN reward function are as follows:

- Give a high reward for substantial improvements in performance
- Give a penalty if the performance degrades significantly
- Give a reward based on the previous performance

To represent the rules of an algorithm, default_bench is a variable used as a performance standard. This is the test workload result under the Linux default settings without changing the parameters. Note that the result means the performance of test workload. As in the filtering process, the reward is different based on the results greater than 20% and less than 20% in order to check whether the performance was significantly affected. A result that is improved by 20% more from default_bench is set to upper, and a result of below 20% is set to under. If the result is better than upper, the reward is 200, a large reward; and if it is lower than under, the reward is −50, which is a penalty. When the performance changes within the range of 20%, if the performance is higher than the previous result, a value of 100 is given; otherwise, a value of 0 is returned. Algorithm 1 is an algorithmic representation of obtaining a reward.
Algorithm 1: Algorithm for reward.

```
upper : 20% greater than default_bench
under : 20% less than default_bench
if result_of_testworkload > upper then
    reward = 200
else if result_of_testworkload < under then
    reward = -50
else if result_of_testworkload > last_result then
    reward = 100
else
    reward = 0
end
```

5. Evaluation

In this section, we evaluate the performance of STUN implemented using reinforcement learning. The following are the main points we wanted to confirm through the evaluation:

- **STUN performance based on number of learning iterations.** We analyzed the results by changing the values of the episode count N and Max_T, which are factors that determine the total number of iterations in reinforcement learning.

- **Performance with micro-benchmark.** To confirm the detailed performance improvement of STUN, we chose a micro-benchmark, i.e., hackbench, and compare its performance under the optimized scheduler parameters by STUN with that under the default scheduler environment.

- **Performance under real workload.** To evaluate the performance impact of STUN for a real workload, we ran a face detection application using Haar Cascades and compare the execution time and frames per second of the application between the cases with default and optimized settings.

- **Improvement based on the number of CPU cores.** To confirm whether the number of CPU cores affects the performance of STUN, we compared the performance improvement rates by optimizing Sysbench in 4-core, 44-core, and 120-core machines.

5.1. Environment

Table 2 shows the hardware specifications of the machine used in the evaluation. STUN was implemented in Python 3.6.9 version environments, and the 4-core and 44-core machines used Linux kernel version 4.15 and 120-core Linux kernel version 5.4. In addition, the test workload used for learning was as follows.

Table 2. Experimental machine specifications.

| Number of Core | CPU                     | Memory |
|----------------|-------------------------|--------|
| 4 core         | Intel(R) Core(TM) i7-6700 CPU @ 3.40 GHz | 8 GB   |
| 44 core        | Intel(R) Xeon(R) CPU E5-2696 v4 @ 2.20 GHz | 132 GB |
| 120 core       | Intel(R) Xeon(R) CPU E7-8870 v2 @ 2.30 Ghz | 800 GB |

- **Hackbench:** As a benchmark [18] for evaluating the performance of the Linux kernel scheduler, it creates processes to communicate through sockets or pipes and measures how much time it takes for each pair to send and receive data. For the option to run hackbench, the number of cores of the target machine was given, and the number of cores multiplied by 40 tasks was applied as a process to measure the execution time.

- **Face detection application using Haar Cascades:** The face detection application uses machine learning-based object detection algorithms [19] provided by OpenCV, one of
the image processing libraries. It finds the face and eyes of a person from the frames of the video file. We modified the default application to a highly parallel multi-threaded application to exploit many underlying cores.

- **Sysbench**: Sysbench is a multi-threaded benchmark tool set based on LuaJIT [20]. Although it is mainly used to benchmark the databases, it is also used to create arbitrary complex workloads. In this evaluation, we generated the number of CPU cores × 10 threads and used the total number of events executed within 1 s as the result.

### 5.2. Number of Learning Iterations

To confirm the correctness of STUN according to the number of learning iterations, we changed the values of the episode count and the step function execution count on a 120-core machine and analyzed the results. To improve the accuracy of the analysis, the evaluation used the recorded file from the result of the test workload under all parameters to avoid the fluctuation of the workload results.

First, to confirm the effect of the number of episodes, the number of the step was fixed to 500, and the number of episodes was set to 25, 50, or 100. We then analyzed the parameter state changes and the performance of Sysbench. Figure 4 shows the graphical representation of each evaluation process. For Figure 4A,B, the number of episodes was 25; for Figure 4C,D, 50; and for Figure 4E,F, 100. The x-axis of each graph is the number of learning iterations, and the y-axis of Figure 4A,C,E indicates the performance of Sysbench (the total number of events). The y-axis of Figure 4B,D,F represents the state values of each parameter.

The results show that if the number of episodes is not enough, the learning cannot find the optimal value. In Figure 4A,B, when the number of iterations is about 8000, the parameters converge, but it does not show the best performance. On the other hand, in Figure 4E, if the number of episodes is too large, the learning can find the optimal performance when the number of training iterations is about 45,000. However, Figure 4F shows that the parameter values do not converge because of the over-fitting problem. When the number of episodes is too large, the rewards are accumulated around the optimal value, so the result becomes unstable. Therefore, the number of episodes has to be set to an appropriate value to find the optimal and stable parameter value, as shown in Figure 4C,D.

Figure 5 shows the analysis of changes in the parameters obtained by a fixed number of episodes (50) while changing the number of steps to 200, 500, or 1000 for Sysbench. The x-axis and y-axis are the same as those in Figure 4. For Figure 5A,B, the number of the steps is 200; for Figure 5C,D, it is 500; and for Figure 5E,F, it is 1000. The analysis results are similar to those for changing the number of episodes. We confirm that if the number of steps is less than 200, the learning is insufficient, and if the number of the steps is over 1000, the model is excessively over-trained. Based on these evaluation results, we used the number of episodes as 50 and the number of steps as 500 for all subsequent evaluations.
Figure 4. STUN performance according to the number of episodes. The x-axis of each graph is the number of learning iterations. The y-axes of (A,C,E) show the performance of Sysbench (the total number of events), and the y-axes of (B,D,F) show the state values of each parameter.
Figure 5. STUN performance according to the number of steps. The x-axis of each graph is the number of learning iterations. The y-axes of (A,C,E) represent the performance of Sysbench (the total number of events), and the y-axes of (B,D,F) represent the state values of each parameter.

5.3. Micro-Benchmark Analysis

To confirm the performance impact of STUN, we optimized Hackbench, a micro-benchmark for the kernel scheduler, using STUN. The Hackbench execution option had 120 processes with 1000 iterations. Through the filtering process, the scheduler policy chose “Normal” and the parameters kernel.sched_latency_ns and kernel.sched_wake
up_granularity_ns as optimization variables. The optimal values of the learning result are as follows:

- Scheduler Policy: Normal;
- kernel.sched_latency_ns = 20,100,000;
- kernel.sched_wakeup_granularity_ns = 190,000,000.

The learning results show a 27.7% reduction in the Hackbench execution time from the default setting: 2.72 to 1.95 s. Figure 6 shows the execution time of Hackbench at each step of the learning process.

While the learning was in process, the results of Hackbench randomly increased and decreased, but after 16,000 steps, the learning process showed stable performance. Figure 7 shows the changes in state of two parameters for optimization during the learning process. Although each parameter initially increased and decreased randomly during the learning process, STUN finally found the optimal values.

![Figure 6. Change in Hackbench performance time in the STUN learning process, where the x-axis is the number of runs and the y-axis is the time of Hackbench.](image)

**Figure 7.** State changes in two parameters during STUN learning. The x-axis is the number of runs, and the y-axis is the state of each parameter.

### 5.4. Real Workloads Analysis

To verify the performance under a real workload, we optimized the face detection program using STUN in a 44-core machine, and analyzed the face recognition time of a video and the frames per second for default and optimized parameter values. The optimal parameter values obtained by STUN were as follows:

- Scheduler Policy: Normal;
- kernel.sched_rt_runtime_us = 1,000,000;
- kernel.sched_min_granularity_ns = 680,100,000.

The results of the face detection program on the actual image after applying the optimal value are as shown in Figure 8. The total execution time decreased by 18.3% from 58.998 to 48.198 s, and the number of frames per second increased by 22.4% from 16.95 to 20.748.
The performance impact of a scheduler is largely dependent on the number of cores, owing to the scheduler’s functionalities. We therefore compared the performance improvement in Sysbench by STUN on 4-core, 44-core, and 120-core machines. The Sysbench thread option was set to 10 times the number of CPU cores in each machine. In addition, the policy and parameters were set to the optimal values obtained through the filtering process. Table 3 shows a scheduler policy and parameter value that optimized Sysbench for each number of cores via STUN.

### Table 3. STUN optimization results for Sysbench for each number of cores.

| Policy  | Optimal Parameter Values                                      |
|---------|---------------------------------------------------------------|
| 4 core  | Normal kernel.sched_latency_ns = 800,000,000                  |
|         | kernel.sched_migration_cost_ns = 0                           |
| 44 core | Normal kernel.sched_latency_ns = 860,000,000                  |
|         | kernel.sched_wakeup_granularity_ns = 0                       |
| 120 core| Normal kernel.sched_rr_timeslice_ms = 0                       |
|         | kernel.sched_wakeup_granularity_ns = 0                       |

The performance improvements in Sysbench with STUN on each machine were as follows. As a result of the 4-core machine optimization with STUN, the number of events per second in Sysbench increased by 26.97% from 4419 to 5611. On a 44-core machine, STUN showed a performance improvement of 54.42%, from 3763 to 5811, and on a 120-core machine, it showed an improvement of 256.13%, from 1206 to 4295. Figure 9 shows the improvement rate on each machine. It was confirmed that the performance improvement was higher in the machine with the largest number of cores.

![Figure 8](image1.png)

**Figure 8.** Performance comparison of face detection program using STUN.

5.5. Performance Impact of Number of Cores

![Figure 9](image2.png)

**Figure 9.** Performance improvement over default for each number of cores.
6. Conclusions

A CPU scheduler is an important factor that affects the system performance. In this paper, we suggested STUN, an optimization framework of the Linux kernel scheduler parameter using reinforcement learning. The use of STUN can enhance the various scheduler environments without human intervention.

STUN has features for effective and rapid optimization. First, the selected parameters significantly affect the performance through a filtering process for an efficient optimization. Additionally, we subdivided the reward algorithm such that the agents used in reinforcement learning can learn effectively.

A performance evaluation confirmed a 27.7% performance improvement in Hackbench over the default parameters of Linux. For a real workload, a face detection application showed an 18.3% reduction in the execution time and a 22.4% increase in the number of frames per second. By optimizing Sysbench using 4 cores, 44 cores, and 120 cores, we showed performance improvements of 26.97%, 54.43%, and 256.13%, respectively, in comparison to the default performance of the machine.

As future study, we plan to adopt another reinforcement learning algorithm, such as Asynchronous Advantage Actor-Critic (A3C) or policy gradient, for more exact optimized parameter values and more concurrent parameters. In addition, we will integrate the logic of STUN with the Linux kernel to make a self-adaptive scheduler.

Author Contributions: Conceptualization, H.L. and H.J.; software, H.L.; experiments, H.L. and S.J.; writing—original draft preparation, H.L. and H.J.; writing—review and editing, H.L., S.J. and H.J.; supervision, H.J.; project administration, H.J. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2020R1F1A1075941).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the first author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Orlando, S. Available online: www.top500.org (accessed on 12 July 2022).
2. Hines, J. Stepping up to summit. Comput. Sci. Eng. 2018, 20, 78–82. [CrossRef]
3. Shin, D. A Study of Linux Schedulers in Multi-core Environment. In Proceedings of the IEIE 2012 International Autumn Annual Conference of IEIE, Zhengzhou, China, 15–17 August 2012; pp. 828–831.
4. CFS Scheduler. Available online: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt (accessed on 12 July 2022).
5. Roberson, J. ULE: A Modern Scheduler for FreeBSD; USENIX–The Advanced Computing Systems Association: Berkeley, CA, USA, 2003.
6. Bouron, J.; Chevalley, S.; Lepers, B.; Zwaenepoel, W.; Gouicem, R.; Lawall, J.; Muller, G.; Sopena, J. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In Proceedings of the 2018 USENIX Annual Technical Conference (USENIXATC 18), Boston, MA, USA, 11–13 July 2018; pp. 85–96.
7. Kolivas, C. BFS–The Brain Fuck Scheduler; Linux Kernel: Melbourne, Australia, 2010.
8. Lama, P.; Zhou, X. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In Proceedings of the 9th International Conference on Autonomic Computing, San Jose, Ca, USA, 18–20 September 2012; pp. 63–72.
9. Wang, G.; Xu, J.; He, B. A novel method for tuning configuration parameters of spark based on machine learning. In Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications, IEEE 14th International Conference on Smart City and IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, Australia, 12–14 December 2016; pp. 586–593.
10. Han, J.; Lee, S. Performance Improvement of Linux CPU Scheduler Using Policy Gradient Reinforcement Learning for Android Smartphones. IEEE Access 2020, 8, 11031–11045. [CrossRef]
11. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.; Veness, J.; Bellemare, M.; Graves, A.; Riedmiller, M.; Fidjel, A.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* 2015, 518, 529–533. [CrossRef] [PubMed]

12. Christiano, P.; Leike, J.; Brown, T.; Martic, M.; Legg, S.; Amodei, D. Deep reinforcement learning from human preferences. *arXiv* 2017, arXiv:1706.03741.

13. Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Dębiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; et al. Dota 2 with large scale deep reinforcement learning. *arXiv* 2019, arXiv:1912.06680.

14. Dearden, R.; Friedman, N.; Russell, S. *Bayesian Q-Learning*; AAAI: Menlo Park, CA, USA, 1998; pp. 761–768.

15. Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. Openai gym. *arXiv* 2016, arXiv:1606.01540.

16. Available online: [https://man7.org/linux/man-pages/man7/sched.7.html](https://man7.org/linux/man-pages/man7/sched.7.html) (accessed on 12 July 2022).

17. Ceron, J.; Castro, P. Revisiting Rainbow: Promoting more insightful and inclusive deep reinforcement learning research. In Proceedings of the 38th International Conference on Machine Learning, Virtual, 18–24 July 2021; Volume 139, pp. 1373–1383.

18. Zhang, Y. Hackbench. 2008. Available online: [http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c](http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c) (accessed on 12 July 2022).

19. Singh, V.; Shokeen, V.; Singh, B. Face detection by haar cascade classifier with simple and complex backgrounds images using opencv implementation. *Int. J. Adv. Technol. Eng. Sci.* 2013, 1, 33–38.

20. Kopytov, A. *Sysbench Manual*; MySQL AB: Cupertino, CA, USA, 2012; pp. 2–3.