

Article

# BIOS-Based Server Intelligent Optimization

Xianxian Qi, Jianfeng Yang <sup>\*</sup>, Yiyang Zhang and Baonan Xiao

School of Electronic Information, Wuhan University, Wuhan 430072, China

<sup>\*</sup> Correspondence: yjf@whu.edu.cn; Tel.: +86-135-0719-1603

**Abstract:** Servers are the infrastructure of enterprise applications, and improving server performance under fixed hardware resources is an important issue. Conducting performance tuning at the application layer is common, but it is not systematic and requires prior knowledge of the running application. Some works performed tuning by dynamically adjusting the hardware prefetching configuration with a predictive model. Similarly, we design a BIOS (Basic Input/Output System)-based dynamic tuning framework for a Taishan 2280 server, including dynamic identification and static optimization. We simulate five workload scenarios (CPU-instance, etc.) with benchmark tools and perform scenario recognition dynamically with performance monitor counters (PMCs). The adjustable configurations provided by Kunpeng processing reach  $2^N$  ( $N > 100$ ). Therefore, we propose a joint BIOS optimization algorithm using a deep Q-network. Configuration optimization is modeled as a Markov decision process starting from a feasible solution and optimizing gradually. To improve the continuous optimization capabilities, the neighborhood search method of state machine control is added. To assess its performance, we compare our algorithm with the genetic algorithm and particle swarm optimization. Our algorithm shows that it can also improve performance up to  $1.10\times$  compared to experience configuration and perform better in reducing the probability of server downtime. The dynamic tuning framework in this paper is extensible, can be trained to adapt to different scenarios, and is more suitable for servers with many adjustable configurations. Compared with the heuristic intelligent search algorithm, the proposed joint BIOS optimization algorithm can generate fewer infeasible solutions and is not easily disturbed by initialization.

**Keywords:** reinforcement learning; BIOS; server; performance optimization



**Citation:** Qi, X.; Yang, J.; Zhang, Y.; Xiao, B. BIOS-Based Server Intelligent Optimization. *Sensors* **2022**, *22*, 6730. <https://doi.org/10.3390/s22186730>

Academic Editor: Antonio Puliafito

Received: 29 July 2022

Accepted: 1 September 2022

Published: 6 September 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Driven by the strong demand of the internet industry, more traditional industries have undergone digital transformation, and servers are also widely used by enterprises to build network applications. For server buyers, high-performance servers can provide their users with a better experience and meet business needs. However, the performance of the server mainly depends on its hardware configuration. The better the configuration is, the higher the performance, and the corresponding cost will also increase. Therefore, it is necessary to improve server performance based on inherent hardware devices. Server users can conduct performance tuning at the operating system layer or at the application layer according to the specific application. For example, Li et al. implemented dynamic performance optimization of the Apache Web Server by tuning some important parameters, such as 'MaxClients' and 'KeepAlive' [1]. For database application servers, SQL queries always have an optimization space. Reference [2] investigated a series of query optimization techniques for improving the energy efficiency of relational databases and NoSQL databases.

Another effective technique is to improve the performance of a single server through a reasonable hardware configuration. Reference [3] showed that hardware data prefetching could improve server application performance, but it did not provide a detailed description of how to configure it. References [4–6] produced a remarkable performance improvement by adjusting the hardware configuration on Intel and POWER8 servers. Nevertheless, these

are all hardware prefetching configurations. On the Taishan 2280 server, the Kunpeng 920 processor [7] also provides a large number of adjustable configurations, such as L3 cache prefetch configurations and HHA (HCCS Home Agent) [7] related configurations. The Basic Input/Output System (BIOS) [8] on the server can control the underlying registers to set these configurations. For example, the L3T\_PREFECTH register can control the “prefetch\_utl\_ddr\_en” function, which represents “whether to allow automatic threshold reduction according to the utilization of DDR”. For more registers and configurations, please refer to [9]. The configurable register bits can be regarded as BIOS switch control items. Reasonable configuration can improve the performance of a multicore server system. Since there are only four configuration items [4,5], exhaustive search methods were used to find the best configuration for a specific application. Reference [6] considered the configuration optimization of 25 bits, but they only filtered out the parts that had substantial performance gains and did not fully consider the possible nonlinear relationship between configurations. In this paper, the number of BIOS configuration items is larger, and it is impossible to manually select a configuration representing near-optimal performance out of such a huge search space. Therefore, it is necessary to resort to intelligent optimization algorithms. Evolutionary algorithms and local search algorithms based on swarm intelligence are common approaches for solving this type of problem. Genetic algorithm (GA) [10,11], particle swarm optimization (PSO) [12–14], whale optimization algorithm (WOA) [15], gray wolf optimization (GWO) [16,17] and other intelligent algorithms are widely used for parameter optimization, and there are versions for discrete parameters. These methods are iterative optimization algorithms that cannot learn the configuration process and may have a higher probability of causing server instability. Since the process of optimizing configuration can be modeled as a sequential decision-making process, we can also adopt reinforcement learning methods. With the development of artificial intelligence technology, deep reinforcement learning has shown great research potential on sequential decision-making problems [18,19].

It can be seen from the literature [4–6] that a fixed configuration does not help performance in many cases, and in fact, it may degrade performance due to useless bus bandwidth consumption and cache pollution. Similarly, it is also not appropriate to set a fixed BIOS configuration for the server because servers may run under different workloads, such as CPU-intensive, disk-intensive, memory-intensive, network-intensive, etc. Due to the different performance indicators of various workload scenarios, the optimal hardware configuration is also different. When the server workload is frequently switched, to avoid the trouble of manually checking and adjusting the configuration, this paper proposes a dynamic tuning framework for dynamically adjusting the configuration. First, the workload scenario is dynamically identified through the server performance monitor counter (PMC) data [20], and then, the optimal configuration can be switched for the identified scenario. Server workload monitoring has been discussed in many works [21,22], but these monitoring data come from operating systems, such as CPU utilization and disk read and write rates. Similar to our work, [4–6] use PMCs as inputs for machine learning strategies to select the best configuration, but due to the few configuration items, their machine learning strategy directly outputs the optimal configuration, not the workload scenario. When there are many configurations, this method is not suitable. Therefore, this paper uses the workload scenario as the label of the machine learning strategy. BIOS can also collect hardware PMCs by accessing the underlying event register, including cycles, iTLB-loads, branch-load-misses, dTLB-loads, and other event-counting data [23]. To the best of our knowledge, there has been little research on optimizing server performance by dynamically adjusting the BIOS hardware configuration. Therefore, this study makes the following contributions:

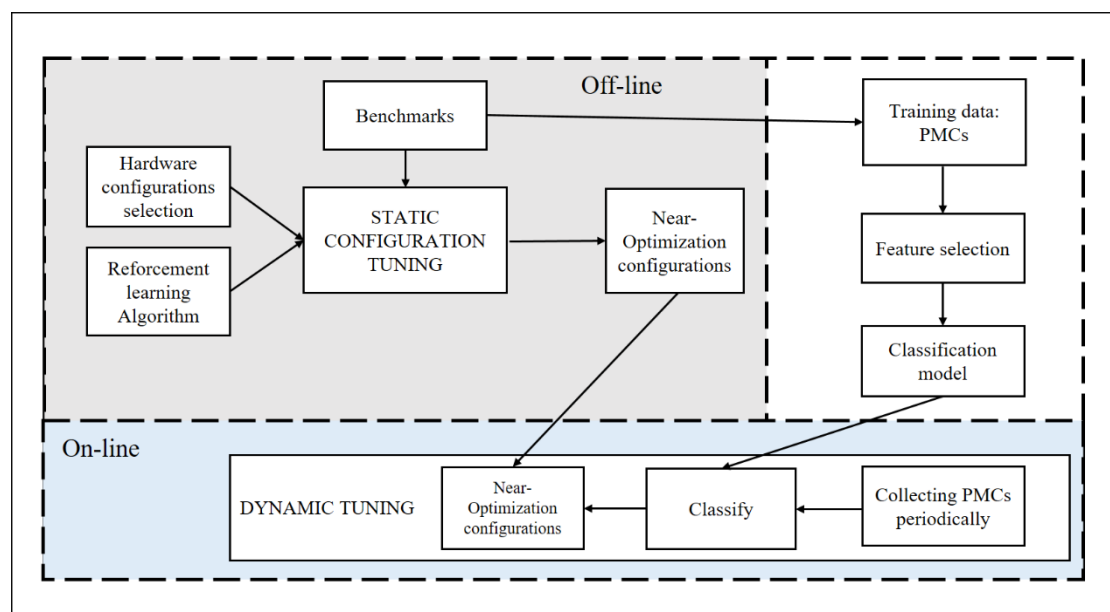
- (1) To improve the performance of Kunpeng processor-based servers, we proposed a performance tuning framework for dynamically adjusting the BIOS configuration. It monitors the server workload information to identify scenarios and implements performance optimization based on the results of static tuning or empirical configuration.

- (2) At the static configuration tuning stage, finding a near-optimal BIOS configuration is modeled as starting from a feasible initial configuration and adjusting the BIOS configuration to obtain an improvement. Based on this model, we propose a joint BIOS optimization algorithm using a deep Q-network combining reinforcement learning and nearest neighbor search.
- (3) With the proposed optimization algorithm, we significantly improve the memory bandwidth rate in memory-intensive scenarios. To further evaluate the proposed static tuning method, we compare it with two metaheuristic methods: genetic algorithm and particle swarm optimization algorithm. The algorithm in this paper is more stable and has a lower probability of server downtime.
- (4) We have also carried out optimization work in other load scenarios and found that in some scenarios, performance indicators are no longer critical optimization indicators.

The remainder of the paper is organized as follows. Section 2 describes the dynamic-tuning framework. Sections 3 and 4 discuss methods for load scenario identification and optimization, respectively. Section 5 presents the results of operating scenario recognition and optimization. Finally, this paper is summarized in Section 6.

## 2. Dynamic Tuning Framework

In this section, we describe the dynamic-tuning framework, which is separated into three stages: an offline training stage, an offline static configuration tuning stage, and an online dynamic-tuning stage, as depicted in Figure 1.



**Figure 1.** The dynamic tuning framework and workflow.

*Online dynamic-tuning stage.* This paper applies machine learning to predict server workload scenarios and periodically sets the corresponding optimal hardware configuration during the online dynamic-tuning stage. It collects PMCs periodically and feeds them into a classifier after a fixed time to identify the current workload scenario. Once it recognizes that the scenario changes, the near-optimization configuration is switched in real-time.

*Offline training stage.* The training data for machine learning are prepared offline. We simulate workload scenarios with benchmark software to collect training data in Section 3. Similar to [5], we measure all performance events supported by our processor and perform feature selection. The final dataset is a matrix, and each row consists of performance counter data and scenario labels. The performance counter data are the values of the specific scenario fixed time. Then, we build a classifier using machine learning strategies.

*Offline static configuration tuning stage.* To find better configurations from the inexhaustible configuration, we do not use iterative search methods. Instead, we use reinforcement learning to gradually optimize the initial configuration to find a better configuration. This is the main part of the article. The details are provided in Section 4.

### 3. Workload Scenario Recognition

#### 3.1. Scenario Preparation

Scenario identification of the server load is a prerequisite for dynamic optimization. However, it is impossible to simulate all workload scenarios, and the load scenarios running on real servers are relatively fixed. Therefore, in this paper, a coarse-grained division of load scenarios is made. Because of the scalability of the proposed dynamic monitoring algorithm, users can train fine-grained classifiers according to the actual server usage. According to different workload conditions, server workload scenarios can be roughly divided into CPU-intensive scenarios, memory-intensive scenarios, disk-intensive scenarios, network-intensive scenarios, and idle scenarios. The difference in load is mainly due to the different tasks currently running on the server. By running the benchmark tool on the server, various scenarios can be simulated, as seen in Table 1. FIO [24] is a benchmark software used to test hard disk IO, which can stress test hard disks, including sequential read and write and random read and write. IOzone [25] is a file system benchmark tool that provides a variety of IO operations that can comprehensively test the performance of the file system. Sysbench [26] is a multithreading and multifunctional benchmarking tool commonly used to test databases, CPUs, etc. The STREAM [27] benchmark is a comprehensive memory test that is widely used in the testing and server market industries. It supports four operation modes, copy, scale, add, and triad, to test the memory bandwidth performance. Iperf3 [28] is a widely used network bandwidth testing tool that supports IPv4 and IPv6 and can run on Windows, Linux, Android, and other platforms.

**Table 1.** Scenario simulation tools.

Scenario Type	Benchmarking Tools
I/O-intensive scenario	FIO, IOZone
Network-intensive scenario	iPerf3
CPU-intensive scenario	Sysbench-CPU
Memory-intensive scenario	STREAM
Idle scenario	None

#### 3.2. Data Processing and Scenario Recognition

A typical multiclass supervised learning problem generally follows four steps: data collection, feature selection, model training, and model evaluation. In various load scenarios simulated by the benchmark test, the BIOS is used to collect PMC data per second. There is redundancy in the original data, and feature correlation analysis can be performed to eliminate some features. The next step is to use traditional machine learning methods for training, such as decision trees (DT) [29], logistic regression (LR) [30], K-nearest neighbors (KNN) [31], and deep neural networks (DNN). In recent studies, DNN can handle complex classification problems well [32,33]. To judge the quality of the final classification model, some common evaluation indicators can be used for evaluation, such as accuracy and precision.

### 4. Workload Scenario Optimization

#### 4.1. Markov Model for BIOS Control Optimization

In a specific scenario, the problem of finding a near-optimal BIOS configuration is modeled as starting from a feasible initial configuration and adjusting the BIOS configuration to obtain an improvement. When the BIOS controls the optimization, the state is characterized by the current server's absolute BIOS configuration and performance evaluation. BIOS configuration can be divided into two categories: integer type, and binary type. There are very few configurations of integer types, so we treat them as multiple

binary configuration items by converting them to binary. Therefore, all configurations can be regarded as configuration switch items. The number of BIOS configuration switch items is  $N$  ( $N > 100$ ), and  $M$  performance indicators are used for the server's performance evaluation; thus, the state is represented with an  $(N + M)$ -dimensional vector. With the definition of the environment state, the action space can be obtained naturally. Actions can be defined as the flip of  $N$  BIOS configuration items and represented by an  $N$ -dimensional vector. It shows the BIOS register configuration items that need to be flipped. The position that needs to be flipped is set to 1, and the position that does not need to be flipped is set to 0. It can be seen that the action space is large and reaches  $2^N$ . To narrow the scope of the action space, we make conventions on actions. Given the state  $s$ , the optional action can only be to flip a single configuration item. In this way, the configuration optimization problem can be abstracted into multistep tuning rather than one-step tuning. Additionally, the state at the next moment is only related to the current state and the action and has nothing to do with the state of the earlier moment, which satisfies the Markov properties. Therefore, this paper uses the Markov decision process (MDP) [34] to complete the BIOS control optimization model design. The BIOS control optimization model based on MDP is defined as follows:

**Definition 1.** BIOS control optimization is based on MDP (MDP-BIOSCO). An MDP-BIOSCO can be defined as a tuple:  $MDP - BIOSCO = \langle S, A, P, R, \gamma \rangle$ .  $S$  is a set of data describing the server state, which is expressed above;  $A(s)$  is the restricted set of actions that can be performed in the state  $s \in S$ .  $R$  is a reward function, and when an action is performed, the server changes from  $s$  to  $s'$ . With the state change, the server performance also changes. Based on this change, an instant return  $r$  is fed back, and its expected value is  $r = E(R(s'|s, a))$ ;  $\gamma \in [0, 1]$  is a discount factor to distinguish the importance of future rewards and immediate rewards.

#### 4.2. Deep Q-Network

Deep reinforcement learning (Deep RL) is a combination of deep learning (DL) and reinforcement learning (RL), which is mainly used to deal with high-dimensional states and action spaces [35]. In [36], Mnih et al. proposed a structure named DQN, which could learn to play a range of Atari 2600 video games at a superhuman level. In this study, we employ a DQN because the state-action space is considerably large.

In the high-dimensional state or action space, traditional Q-learning cannot estimate the  $Q$  value corresponding to each large state and action space [37]. In DQN, a deep convolutional neural network is used to approximate the optimal action-value function. Nevertheless, when a nonlinear function approximator, such as a neural network, is used to represent the action-value function, RL is unstable or even divergent. To address this problem, Mnih et al. proposed an experienced pool and target network. As illustrated in Figure 2, replay memory is used to store samples  $(s, a, r, s')$  and perform random replay, thereby eliminating serial data correlation. The target  $Q$  network is used to assist in calculating the following loss function equation:

$$L_i(\theta_i) = E(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2 \quad (1)$$

where  $i$  is the number of iterations,  $s$  and  $a$  represent the current state and action, respectively.  $s'$  and  $a'$  represent the next state and action, respectively.  $\theta_i$  are the parameters of the estimation  $Q$ -network at iteration  $i$ , and  $\theta_i^-$  are the target  $Q$ -network parameters at iteration  $i$ .  $r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$  is the target value. The parameters  $\theta_i^-$  are updated at every  $C$  step from the estimation  $Q$ -network.

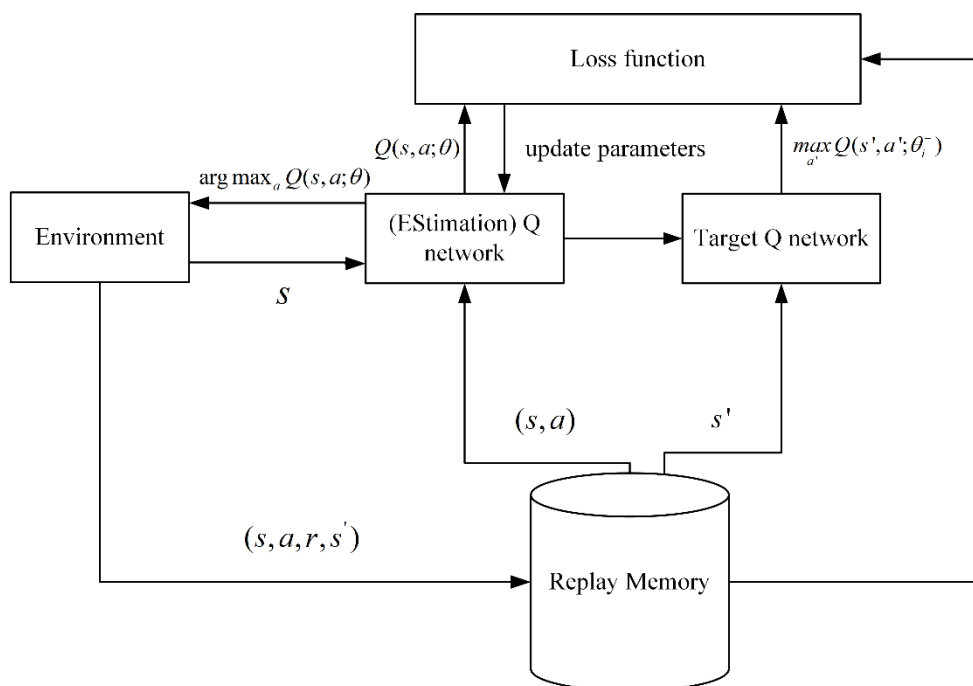


Figure 2. Principle of DQN.

4.3. Joint BIOS Optimization Algorithm Using DQN

To optimize BIOS control in a real server, a joint BIOS optimization algorithm using DQN is proposed based on the above MDP-BIOSCO model. The overall structure is illustrated in Figure 3. The first part involves the training phase and includes three subparts: environment, agent, and experience replay. The second part is designed for continuous optimization during the testing phase.

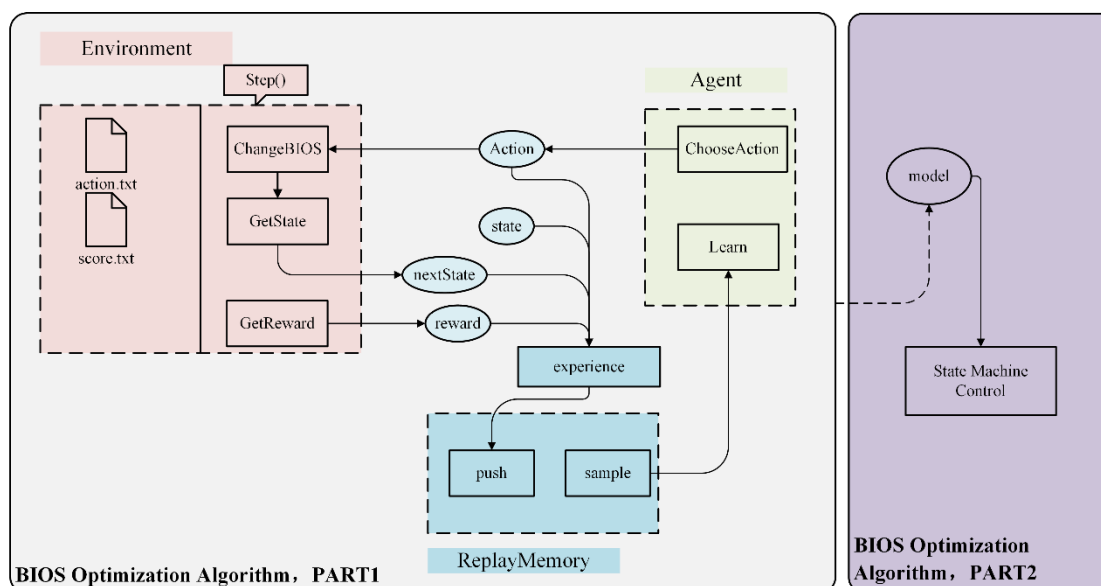


Figure 3. Structure of the BIOS optimization algorithm.

4.3.1. Environment Design

The environment is the object of agent interaction in a reinforcement learning system. The design of the environment mainly includes two parts: an interactive data record and a function, step().

It takes a few minutes to configure the BIOS and test the performance data on the server. The bios.txt and score.txt files shown in Figure 3 store the BIOS absolute configuration and the corresponding performance score data, respectively.

The core of the environment is the step() function, which contains the contents of the entire environment in Figure 3. The input of the step() function is the action, and the output is the state at the next moment, the reward of the current action, and whether to terminate the training episode. This function directly controls the server to conduct field measurements and calculate the state at the next moment.

- A. The state of the system is obtained at the next moment. The state at the next moment is obtained through end-to-end testing. First, we calculate the absolute BIOS configuration at the next moment from the predicted action at the current time, and then we test the server to obtain the performance evaluation. Finally, the two parts are merged to form a state.
- B. An instant reward is obtained. In contrast to the general reinforcement learning task, BIOS control optimization has no specific target, and the desired effect is that the algorithm can obtain better server performance quickly while ensuring the ability to jump out of local optimization. For the STREAM test scenario, the goal is to adjust the configuration, ensuring that the memory scores can increase rapidly and have the ability to find higher scores. Therefore, the reward function is set as follows:

Setting reward,  $r_1$ . The action of BIOS configuration modification causes the performance scores to rise or fall, and  $r_1$  provides feedback for the current action. There is a large gap in performance evaluation for different servers and operating scenarios. To ensure scalability, this reward adopts the baseline design method. The baseline is the known maximum difference of automated test acquisitions in the initial pre-experimental stage as follows:

$$r_1 = (score_{i+1} - score_i) / baseline * scale \quad (2)$$

$score_i$  and  $score_{i+1}$  represent the performance scores of the current moment and the next moment, respectively. Since the score gap may be small compared to the baseline in the actual experiment, we have incorporated a scaling factor  $scale$ . By default,  $scale = 100$ .

Setting a fixed reward,  $r_2$ , when performance scores are higher than the record in this episode. This reward,  $r_2$ , is mainly to drive the algorithm to find higher scores and maintain the ability to jump away from the local optima.

Setting reward,  $r_{down}$ , when the server is down or works unstably. Taking all of the above into consideration, the instant reward module is described as follows:

$$r = \begin{cases} r_1 + r_2 & \text{other} \\ r_{down} & \text{if server down} \end{cases} \quad (3)$$

Moreover, the generated samples  $(s, a, r, s')$  are stored in an experience replay memory. These samples are then retrieved randomly from the experience replay and fed into the training process. If it is a downtime sample, repeat the storage  $k$  times to increase the proportion of downtime samples.

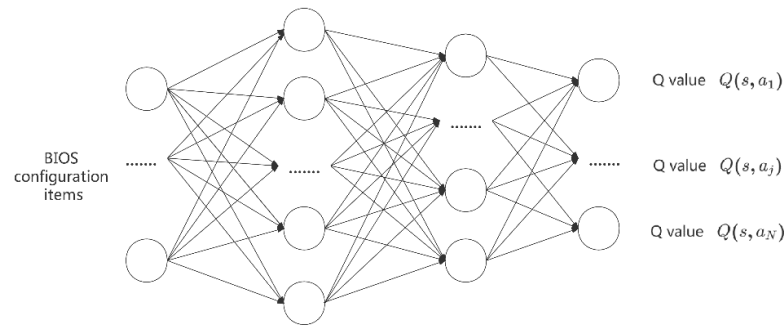
#### 4.3.2. Agent Decision-Making and Learning

The agent is the main actor in strategy learning. It finds the optimal action strategy through continuous trial and error learning, which mainly includes two aspects: action selection and strategy learning.

Figure 4 is the Q-network structure, the input is the BIOS configuration, and the output is the Q value for different actions. For instance,  $Q(s, a_j)$  represents the Q value for selecting the  $j$ th action at state  $s$ . In Section 4.1, we define and restrict actions. Action  $a$  means to modify only one configuration, and thus, the output dimension of the Q network is the same as that of the configuration item. At state  $s$ , to choose the best action in the set of alternative actions, the decision-maker will use a greedy strategy to take the action

with the largest  $Q$  value. However, adopting the greedy strategy alone will cause missing the optimal action, so this paper uses the  $\varepsilon$  greedy policy to ensure a certain probability of exploration. At the same time, to make up for the low intensity initial search problem, a large value of  $\varepsilon_{max}$  is set at the beginning of the training of the algorithm, and it decreases according to the decline coefficient  $\varepsilon_{decay}$ , as the training goes on until it drops to the minimum value,  $\varepsilon$  as follows:

$$\varepsilon = \max(\varepsilon * \varepsilon_{decay}, \varepsilon_{min}) \quad (4)$$



**Figure 4.** Schematic illustration of the q-network.

During the policy update phase, the parameters of the  $Q$ -network are updated as follows:

$$\theta_{i+1} = \theta_i + \alpha [r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)] \nabla Q(s, a; \theta_i) \quad (5)$$

where  $i$  is the number of iterations,  $s$  and  $a$  represent the current state and action, respectively.  $s'$  and  $a'$  represent the next state and action, respectively.  $\theta_i$  are the parameters of the estimation  $Q$ -network at iteration  $i$ , and  $\theta_i^-$  are the target  $Q$ -network parameters at iteration  $i$ .  $\alpha$  and  $\gamma$  represent the learning rate and discount factor.

In summary, the BIOS optimization algorithm (PART 1) is applied to the server BIOS control optimization problem, as shown in Algorithm 1.

---

**Algorithm 1** BIOS optimization algorithm based on DQN (Part 1).

---

**Input:** N Initial capacity of the playback pool,  
M Number of final exploration frames for optimization,  
T Number of optimization steps per iteration process,  
 $s_{initial}$  State corresponding to the initial configuration

- 1 Initialize replay memory (D) to capacity N;
- 2 Initialize Q-network with random weights  $\theta$ ;
- 3 Initialize environment;
- 4 For episode = 1, M do
  - 5 Initialize environment state  $s = s_{initial}$ ;
  - 6 For t = 1, T do
    - 7 Calculate  $Q(s, a; \theta)$  in state  $s$ ;
    - 8 With probability  $\varepsilon$  select a random action  $a$ ,  
Otherwise, select  $a = \operatorname{argmax}_a(Q(s, a; \theta))$ ;
    - 9 Execute action  $a$  in the environment,  
obtain a reward  $r$ , next state  $s'$ , and whether server downtime;
    - 10 If server downtime: store the same transition in D  $k$  times;
    - 11 else: store one transition in D;
    - 12 Sample random minibatch of transitions from D
    - 13 Update parameters  $\theta$  in Q-network with Formula (5)
    - 14 Every C steps reset  $\theta^- = \theta$
  - 15 End For
- 16 End For

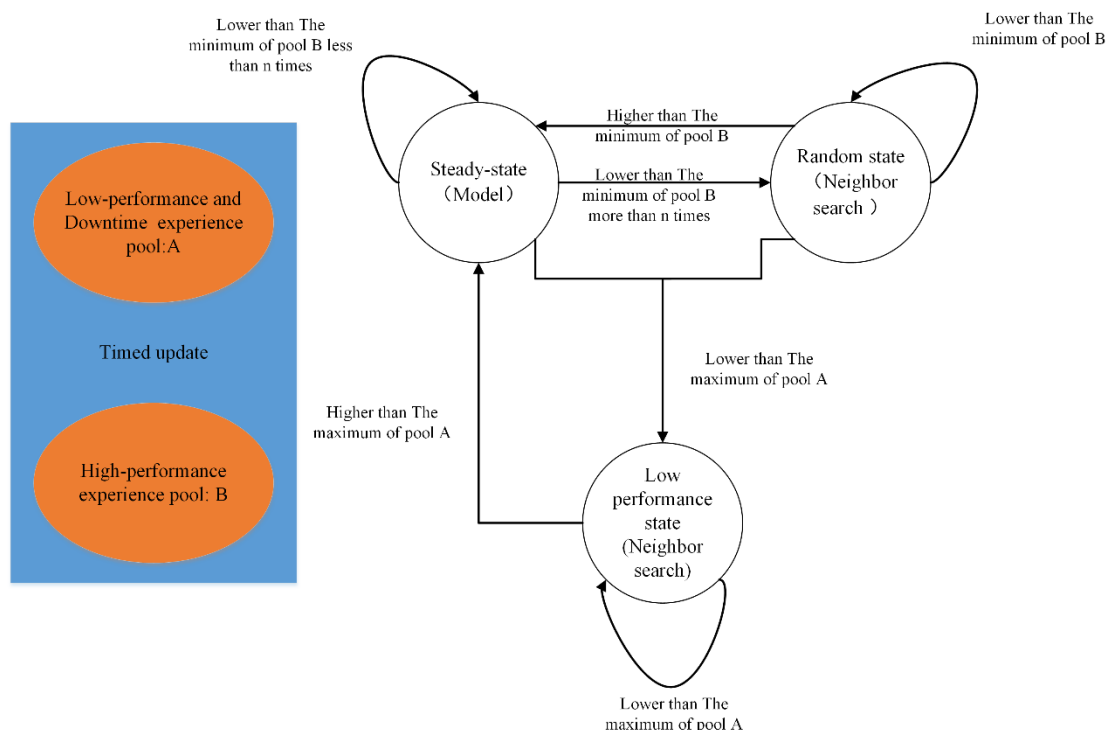
---



### 4.3.3. State Control Optimization Algorithm

In Part 2 of Figure 3, state machine control is added to obtain a model-assisted joint optimization method. The state machine can record the good performance state. Based on the optimized model and the state machine, the joint optimization method can realize the continuous optimization of the server to improve global optimization.

After the first part of the algorithm is executed, some better-performing configurations are obtained, as well as a trained and tuned model. These contents will be used to guide subsequent state-controlled neighborhood searches. As depicted in Figure 5, the two experience pools that continue to update come from the data of the training process. There are three states: steady state, random state, and low-performance state. In the steady state, it uses the model trained above to select an action. In the random state, it will find the past  $n$  high-performance experiences based on memory and then perform neighbor searching based on Hamming Distance. In the low-performance state, it will find the first  $n$  high-performance experiences from pool B and then search nearby. The process of generating a new configuration for neighbor searching is as follows: (1) Randomly generate a candidate configuration. (2) Calculate the Hamming distance between the candidate configuration and  $n$  reference configurations. In the random state, the reference configuration is the past  $n$  high-performance experiences. (3) Determine whether more than half of the  $n$  distances are lower than the set distance threshold, and if so, generate a new solution; otherwise, repeat (1). At the beginning of the algorithm, the last 5% of the first stage and the downtime configuration are filled into experience pool A, and the first 1% of the configuration is filled into B. The initial configuration is set the same as the first stage and defined as a stable state. Then, state switching is performed according to Figure 5, and each state transition is an iterative process. If the resulting new solution performance is lower than the maximum value of A, it is added to A, and if it is higher than the minimum value of B, it is added to B.



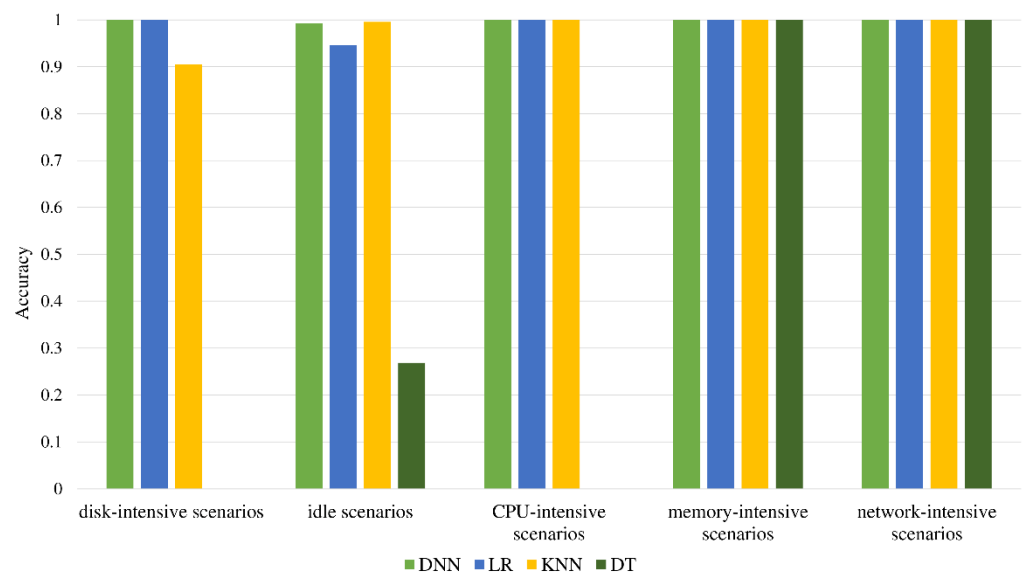
**Figure 5.** State machine process in optimization.

## 5. Experimental Results

### 5.1. Workload Scenario Recognition Results

In each scenario, a similar amount of data is collected, with a total of 497,640 combined data. With the help of the Pearson correlation coefficient [38] for feature screening, the

features with high correlation are removed, and finally, 38 features remain in Table A1. After dividing the dataset, there are 338,396 pieces of data in the training set, 59,716 pieces of data in the validation set, and 99,528 pieces of data in the test set. As seen in Figure 6, each classifier after training is evaluated using the test set. It was found that the performance of the decision tree is the worst. Its classification ability for hard disk-intensive scenarios, idle scenarios, and CPU-intensive scenarios is too poor, resulting in low accuracy. The other three algorithms have good classification capabilities for CPU-intensive scenarios, memory-intensive scenarios, and network-intensive scenarios and can achieve close to 100% accuracy. The performance difference between these four algorithms is mainly reflected in the classification ability of hard disk-intensive scenarios and idle scenarios. According to the results, DNN has the best classification ability, with an accuracy rate of 99.7%.



**Figure 6.** Comparison of classification accuracy.

### 5.2. Operating Scenario Performance Optimization Experiment

According to prior knowledge, the server does not work if some BIOS switch configuration items change. Therefore, these configuration items will be directly removed. The final configuration items are shown in Table 2, and 104 switch configuration items are reserved for the experimental STREAM scenario.

**Table 2.** Configuration register [9].

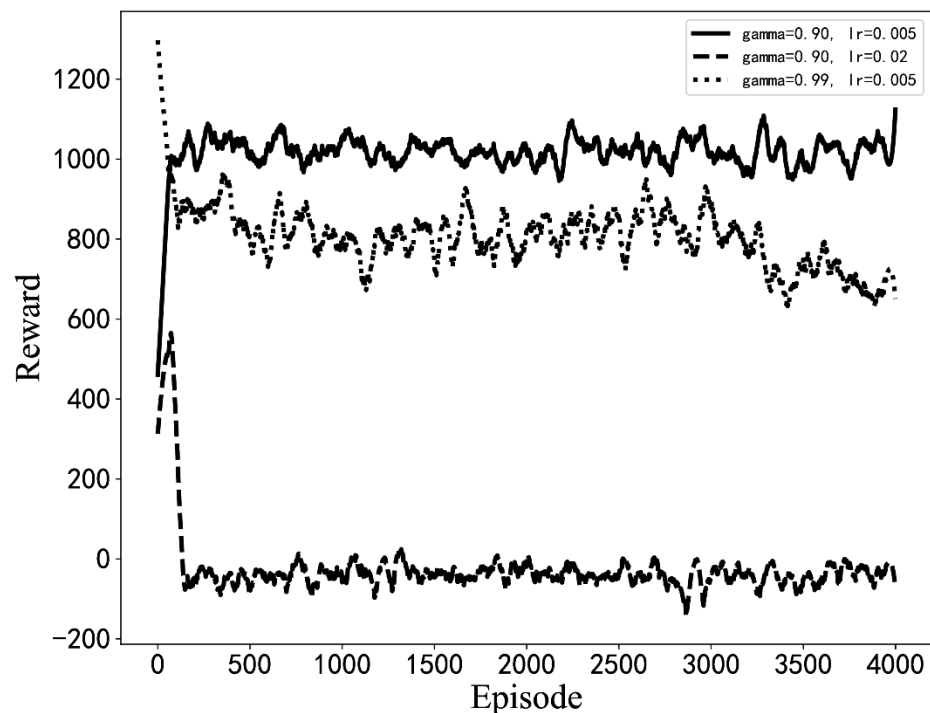
Register Name	Number of Configuration Bits Selected for Optimization
L3T_STATIC_CTRL	8
L3T_DYNAMIC_CTRL	16
L3T_DYNAMIC_AUCTRL0	8
L3T_DYNAMIC_AUCTRL1	21
L3T_PREFECTH	8
HHA_DIR_CTRL	15
HHA_FUNC_DIS	17
HHA_TOTEMNUM	11

### 5.2.1. Simulation

To verify the effectiveness of the optimization algorithm detailed in Section 3 and to find a good parameter design scheme promptly, the algorithm simulation experiment is designed before the experiment.

The BIOS on the server is configured dynamically, and the STREAM benchmark tool is run to collect simulation model training data. STREAM uses four memory operations, copy, scale, add, and triad, to test the bandwidth performance of the system memory. These four rates are used as indicators to measure the server's performance. From the collected data, the minimum average memory rate of the server in the experiment is close to 150,000 MB/s. Then, a performance score regression model is established based on the collected data.

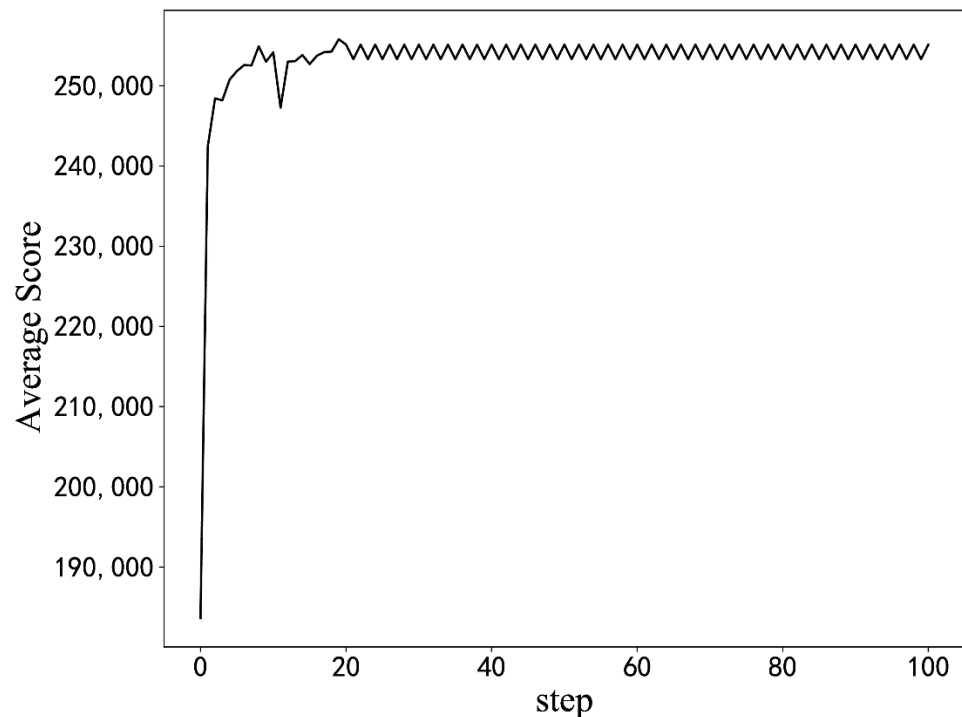
Experiments with different hyperparameters are conducted based on the optimization algorithm in this paper, and the parameter setting scheme is selected by comparison. The influence of some hyperparameter settings on training is shown in Figure 7. When the learning rate or discount coefficient is too large, the average return shows a downward trend. The final hyperparameter design scheme is shown in Table 3. The model is used for optimization from the initial BIOS configuration, as seen in Figure 8. Moreover, the model can quickly optimize from a low-score state to a high-score state.



**Figure 7.** The average reward curves of the training process for different hyperparameter settings. (The results of simulation experiments).

**Table 3.** List of hyperparameters and their values.

Hyperparameters	Value
minibatch size	32
replay memory size	10,000
discount factor $\gamma$	0.90
learning rate $\alpha$	0.005
initial exploration $\epsilon_{max}$	1
exploration decay $\epsilon_{decay}$	0.995
final exploration $\epsilon_{min}$	0.1
final exploration frame $M$	2000
final step $T$	100



**Figure 8.** Optimization result from the initial configuration. (The results of simulation experiments).

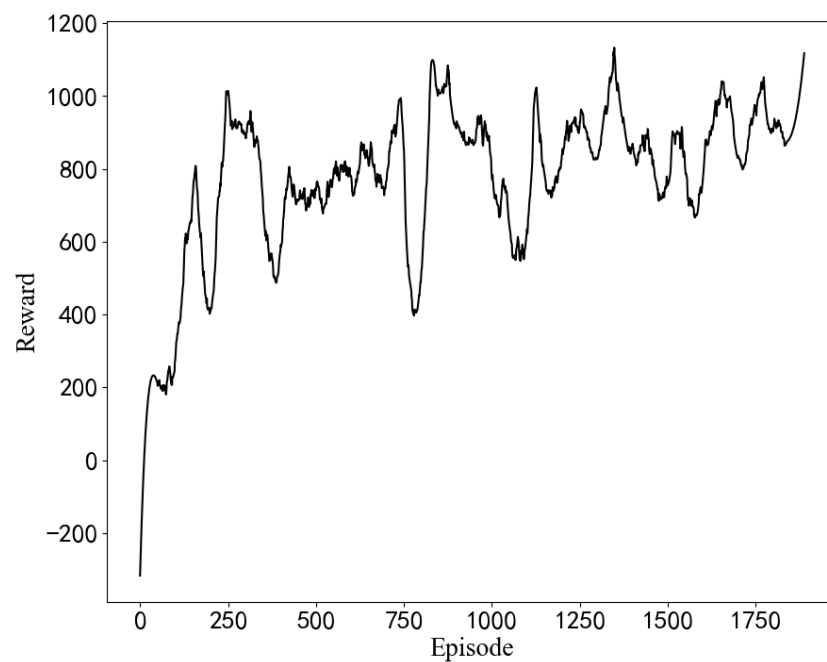
### 5.2.2. Measured Experiment

We refer to the parameter setting plan of the simulation experiment and make minor modifications to it. The actual training requires a test server, which takes a long time, thus reducing the capacity of the experience pool and the final step  $T = 40$ . The performance score for the initial training state is 202,000 MB/s. The average reward is used to evaluate the effect of the algorithm training on the actual server.

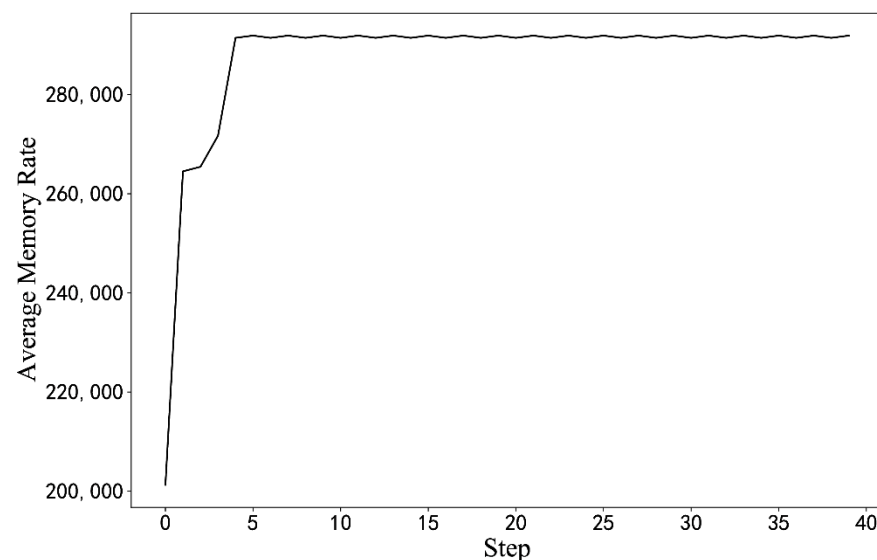
In Figure 9, as the number of iterations increases, the average return obtained by the algorithm continues to increase, but the shock is greater than the simulation result. In an actual server environment, some BIOS configuration items and configuration progress cause the server to crash or work unstably. When this happens, the cumulative return of the round drops considerably, and then the reward curve exhibits a sudden drop. Additionally, the trained model is used to optimize the initial BIOS configuration in Figure 10.

The state machine control of the BIOS optimization algorithm (Part 2) is added to realize continuous optimization. Table 4 shows the comparison of this paper's joint optimization algorithm and other traditional heuristic algorithms, in which *popsiz*e represents the number of individuals in the population, and *iter* represents the number of iterations. The experience configuration is provided by the cooperative server manufacturer unit, which is obtained by a large number of manually automated tests. The optimal configuration performance of the three algorithms is all higher than the empirical configuration, but the BPSO algorithm has the worst optimization effect and the highest probability of downtime. This is because BPSO, despite having a strong global search capability, becomes more random with the iterative search of the algorithm, resulting in an increased probability of generating downtime configurations. Since too many downtimes will damage the server, the BPSO optimizations are all manually terminated in about 30 iterations, leading to the worst optimization effect. Different population sizes in the genetic algorithm also affected the downtime probability and optimization effect. When the number of individuals in the population is larger, the difference between populations will be larger, and the global optimization ability will also be stronger. Therefore, the performance optimization effect of *popsiz*e = 100 is better than that of *popsiz*e = 20. Moreover, individuals with greater differences are more likely to cause downtime when performing crossover operations. In

addition, both algorithms need to initialize the population, which also brings some downtime configuration due to the random initialization. Although the algorithm proposed in this paper limits the number of tuning steps, the search space is also large enough, so the performance optimization effect can be close to that of the genetic algorithm. More importantly, the algorithm is based on reinforcement learning with a learning mechanism and is not disturbed by the initialization process; thus, the performance is more stable with a smaller downtime probability. After analyzing the downtime configurations, it was found that the server being down not only depends on the BIOS configuration but also on the configuration process. For example, configuration-A and configuration-B do not cause server downtime, but adjusting to configuration-B from configuration-A can cause server downtime. However, the GA algorithm has no memory of the configuration process.



**Figure 9.** The average reward curves of the training process in the STREAM scenario. (The results of the measured experiments).

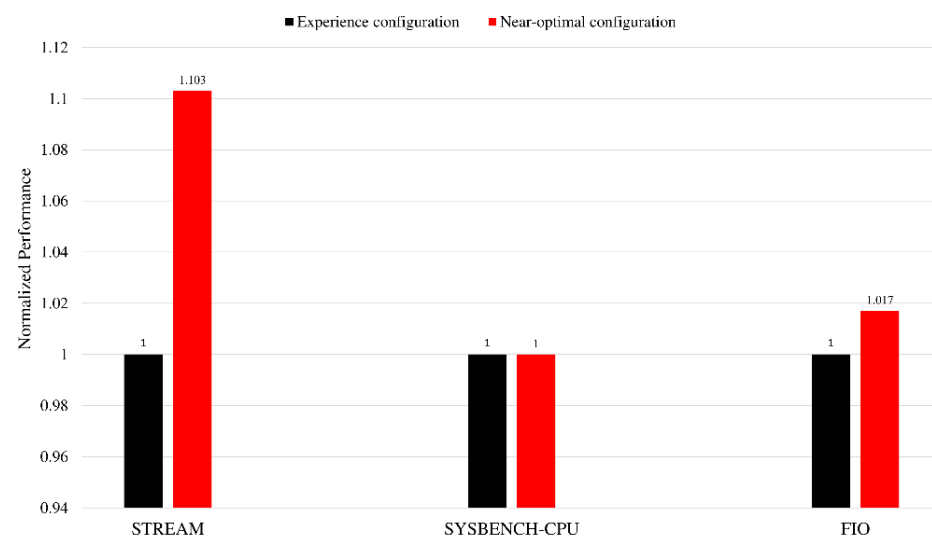


**Figure 10.** Optimization results from the initial configuration. (The results of the measured experiments).

**Table 4.** Comparison of different algorithm optimization results for the STREAM scenario.

Methods	Main Parameter Settings	Best Performance Score (Average Memory Rate MB/s)	Server Downtime Probability during the Experiment Process
Prior Knowledge		263,000	
Joint BIOS optimization algorithm using DQN (including the training process)	Iter = 50	290,400	4.1%
Genetic algorithm	Iter = 100, popsize = 20	281,000	8.1%
	Iter = 100, popsize = 100	289,700	20.3%
Binary particle swarm algorithm	PopsiZe = 100	276,000	39.8%

We also perform static tuning on several other benchmark software, and the results are shown in Figure 11. The performance metric of Sysbench-CPU is the 95th percentage of the prime calculation event latency. The performance metric of Fio is the IOPS (input/output operations per second) under the random-read workload. For the Sysbench-CPU benchmark, there is almost no room for improvement of the empirical configuration. The main reason is that for the performance of the core, the empirical configuration has reached the optimal solution, and it is difficult to further optimize it further. In addition, although kumpeng920 provides a lot of configuration items, only some configurations are selected for the experiment in this article. Considering that the configurations adjusted in this article are mainly related to LLC and HHA, there is little opportunity for adjusting these configurations to improve the core performance but a greater opportunity for optimization in computing and communication scenarios, such as memory. In the follow-up work, energy efficiency can be used as the tuning index for scenarios where there is little room for performance improvement. Additionally, more memory-type benchmarks can be added for fine-grained scenario classification and static tuning.

**Figure 11.** Normalized performance of experience configuration and near-optimal configuration obtained by static offline tuning.

## 6. Conclusions

In this study, we design a dynamic tuning framework under unknown operating scenarios based on the BIOS system. The aim is to give full consideration to the hardware capabilities of the system. First, dynamic scenario classification is implemented on the BIOS system using real-time server monitoring data. The accuracy rate reaches 99.7%. Then, the Markov model of the BIOS control decision is established. Combining reinforcement learning with the nearest neighbor search, we propose a joint BIOS control optimization method. Simulation experiments show that the algorithm proposed in this paper can optimize the server to a higher performance state. The actual experiments on the server prove that the algorithm performs well in dynamic situations, and in the STREAM scenario, it achieves good memory rate performance optimization (43.7% higher than the initial training configuration state and 10.3% higher than the experience configuration state). Compared with the genetic algorithm, the obtained optimal performance is close, and the probability of downtime during optimization is smaller. Static configuration optimization also shows that the performance in partial load scenarios is not sensitive to the configuration in this article, such as sysbench-CPU, and energy efficiency may be a good tuning indicator. Given this situation, the static optimization algorithm in this paper can still work. In summary, the dynamic tuning framework makes full use of system hardware functions and shows its effectiveness.

**Author Contributions:** Conceptualization, X.Q.; writing—original draft, X.Q. and Y.Z.; methodology, X.Q., J.Y., Y.Z. and B.X.; writing—review and editing, J.Y. and B.X.; visualization, X.Q. and Y.Z.; formal analysis, X.Q.; validation, B.X.; resources, J.Y.; project administration, J.Y.; supervision, J.Y. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare that they have no conflict of interest.

## Appendix A

**Table A1.** Data features remaining after data preprocessing.

Serial Number	Feature Name	Data Type
1	Cpu_cycles	int
2	L1_cache_refill	int
3	L1d_cache_refill	int
4	L1d_cache	int
5	L1d_tld_refill	int
6	L1i_cache	int
7	L2d_cache	int
8	L2d_cache_refill	int
9	L1d_tlb	int
10	Prf_req	int
11	Hit_on_prf	int
12	Mem_stall_l1miss	int
13	Mem_stall_l2miss	int
14	Atc_cmd	int
15	Flux_rd	int
16	Flux_wr	int
17	Prc_cmd	int
18	Rnk_chg	int

Table A1. Cont.

Serial Number	Feature Name	Data Type
19	Rw_chg	int
20	Fluxid_wcmd	int
21	Fluxid_rcmd	int
22	Bnk_chg	int
23	Rd_cpiped	int
24	Rx_ops_num	int
25	Rx_outer	int
26	Rx_sccl	int
27	Retry	int
28	Tx_snp_outer	int
29	Hac_smmu_transaction	int
30	Net_smmu_transaction	int
31	Net_smmu_l2_ltb_hit	int
32	Net_smmu_ltb_miss	int
33	Pcie_smmu_transaction	int
34	Pcie_smmu_trans_table_walk_access	int
35	Pcie_smmu_context_bank_cache_miss	int
36	Pcie_smmu_ltb_miss	int
37	Pcie_smmu_l1_tlb	int
38	Pcie_smmu_l2_ltb_hit	int

## References

- Li, J.; Lu, M. The performance optimization and modeling analysis based on the Apache Web Server. In Proceedings of the 32nd Chinese Control Conference, Xian, China, 26–28 July 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 1712–1716.
- Mahajan, D.; Blakeney, C.; Zong, Z. Improving the energy efficiency of relational and NoSQL databases via query optimizations. *Sustain. Comput. Inform. Syst.* **2019**, *22*, 120–133. [CrossRef]
- Bakhshalipour, M.; Tabaeiaghdaei, S.; Lotfi-Kamran, P.; Sarbazi-Azad, H. Evaluation of Hardware Data Prefetchers on Server Processors. *ACM Comput. Surv.* **2020**, *52*, 1–29. [CrossRef]
- Liao, S.; Hung, T.H.; Nguyen, D.; Chou, C.; Tu, C.; Zhou, H. Machine learning-based prefetch optimization for data center applications. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, New York, NY, USA, 14–20 November 2009; pp. 1–10.
- Rahman, S.; Burtscher, M.; Zong, Z.; Qasem, A. Maximizing hardware prefetch effectiveness with machine learning. In Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, New York, NY, USA, 24–26 August 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 383–389.
- Li, M.; Chen, G.; Wang, Q.; Lin, Y.; Hofstee, P.; Stenstrom, P.; Zhou, D. PATer: A Hardware Prefetching Automatic Tuner on IBM POWER8 Processor. *IEEE Comput. Arch. Lett.* **2015**, *15*, 37–40. [CrossRef]
- Xia, J.; Cheng, C.; Zhou, X.; Hu, Y.; Chun, P. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* **2021**, *41*, 67–75. [CrossRef]
- Regenscheid, A. BIOS Protection Guidelines for Servers. *NIST Spec. Publ.* **2014**, *800*, 147B. [CrossRef]
- openEuler: Prefetch\_tuning. Available online: [https://gitee.com/openeuler/prefetch\\_tuning/tree/master](https://gitee.com/openeuler/prefetch_tuning/tree/master) (accessed on 19 May 2022).
- Katoch, S.; Chauhan, S.S.; Kumar, V. A review on genetic algorithm: Past, present, and future. *Multimed. Tools Appl.* **2021**, *80*, 8091–8126. [CrossRef]
- Wu, D.; Wu, C. Research on the Time-Dependent Split Delivery Green Vehicle Routing Problem for Fresh Agricultural Products with Multiple Time Windows. *Agriculture* **2022**, *12*, 793. [CrossRef]
- Koyuncu, H.; Ceylan, R. A PSO based approach: Scout particle swarm algorithm for continuous global optimization problems. *J. Comput. Des. Eng.* **2019**, *6*, 129–142. [CrossRef]
- Beheshti, Z.; Shamsuddin, S.M.; Hasan, S. Memetic binary particle swarm optimization for discrete optimization problems. *Inf. Sci.* **2015**, *299*, 58–84. [CrossRef]
- Almahdi, S.; Yang, S.Y. A constrained portfolio trading system using particle swarm algorithm and recurrent reinforcement learning. *Expert Syst. Appl.* **2019**, *130*, 145–156. [CrossRef]
- Goyal, S.; Bhushan, S.; Kumar, Y.; Rana, A.; Bhutta, M.; Ijaz, M.; Son, Y. An Optimized Framework for Energy-Resource Allocation in a Cloud Environment based on the Whale Optimization Algorithm. *Sensors* **2021**, *21*, 1583. [CrossRef] [PubMed]
- Faris, H.; Aljarah, I.; Al-Betar, M.A.; Mirjalili, S. Grey wolf optimizer: A review of recent variants and applications. *Neural Comput. Appl.* **2018**, *30*, 413–435. [CrossRef]



17. Chen, H.; Miao, F.; Chen, Y.; Xiong, Y.; Chen, T. A Hyperspectral Image Classification Method Using Multifeature Vectors and Optimized KELM. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2021**, *14*, 2781–2795. [[CrossRef](#)]
18. Ghoul, T.; Sayed, T. Real-Time Safety Optimization of Connected Vehicle Trajectories Using Reinforcement Learning. *Sensors* **2021**, *21*, 3864. [[CrossRef](#)]
19. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv* **2013**, arXiv:1312.5602.
20. Zamani, R.; Afsahi, A. A study of hardware performance monitoring counter selection in power modeling of computing systems. In Proceedings of the 2012 International Green Computing Conference (IGCC), San Jose, CA, USA, 4–8 June 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 1–10.
21. Smith, J.W.; Sommerville, I. Workload classification software energy measurement for efficient scheduling on private cloud platforms. *arXiv* **2011**, arXiv:1105.2584.
22. Myint, S.H. Server Workload Classification and Analysis with Machine Learning Algorithms. Available online: <https://meral.edu.mm/record/4398/files/11108.pdf> (accessed on 19 May 2022).
23. Linux/Tools/Perf/Pmu-Events. Available online: <https://github.com/torvalds/linux/tree/master/tools/perf/pmu-events/arch/arm64> (accessed on 19 May 2022).
24. FIO-Flexible I/O Benchmark. Available online: [https://fio.readthedocs.io/en/latest/fio\\_doc.html](https://fio.readthedocs.io/en/latest/fio_doc.html) (accessed on 19 May 2022).
25. IOzone Filesystem Benchmark. Available online: <https://www.iozone.org/> (accessed on 19 May 2022).
26. Sysbench Benchmark. Available online: <https://wiki.gentoo.org/wiki/Sysbench> (accessed on 19 May 2022).
27. STREAM Benchmark. Available online: <https://github.com/jeffhammond/STREAM> (accessed on 19 May 2022).
28. iPerf3. Available online: <https://iperf.fr/> (accessed on 19 May 2022).
29. Patel, H.H.; Prajapati, P. Study and analysis of decision tree based classification algorithms. *Int. J. Com-puter. Sci. Eng.* **2018**, *6*, 74–78. [[CrossRef](#)]
30. Wright, R.E. Logistic regression. In *Reading and Understanding Multivariate Statistics*; Grimm, L.G., Yarnold, P.R., Eds.; American Psychological Association: Washington, DC, USA, 1995; pp. 217–244.
31. Guo, G.; Wang, H.; Bell, D.; Bi, Y.; Greer, K. KNN model-based approach in classification. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: Proceedings of the OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2003 Catania, Sicily, Italy, 3–7 November 2003*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 986–996.
32. Haldorai, A.; Ramu, A. Canonical correlation analysis based hyper basis feedforward neural network classification for ur-ban sustainability. *Neural Processing Lett.* **2021**, *53*, 2385–2401. [[CrossRef](#)]
33. Zhao, H.; Liu, J.; Chen, H.; Chen, J.; Li, Y.; Xu, J.; Deng, W. Intelligent Diagnosis Using Continuous Wavelet Transform and Gauss Convolutional Deep Belief Network. *IEEE Trans. Reliab.* **2022**. [[CrossRef](#)]
34. Puterman, M.L. Markov decision processes. In *Handbooks in Operations Research and Management Science*; Elsevier: Amsterdam, The Netherlands, 1990; Volume 2, pp. 331–434.
35. Arulkumaran, K.; Deisenroth, M.P.; Brundage, M.; Bharath, A.A. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Process. Mag.* **2017**, *34*, 26–38. [[CrossRef](#)]
36. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529–533. [[CrossRef](#)] [[PubMed](#)]
37. Alom, M.Z.; Taha, T.M.; Yakopcic, C.; Westberg, S.; Sidike, P.; Nasrin, M.S.; Hasan, M.; Van Essen, B.C.; Awwal, A.A.; Asari, V.K. A state-of-the-art survey on deep learning theory and architectures. *Electronics* **2019**, *8*, 292. [[CrossRef](#)]
38. Benesty, J.; Chen, J.; Huang, Y.; Cohen, I. Pearson correlation coefficient. In *Noise Reduction in Speech Processing*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 1–4.