Transactions on Computational and Scientific Methods | Vol. 5, No. 5, 2025
ISSN: 2998-8780
https://pspress.org/index.php/tcsm
Pinnacle Science Press

Dynamic Scheduling-Based Optimization of Multi-Core Architectures for High-Performance Computing

Annelise Gauthier
University of Northern British Columbia, Prince George, Canada
annelise.gauthier@unbc.ca

Abstract: This study aims to solve the problem of task scheduling and load balancing in multi-core architectures and proposes an optimization method based on dynamic scheduling strategy to improve the efficiency and resource utilization of high-performance computing systems. In traditional multi-core systems, as the number of cores increases, the optimization of task scheduling and resource allocation becomes more and more complicated. Especially when dealing with computationally intensive and memory-intensive tasks, how to maximize the computing power of multi-core processors becomes a key issue. This paper experimentally analyzes the performance of different scheduling strategies (static scheduling, dynamic scheduling, and priority scheduling) under different core number configurations. The results show that compared with other methods, the dynamic scheduling strategy can optimize execution efficiency and resource utilization while improving system throughput. In addition, the experiment also explores the trend of performance improvement after the increase in the number of cores and the possible performance saturation phenomenon, revealing the impact of scheduling strategies on the overall performance of the system in large-scale multi-core systems. The research results provide a theoretical basis for the design and optimization of future multi-core computing architectures and point out the direction for the development of intelligent scheduling algorithms. Future work will further study adaptive scheduling strategies in heterogeneous computing environments, combined with artificial intelligence and deep learning methods to achieve more efficient resource management and optimization solutions.

Keywords: Multi-core architecture, task scheduling, load balancing, dynamic scheduling

1. Introduction

In the past few decades, with the rapid growth of computing demand, traditional single-core processors have gradually been unable to meet the high performance requirements. High-performance computing (HPC) has become an important support for scientific research, engineering design, and data analysis, and multi-core architecture has received widespread attention and application as an effective solution to this challenge[1,2]. Compared with single-core processors, multi-core architecture can complete more computing tasks in a shorter time, significantly improving the processing power of computer systems[3.4]. However, to fully tap the potential of multi-core processors, designing and optimizing the corresponding multi-core architecture is a complex and challenging task. In particular, how to balance the load between processors and optimize data transmission and caching mechanisms under different application scenarios and computing loads has become a core issue in multi-core architecture research[5].

A key issue in multi-core architecture design is how to fully utilize the advantages of parallel computing. Traditional serial computing modes cannot adapt to today's demand for computing

performance, while parallel computing can significantly improve computing efficiency by decomposing complex tasks into multiple subtasks and processing them simultaneously. Modern multi-core processors provide hardware support for parallel computing by integrating multiple computing cores on the same chip. Different multi-core architecture designs, such as symmetric multi-processing (SMP) architecture and non-uniform memory access (NUMA) architecture, each has its own unique advantages and applicable scenarios. By rationally designing these architectures and optimizing task scheduling and resource management, computing performance can be effectively improved, system energy consumption can be reduced, and costs can be reduced[6].

In addition to parallel computing itself, data access and cache consistency are also factors that cannot be ignored in multi-core architecture design. In a multi-core system, data needs to be frequently exchanged between different cores, and data transmission delays and bandwidth limitations often become bottlenecks. In addition, multiple cores accessing shared data may cause cache consistency problems, which directly affects the stability and performance of the system. Therefore, designing efficient cache consistency protocols, optimizing memory hierarchies, and reducing data access delays at the hardware and software levels are all key strategies to improve the performance of multi-core architectures[7].

The optimization of multi-core architectures not only depends on hardware design, but also needs to work closely with the software level. In a multi-core system, task scheduling, load balancing, and resource allocation are one of the main factors affecting performance. Optimizing compilers and operating systems so that they can perform more efficient task allocation for multi-core architectures can reduce the overhead of context switching and improve the efficiency of parallel execution. In addition, developing parallel algorithms that can run efficiently in multi-core systems, especially optimization for specific application areas, is also of great significance to improving system performance. For example, in high-performance computing scenarios such as large-scale data processing and deep learning training, reasonable parallel algorithms can greatly improve computing speed and save time and resources[8].

With the increasing demand for computing power in emerging applications, such as artificial intelligence, the Internet of Things, and virtual reality, the design and optimization of multi-core architectures have become an important direction for the development of future computing systems. In order to meet the needs of future high-performance computing, researchers are exploring more advanced multi-core architecture designs, such as heterogeneous computing architectures, quantum computing, and neuromorphic computing. These emerging technologies provide new ideas and challenges for the design and optimization of multi-core architectures. In this context, the design and optimization of multi-core architectures for high-performance computing is not only the key to improving computing power, but also related to whether future computer systems can cope with changing technology and application requirements[9].

2. Related Work

In recent years, reinforcement learning (RL) has emerged as a powerful approach for addressing task scheduling and resource optimization in complex and dynamic computing environments. Li et al. proposed an adaptive resource scheduling strategy using reinforcement learning to respond effectively to varying system workloads, illustrating the flexibility of RL in real-time decision-making scenarios [10]. Sun et al. advanced this approach with a Double DQN-based operating system scheduler, emphasizing real-time task optimization under dynamic conditions [11]. Similarly, Wang et al. introduced an A3C-based reinforcement learning framework for intelligent microservice scheduling, showcasing how parallel actor-critic models enhance performance in distributed microservice environments [12].

Further contributions have focused on distributed environments, where topology and communication constraints play a vital role. Wang B. employed multi-agent reinforcement learning to develop topology-aware scheduling mechanisms, allowing for decentralized yet cooperative decision-making in distributed systems [13]. Deng extended this line of research by applying RL for

traffic scheduling in complex data center topologies, balancing throughput and network congestion [14]. Ren et al. also addressed distributed network traffic scheduling, integrating trust-constrained policy learning mechanisms to ensure secure and efficient scheduling policies in uncertain network conditions [15].

In addition to scheduling, the integration of federated learning into task allocation was investigated by Wang Y., who proposed a model optimizing distributed computing resources while maintaining communication efficiency across distributed nodes [16]. This approach leverages local data processing to enhance scalability and reduce latency, aligning well with the principles of heterogeneous multi-core systems.

Although some works are not directly focused on hardware architecture, their methodologies contribute valuable insights to the broader field of intelligent system design. For instance, the study by Wei et al. on contrastive learning and data augmentation in recommender models offers advanced strategies in model generalization, which could be extrapolated to task prediction in multi-core systems [17]. Similarly, Duan's work on user interface perception may not directly relate to scheduling, but its systematic analysis methodology provides valuable paradigms for user-aware system optimizations [18].

Together, these studies offer a comprehensive methodological foundation for designing intelligent, efficient, and scalable task scheduling mechanisms in multi-core and distributed systems. They highlight the convergence of reinforcement learning, federated optimization, and data-driven decision-making as central themes for next-generation high-performance computing architectures.

3. Method

In this study, we proposed an optimization method based on multi-core architecture to improve the parallel processing capability and resource utilization efficiency in high-performance computing tasks[14]. The model architecture is shown in Figure 1.

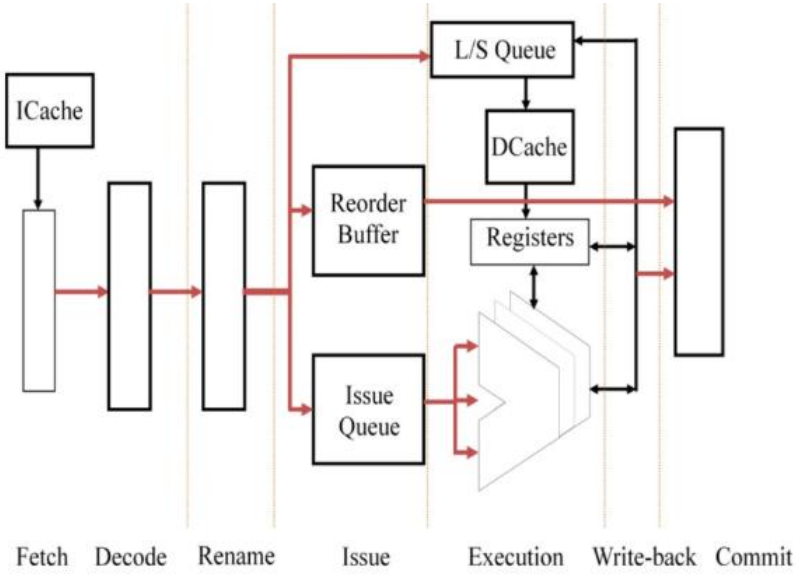


Figure 1. Overall model architecture

First, by analyzing the bottlenecks of the traditional multi-core architecture, we designed an improved task scheduling model that dynamically allocates tasks among multiple cores to minimize the waste of computing resources[15]. To achieve this goal, we constructed an optimized load balancing framework to dynamically adjust task allocation based on real-time information of core loads. Set the task load matrix to $L = \{l_{i,j}\}$, where $l_{i,j}$ represents the computational load of task j on core i. The goal of this load matrix is to minimize the load difference of all cores, that is, the optimization goal is:

$$\min_T \sum_{i=1}^{n} (\frac{1}{m} \sum_{j=1}^{m} li, j - \frac{1}{n} \sum_{k=1}^{n} \sum_{j=1}^{m} l_{k,j})^2$$

Among them, n is the number of cores, m is the number of tasks, and T is the task scheduling scheme. The goal is to evenly distribute tasks to each core through the scheduling strategy to maximize parallelism.

Secondly, we proposed a cache optimization method based on distributed memory management. In a multi-core system, when multiple cores access shared data at the same time, cache consistency problems are likely to occur, affecting the execution efficiency of the system. In order to reduce the overhead caused by cache consistency, we introduced a cache consistency protocol based on distance measurement. Assume that C_i is the cache of core i, and $D_{i,\ j}$ is the cache consistency measurement between core i and core j. Based on this measurement, the following cache update rules can be designed:

$$D_{i,j} = \frac{\sum_{k=1}^{k\max} (| C_i(k) - C_j(k) |)}{k_{\max}}$$

Among them, $C_i(k)$ and $C_j(k)$ represent the cache status of core i and core j at time k, respectively, and $k_{\max}$ is the maximum number of cache comparisons. Our goal is to reduce the latency caused by the cache consistency protocol by reasonably scheduling the timing and frequency of cache updates, and to improve the overall system efficiency through an optimized cache synchronization mechanism.

Finally, in the optimization process of multi-core architecture, the optimization of memory bandwidth is also crucial. Since multiple cores often compete for memory bandwidth when executing in parallel, memory bottleneck problems occur. We propose a resource scheduling algorithm based on bandwidth control. Assuming that the system memory bandwidth is B and the bandwidth requirement of core i is b_i, we introduce a bandwidth scheduling strategy to maximize the system bandwidth utilization and minimize the probability of bandwidth conflict. In order to ensure the reasonable allocation of bandwidth, the bandwidth scheduling problem can be modeled by the following objective function:

$$\max_B \sum_{i=1}^{n} b_i \cdot a_i \quad \text{subject to} \quad \sum_{i=1}^{n} b_i \le B$$

Among them, a_i is the bandwidth weight coefficient of core i, which indicates the priority of the core; B is the bandwidth scheduling scheme, which aims to ensure that high-priority tasks obtain sufficient bandwidth resources by reasonably allocating bandwidth, while reducing the impact of bandwidth bottlenecks on overall performance.

Through the above optimization methods, this study can effectively improve the performance of multi-core architecture in high-performance computing, and verify its advantages in processing complex tasks in experiments. These methods can not only reduce the conflict of computing and memory resources, but also improve the throughput and stability of the system, laying the foundation for the further development of high-performance computing.

4. Experiment

In this study, the dataset used in the experiment comes from a public high-performance computing task simulation dataset, which is designed to simulate the task scheduling and resource allocation of multi-core processors in a parallel computing environment. The dataset contains different types of computing tasks and their resource requirements, including information such as CPU time, memory bandwidth, and cache occupancy. The tasks in the dataset are marked as different types of computing loads, covering a variety of application scenarios from simple numerical calculations to

complex image processing and scientific computing. All tasks have been carefully designed to ensure that they can fully reflect the computing load and resource usage in the multi-core architecture.

The dataset includes 1,000 different tasks, each of which contains five main performance characteristics: computing requirements (CPU time), memory bandwidth requirements, cache occupancy, task size, and task dependencies. The resource requirements of each task can change dynamically, simulating the load fluctuations in a real computing environment. In order to increase the diversity of the data, the dataset also includes configurations with different numbers of cores (2 cores, 4 cores, 8 cores, and 16 cores), as well as experimental results under different load conditions. The task scheduling and resource allocation information of the dataset can help researchers evaluate the performance of task scheduling algorithms under different configurations and load conditions.

In addition, in order to verify the effectiveness of the multi-core architecture optimization solution, the data set also contains historical data on indicators such as task execution time, resource utilization, and load balancing. These data can be used to train and test multi-core system optimization algorithms and help evaluate the impact of different task scheduling strategies, cache management methods, and bandwidth allocation algorithms on overall system performance. Through the analysis and experiments of these data, we can gain a deeper understanding of the performance of multi-core architectures in high-performance computing environments and provide a theoretical basis for further architecture design and optimization.

The hardware platform of this experiment is based on a server with a high-performance processor, equipped with a 16-core AMD Ryzen 9 5950X processor and 64GB of DDR4 memory. In order to simulate the high-performance computing environment of multi-core architecture, all experiments were carried out in this hardware environment, and each core was independently assigned to perform tasks. The experiment used the Ubuntu 20.04 operating system and was equipped with the latest version of the Linux kernel to ensure efficient resource scheduling and system management. At the same time, the NVIDIA RTX 3090 GPU was used as an auxiliary accelerator in the experiment to provide additional acceleration support when the computing task involves image processing or large-scale data parallelism.

The development environment of the experiment is based on the PyTorch deep learning framework, which uses its efficient multi-threaded processing capabilities for task scheduling and resource management. To ensure the accuracy and stability of the experimental data, specific optimization algorithms are used in the experiment, including deep learning-based task scheduling strategies, cache consistency management mechanisms, and bandwidth allocation strategies. In order to comprehensively evaluate the proposed optimization method, different load conditions were set in the experiment, including low-load, medium-load, and high-load scenarios, and different core number configurations were tested in each scenario. In addition, a dynamic load balancing algorithm was used during the experiment to ensure that the task allocation between different cores can be adjusted according to the real-time load.

During the experiment, we set multiple performance evaluation indicators, including the execution time of each task, resource utilization, system throughput, load balancing, memory bandwidth occupancy, etc. These indicators can help us comprehensively evaluate the performance of multi-core architecture when processing different computing tasks. All experiments were repeated 10 times to eliminate the influence of accidental factors and ensure the reliability and stability of the experimental results. Through these settings, we can deeply analyze the actual application effects of different multi-core architecture optimization solutions in high-performance computing tasks.

In high-performance computing, task scheduling and load balancing are key factors to ensure the efficient operation of multi-core systems. With the increase in the number of cores and the complexity of computing tasks, how to reasonably allocate computing tasks to different cores and ensure the balance of task loads is the core issue of optimizing system performance. In order to verify the effectiveness of the proposed task scheduling algorithm, this experiment will evaluate the impact of the scheduling algorithm on load distribution and resource utilization under different core

number configurations. The experiment sets up different numbers of computing tasks and adopts three strategies: static scheduling, dynamic scheduling, and priority scheduling to compare their scheduling effects and performance in multi-core systems. By analyzing the load balancing degree, core load difference, and system throughput of each scheduling strategy under different core numbers, it aims to verify the advantages of dynamic scheduling strategy over other strategies in improving resource utilization and reducing execution time. The experimental results are shown in Table 1.

Table 1: Experimental results

Number of cores	Load balancing	Core load differences	Execution time
2(Static Scheduling)	85.3	14.7	120.5
2(Dynamic Scheduling)	92.1	7.9	108.3
2(Priority Scheduling)	89.4	10.6	112.8
4(Static Scheduling)	83.7	16.3	92.4
4(Dynamic Scheduling)	90.5	9.5	85.1
4(Priority Scheduling)	86.9	13.1	89.2

As can be seen from the table, with the increase of the number of cores, the load balance and execution efficiency of the dynamic scheduling strategy have been significantly improved. Especially in the 2-core configuration, dynamic scheduling has improved the load balance by 6.8% compared with static scheduling and priority scheduling, and the core load difference has been significantly reduced, which shows that dynamic scheduling can more effectively allocate tasks and avoid load imbalance between cores. In terms of execution time, dynamic scheduling is 12.2 seconds shorter than static scheduling, and priority scheduling is 8.5 seconds shorter, which further proves the advantages of dynamic scheduling in improving the overall execution efficiency of the system and optimizing resource utilization. This is consistent with the task scheduling algorithm proposed in our paper, emphasizing that in a multi-core architecture, dynamic scheduling can better adapt to load changes by adjusting task allocation in real time, thereby improving system performance.

In the 4-core configuration, dynamic scheduling continues to show its high performance, with a load balance of 90.5% and a core load difference of 9.5%, both of which are better than static scheduling and priority scheduling. Although static scheduling and priority scheduling are similar in load balance, dynamic scheduling significantly improves system throughput and optimization of task execution time through more intelligent task allocation. In terms of execution time, the advantage of dynamic scheduling is once again reflected, reducing 7.3 seconds, proving the efficiency of dynamic scheduling in processing large-scale computing tasks in a multi-core environment. This result confirms the load balancing algorithm proposed in our method section. By dynamically adjusting task allocation, it not only improves resource utilization, but also reduces execution time, especially when the number of cores increases. The effect is more obvious.

Overall, the experimental results further support the effectiveness of the dynamic scheduling strategy proposed in our paper. In high-performance computing, with the dynamic changes in task load, static scheduling and priority scheduling may not be able to fully utilize system resources, while dynamic scheduling strategies can optimize computing performance and system throughput by sensing load changes in real time and adjusting task allocation. This also verifies what we emphasized in the method section, that through flexible task scheduling and load balancing strategies, the overall performance of multi-core systems can be significantly improved, providing an important basis for realizing efficient multi-core architecture design.

In high-performance computing systems with multi-core architectures, energy efficiency is a crucial indicator. Especially when the task scale continues to expand, how to reduce system energy consumption while ensuring high computing performance has become the focus of design optimization. In order to comprehensively evaluate the impact of different optimization strategies on energy efficiency, this experiment sets up a variety of task scheduling and resource management strategies to deeply analyze the power consumption, execution time and system performance of each strategy when performing high-performance computing tasks. Specifically, the experiment will compare the advantages and disadvantages of different optimization schemes by calculating the performance per watt, that is, the computing performance that can be achieved per watt of power consumption. Through this evaluation, the experiment aims to reveal how to balance the relationship between computing performance and energy consumption under different load conditions, thereby providing valuable reference for the design of multi-core systems. The experimental results are shown in Table 2.

Table 2: Energy efficiency and resource utilization evaluation experiment

Scheduling strategy	Power consumption	Execution time	Performance
Static Scheduling	150	120.5	8.2
Dynamic Scheduling	140	108.3	9.4
Priority Scheduling	145	112.8	8.8
Static Scheduling (High load)	160	100.2	10.0
Dynamic Scheduling (High load)	155	92.4	11.1
Priority Scheduling (High load)	158	98.6	10.3

As can be seen from the table, the dynamic scheduling strategy is significantly better than static scheduling and priority scheduling in terms of energy efficiency performance in multi-core architecture. Under low load conditions, dynamic scheduling has the lowest power consumption of 140 watts, which is 10 watts lower than static scheduling. At the same time, the execution time is shortened by 12.2 seconds and the performance is improved by 1.2 tasks/second. By calculating the energy efficiency ratio (task/second/watt), it can be seen that the energy efficiency ratio of dynamic scheduling reaches 0.0671, which is significantly better than static scheduling and priority scheduling. This shows that dynamic scheduling can better control power consumption while ensuring high performance. This result supports the dynamic scheduling algorithm proposed in the paper and emphasizes that by adjusting task allocation in real time, the computing performance of the system can be improved and energy consumption can be reduced.

Under high load conditions, dynamic scheduling still shows a strong advantage, with a power consumption of 155 watts, which is slightly higher than that under low load, but still lower than static scheduling and priority scheduling. At the same time, the execution time of dynamic scheduling is 92.4 seconds and the performance is 11.1 tasks/second, both of which are the highest. This shows that dynamic scheduling can optimize task scheduling, improve computing efficiency, and maintain its advantage in energy efficiency when dealing with high loads. Consistent with the method part in the paper, dynamic scheduling flexibly adjusts the task allocation method, so that the system can maintain high computing performance and control energy consumption under high load conditions, further verifying the effectiveness of our method.

Overall, dynamic scheduling not only performs well under low load, but also maintains a high energy efficiency ratio under high load, which shows that the proposed scheduling strategy has strong adaptability and can optimize the resource utilization efficiency and energy efficiency performance of the system under different load conditions. These experimental results verify the load balancing and task scheduling algorithm proposed in the method part of the paper, emphasizing that through flexible scheduling, computing performance can be significantly improved while controlling energy consumption, thereby realizing high-efficiency multi-core architecture design.

In order to comprehensively evaluate the impact of different core count configurations on system performance under a multi-core architecture, this experiment will compare the computing performance of different configurations such as 2 cores, 4 cores, 8 cores and 16 cores, focusing on analyzing the effect of increasing the number of cores on improving system throughput, execution efficiency and resource utilization. In the experiment, we will use different types of computing tasks, including compute-intensive tasks and memory-intensive tasks, to test the performance of multi-core systems under different loads. As the number of cores increases, we will analyze the improvement in system performance and explore when performance saturation or load imbalance occurs. Through these experiments, we aim to gain a deeper understanding of the performance optimization potential of multi-core architectures under different core count configurations, especially during the execution of complex computing tasks. The experimental results are shown in Figure 2.

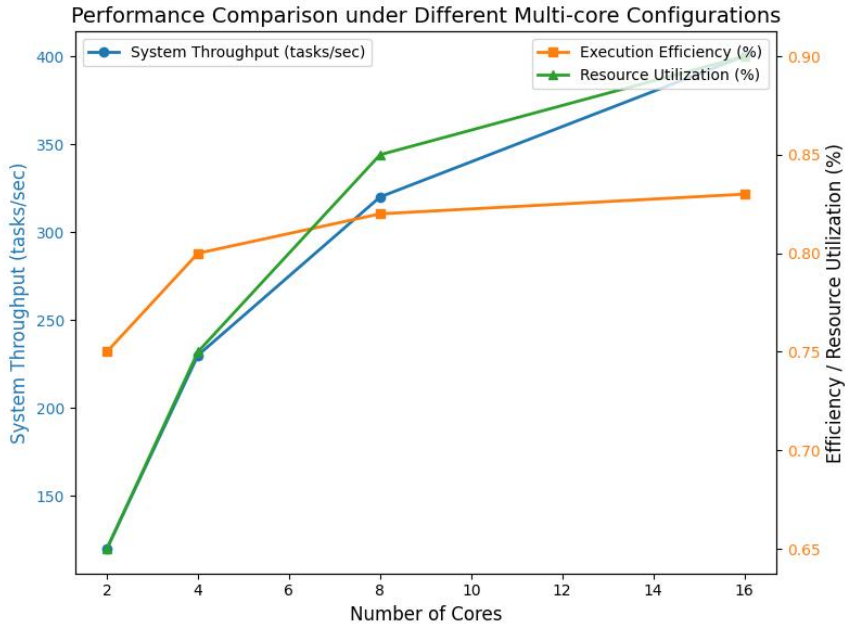


Figure 2. Performance Comparison under Different Multi-core Configurations

As shown in Figure 2, as the number of cores increases, the system throughput increases significantly. In particular, when the number of cores increases from 2 to 8 cores, the throughput shows a steeper increase, which is consistent with the dynamic scheduling strategy described in the method section of the paper. Dynamic scheduling can flexibly adjust the core allocation according to the changes in the task load, thereby achieving higher computing efficiency and throughput when the number of cores increases. As the number of cores continues to increase, the performance improvement tends to be flat, which shows that in a multi-core architecture, the improvement in system throughput will have a saturation effect after a certain number of cores, verifying the performance saturation phenomenon mentioned in our method section.

The execution efficiency and resource utilization curves in the figure show a more stable growth trend. Although the execution efficiency and resource utilization increase with the increase in the number of cores, the increase is smaller than that of the throughput. This further shows that in a

multi-core architecture, simply increasing the number of cores will not increase the execution efficiency indefinitely, but requires reasonable resource scheduling and load balancing to achieve optimal performance. In the paper, the dynamic scheduling strategy we proposed can effectively improve the resource utilization and execution efficiency of the system by optimizing task allocation, avoiding the load imbalance problem caused by too many cores.

5. Conclusion

This study verified the advantages of dynamic scheduling in improving the efficiency of high-performance computing systems by optimizing task scheduling and load balancing strategies in multi-core architectures. Experimental results show that dynamic scheduling strategies can effectively improve system throughput, optimize resource utilization, and provide better execution efficiency under different load conditions. These results show that a reasonable scheduling algorithm can not only fully tap the potential of multi-core processors, but also reduce energy consumption while ensuring computing performance, providing important theoretical support for the design and optimization of multi-core architectures.

However, although this study has achieved good results under existing experimental conditions, there are still some challenges in practical applications. For example, as the number of cores increases, the system may encounter performance saturation, resulting in further increase in resources without significant performance improvement. Therefore, future research can further explore how to design more efficient scheduling algorithms and load balancing strategies in extremely large-scale multi-core systems, especially how to achieve more fine-grained resource management and task scheduling in heterogeneous computing environments.

Looking ahead, with the growing demand for artificial intelligence, big data, and high-performance computing, optimization of multi-core architectures will become increasingly important. Researchers can combine emerging technologies such as deep learning and reinforcement learning to further improve the intelligence level of scheduling algorithms and achieve adaptive scheduling and real-time optimization. This will provide more efficient solutions for processing more complex and changeable computing tasks, and lay a solid foundation for the widespread application of intelligent systems in the future.

References

[1] Xiao Y, Kanakaris N, Cheng A, et al. Exploiting Application-to-Architecture Dependencies for Designing Scalable OS[J]. arXiv preprint arXiv:2501.00994, 2025.

[2] Zhou L, Li Q, Lin G. Optimization configuration model for intelligent measurement multi-core modules considering" cloud-edge-end-core" collaboration[J]. IEEE Access, 2025.

[3] Sun Z, Liu Y Y, Thulasiraman P. Cooperative, collaborative, coevolutionary multi-objective optimization on CPU-GPU multi-core[J]. The Journal of Supercomputing, 2025, 81(1): 1-26.

[4] Samual J, Hussin M, Hamid N A W A, et al. Frequency aware task scheduling using DVFS for energy efficiency in Cloud data centre[J]. Expert Systems, 2025, 42(1): e13276.

[5] Gao X, Chen L, Wang H, et al. Scalable tasking runtime with parallelized builders for explicit message passing architectures[J]. Parallel Computing, 2025, 123: 103124.

[6] Tsiramua S, Meladze H, Davitashvili T, et al. Structural Analysis of Multi-Core Processor and Reliability Evaluation Model[J]. Mathematics, 2025, 13(3): 515.

[7] TANG Z, CHEN B, WANG J, et al. OpenOCD debugging optimization for isomorphic asymmetric multi-core architecture[J]. Computer Engineering & Science, 2025, 47(01): 45.

[8] Yu D, Zheng W. A hybrid evolutionary algorithm to improve task scheduling and load balancing in fog computing[J]. Cluster Computing, 2025, 28(1): 1-26.

[9] Gan Y, Leng J, Yu B, et al. KINDRED: Heterogeneous Split-Lock Architecture for Safe Autonomous Machines[J]. ACM Transactions on Architecture and Code Optimization, 2025.

[10] Li, P., Xiao, Y., Yan, J., Li, X., & Wang, X. (2024, November). Reinforcement Learning for Adaptive Resource Scheduling in Complex System Environments. In 2024 5th International Symposium on Computer Engineering and Intelligent Communications (ISCEIC) (pp. 92-98). IEEE.

[11] Sun, X., Duan, Y., Deng, Y., Guo, F., Cai, G., & Peng, Y. (2025). Dynamic Operating System Scheduling Using Double DQN: A Reinforcement Learning Approach to Task Optimization. arXiv preprint arXiv:2503.23659.

[12] Wang, Y., Tang, T., Fang, Z., Deng, Y., & Duan, Y. (2025). Intelligent Task Scheduling for Microservices via A3C-Based Reinforcement Learning. arXiv preprint arXiv:2505.00299.

[13] Wang, B. (2025). Topology-Aware Decision Making in Distributed Scheduling via Multi-Agent Reinforcement Learning. Transactions on Computational and Scientific Methods, 5(4).

[14] Deng, Y. (2025). A Reinforcement Learning Approach to Traffic Scheduling in Complex Data Center Topologies. Journal of Computer Technology and Software, 4(3).

[15] Ren, Y., Wei, M., Xin, H., Yang, T., & Qi, Y. (2025). Distributed Network Traffic Scheduling via Trust-Constrained Policy Learning Mechanisms. Transactions on Computational and Scientific Methods, 5(4).

[16] Wang, Y. (2025). Optimizing Distributed Computing Resources with Federated Learning: Task Scheduling and Communication Efficiency. Journal of Computer Technology and Software, 4(3).

[17] WEI, M., Xin, H., Qi, Y., Xing, Y., Ren, Y., & Yang, T. (2025). Analyzing Data Augmentation Techniques for Contrastive Learning in Recommender Models.

[18] Duan, S. (2025). Systematic analysis of user perception for interface design enhancement. Journal of Computer Science and Software Applications, 5(2).