

Hybrid-Adaptive Thread Tuning to Mitigate Simulation Execution Bottlenecks in High-Performance Reinforcement Learning Inference

Jiming Su¹, Hantao Hua², Lujia Yin¹, Yiping Yao¹ and Feng Zhu^{1*}

¹College of Systems Engineering, National University of Defense Technology

²College of Computer Science and Technology, National University of Defense Technology

{jimings, ht.hua, yinlujia10, ypyao, zhufeng}@nudt.edu.cn

Abstract

In simulation-in-the-loop decision-making systems, reinforcement learning (RL) inference is often constrained by simulator-side execution overhead, where workloads are highly dynamic and sensitive to runtime thread configurations. Existing multithreaded strategies struggle to match thread resources before or during execution, causing resource contention, scheduling overhead, and reduced throughput. Through empirical analysis, we identify the ratio of task execution time to scheduling time as the key factor determining the optimal thread count. Building on this insight, we propose AutoThread, a hybrid adaptive thread-tuning method for mitigating simulation bottlenecks in RL inference. AutoThread employs a Physics-Informed Neural Operator (PINO) as a thread-count predictor and incorporates a finite-source M/M/1 queueing model to constrain and guide prediction, enabling fast and accurate estimation under dynamic workloads. It further performs load-aware online fine-tuning to compensate for prediction errors and refine resource allocation. Experiments show that AutoThread improves average speedup by 18.4% over static strategies, achieves average throughput of 1.7× and 1.8× that of XGBoost and Reinforcer, respectively, and reduces execution time by up to 83.8% compared with state-of-the-art methods. Our code and dataset are publicly available at <https://github.com/suchenjm/AutoThread>.

1 Introduction

As artificial intelligence is increasingly used for decision-making in complex dynamic systems, reinforcement learning with simulation-in-the-loop (SiL-RL) has become a core paradigm for wireless communication network optimization and swarm intelligence control[Feriani and Hossain, 2021; Hu *et al.*, 2024]. Its operational efficiency, however, directly limits real-time decision-making. SiL-RL performance depends on three components: simulation-side model computation (e.g., social network dynamic evolution models[Wu *et al.*, 2022]), agent-side algorithm inference (e.g., Deep Q-Network decision-making[Zhang *et al.*, 2021]), and their

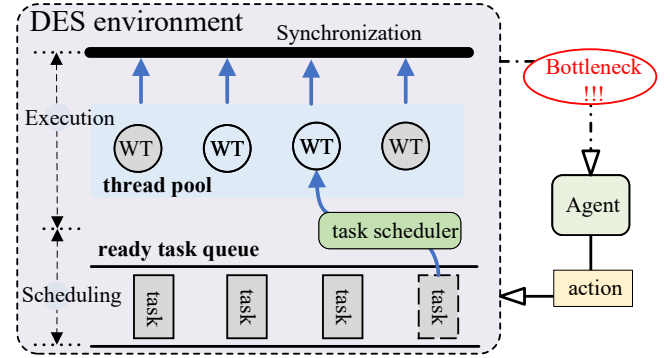


Figure 1: Diagram of the DES environment execution process

communication/synchronization. As simulation models grow increasingly complex (e.g., high-precision channel interference models[Eldeeb and Alves, 2025] in communication networks or high-fidelity aerodynamic models[Riboldi *et al.*, 2025] in UAV swarms), the simulation side becomes the critical bottleneck[Morgado *et al.*, 2025], increasing end-to-end latency and impairing real-time decisions. This paper therefore accelerates simulation-side performance to break this bottleneck and support efficient SiL-RL systems.

Existing SiL-RL simulation environments typically adopt Discrete Event Simulation (DES) and use multithreading on multicore processors for parallel acceleration[Belsare *et al.*, 2022]. They mainly employ two architectures: centralized queues with thread pools and distributed queues with work stealing[Risco-Martín *et al.*, 2022]. The centralized queue, with dynamic load balancing and lower synchronization overhead, is well suited to SiL-RL inference, where agent activation rates and task loads fluctuate across simulation cycles[Liu *et al.*, 2025]. This work therefore focuses on simulation acceleration under the centralized queue architecture.

However, centralized queues are highly sensitive to the number of worker threads (WTs): all WTs share one task queue, and task acquisition and distribution are handled serially by the scheduler[Samadi *et al.*, 2024]. Thus, the scheduler may become a serial bottleneck: as WTs increase, scheduler contention, lock waiting, and busy waiting rise sharply, negating or even reversing parallel gains. Selecting WTs according to workload characteristics is therefore key to im-

proving simulation-side performance [Li *et al.*, 2025]. First, the optimal WT count depends on workload intensity and hardware factors such as core frequency, memory bandwidth, and cache hierarchy, which vary across platforms and are hard to know a priori. Second, SiL-RL workloads are highly dynamic; agent activation rates, task execution times, and interaction intensities fluctuate across simulation stages, making fixed WT counts suboptimal. Runtime noise and external disturbances further obscure transient interference versus persistent workload shifts. Together, these factors prevent precise WT allocation before or during runtime.

To address this, we propose AutoThread, a hybrid adaptive thread-tuning method for SiL-RL simulation acceleration. The core idea is to construct a WT count predictor where a neural network is guided and constrained by a theoretical performance model. By introducing a dynamic load-aware adaptive fine-tuning strategy, we achieve significant improvements in simulation performance, thereby supporting high-performance inference in RL decision-making systems. The main contributions of this paper are as follows:

- We open-source a large-scale multithreaded trajectory dataset for DES environment. Through empirical analysis, we derive three findings regarding the relationship between WT count and simulation performance.
- We develop a Physics-Informed Neural Operator (PINO) thread predictor, utilizing a queueing model to constrain and guide the training process.
- We propose AutoThread, which leverages PINO-based thread prediction and load-aware online fine-tuning to achieve fast and adaptive WT count allocation.

2 Empirical Study

We first introduce the dataset, then empirically reveal the relationship between WT count and execution performance to guide AutoThread’s model construction and tuning strategy.

2.1 Dataset

We constructed the multithreaded trajectory dataset for the DES environment by collecting over 10,000 multithreaded execution samples across both Intel and AMD CPU hardware platforms. The dataset includes two application cases:

PCS: The Personal Communication System (PCS) [Carothers *et al.*, 2002] is a benchmark DES environment that simulates wireless communication processes within a network topology and is widely used to test decision algorithms in communication networks. In this application, each agent represents a communication node, continuously generating five types of communication tasks.

UAV: This is a case study based on the DES paradigm, simulating the autonomous navigation and obstacle avoidance of multiple unmanned aerial vehicles (UAVs) within a specific area. The paths are derived from real-world flight data [FlightAware, 2025]. Each agent represents a UAV, involving both perception and navigation tasks.

To emulate high-fidelity simulation, we embedded array dot products and iterative power computations into each task, producing execution times between 10 μ s and 800 μ s. With

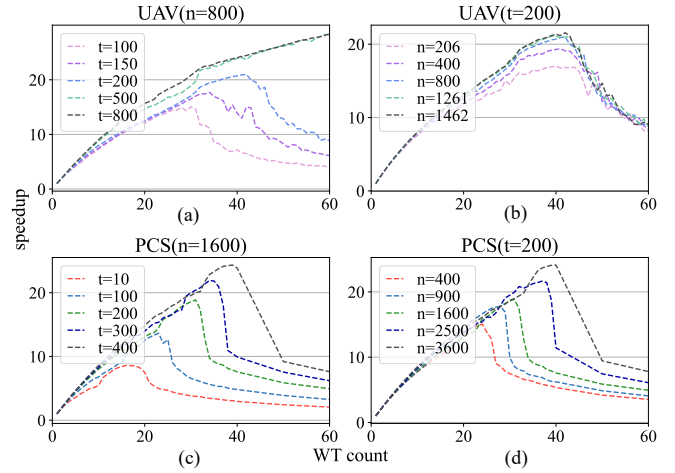


Figure 2: Variation of speedup with WT count under different application parameters, where t denotes the average task execution time and n denotes the number of agents

profiling tools, we captured each sample’s execution trajectories under various WT counts. The data include hardware utilization, application logs, and micro-architectural features, totaling 34 variables. To our knowledge, this is the first public DES trajectory dataset for RL systems and among the largest single-node thread-level trace sets; we release it to support studies on multithreaded behavior and DES acceleration.

2.2 Empirical Observations

We conduct three experiments to characterize how workload, hardware, and micro-architectural metrics jointly affect performance. Our analysis yields the following observations.

Dependence of Optimal WTs on Workload Intensity

Figure 2 shows speedup versus WT count under different agent populations and task execution times. In UAV, each agent always generates two fixed tasks with execution time t ; in PCS, each agent generates a variable number of tasks whose execution times follow a Poisson distribution with mean t . The optimal WT count is positively correlated with average task execution time: higher workload intensity increases the optimal WT count because longer tasks enlarge parallelizable regions and let more threads improve performance. Conversely, with fixed execution time, the optimal WT count remains nearly unchanged regardless of agent scale (Figure 2(b)). The variance in Figure 2(d) arises because changing the number of PCS agents alters the task-queue execution-time distribution and thus its mean.

Contention-Induced Waiting as the Cause of Degradation

We conduct detailed analysis on a subset of samples using VTune. Figure 3 displays the distribution of hotspot functions and thread time consumption. As shown in Figure 3(a), the proportion of time spent on lock operations rises rapidly as the WT count increases. Once the lock-wait time exceeds the task execution time, system performance begins to degrade. Figure 3(b) further shows that the critical point at which busy waiting starts to increase sharply around the op-

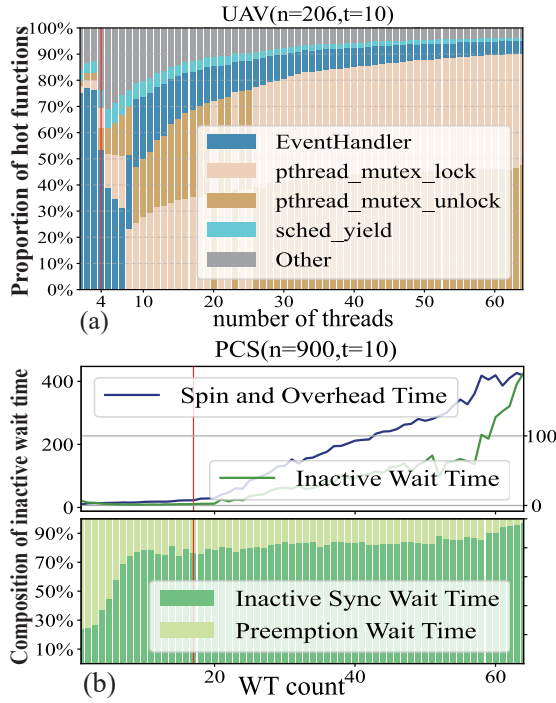


Figure 3: Hotspot function distribution and thread time consumption in DES environment

timal WT count is insensitive to inactive wait times such as inactive synchronization and preemption.

Although busy waiting time is key to the optimal WT count, measuring it online is challenging [Jancauskas *et al.*, 2019]. We therefore use lightweight metrics to indirectly reflect busy-waiting status, as shown in Fig 4. Figure 4(a) shows that *usr_cpu* peaks near the optimal WT count, where CPU cycles are most efficiently used for parallel tasks. Once the WT count exceeds the optimum, *system_cpu* (Figure 4(b)) rises sharply, indicating a shift from effective computation to OS scheduling and thread waiting. Since busy-waiting threads still occupy physical cores, aggregate CPU utilization does not accurately reflect the optimal WT count (see Figure 4(c)). Figure 4(d) further shows that the context switch rate reaches a local minimum near the optimum, suggesting that an appropriate thread count reduces unnecessary scheduling switches. In contrast, cache miss rate (Figure 4(e)) and memory consumption (Figure 4(f)) affect DES performance but not optimal WT selection.

Linear Interference Under Multicore Concurrency

Figure 5 records the task execution time and task scheduling time for an individual working thread under varying numbers of active threads. The results show that as the number of active threads increases, both the execution and scheduling times for a single thread grow linearly. This indicates that the performance degradation of individual threads in a multithreaded environment can be approximated by a linear model, which captures the aggregate pressure exerted by all threads on a single thread’s runtime. The slope and intercept of these linear functions depend on hardware characteristics

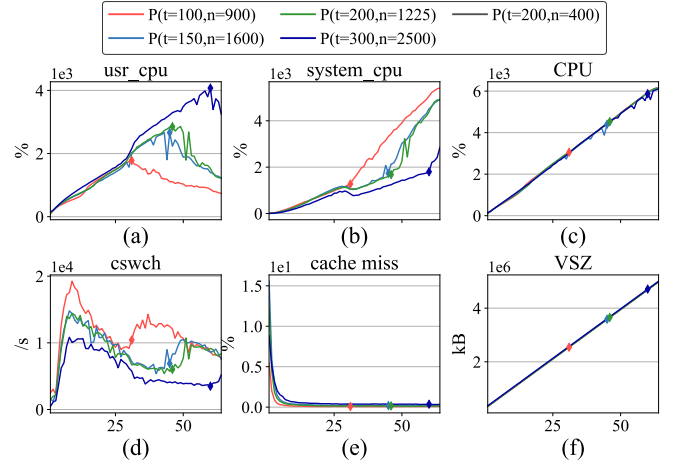


Figure 4: Variation of DES environment’s micro-architectural Metrics with WT count

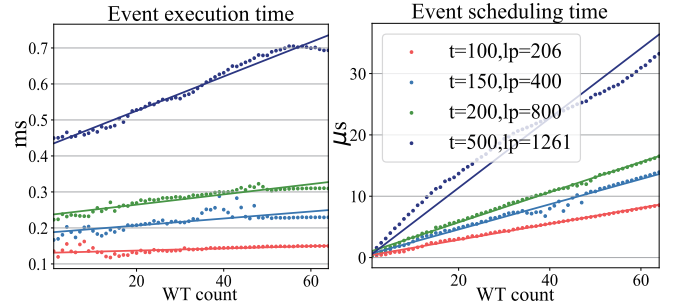


Figure 5: Performance interference among multiple WTs

(e.g., core frequency, cache size) and workload intensity.

3 Methodology

Based on the empirical findings in Section 2, we propose AutoThread. AutoThread consists of two core components: a PINO predictor and a dynamic tuner. This approach adopts a two-stage optimization strategy of online prediction followed by dynamic adjustment: during the online prediction stage, it rapidly estimates the optimal initial thread count based on the characteristics of the current workload; during the dynamic adjustment stage, it performs fine-grained adaptive thread tuning according to runtime feedback.

3.1 Model-Informed PINO Predictor

Performance Model. In typical SiL-RL inference workflows, the simulator must process a large number of computation tasks generated by multiple agents within each simulation step. We decompose the execution of a single simulation cycle into three sequential phases: task execution, task scheduling, and synchronization. WT reconfiguration is performed at the granularity of simulation cycles, during which the WT count remains fixed.

During the task execution phase, WTs process assigned tasks in parallel. Since the system must wait for all threads to finish before synchronization, the total execution time is

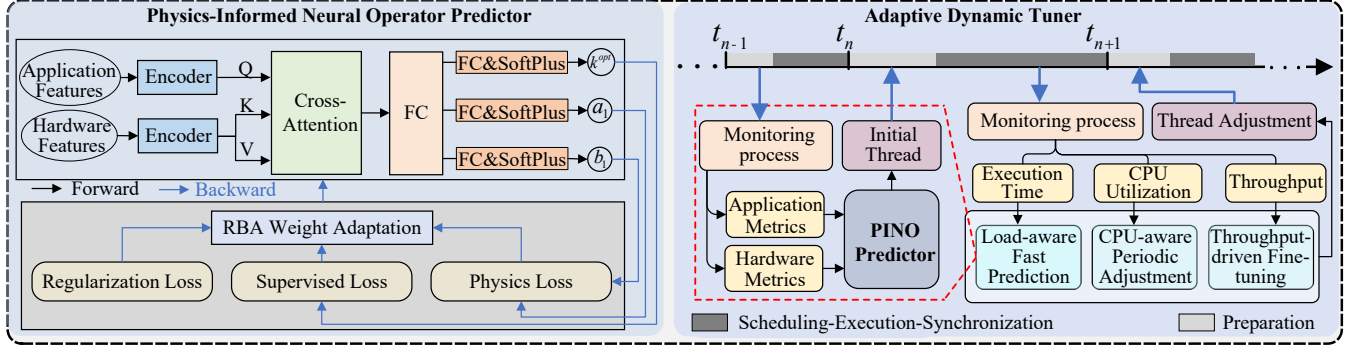


Figure 6: Overall workflow of AutoThread

determined by the slowest thread:

$$T_{exe}(k) = \left\lceil \frac{N}{k} \right\rceil * \frac{1}{\mu_e}$$

where N is the total number of tasks within the current simulation step; k is the number of WTs; and μ_e denotes the average service rate of tasks within the current simulation step.

Scheduling is inherently serial because the shared scheduler serves at most one WT at a time. We model the scheduler as an M/M/1 server and WTs as customers in a finite-source closed queueing system, where a WT requests scheduling only after finishing its current task, yielding an effective arrival rate $\lambda = k\mu_e$. When the system is stable, steady-state Continuous-Time Markov Chain (CTMC) analysis gives:

$$T_{sch} = N * \left(\frac{k}{\mu_b(1-P_0)} - \frac{1}{k\mu_e} \right), P_0 = \frac{1}{\sum_{i=0}^k \frac{k!}{(k-i)!} \left(\frac{k\mu_e}{\mu_b} \right)^i}$$

where μ_b denotes the scheduler service rate and P_0 is the idle probability of the scheduler. When $k\mu_e$ exceeds μ_b , the scheduler becomes the dominant bottleneck and the total time is bounded by the scheduler capacity:

$$T_{exe}(k) + T_{sch}(k) = \frac{N}{\mu_b} + \frac{1}{\mu_e}$$

This captures the saturation regime under oversubscription.

The synchronization phase (e.g., global state update and clock advance by the master thread) is treated as constant with respect to k when focusing on WT selection.

Based on Section 2.2.3, resource contention causes performance degradation that grows linearly with active WT count:

$$\mu_b(k) = \frac{1}{a_1 k + t_{base1}}, \quad \mu_e(k) = \frac{1}{a_2 k + t_{base2}}$$

where a_1 and a_2 denote contention intensity coefficients for scheduling and execution, respectively, and t_{base1} , t_{base2} represent baseline times under single-thread execution. These parameters are determined by hardware platform and application context.

Combining the above, the total execution time of a single simulation cycle can be expressed as:

$$T(k) = \begin{cases} \left\lceil \frac{N}{k} \right\rceil * \frac{1}{\mu_e} + N * \left(\frac{k}{\mu_b(1-P_0)} - \frac{1}{k\mu_e} \right), & \text{if } k\mu_e \leq \mu_b \\ \frac{N}{\mu_b} + \frac{1}{\mu_e}, & \text{if } k\mu_e > \mu_b. \end{cases} \quad (1)$$

The above equation shows that execution time is a piecewise-continuous, non-smooth function of the WT count k . Numerical analysis reveals that the optimal configuration occurs at $k^* \approx \mu_b/\mu_e$, indicating optimal performance when scheduler service rate balances with task execution rate.

PINO Predictor. While μ_e can be estimated online from probe-collected execution times, directly measuring μ_b is difficult due to microsecond-scale sensitivity and measurement overhead. We therefore introduce a Physics-Informed Neural Operator (PINO) to fuse the above model with historical traces for fast online prediction.

We define a neural operator $\mathcal{G}_\theta : a \rightarrow k^* = \mathcal{G}_\theta(a)$, which maps system state parameters a to the optimal WT number k^* . We construct the feature vector $a = \{N, \mu_e, k_{obs}, usr_cpu, system_cpu, VSZ, csuch\}$ from both application and hardware features. Although virtual memory usage (VSZ) has limited impact on thread decisions, excessive memory consumption may trigger resource contention or even abnormal program termination; thus, it is included as an input to enhance robustness.

As illustrated in Figure 6, PINO predictor employs a multi-output head cross-attention architecture. Encoders separately process application and hardware features, cross-attention captures their deep interactions, and FC layers decode fused features into optimal WT count k^* and contention coefficients a_1, b_1 . Physical loss:

$$\mathcal{L}_{phy} = \|T(N, k, \mu_b, \mu_e) - T_{obs}\|_2 + \alpha \left\| \frac{\mu_b}{\mu_e} - k^* \right\|_2$$

Supervised loss and Regularization term:

$$\mathcal{L}_{sup} = \|k^* - k_{obs}\|_2, \quad \mathcal{L}_{reg} = \gamma \|\theta\|_2$$

where T represents the theoretical performance model (Equation 1), μ_b is derived from the contention model as $\mu_b = a_1 k + b_1$, and T_{obs} and k_{obs} denote the observed task queue execution time and iteratively measured optimal WT number, respectively. The weight α normalizes contributions of the two physical components. The overall training objective is $\mathcal{L}_{loss} = \mathcal{L}_{phy} + \lambda \mathcal{L}_{sup} + \mathcal{L}_{reg}$. We incorporate Residual-Based Attention (RBA) [Anagnostopoulos *et al.*, 2024] to dynamically adjust λ between physical and supervised losses. In out-of-distribution scenarios, RBA automatically increases the physical-loss weight, strengthening reliance on the performance model and improving prediction robustness.

3.2 Adaptive Dynamic Tuner

Based on the trained PINO predictor, we design an adaptive dynamic tuner adopting a three-tier progressive adjustment strategy. The tuner aims to establish a closed-loop control process of prediction–execution–feedback–readjustment, so as to mitigate adverse effects caused by prediction errors, system noise, and measurement bias. The complete workflow is presented in Algorithm 1.

Load-Aware Fast Prediction. During each execution cycle, the system automatically records the average service time of the current task queue t_{cur} using probe pointer and computes the average processing time of the queue T_{cur} (lines 1-3). When a change in t_{cur} exceeding a preset threshold is detected, the tuner immediately invokes PINO to predict the optimal WT count and update it (lines 4-6). This mechanism ensures fast responsiveness to sudden load changes.

CPU-Aware Periodic Adjustment. To avoid continuous prediction overhead and handle gradually varying loads, a periodic checking mechanism is introduced. The check interval is dynamically adaptive: if prediction was triggered in the previous cycle, a short interval C_1 is used to verify and correct predictions quickly; otherwise, a longer interval C_2 is adopted to reduce monitoring overhead (line 7). At a checkpoint, if the observed $system_cpu$ exceeds the usr_cpu , the algorithm proactively reduces the number of WTs to mitigate contention (lines 8–16). This mechanism effectively maps CPU indicators of resource contention into direct signals for thread adjustment, thereby preventing performance degradation due to oversubscription.

Throughput-Driven Fine-tuning. When the system is detected to be in a high-contention state, the tuner switches to a fine-tuning phase. In this phase, throughput of the task queue is directly optimized as the adjustment objective. If reducing threads leads to improved throughput, the adjustment is accepted and further attempts are made; otherwise, the change is rolled back and the phase is terminated (lines 17–25). Considering that undersubscription has a less detrimental effect on performance than the severe contention caused by oversubscription, the fine-tuning phase only considers reducing threads. Moreover, to avoid erroneous adjustments due to random fluctuations or transient anomalies, the tuner exits this phase only if throughput fails to improve over three consecutive simulation steps or if the thread count reaches the minimum allowed value (line 20).

4 Evaluation

In this section, we first introduce the experimental setup and current thread tuning methods. Subsequently, we evaluate the proposed method against these baselines in terms of system acceleration performance and adaptability. Finally, we conduct ablation studies to quantify the contributions of each optimization module and verify the robustness of our method with respect to various hyperparameters.

4.1 Setup

The training set is derived from averaged statistics over the entire execution under constant-load scenarios, where the

Algorithm 1 Three-Level Progressive Thread Adjustment

Input: δ (load change threshold), C_1 (short period), C_2 (long period)
Output: WT count k for each simulation step

```

1: while simulation not finished do
2:    $t_{cur} \leftarrow \text{PROBETASKTIME}()$ 
3:    $T_{cur} \leftarrow \text{EXECUTECYCLE}()$ 
4:   if  $|t_{cur} - t_{prev}| / (t_{prev} + \epsilon) > \delta$  and  $tuning = 0$  then
5:      $k \leftarrow \text{PINO-PREDICT}$ 
6:      $predicted \leftarrow \text{TRUE}$ 
7:   else if  $predicted$  and  $(tick \bmod C_1 = 0)$  then
8:      $(system\_cpu, usr\_cpu) \leftarrow \text{GETCPUMETRICS}()$ 
9:     if  $usr\_cpu < system\_cpu$  then
10:       $k \leftarrow k - 1, tuning \leftarrow 1$ 
11:     end if
12:      $predicted \leftarrow \text{FALSE}$ 
13:   else if  $\neg predicted$  and  $(tick \bmod C_2 = 0)$  then
14:     if  $usr\_cpu < system\_cpu$  then
15:        $k \leftarrow k - 1, tuning \leftarrow 1$ 
16:     end if
17:   else if  $tuning \geq 1$  and  $k > k_{min}$  then
18:     if  $T_{cur} < T_{prev}$  then  $\triangleright$  Throughput improved
19:        $k \leftarrow k - 1, tuning \leftarrow 1$ 
20:     else if  $tuning < 3$  then  $\triangleright$  Tolerate fluctuation
21:        $tuning \leftarrow tuning + 1$ 
22:     else  $\triangleright$  Rollback and exit tuning
23:        $k \leftarrow k + 1, tuning \leftarrow 0$ 
24:     end if
25:   end if
26:    $t_{prev} \leftarrow t_{cur}, T_{prev} \leftarrow T_{cur}$ 
27:    $\text{SYNCHRONIZE}(); \text{NEXTTICK}()$ 
28: end while

```

execution time of each task remains unchanged throughout the simulation horizon. We split the dataset into training/testing/validation sets with a ratio of 0.7/0.2/0.1 to offline train different predictors. During training, each feature vector is taken as the average over the whole execution; since the load is constant, the optimal thread count label for each sample can be obtained by sweeping different thread counts and recording the total execution time.

In the testing phase, we use dynamic workloads to evaluate the speedup and adaptivity of each method. Specifically, within each simulation step, task execution times follow a Poisson distribution parameterized by t , while the parameter t varies across steps as a square wave with frequency roh (switching among $[t, 1.5t, 2t]$). Each method adjusts based on instantaneous values of the feature variables at a single time point, so as to emulate a practical deployment setting with offline training and online testing.

Experiments are conducted on two multicore architectures: AMD EPYC 9734 (112 Cores/224 Threads) and Intel Xeon Gold 6338 (32 Cores/64 Threads). The baselines include **ADAPT-T**[Costa *et al.*, 2019] based on runtime feedback control; **Otter**[Luan *et al.*, 2022] employing search strategies; **Thread Reinforcer**[Pusukuri *et al.*, 2011] utilizing system resource utilization; **Xgboost**[Akash *et al.*, 2021] as an advanced representative of machine learning-based prediction.

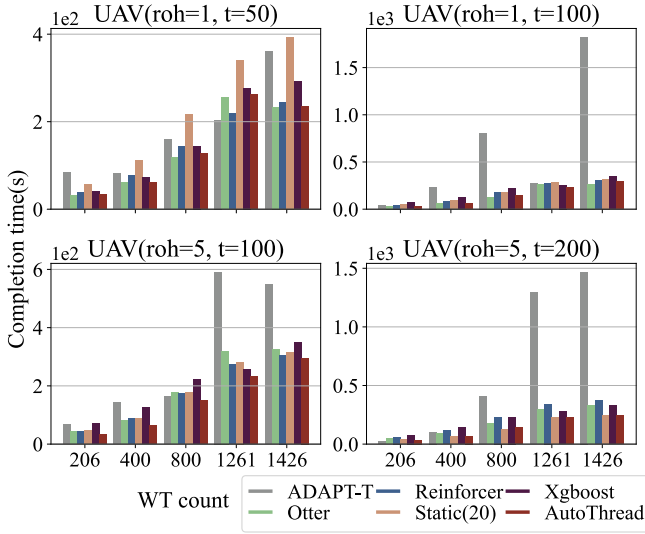


Figure 7: Runtime comparison of different methods on AMD platform for UAV

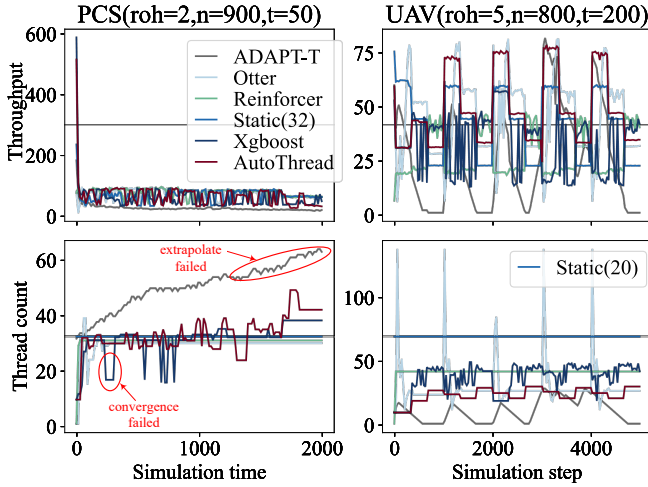


Figure 8: Adaptiveness comparison on PCS and UAV

Additionally, a fixed thread count configuration was used as a control experiment, denoted as **Static**. All methods were implemented on the Repast HPC system [Zhu *et al.*, 2017]. Each test case was executed at least five times to obtain optimal measurement data.

4.2 Performance Experiments

We run the UAV benchmark on the AMD platform with the execution horizon set to 5000 and the load threshold $\delta = 0.3$, and record the total runtime from end to end under different thread adjustment schemes; the results are shown in Fig. 7. Compared to ADAPT-T, AutoThread saves up to 83.8% of runtime in different scenarios. Over 20 test scenarios, AutoThread achieves an average runtime of 149.32 s and delivers 10.3% and 17.2% higher speedup than the state-of-the-art Otter and Reinforcer, respectively. Notably, we set the thread count of the fixed strategy to 20, i.e., the aver-

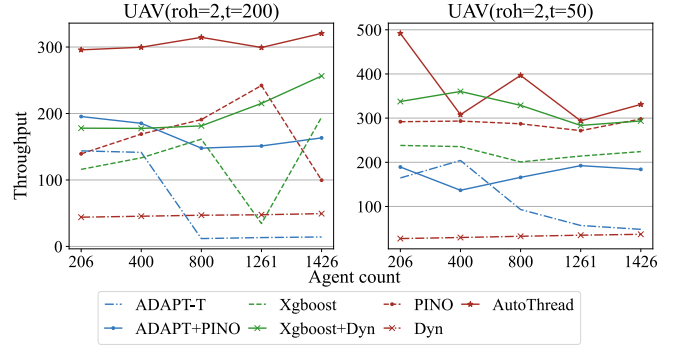


Figure 9: Results of ablation experiments

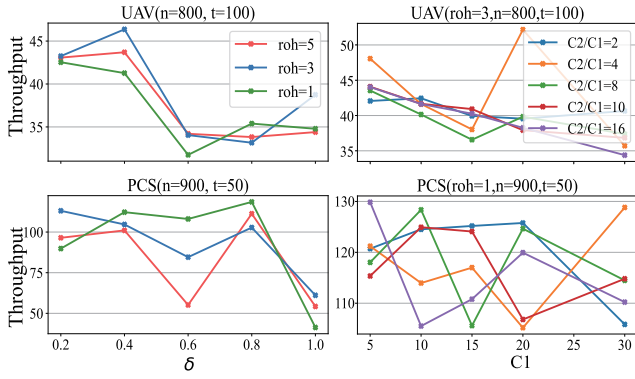
age output of all tuning schemes. Even when accounting for the additional overhead introduced by frequent thread adjustments, AutoThread still attains, on average, an 18.4% higher speedup than the fixed strategy with prior knowledge, with a maximum performance improvement of 62.6%.

4.3 Adaptiveness Experiments

We run the PCS and UAV applications on both Intel and AMD platforms under different load-variation intensities (frequency roh , amplitude t), and record the system throughput over environment steps to evaluate the adaptability of different schemes, where throughput denotes the number of tasks processed per millisecond. Figure 8 shows that, in practical scenarios, when competing methods successively exhibit non-convergent oscillations and extrapolation failures, AutoThread still maintains strong parallel speedup. On the PCS case, AutoThread attains an average throughput of 536.83, improving over ADAPT-T and Xgboost by 37.7% and 2.3%, respectively, while remaining below Reinforcer's 718.55. Under the UAV case with larger workload fluctuations, AutoThread's average throughput is 1.8 $\times$ and 1.7 $\times$ that of Reinforcer and Xgboost, respectively.

4.4 Ablation Experiments

We ablate the PINO predictor and the Adaptive Dynamic Tuner to investigate the performance contribution of each component. The compared variants include: using only the physics-informed neural operator network prediction (**PINO**); using the dynamic tuner with the maximum thread count (**Dyn**); applying the Dyn tuner on top of the Xgboost output (**Xgboost+Dyn**); and pairing the ADAPT-T tuner with the output of PINO (**ADAPT+PINO**). Figure 9 reports the average throughput of different schemes when running the UAV application on the ADM platform. The results show that both the PINO predictor and the Adaptive Dynamic Tuner effectively improve runtime performance. Across 10 test scenarios, the Dyn technique increases the task processing speed of Xgboost by 87.8% on average, with a maximum improvement of 55.2 $\times$. PINO increases the task processing speed of the ADAPT-T tuner by 3.9 $\times$ on average. Moreover, AutoThread improves average throughput by 1.3 $\times$ and 2.0 $\times$ over Xgboost+Dyn and ADAPT+PINO, respectively, demonstrating the combined benefits of the two techniques.


 Figure 10: Robustness analysis of δ and C_1/C_2 parameter

4.5 Robustness Experiment

We further examine the impact of parameters δ , C_1 and C_2 on AutoThread’s performance. The parameter δ represents the maximum tolerable degree of workload change for thread adjustment. An overly small value leads to frequent thread changes due to random hardware jitter, whereas an overly large value prevents timely adaptation to workload changes. AutoThread’s robustness to δ is shown in Figure 10. Benefiting from the theoretical constraints of the physical model and the error-correction capability of the dynamic tuner, AutoThread provides stable speedup even under workload inputs with substantial randomness. In contrast, an excessively large δ weakens the predictor’s sensitivity to workload changes, resulting in performance degradation.

Figure 10 shows the impact of different C_1/C_2 combinations on system performance. We find that a smaller C_1 enables faster response; however, an overly small C_2 causes frequent periodic checks and increases system overhead, while an overly large C_2 may cause the system to miss critical tuning opportunities. The optimal ratio is approximately 2–4, within which short-cycle validation and long-cycle monitoring achieve a good balance—responding quickly to workload changes while avoiding excessive monitoring overhead.

4.6 Discussion

The experimental results clearly demonstrate the advantages of AutoThread in sustaining parallel speedup and substantially reducing execution time, particularly in environments with fluctuating workloads. The key strength of AutoThread lies in the deep integration of data-driven representational capacity with the structural constraints imposed by physical models: on the one hand, it learns latent prior patterns from runtime data to enable rapid prediction under dynamic workloads; on the other hand, by introducing interpretable theoretical models as inductive biases and training constraints, it anchors the prediction process in consistent mechanistic logic, thereby significantly improving robustness and the achievable performance upper bound while preserving interpretability and transferability. This form of mechanism-guided learning coupled with data-corrected inference is central to AutoThread’s superior effectiveness.

Specifically, first, guided by a performance model, PINO enforces queueing-theoretic constraints on the neural net-

work’s predictive outputs, endowing the predictions with strong generalization and physical plausibility; consequently, it can provide reliable initial estimates of thread counts even under workload conditions whose distributions differ from those seen during training. Second, the dynamic tuning mechanism employs a three-level strategy to perform online corrections to the predicted results, while accounting for the overhead of monitoring and adjustment. This enables the system to rapidly converge to a near-optimal thread configuration when the workload changes, thereby accelerating the simulation environment’s runtime efficiency.

5 Related Work

Many WT tuning methods have been proposed across domains; however, compared with this study, they insufficiently address dynamic workload fluctuations or generalizability. Static modeling methods set fixed thread counts through empirical heuristics or offline analysis, such as using a multiple of physical cores [Montesano *et al.*, 2024] or estimating optimum from the ratio of thread creation to maintenance overhead [Luo *et al.*, 2025]. Although simple and efficient, these strategies struggle in complex, volatile environments because they cannot adjust to workload fluctuations in real time.

Machine learning has enabled data-driven thread-configuration prediction, including Reinforcement Learning [Xu *et al.*, 2012], Ensemble Learning [Chao *et al.*, 2025], and online search optimization [Stetsenko and Dyfuchyna, 2019], with applications in large-scale cluster management. However, these methods often rely on high-quality training data; under dynamic scenarios absent from training, pure data-driven models may generalize poorly. Another common category uses adaptive regulation based on online feedback control [Albalawi, 2025]. These methods incrementally adjust WT count, one thread at a time, using metrics such as request arrival rates and service times until performance no longer improves. Although practical for cloud services, such feedback-driven regulation often suffers from slow convergence, latency, and performance oscillations in large thread spaces or under frequent workload variations [Bahadur *et al.*, 2025].

6 Conclusion

This paper proposes AutoThread, a thread-count tuning method for ABS systems to accelerate RL inference. The core contributions are as follows: (1) we open-source a multithreaded trajectory dataset collected from a DES simulation environment, and empirically derive three key insights; (2) we design a PINO-based thread predictor and establish a theoretical model relating the number of WTs to runtime, which is used to constrain and guide model training; and (3) we develop an efficient hybrid thread adjustment method that incorporates a load-aware adaptive tuning strategy on top of PINO’s predictions. Experimental results show that AutoThread delivers excellent parallel speedup and strong workload adaptivity. The proposed approach is portable and can be applied to accelerate other systems based on a shared event queue. Future work will explore batch-sample thread tuning methods for accelerating RL training.

Acknowledgments

We thank the anonymous IJCAI reviewers for their insightful comments. This work was partially supported by the National Natural Science Foundation of China (NSFC) under Grant No.62503491.

Contribution Statement

Jiming Su and Hantao Hua contributed equally to this work. Feng Zhu is the corresponding author.

References

- [Akash *et al.*, 2021] Lakindu Akash, Duneesha Fernando, Malith Jayasinghe, Chamath Keppitiyagama, and Kishanthan Thangarajah. Machine learning based thread pool tuning via program analysis. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 648–653. IEEE, 2021.
- [Albalawi, 2025] Nasser S Albalawi. Dynamic scheduling strategies for cloud-based load balancing in parallel and distributed systems. *Journal of Cloud Computing*, 14(1):33, 2025.
- [Anagnostopoulos *et al.*, 2024] Sokratis J Anagnostopoulos, Juan Diego Toscano, Nikolaos Stergiopoulos, and George Em Karniadakis. Residual-based attention in physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 421:116805, 2024.
- [Bahadur *et al.*, 2025] Faisal Bahadur, Zulfiqar Ahmad, and Abdulmohsen Algarni. Poolrunner: An extensible performance testing simulation tool for thread-pool middleware. *IEEE Access*, 2025.
- [Belsare *et al.*, 2022] Sahil Belsare, Emily Diaz Badilla, and Mohammad Dehghanimohammadabadi. Reinforcement learning with discrete event simulation: the premise, reality, and promise. In *2022 Winter Simulation Conference (WSC)*, pages 2724–2735. IEEE, 2022.
- [Carothers *et al.*, 2002] Christopher D Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of parallel and distributed computing*, 62(11):1648–1669, 2002.
- [Chao *et al.*, 2025] Zhiteng Chao, Feng Gu, Junying Huang, Wenjie Li, Jing Ye, Huawei Li, and Xiaowei Li. Memory-efficient and adaptive heterogeneous framework for gate-level fault simulation. *ACM Transactions on Design Automation of Electronic Systems*, 30(5):1–27, 2025.
- [Costa *et al.*, 2019] Nilushan Costa, Malith Jayasinghe, Ajantha Atukorale, Supun Abeysinghe, Srinath Perera, and Isuru Perera. Adapt-t: An adaptive algorithm for auto-tuning worker thread pool size in application servers. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6. IEEE, 2019.
- [Eldeeb and Alves, 2025] Eslam Eldeeb and Hirley Alves. Offline and distributional reinforcement learning for wireless communications. *arXiv preprint arXiv:2504.03804*, 2025.
- [Feriani and Hossain, 2021] Amal Feriani and Ekram Hossain. Single and multi-agent deep reinforcement learning for ai-enabled wireless networks: A tutorial. *IEEE Communications Surveys & Tutorials*, 23(2):1226–1252, 2021.
- [FlightAware, 2025] FlightAware. Flightaware - flight tracker / flight status. <https://uk.flightaware.com/>, 2025. Accessed: 2025-10-25.
- [Hu *et al.*, 2024] Kai Hu, Mingyang Li, Zhiqiang Song, Keer Xu, Qingfeng Xia, Ning Sun, Peng Zhou, and Min Xia. A review of research on reinforcement learning algorithms for multi-agents. *Neurocomputing*, 599:128068, 2024.
- [Jancauskas *et al.*, 2019] Vytautas Jancauskas, Tomasz Pirotek, Piotr Kopta, and Bartosz Bosak. Predicting queue wait time probabilities for multi-scale computing. *Philosophical Transactions of the Royal Society A*, 377(2142):20180151, 2019.
- [Li *et al.*, 2025] Cuixia Li, Junhai Wang, Jiahao Shi, Liqiang Liu, and Shuyan Zhang. Adwtune: an adaptive dynamic workload tuning system with deep reinforcement learning. *Complex & Intelligent Systems*, 11(4):192, 2025.
- [Lin *et al.*, 2025] Haoran Lin, Xianzhi Yu, Kang Zhao, Han Bao, Zongyuan Zhan, Ting Hu, Wulong Liu, Zekun Yin, Xin Li, and Weiguo Liu. Hap: Hybrid adaptive parallelism for efficient mixture-of-experts inference. *arXiv preprint arXiv:2508.19373*, 2025.
- [Luan *et al.*, 2022] Guangqiang Luan, Pu Pang, Quan Chen, Shuai Xue, Zhuo Song, and Minyi Guo. Online thread auto-tuning for performance improvement and resource saving. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3746–3759, 2022.
- [Luo *et al.*, 2025] Xiaoxuan Luo, Weiwei Lin, Jiachun Li, Fan Chen, Haocheng Zhong, and Keqin Li. An extensible thread throttling method for multiple openmp parallel programs. *ACM Trans. Embed. Comput. Syst.*, September 2025.
- [Montesano *et al.*, 2024] Federica Montesano, Romolo Marotta, and Francesco Quaglia. Spatial/temporal locality-based load-sharing in speculative discrete event simulation on multi-core machines. *ACM Transactions on Modeling and Computer Simulation*, 35(1):1–31, 2024.
- [Morgado *et al.*, 2025] Ângelo Morgado, Kaoru Ota, Mi-anxiong Dong, and Nuno Pombo. Evaluating end-to-end autonomous driving architectures: a proximal policy optimization approach in simulated environments. *Autonomous Intelligent Systems*, 5(1):14, 2025.
- [Pusukuri *et al.*, 2011] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125. IEEE, 2011.

- [Riboldi *et al.*, 2025] Carlo ED Riboldi, Marco Tomasoni, et al. Formation flight of fixed-wing uavs: Dynamic modeling, guidance design, and testing in realistic scenarios. *Aerospace*, 12(3):1–49, 2025.
- [Risco-Martín *et al.*, 2022] José L Risco-Martín, Kevin Henares, Saurabh Mittal, Luis F Almendras, and Katzalin Olcoz. A unified cloud-enabled discrete event parallel and distributed simulation architecture. *Simulation Modelling Practice and Theory*, 118:102539, 2022.
- [Samadi *et al.*, 2024] Mohammad Samadi, Sara Royuela, Luis Miguel Pinho, Tiago Carvalho, and Eduardo Quiñones. Time-predictable task-to-thread mapping in multi-core processors. *Journal of Systems Architecture*, 148:103068, 2024.
- [Stetsenko and Dyfuchyna, 2019] Inna V Stetsenko and Oleksandra Dyfuchyna. Thread pool parameters tuning using simulation. In *International Conference on Computer Science, Engineering and Education Applications*, pages 78–89. Springer, 2019.
- [Wu *et al.*, 2022] Zhibin Wu, Qinyue Zhou, Yucheng Dong, Jiuping Xu, Abdulrahman H Altalhi, and Francisco Herrera. Mixed opinion dynamics based on degroot model and hegselmann–krause model in social networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 53(1):296–308, 2022.
- [Xu *et al.*, 2012] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. Url: A unified reinforcement learning approach for autonomic cloud management. *Journal of Parallel and Distributed Computing*, 72(2):95–105, 2012.
- [Zhang *et al.*, 2021] Chi Zhang, Sanmukh Rao Kuppannagari, and Viktor K Prasanna. Parallel actors and learners: A framework for generating scalable rl implementations. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 1–10. IEEE, 2021.
- [Zhu *et al.*, 2017] Feng Zhu, Yiping Yao, Wenjie Tang, and Jun Tang. A hierarchical composite framework of parallel discrete event simulation for modelling complex adaptive systems. *Simulation Modelling Practice and Theory*, 77:141–156, 2017.