Deploying an ML Agent for OS: A Comparison between User-Space and Kernel-Space

Zhuoyi Huang, Long Peng*, Zhuo Li, Xiaodong Liu, and Jie Yu
College of Computer Science and Technology, National University of Defense Technology
Changsha, Hunan, China

***Abstract*—Machine learning (ML) has shown great potential for improving operating system (OS) performance, yet it is currently deployed primarily in user space. This leaves the performance, overhead, and effectiveness of kernel-space ML largely unquantified. To address this gap, we present a controlled empirical study that compares user-space and kernel-space deployment of the same ML agent. Our approach utilizes a dual-mode ML framework extended to support reinforcement learning (RL), allowing an identical Deep Q-Network (DQN) model to run in both spaces, thereby isolating deployment location as the only variable. We evaluate this using proactive page reclaim, a kernel-intensive task requiring deep access to system state. Experiments show that our kernel-space DQN agent reduces page refaults by 15.9% compared to the conventional LRU policy. More critically, it reduces total decision latency by 98.9% and achieves a 75.8% reduction in context switches, a 62.4× higher inference throughput and a 16.7× reduction in runtime memory overhead compared to user-space deployment. These results provide a quantitative foundation for integrating ML algorithms into OS, confirming that kernel-space integration is essential for realizing low-latency, high-throughput, and efficient learning-driven OS optimizations.**

***Index Terms*—Efficient and Tiny Neural Networks, Neural Network Applications, Reinforcement Learning.**

I. Introduction

The integration of ML into OS, presents a significant opportunity for enhancing system adaptability and efficiency. A key unanswered question in this area is where the ML agent should be placed: in user space or inside the kernel? This decision directly affects the performance, practicality, and long-term sustainability of using ML to optimize an OS.

This design choice involves a core trade-off. Placing the agent in user space allows the use of mature, full-featured ML frameworks and can leverage specialized hardware accelerators (e.g., GPUs/TPUs), making development easier and faster. However, because the agent runs outside the OS kernel, every decision requires communication across the kernel–user boundary, which introduces delays due to repeated context switches and data copying. For tasks that require real-time control, this added latency can significantly reduce the benefits of a learned policy. On the other hand, running the agent inside the kernel avoids these cross-boundary delays entirely, enabling truly low-latency decisions. But the kernel is a constrained environment: it has limited resources and lacks built-in support for ML libraries (e.g., PyTorch[17], CUDA[16]) or easy access to accelerators. As a result, prior work has often been limited to simplified, lightweight models, constraining their expressive power and achievable performance. This leaves a key gap: we still lack a clear, measurement-based comparison of the two deployment strategies.

To address this research gap, we conduct a controlled empirical study to systematically compare user-space and kernel-space deployment for a representative OS task. Our methodology follows three logical steps: (1) Controlled Comparison Foundation: We use the KML framework [2], whose dual-mode architecture allows the same ML model to run in either space, isolating deployment location as the only experimental variable. (2) Model Selection and Framework Extension: To examine the trade-offs with a modern ML agent, we implement a DQN [13, 14]. This required extending KML to support building, training, and deploying DQN models in its dual-mode paradigm, a core technical contribution of this work. (3) Use Case and Objective: We select proactive page reclaim as the target task because it requires deep kernel access and low-latency decisions, making it an ideal benchmark for evaluating deployment overheads and effectiveness. The goal of this case study is to demonstrate that our DQN agent can improve memory reclaim efficiency while quantifying the overheads (e.g., latency and context-switch costs) between deployment modes, thereby providing evidence-based guidance for choosing where to place ML agents in an OS.

This paper makes the following contributions:

- **A Dual-Mode DQN Framework via KML Extension:** We extend the KML framework to support DQN agents on both sides of the user-kernel boundary. This enables the construction and deployment of **identical** DQN models in either space, establishing a fair and controlled basis for our comparative study.
- **DQN-Driven Improvement in Memory Reclaim:** We implement a DQN agent for proactive page reclaim, demonstrating that learned policies can enhance a core OS function. Our agent reduces page refaults by **15.9%** compared to the traditional LRU policy, validating the practical efficacy of ML for low-level system optimization.
- **A Comprehensive Quantitative Basis for Kernel-Space ML Deployment:** Through a controlled empirical study, we provide the first comprehensive performance and overhead comparison. Our results show that kernel-space

* Corresponding author: penglong@nudt.edu.cn

deployment achieves a **98.9% reduction in decision latency** and a **62.4× increase in throughput**, while also reducing context switches by **75.8%** and runtime memory overhead by **16.7×**, conclusively demonstrating the decisive advantage of kernel-space integration for latency-sensitive OS optimizations.

The rest of this paper is structured as follows. Section III describes our extension of the KML framework to support DQN models, which is a core technical contribution of this work. Section IV details the design and implementation of our DQN agent for proactive page reclaim. Section V presents the experimental methodology and comparative results, focusing on the overhead and efficacy trade-offs between user-space and kernel-space deployment. Finally, Section VI discusses the broader implications of our findings, derives practical guidelines, and concludes the paper.

II. Background

Applying ML to OS optimization has become a key approach to enhancing the adaptability and performance. Currently, research and practice in this field primarily follow two technical paths, which differ significantly in deployment location and flexibility.

A. User-Space Deployment Path

In this deployment path, the model operates as a privileged user-space process, collecting system state and applying decisions through system call interfaces such as `ioctl` and `sysfs`. The primary advantage of this approach is its strong development flexibility, enabling researchers to leverage mature ML ecosystems including PyTorch and TensorFlow [1] for rapid iteration and complex model training. This approach has been successfully applied to optimize core OS subsystems, such as using LSTM networks to predict object lifetimes for reducing memory fragmentation in allocators like LLAMA [11]. However, its core bottleneck lies in decision latency and overhead: frequent kernel-user space context switches, data copying, and polling mechanisms prevent this approach from meeting the real-time demands of latency-sensitive kernel subsystems such as memory reclaim and scheduling.

B. Kernel-Space Deployment Path

To achieve minimal latency, another approach embeds ML models directly inside the OS kernel, avoiding the overhead of user-kernel communication. Recent systems demonstrate this: LinnOS [7] places a small, quantized neural network in the Linux storage layer to predict I/O latency, enabling microsecond-scale inference. Another work [3] integrates a multi-layer perceptron into the scheduler to improve task migration decisions with high accuracy. A key constraint is the need to avoid costly floating-point operations in the kernel, leading to solutions like quantization. More fundamentally, these models are not built with standard frameworks but are manually implemented in C for speed, limiting their complexity. The custom nature of each implementation and the absence of a comparable user-space counterpart make a fair, controlled performance comparison across deployment modes extremely difficult. Such a comparison enables a principled evaluation of how deployment mode affects the performance overhead of ML models while preserving their effectiveness in system optimization. To enable it, we design a new, OS-specific ML application. This approach allows us to isolate the performance impact attributable solely to the deployment mode.

C. KML Framework

To enable ML within resource-constrained environments such as an OS kernel, the KML framework was introduced. Unlike general-purpose ML frameworks that rely on extensive third-party libraries, KML is a lightweight library designed from the ground up for the kernel and implemented in C. Its core design principles are minimal overhead and a small memory footprint.

A key feature of KML is its support for flexible training and deployment across both user and kernel spaces. It enables user-space model training, leveraging richer tools for development and tuning. Crucially, thanks to a unified API, models trained in user space can be deployed for execution in either user space or kernel space without modification. This dual deployment capability is foundational to our work, as it allows for a direct comparison of performance overheads between the two execution environments using the same model.

As illustrated in Figure 1, KML provides a foundational set of components for neural network operations within the kernel. Its native library includes core mathematical operations for linear algebra, such as matrix multiplication, basic neural network layers, such as linear and sigmoid layers, and essential loss functions, such as cross-entropy. These components, implemented from scratch in C for efficiency, support both forward and backward propagation. To manage floating-point computations in the kernel, where the FPU is often disabled, KML temporarily enables it within small, carefully bounded code sections, leaving further optimizations like quantization to the developer.

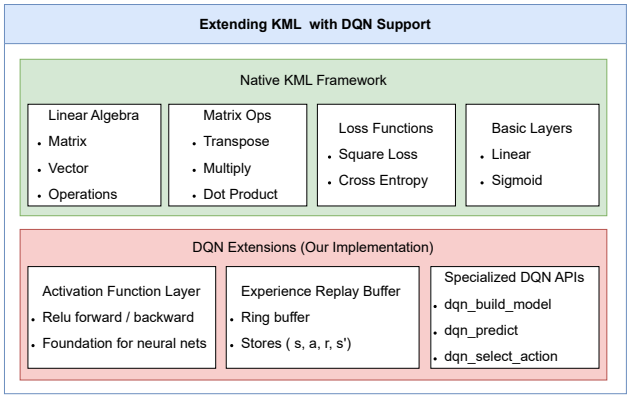

Fig. 1. Extended DQN core components for the KML framework.

TABLE I
DQN-SPECIFIC APIS FOR STORAGE OPTIMIZATION IN THE KML FRAMEWORK.

```
dqnevict_net *build_dqnevict_net(dqnevict_config *config);
void store_experience(replay_buffer *buffer, matrix *state, int action,
                      float reward, matrix *next_state,int done);
matrix *dqnevict_net_inference(matrix *input, dqnevict_net *dqnevict);
matrix *predict_dqnevict(dqnevict_net *dqnevict);
```

However, KML's native component set lacks built-in support for RL. To enable in-kernel deep RL, we extended KML with new modules for DQN. Figure 1 presents these key extensions alongside the native KML components. The design and implementation of our DQN modules will be detailed in Section III.

III. A DUAL-MODE IMPLEMENTATION OF DQN AGENTS IN KML

Our work builds upon the KML framework to implement a DQN agent within the Linux kernel, extending KML's dual-mode deployment to RL. Accordingly, the implementation proceeds in three phases: we first extend KML with DQN-specific components, then develop a lightweight DQN model optimized for kernel constraints, and finally establish a dual-mode framework for training and inference across user and kernel spaces.

A. Extending KML with DQN Components

The native KML framework lacks built-in support for RL. To bridge this gap, we implemented three core components that enable DQN construction, training, and inference, as shown in Figure 1. First, we introduced an Activation Function Layer that implements forward propagation for the ReLU function, thereby extending KML's suite of layers with a fundamental nonlinear operator required for deep neural networks. Second, we developed an Experience Replay Buffer for storing and sampling (state, action, reward, next state) tuples. Its design aims to maintain low overhead during offline training. Third, we extended KML with a set of DQN APIs that define key functions for model management, inference, and decision-making. Table I summarizes these new interfaces, which include building the DQN network (build_dqnevict_net), storing experience tuples (store_experience), performing lightweight inference (dqnevict_net_inference), and obtaining the final Q-value predictions (predict_dqnevict). These four APIs were selected to represent the minimal, complete workflow for DQN-based optimization, from model construction and data collection to inference and decision-making. Together, these extensions allow KML to host and execute DQN policies, eliminating external dependencies while preserving the framework's low-overhead design.

B. Lightweight DQN Implementation

To enable a fair and controlled performance comparison between user-space and kernel-space deployments, we designed a lightweight DQN model with a self-contained codebase. The network employs a compact fully-connected architecture and normalizes input features to the range 0 to 1, which jointly keeps computational overhead low and confines model parameters to a limited numerical interval. This design streamlines floating-point operations during inference and ensures that the same computational burden is imposed in both execution environments. Crucially, the model is implemented entirely in C without external dependencies, guaranteeing that the identical model can be deployed and executed in both user and kernel spaces. This approach isolates the performance impact of the deployment mode itself, providing a clean basis for comparison.

C. The Dual-Mode Deployment

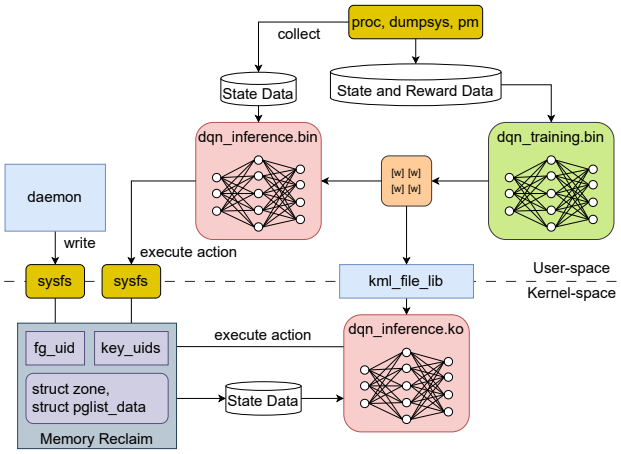

Fig. 2. The Dual-mode Training and Inference Architecture.

We utilize KML's dual-mode design to maintain maximum consistency in the model and code library across both deployment environments, as illustrated in Figure 2. Training is performed in user space using the KML user-space library. It provides a flexible development environment while shares the same API as the in-kernel library. The trained model weights are converted and stored in a compact binary file. For actual use, this same binary file can be loaded in two ways: directly into our lightweight DQN kernel module to run inside the kernel, or into a user-space program for comparison. This allows the exact same trained model to run in both environments, enabling a direct performance comparison. While this

architecture ensures a fair and controlled comparison between deployment modes, the method of interacting with the OS inherently differs. In user space, system state is collected via tools such as `proc`, `dumpsys`, and `ps`, and actions are executed through `sysfs`. In the kernel, state is read directly from internal data structures (such as zone, page list) and the `fg_uid` variable, while decisions are applied by modifying the `key_uids` variable, which governs page-reclaim priority.

IV. Use Case: Page Refault

A. Problem Definition

Page refaults are a critical performance bottleneck in system memory management, especially on resource-limited mobile devices. A page refault occurs when an application requests a memory page that was recently evicted, forcing a slow reload from flash storage instead of a fast access from RAM. This inefficiency stems from a mismatch between generic memory reclaim policies and actual application behavior. Consequently, refaults severely degrade user experience by introducing microsecond-scale delays [9] that are hundreds of times slower than memory accesses, which results in noticeable lag during application launches or switching. Frequent refaults further indicate memory thrashing, wasting I/O and CPU resources. Therefore, minimizing refaults is essential for ensuring smooth system performance.

The traditional LRU-based reclaim scheme evicts pages from different applications with uniform priority [10]. A promising direction is to reduce refaults by identifying pages likely to be re-accessed and lowering their reclaim priority accordingly. This predictive task is where our DQN model offers a novel solution. To manage the scale of innumerable memory pages, we group pages by their owning application and adjust priority at the application level. In mobile systems, each application is uniquely identified by a UID, which serves as our natural grouping key.

Following this formulation, we design a DQN agent that learns to dynamically adjust page-reclaim priorities across applications, with the direct objective of minimizing refaults. Unlike prior heuristic or static approaches [8, 9], we formulate the problem as a RL task, which enables the capture of complex patterns in memory access behavior.

B. State, Action, and Reward Design

The design of a DQN model centers on three core components: state, action, and reward. As illustrated in Figure 3, our state representation is a 6-dimensional vector comprising: the system's page swap usage, available memory, the UID of the foreground application, and the UIDs of the three most recently used applications. The swap usage and available memory metrics indicate memory pressure and are easily obtained via the `/proc` filesystem. The selected UIDs capture the user's recent application interaction pattern, which is important for predicting which applications are likely to incur refaults. To dynamically collect these UIDs, we implement a user-space daemon (see Figure 2). This daemon identifies the foreground application's UID and writes it to a user-defined kernel variable `fg_uid`. When application switches occur, the previous foreground application's UID is logged into the list of recently used applications.

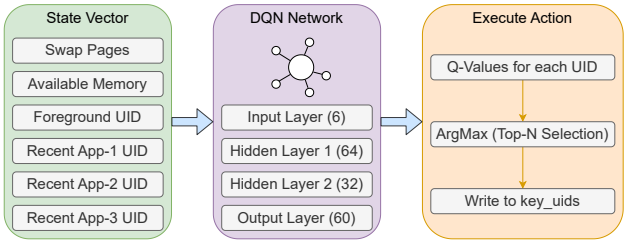

Fig. 3. DQN framework architecture.

Our Q-network outputs Q-values for the 60 user applications installed on the device. The agent selects the top-N UIDs based on these Q-values and writes the selected list into a dedicated kernel variable `key_uids`. This variable is a custom extension we introduced into the kernel memory-reclaim logic. Pages belonging to UIDs listed in `key_uids` are assigned a lower reclaim priority, which makes them less likely to be evicted. Our modifications to the kernel are intentionally lightweight, limited primarily to adding this variable and the corresponding priority check.

$$Reward = 2 \cdot \text{sigmoid}\left(k \cdot (\theta - \text{refault\_count})\right) - 1 \quad (1)$$

The reward function is designed to directly minimize the refault count, as defined in Equation 1. The refault count is obtained from the `/proc` filesystem. If the refault count surpasses a predefined threshold θ, the reward becomes negative, otherwise it is positive. This formulation encourages the agent to seek states minimizing the increase in refault count and to learn policies that reduce them consistently. Furthermore, the reward is bounded between -1 and 1 by using the sigmoid function to stabilize the training process, and the parameter k controls the smoothness of the reward function.

C. Network Architecture and Training

Our Q-network is a fully connected neural network. It takes the 6-dimensional state vector as input, passes through two hidden layers with 64 and 32 neurons (using ReLU activation), and outputs a vector of 60 Q-values, one for each application. To stabilize training, we employ standard DQN techniques: an experience replay buffer to mitigate sequential correlation and a target network to provide consistent learning targets. The model is trained offline on traces collected from real device operation under dynamic workload scenarios, ensuring it converges to a robust policy before being deployed within our dual-mode framework for online inference.

V. Evaluation

This evaluation section is designed to achieve two primary objectives: (1) to demonstrate the effectiveness of our DQN-based approach in reducing refault counts compared to existing

memory reclaim schemes, and (2) to quantify the performance overhead differences between user-space and kernel-space deployments of the intelligent agent. Our code is publicly available at https://www.osredm.com/p49781326/DQNevict.

A. Experimental Setup

Environment: All experiments were conducted on a Khadas Edge2 RK3588S development board featuring a Rockchip RK3588S SoC and 16GB of RAM, running Android 12 with a Linux kernel version 5.10.

Workload: We emulate realistic mobile usage patterns by generating workloads based on the LiveLab dataset [20], which contains real application usage histories. Our simulation includes scenarios such as social media browsing, gaming, and heavy multitasking. Applications are launched programmatically according to the recorded history, while our DQN model runs concurrently to collect system state data dynamically. This automated approach preserves authentic user behavior patterns and ensures experimental consistency.

Evaluation metrics: The evaluation focuses on two dimensions. Memory efficiency is measured by the refault count. Performance overhead is quantified by the end-to-end latency per inference iteration, the number of context switches, inference throughput, and memory footprint.

Baselines: For memory efficiency evaluation, we compare our DQN policy against two baselines: the default Linux LRU algorithm and Acclaim, a rule-based heuristic policy. For analyzing deployment overhead, we compare the performance of our user-space DQN agent with its kernel-space counterpart.

Parameters: The key parameters include N, defined as the number of candidate applications selected for priority adjustment in each decision epoch, the refault threshold θ and the smoothness parameter k in sigmoid function. Empirically, we set $N = 5$ to balance effectiveness and risk, where a larger N would demote page priorities and weaken the policy, while a smaller N increases the risk of misidentifying high-refault applications. The value of θ was set to 10,000 based on the empirical observation that a single iteration of our model takes approximately 20 seconds, during which the number of refaults under the LRU policy typically increases by a comparable magnitude. The smoothness parameter k controls the sensitivity of the reward function. A smaller k value (e.g., $k = 0.1$) produces a mild reward curve that encourages exploration during early training stages, while a larger k value (e.g., $k = 0.3$) creates a steeper response that accelerates convergence. In our experiments, k was set to 0.2 as it provided a good balance between exploration efficiency and learning stability.

B. Memory Effciency Improvement

This subsection evaluates the effectiveness of our intelligent reclaim policy using the refault count as the primary indicator for memory management efficiency. Figure 4 compares four strategies: LRU, Acclaim, DQN (user space), and DQN (kernel space), based on data collected at a stable state. We ran the same workload according to Livalab using the four reclaim

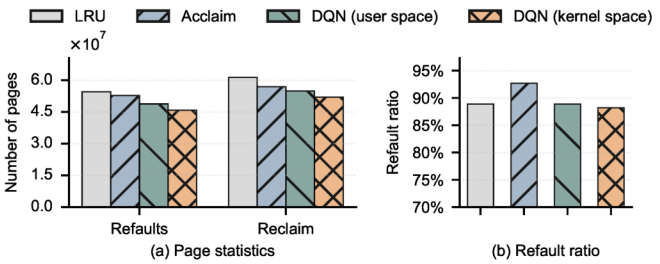

Fig. 4. The statistics of Refault pages and reclaim pages.

schemes, with each test lasting 24 hours. The numbers of refault and reclaim pages were recorded via the `/proc` filesystem. Each test was repeated 5 times, and the average values were calculated for each scheme.

Overall, the kernel-space DQN implementation delivers the best performance. It achieves the lowest refault count and the lowest total number of pages reclaimed, resulting in the best refault ratio of 88.19%. This represents a substantial reduction of approximately 15.9% in refaults compared to the baseline LRU strategy, and 13.2% compared to the Acclaim heuristic. The user-space DQN agent reduces refaults by 10.5% compared to LRU and by 7.6% compared to Acclaim, validating the DQN's learning capability. However, it incurs 6.5% more refaults than the kernel-space version, highlighting the overhead of user-kernel communication. The refault ratio for user-space DQN is similar to LRU but 4.1% better than Acclaim.

The performance of the kernel-space DQN can be attributed to its direct access to system state and immediate action execution, eliminating latency and decision lag that affect the user-space agent. This allows it to make more timely and accurate reclaim decisions.

C. Performance Overhead Analysis

We evaluate the performance overhead of our kernel-space deployment compared with the user-space deployment. The evaluation involves executing 100 consecutive inference iterations in both environments. For each run, we compute averages every 10 iterations. The entire process is repeated 5 times, and the mean values are reported in the following analysis.

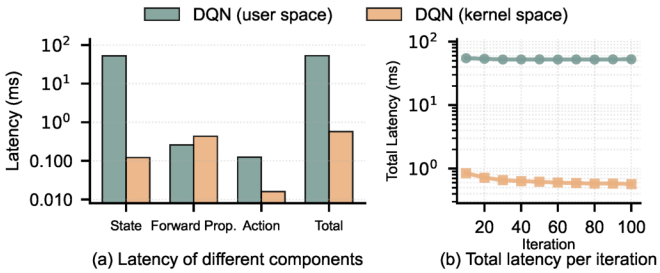

Fig. 5. Inference latency comparison.

1) Inference Latency: Each inference iteration comprises three stages: state acquisition, forward propagation (the core model inference), and action execution. Figure 5 details the latency breakdown for the DQN agent. The kernel-space deployment achieves a transformative reduction in total inference latency despite a minor increase in forward propagation time. The most significant improvement is in the *State Get* operation: its average latency drops from 52.36 ms in user space to 0.12 ms in kernel space, a reduction of 99.77%. This near-elimination of overhead results from direct access to kernel memory structures, bypassing system calls. Similarly, the maximum latency for state acquisition drops by 99.5%. Conversely, the core forward propagation operation is moderately slower in kernel space, taking 0.43 ms on average compared to 0.26 ms in user space (an increase of 66.80%), and its maximum latency increases by 1.88×. This slowdown is likely attributable to the higher cost of floating-point calculations in the kernel context, although the absolute increase remains small. The *Action Execution* latency is also drastically lower in kernel space, with average latency reduced by 87.10% (0.02 ms versus 0.12 ms) and maximum latency reduced by 93.3%. This improvement occurs because the kernel-space agent directly manipulates kernel page lists. Overall, the total average latency per inference iteration is reduced by 98.9% in kernel space, from 52.74 ms to 0.57 ms. The line plot in Figure 5 further shows the evolution of per-iteration total latency across 100 iterations. While the user-space latency remains consistently high (around 52 ms), the kernel-space latency starts near 1 ms and rapidly declines, stabilizing at approximately 0.5 ms after about 40 iterations. This initial overhead and subsequent convergence are likely due to cold-start effects within the kernel, such as initial cache misses and the one-time cost of locking necessary data structures, after which the system reaches a steady, optimal state. Consequently, the total average latency plummets from 52.74 ms to 0.57 ms. This improvement directly translates to the throughput increase shown later, proving that the kernel-space architecture effectively removes the fundamental bottleneck of user-kernel crossing.

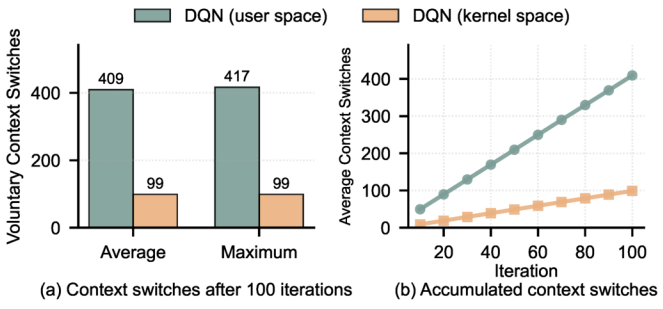

Fig. 6. Context switch count for the user-space and kernel-space DQN.

2) Context Switches: Figure 6 shows that kernel-space operation drastically reduces system overhead. The average total number of context switches over 100 inference iterations drops from 409 to 99, a reduction of 75.8%. The peak total number of switches exhibits a similar decline, from 417 to 99. Furthermore, the line chart indicates that the cumulative number of context switches increases linearly with the number of iterations, and the rate of increase is higher for the user-space agent than for the kernel-space one. These reductions are expected because the kernel-space agent performs state collection, inference, and action execution within a single kernel context, thereby minimizing mode transitions. The sharp decrease in context switching is a primary contributor to the lower latency and higher throughput observed in the kernel-space deployment.

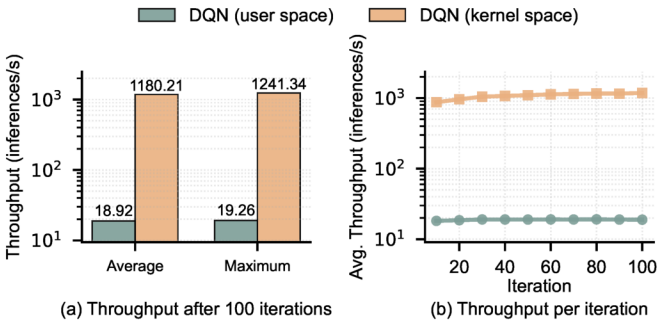

Fig. 7. Inference throughput comparison.

3) Throughput: Figure 7 demonstrates the combined effect of latency reduction and fewer context switches. We measure inference throughput, defined as the number of completed inference iterations per second, calculated by dividing the total inference count by the actual running time in seconds. Inference throughput surges from 18.92 inferences/second in user space to 1180.21 inferences/second in kernel space, an improvement of 62.4×. Peak throughput shows a similar scale of improvement. This demonstrates that kernel-space ML incurs only minimal CPU overhead while delivering substantial performance gains.

4) Memory Overhead: Table II compares memory consumption. The kernel-mode implementation achieves a more efficient *runtime* footprint. Runtime physical memory usage is reduced by a factor of 16.7×, and framework overhead is reduced by 16.2×. This is because the kernel module shares the kernel's address space and avoids the overhead of a separate user-space process and libraries. The trade-off is a larger static file size (413 KB compared to 44 KB), as the kernel module contains more low-level integration code. This trade-off favors long-running systems where runtime efficiency is critical.

VI. Discussion and Comprehensive Analysis

Our experiments reveal a fundamental trade-off, defining two distinct paradigms for deploying ML within an OS. The first paradigm, kernel integration, is optimized for low latency and efficiency. By becoming an integral part of the OS, the agent eliminates the overhead of user-kernel communication.

TABLE II
COMPARISON OF MEMORY USAGE BETWEEN USER-SPACE AND KERNEL-SPACE IMPLEMENTATIONS

Memory Dimension	Memory Consumption		
	User	Kernel	Comparison
Runtime Physical Memory (KB)	2,808	168	16.7×
Framework/Library Overhead (KB)	1,948	120	16.2×
Static File Size (KB)	44	413	11%

This enables near-instantaneous state access and action execution, minimal context switching, and superior runtime memory efficiency. Consequently, this approach is transformative for implementing latency-critical control mechanisms within the kernel. The trade-off includes a larger static footprint, which may be a critical constraint in storage-sensitive embedded environments, and the inherent complexity of kernel programming, which translates to higher development and verification costs. Ultimately, this demonstrates that integrating ML into the kernel is not merely an implementation choice, but a necessary design principle for achieving the low latency and high effectiveness required by core OS control mechanisms.

In contrast, the user-space isolation paradigm prioritizes computational optimality and safety through traditional process separation. Its principal advantage is access to highly optimized numerical libraries, which accelerate the core model inference. This makes it suitable for roles where computational throughput is paramount and sub-millisecond decision latency is not required, such as for policy analysis or as an advisory subsystem. However, its need to interact with the kernel via system calls imposes a severe performance penalty for fine-grained, in-line control tasks.

The choice between these paradigms is therefore an architectural decision aligned with the agent's primary function. Kernel-space deployment is warranted when the agent must implement a direct, low-latency control loop that is deeply integrated with kernel state. User-space deployment is preferable when the agent serves in a computational, analytical, or advisory capacity, where leveraging mature libraries and maintaining safety isolation outweigh the need for minimal latency. This decision can be guided by quantitative thresholds, such as the required decision frequency or maximum tolerable latency, beyond which kernel integration becomes necessary. This work demonstrates that kernel integration is not only feasible but can unlock order-of-magnitude performance gains for control tasks, significantly expanding the design space for ML in OSs. Future work should explore hybrid architectures that strategically split control and computation across the kernel-user boundary. More specifically, to mitigate the identified barriers, research should focus on developing automated toolchains for safe kernel deployment of ML models and on creating lightweight, kernel-friendly inference runtimes to bridge the computational performance gap observed in this study.

VII. RELATED WORK

The Gap in kernel-space ML. Deploying user-space frameworks such as PyTorch, TensorFlow, Keras [4] and scikit-learn [18] directly inside the kernel is infeasible. Solutions like the lightweight KML enable efficient in-kernel inference, while eBPF [6] provides a safe extension mechanism. However, eBPF lacks demonstrated use cases in production OSs, and KML's model support is limited to classic algorithms like decision trees, multi-layer perceptrons, and back-propagation neural networks, excluding more complex, adaptive models such as DQN. To bridge this gap, we extend the KML framework to support DQN-based decision-making and evaluate its performance.

ML for memory management. Research increasingly applies ML to OS and memory management [7]. Specific works use LSTMs to predict object lifetimes [12] or access patterns [15], while others design learning-based pagers [5] or tuners for heterogeneous memory [19]. However, these systems are implemented either in user space or kernel space, and none deploy and evaluate the same model across both environments. To address this, we implement and evaluate one consistent model in both user and kernel space.

VIII. CONCLUSION

This work tackles a fundamental question in ML-enhanced operating systems: where to deploy learning agents for optimal performance. We demonstrate that deep kernel integration is not merely an implementation choice but a necessary design principle for achieving the low latency and high efficacy required in core OS tasks. Through a controlled, direct comparison using a unified DQN agent for page reclaim, we prove that kernel-space integration uniquely enables the real-time decision-making where user-space agents fall short. Consequently, our co-designed architecture establishes a practical and extensible model for applying learned policies to other latency-critical subsystems, such as I/O scheduling. Future work will explore generalizing this co-design principle to a wider suite of OS primitives and will investigate hybrid deployment strategies, thereby providing a more comprehensive framework for evaluating ML integration paradigms across the system stack.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous reviewers for their insightful and detailed comments. This work is supported by the Science and Technology Innovation Program of Hunan Province under Grants 2025RC3121 and 2024RC1048.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283. [Online]. Available: https://arxiv.org/abs/1603.04467

[2] I. U. Akgun, A. S. Aydin, A. Burford, M. McNeill, M. Arkhangelskiy, and E. Zadok, "Improving storage

systems using machine learning," *ACM Trans. Storage*, vol. 19, no. 1, Jan. 2023.

[3] J. Chen, S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "Machine learning for load balancing in the linux kernel," in *APSys '20: Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020.

[4] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015, deep learning library for TensorFlow and other backends.

[5] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A hybrid memory page scheduler with machine intelligence," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 37–48.

[6] B. Gbadamosi, L. Leonardi, T. Pulls, T. Høiland-Jørgensen, S. Ferlin-Reiter, S. Sorce, and A. Brunström, "The ebpf runtime in the linux kernel," 2024. [Online]. Available: https://arxiv.org/abs/2410.00026

[7] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "LinnOS: Predictability on unpredictable flash storage with a light neural network," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 173–190.

[8] C. Li, Y. Liang, R. Ausavarungnirun, Z. Zhu, L. Shi, and C. J. Xue, "Ice: Collaborating memory and process management for user experience on resource-limited mobile devices," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 79–93.

[9] Y. Liang, J. Li, R. Ausavarungnirun, R. Pan, L. Shi, T.-W. Kuo, and C. J. Xue, "Acclaim: adaptive memory reclaim to improve user experience in android systems," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'20. USA: USENIX Association, 2020.

[10] Y. Liang, Q. Li, and C. J. Xue, "Mismatched memory management of android smartphones," in *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'19. USA: USENIX Association, 2019, p. 7.

[11] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based memory allocation for c++ server workloads," in *ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[12] M. Maas, D. G. Andersen, M. Isard, M. M. Javanmard, K. S. McKinley, and C. Raffel, "Learning-based memory allocation for c++ server workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 541–556.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *Computer Science*, 2013.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Petersen, M. Riedmiller, A. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[15] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "Deepcache: A deep learning based framework for content caching," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, ser. NetAI'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 48–53.

[16] NVIDIA Corporation, *CUDA Toolkit Documentation*, 2024, version 12.4. [Online]. Available: https://docs.nvidia.com/cuda/

[17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] S. Phadke and S. Narayanasamy, "Mlp aware heterogeneous memory system," in *2011 Design, Automation Test in Europe*, 2011, pp. 1–6.

[20] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: measuring wireless networks and smartphone users in the field," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 3, p. 15–20, Jan. 2011.