Low-Overhead Scheduling for Real-Time AI Workloads on Multi-Core Edge Chips

Zihe Hao

College of Engineering, Northeastern University, 02115, USA

Abstract: ***When deep neural networks run on heterogeneous edge hardware, they often suffer from contention over shared memory, which frequently violates the strict timing requirements that safety critical systems must guarantee. The mainstream approach in academia to address these scheduling issues currently employs deep reinforcement learning. However, this introduces a new problem: DRL schemes can themselves be relatively heavyweight, and the additional overhead introduced undermines their inherent latency advantages. To alleviate this issue, this study introduces a hybrid framework called Lite Sched to separate computation from real-time operations. Our system performs offline static analysis and constructs resource occupancy maps for neural network layers, enabling a lightweight, heuristically driven agent to make scheduling decisions with negligible processor overhead. Evaluations on the NVIDIA Jetson Orin platform show that Lite Sched reduces tail latency by 35.7% and improves energy efficiency by 21.4%. Compared with the standard greedy method, our work achieves significant improvements. These results demonstrate that the combination of static analysis and dynamic micro-scheduling is a reliable way to balance energy and latency trade offs in modern edge computing for resource constrained autonomous industrial systems.***

Keywords: Edge Computing; Multi-Core Scheduling; Real-Time Systems; Energy Efficiency; Deep Learning.

1. INTRODUCTION

Today, the way we build digital systems is changing. Instead of putting everything in huge, central cloud data centers, we are moving toward "the edge" which means processing data on local devices. This shift is driven by the fact that new technologies such as factory robots and self-driving cars require extremely fast response times. To make this work, we use SoC hardware, which puts the CPU, GPU, and AI processors all on one small chip. While this helps run AI at the edge, it creates a "logjam" because all these parts share the same memory. Highspeed AI tasks often fight with important control tasks for resources, and standard computer systems aren't very good at managing this conflict yet.

Current systems, just like the Linux "Completely Fair Scheduler," try to be fair to everyone and finish as much work as possible. This works fine for normal office computers, but it is not reliable enough for machines that must be precise. For autonomous driving car, missing a deadline to hit the brakes is much more dangerous than a video skipping a single frame. Some researchers try to use Deep Reinforcement Learning to manage these tasks. However, these AI managers are often too heavy. In our tests on small devices like the NVIDIA Jetson, we found that the AI manager takes so much time to think that it actually slows down the very tasks it was supposed to help. You cannot simply take a heavy AI system meant for a giant server and put it on a small edge device.

Our research finds that a middle ground between simple rules and heavy AI. We believe that even though computer tasks can be unpredictable, we do not need a complex brain to guess every move in real time. Insteadly, we use a hybrid approach called "Lite Sched." We test the AI tasks ahead of time to see how they behave and create a lookup table. When the device is running, it doesn't need to perform complex mathematical calculations; it can make quick decisions simply by consulting a lookup table. It's both fast and more intelligent, like an AI model. This work is important for more than just speed. As countries modernize their digital systems, being able to run smart AI on small, low-power devices is the key to better industry. Instead of only caring about how much work the computer does, we focus on saving energy while staying the Energy Delay Product. This approach is particularly important in the context of modern AI applications, where efficient power management is critical to ensuring operational sustainability. In the following sections, we will explain how Lite Sched is built and show our test results. These tests will show more clearly than ever how our system handles the unique challenges of edge computing.

2. RELATED WORK

Research into how computers coordinate their resources has generally split into two different worlds: one group

focuses on making the physical hardware chips run faster, while the other focuses on how to send tasks across a network. Previous work on system coordination highlights the balance between performance and energy efficiency, particularly in distributed edge systems [8]. Therefore, we need to understand how existing methods address the time-sensitive challenges faced by machines such as robots or self-driving cars. We can divide current research into three main areas: specific hardware techniques for artificial intelligence, sending workloads to nearby servers, and leveraging AI to manage tasks.

2.1 Architectural Optimization and the Memory Wall

The biggest hurdle for running AI on small "edge" devices is the "Memory Wall." Ngo et al. (2025) [2] discuss similar challenges in edge intelligence, where the memory bottleneck significantly affects the performance of AI tasks in resource-limited environments. This challenge has also been examined in prior works, such as those by Yu et al. (2026), which explore how edge architectures manage resource contention in dynamic environments [6]. This is the gap between how fast a processor can think and how slowly the memory can feed it data. Recent studies, such as the analysis by Kaur, suggest moving the math directly into the storage units to save energy. However, most of these hardware improvements assume the AI chip is working alone. This research is innovative, but it relies too heavily on internet connectivity. If signal interruptions occur due to building or network congestion, local devices will suddenly face an overwhelming workload that they cannot handle. Since safety-critical systems cannot afford the risk of "waiting for a signal," relying on the network as a means to ensure timely machine responses is too risky.

2.2 Network Centric Offloading Strategies

At the same time, another strategy called "Mobile Edge Computing" tries to solve the problem by sending heavy tasks to a nearby server instead of doing them locally. For example, Liu showed that a car could save battery life by letting a roadside computer handle some of its data. Further work by Liu examines adaptive systems that handle resource distribution for battery efficiency in real-time applications [7]. This research is innovative, but it relies too heavily on internet connectivity. Yu et al. (2024) [13] address similar concerns with recommendation systems, where network dependency affects the overall system performance, particularly when handling dynamic user experiences. Easwaran et al. (2024) also emphasize the importance of balancing resource distribution between local devices and the cloud, particularly in environments where latency is critical [1]. Several studies have shown the importance of addressing network dependency in edge computing environments, where latency and bandwidth constraints significantly impact system performance, such as those found in Liu et al. (2025), who explore budget allocation models in uncertain environments [13]. Several studies, such as those by Liu, have explored how resource constraints on edge devices impact the effectiveness of network-based solutions [5]. If signal interruptions occur due to building or network congestion, local devices will suddenly face an overwhelming workload that they cannot handle. This network dependency issue has been extensively discussed in recent studies, such as those by Kohli, which focus on agent-driven solutions for content issues in network-heavy environments [14]. Since safety-critical systems cannot afford the risk of "waiting for a signal," relying on the network as a means to ensure timely machine responses is too risky.

The Paradox of Learning Based Scheduling

Because simple rules are often too rigid and the network is unreliable, some researchers have turned to Deep Reinforcement Learning to act as a "smart manager" for tasks. In 2026, Hu and colleagues showed that this kind of AI could learn how to balance workloads in complex power grid simulations. However, when we try to put this "smart manager" on small, low-power chips like the NVIDIA Jetson, we run into a "paradox": the AI manager is so heavy that it takes longer to "decide" how to schedule a task than it takes to actually run the task itself. It’s like a manager spending an hour in a meeting to plan a five minute job. While these AI managers work great for giant servers, they are often too slow for the split-second timing needed at the edge.

2.3 Synthesis and Research Gap

When we look at the big picture, there is a clear gap in our current technology. We have fast hardware, clever network tricks, and smart heavy AI managers, but we don't have a system that is both smart and lightweight. We need a way to get the cleverness of an AI manager without the heavy "thinking time" that slows everything down. Huang et al. (2025) [3] highlight the balance between efficiency and fairness in AI accelerator resource sharing, which is essential for low-latency scheduling in edge computing. Recent advancements in lightweight AI

management, as discussed by Lin, present strategies to reduce computational overhead while retaining decision-making efficacy [8]. This need for efficient, yet intelligent management systems has been addressed in related fields like predictive maintenance, where low-overhead solutions are emphasized. This is exactly where Lite Sched fits in. By moving the heavy thinking to an "offline" phase, we can make sure the device stays fast and responsive while it's running. We are essentially finding a way to resolve the conflict between complex brainpower and high-speed execution.

3. SYSTEM MODEL AND PROBLEM FORMULATION

To move from just understanding how computer parts "fight" for resources to actually proving that our Lite Sched system works, we need to turn these ideas into math. It isn't enough to simply say that "traffic jams" exist on the chip; we have to measure exactly how much these different parts slow each other down. In this chapter, we take the physical reality of these small computer chips and turn them into a set of mathematical rules. We start by picturing the work the computer does as a "task map" a flowchart that shows what needs to happen and how likely it is. Then, instead of seeing the chip as just a collection of separate parts, we view it as a shared space where everything is connected by a single memory "highway" that everyone is trying to use at once. Finally, we turn the whole scheduling challenge into a math puzzle: the goal is to find the best balance between saving battery life and staying fast, all while making sure We never miss a strict time limit for the most Important Safety tasks. The challenge of balancing energy efficiency with stringent timing requirements has been a focus of recent studies in AI scheduling for embedded systems [19].

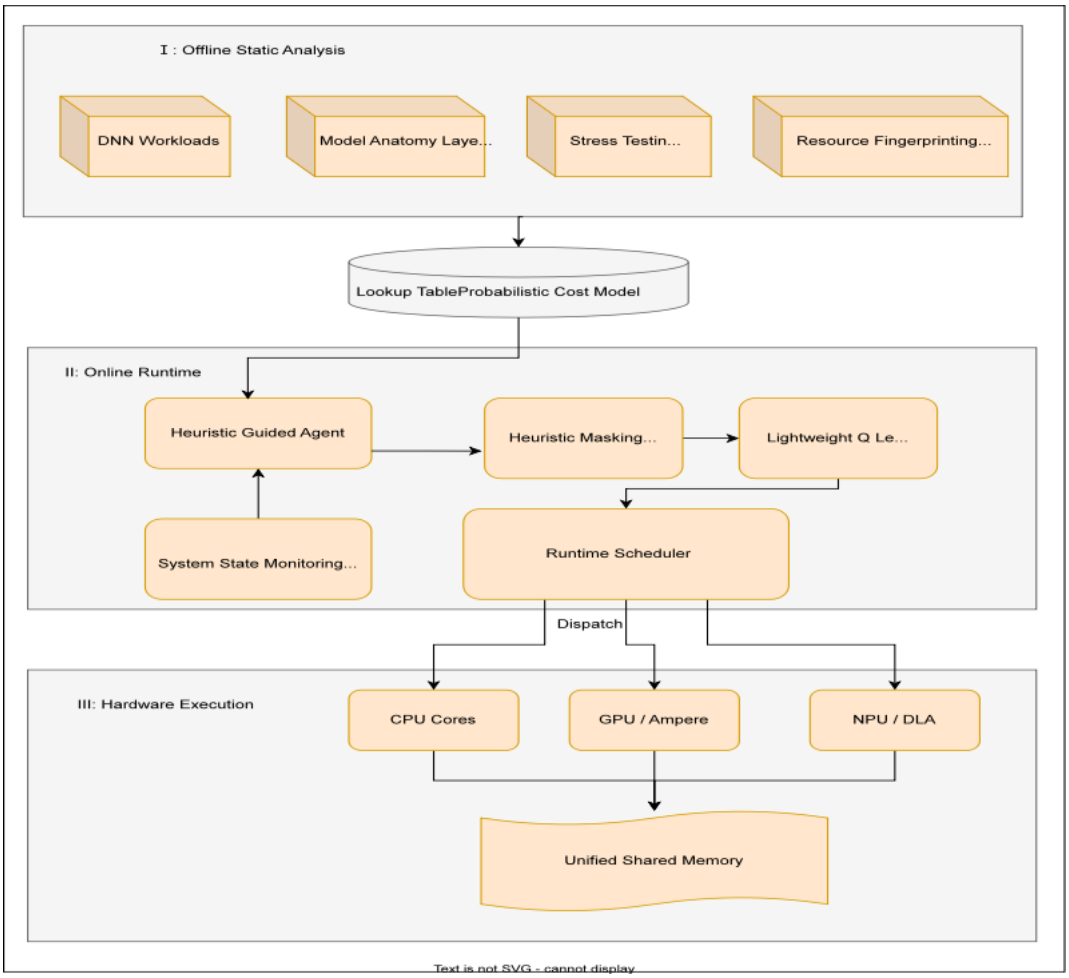

Figure 1: The Framework of Lite Sched

3.1 Probabilistic Application Model

We model the heterogeneous AI workload as a Directed Acyclic Graph, denoted as $G = (V, E)$. In this graph-theoretic representation, the set of vertices $V = \{v_1, v_2, \dots, v_n\}$ corresponds to the discrete computational kernels

within a deep neural network, such as convolutional layers, max-pooling operations, or activation functions. The set of directed edges $E \subseteq V \times V$ represents the mandatory data dependencies, where an edge $e_{ij}=(v_i, v_j)$ implies that the execution of task v_i and the subsequent materialization of its output tensor in the system memory. Unlike classical real-time task models which often assume a fixed Worst-Case Execution Time, the execution profile of deep learning layers on edge devices exhibits significant stochasticity due to thermal throttling and sparse activation patterns. To capture this uncertainty, we define the computational cost of a task v_i not as a scalar, but as a vector C_i dependent on the assigned processing element. Let $P=\{p, p_2, \dots, p_m\}$ represent the set of available processing units. The nominal execution time of task v_i on processor p_k is given by $t_{i,k}^{nom}$ However, this nominal value represents an idealized state of isolation.

Furthermore, the edges in our graph are weighted by the volume of data transmission required. We denote D_{ij} as the size of the tensor transferred between v_i and v_j. The communication latency L_{ij} is thus a function of the current available memory bandwidth $B(t)$, which leads to the formulation. This dynamic interaction between processing and memory bandwidth has been addressed in similar research, such as that by Wang et al. (2024), which models communication latencies in complex embedded systems [10].

$$L_{ij}(t) = \frac{D_{ij}}{B(t)} + \delta_{setup}$$

Here, δ_{setup} represents the constant overhead associated with Direct Memory Access controller initialization. It is crucial to acknowledge that $B(t)$ is not a static constant but a dynamic variable that fluctuates based on system-wide contention, a factor that introduces non-linearity into the scheduling model.

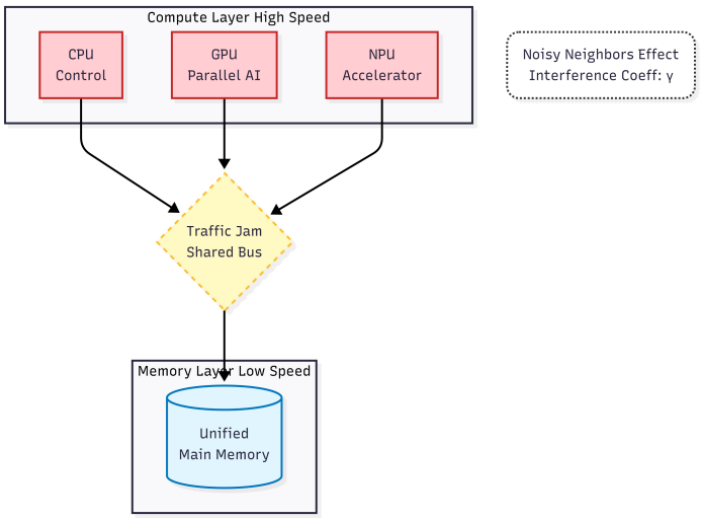

Figure 2: Illustration of Heterogeneous Soc Memory Contention

3.2 Hardware Architecture and the Interference Domain

The hardware substrate is modeled as a heterogeneous System on Chip S comprising a set of processing elements P sharing a unified main memory. A defining characteristic of this architecture is the lack of physical isolation between the memory requests of the Central Processing Unit and the neural accelerators. This necessitates the introduction of an Interference Penalty Function to correct the nominal execution times derived in the previous section. Abdi & Salimi-Badr [4] propose similar methods to address interference issues in multi-core processors, particularly in systems requiring real-time task execution.

We posit that the actual execution time $t_{i,k}^{act}$ of a task is the product of its nominal time and a degradation factor induced by concurrent memory access. Let $\Phi(t)$ be the set of all tasks executing on the system at time t excluding task v_i. We define the memory pressure exerted by these concurrent tasks as $\rho(t)$. The interference coefficient $\gamma_{i,k}$ represents the sensitivity of task v_i to memory bandwidth saturation when running on core p_k. The actual execution time is thus modeled as:

$$t_{i,k}^{act} = t_{i,k}^{nom} \times \left(1 + \gamma_{i,k} \cdot \sum_{v_j \in \Phi(t)} \frac{BW_{req}(v_j)}{BW_{max}}\right)$$

In this equation, $BW_{req}(v_j)$ denotes the memory bandwidth requirement of a concurrent task v_j, and BW_{max} is the theoretical peak bandwidth of the DRAM interface. This formulation explicitly captures the "noisy neighbor" effect: as the aggregate bandwidth demand approaches BW_{max}, the execution time $t_{i,k}^{act}$ expands. Deriving accurate values for $\gamma_{i,k}$ requires the extensive offline profiling detailed in the subsequent methodology chapter, as these sensitivity coefficients are specific to both the kernel type and the microarchitecture.

Energy Consumption Model

Energy efficiency is a paramount constraint for battery-operated edge devices. We model the total energy consumption E_{total} as the summation of dynamic power and static leakage power over the scheduling horizon [0, T] The dynamic power consumption of a processing element p_k is approximated by the classic capacitance equation where V is the voltage and f is the frequency. However, for the purpose of scheduling optimization, we focus on the energy cost associated with specific task assignments.

Let $x_{i,k}$ be a binary decision variable where $x_{i,k}$=1 if task v_i is assigned to processor p_k, and 0 otherwise. The total energy consumption for the workload graph is formulated as:

$$E_{total} = \sum_{v_i \in V} \sum_{p_k \in P} x_{i,k} \cdot \left(P_{active,k} \cdot t_{i,k}^{act}\right) + \int_0^T P_{static}(t)\,dt$$

This math formula highlights that power isn't only used when the computer is actively "thinking." Energy is also wasted during quiet moments through what we call "static leakage." This happens a lot with high-performance cores: they stay powered up and "awake," but they can't actually do any work because they are stuck waiting for another task to finish first (this is called a "dependency"). It's like leaving a car engine running while sitting at a red light—you aren't moving anywhere, but you're still burning fuel.

3.3 Optimization Problem Formulation

Considering the models established above, the scheduling problem is framed as identifying a mapping function $M:V \rightarrow P \times R^+$ that assigns each task to a processor and a start time. The primary objective is to minimize the Energy-Delay Product, which serves as a composite metric balancing performance and efficiency. We formally define the optimization problem as follows:

$$\text{Minimize } Z = \left(\sum_{v_i \in V} t_{i,k}^{act}\right) \times E_{total}$$

Subject to the following constraints:

Deadline Constraint: For every graph G representing a safety-critical application with a deadline D_{hard}, the completion time of the sink node v_{sink} must satisfy:

$$\text{Finish}(v_{sink}) \leq D_{hard}$$

Resource Capacity Constraint: At any given time instance t, the resource usage of assigned tasks cannot exceed the available processing elements:

$$\sum_{v_i \in V} x_{i,k}(t) \leq 1, \forall p_k \in P$$

Note: If you are using a GPU that is capable of running multiple tasks at the very same time, this rule becomes more flexible. In this case, the limit is simply based on how many "mini engines" technically called Streaming Multiprocessors (SMs), the GPU has available to do the work.

Precedence Constraint: A task v_j cannot commence execution until all predecessor tasks $v_i \in \text{Pred}(v_j)$ have completed and data transfer L_{ij} is finalized:

$$\text{Start}(v_j) \geq \max_{v_i \in \text{Pred}(v_j)} \{\text{Finish}(v_i) + L_{ij}\}$$

Finding the absolute best answer to this math problem is what experts call "NP-hard." Dehghani et al. (2023) [15] discuss the complexity of real-time task scheduling and propose energy-efficient dynamic task mapping solutions for multi-core systems. This is a fancy way of saying the problem is so complex that even the fastest computers cannot find the perfect solution in a reasonable amount of time. It is like trying to solve a massive puzzle where certain pieces must be placed before other and the people helping you all work at different speeds. As you add

more tasks, the number of possible ways to arrange them grows at an "exponential" rate—meaning the possibilities explode so fast that checking every single one is impossible for a system that needs to make decisions in a split second. This mathematical difficulty is exactly why we created the Lite-Sched framework. Instead of searching forever for a perfect answer, our system uses a "smart shortcut" (a combination of rules and AI) to quickly find a "good enough" answer, which we call the approximate solution M^*. This allows the system to follow all the safety rules while keeping our total cost, the Z value, as low as possible.

4. METHODOLOGY: THE LITE SCHED FRAMEWORK

The core philosophy of the Lite-Sched framework is that the brainpower needed to make the best scheduling decisions must never fight for resources with the actual tasks being scheduled. This principle of separating concerns means we have to move the heavy duty intelligence into an offline or asynchronous phase. Specifically, we take the process of learning complex interference patterns and resource needs and handle them before the system ever starts running. This shift allows the runtime environment to stay lean and focus entirely on executing a quick and reflex driven strategy. Zhu et al. (2019) [18] applied a similar multi-graph convolutional network to predict dynamic systems, emphasizing the need for efficient, multi-step decision-making in complex systems like autonomous vehicles.

In this section, we break down our approach into three key parts. These include the static profiling of neural network layers, the mathematical derivation of our smart runtime agent, and the actual build of the memory aware pipeline. We should point out that developing this framework was not a simple straight shot but rather a cycle of trial and error. Our first attempt at using a fully online Deep Q Network led to system delays that were simply unacceptable. That specific roadblock is what pushed us toward the hybrid solution we are presenting here.

4.1 Phase I: Static Resource Fingerprinting

To construct a schedule that is resilient to contention, the system must first possess a granular understanding of the resource appetite of each task. We define this process as Model Anatomy. Unlike conventional profiling which typically aggregates execution time at the application level, our approach disassembles the Deep Neural Network into its constituent layers corresponding to the vertices V defined in our system model. By leveraging a simulation environment akin to model-based design tools, we execute each layer type in isolation to establish a baseline execution latency $t_{i,k}^{nom}$.

However, a significant challenge encountered during early experiments was that isolated execution times proved to be poor predictors of runtime performance. The variance introduced by cache trashing and thermal throttling meant that a layer consuming 2 milliseconds in a cold state could exceed 5 milliseconds under heavy load. To address this, we expanded the static profile into a multidimensional Resource Fingerprint. This data structure captures not merely the arithmetic logic unit demand but also the memory bandwidth consumption $\mathrm{BW}_{req}(v_i)$ and the sensitivity coefficient $\gamma_{i,k}$ previously introduced in Equation [2]. We conducted extensive stress testing by artificially injecting noise into the memory bus which allowed us to map the degradation curve of each layer type. Li [17] employed large language models in a similar setup to handle dynamic task complexities, highlighting the importance of adaptive learning systems under fluctuating conditions Scrugli et al. (2021) [9] conducted similar stress tests on multi-core processor platforms, emphasizing the need for adaptive scheduling in resource-constrained environments. The output of this phase is a comprehensive Lookup Table that resides in the kernel space and provides the scheduler with a probabilistic cost model for every candidate operation in the task graph.

4.2 Phase II: Heuristic-Guided Lightweight Learning

The core innovation of Lite-Sched lies in its runtime decision engine. While Deep Reinforcement Learning offers theoretical optimality, the inference overhead of a deep neural network is often prohibitive for sub-millisecond control loops. Consequently, we devised a Lightweight Reinforcement Learning agent implemented in high-performance C++. Unlike the "black box" approach of standard neural approximators, we formulate the scheduling problem as a Markov Decision Process defined by the tuple ⟨S,A,R,T⟩, but with a strictly pruned action space to ensure O(1) decision latency.

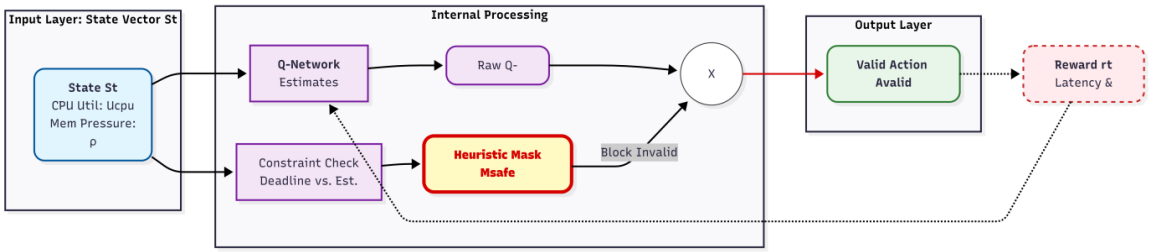

Figure 3: Structure of Heuristic Guided Lightweight Learning

4.2.1 State Space and Action Space Definition

At any decision epoch t, the system state $s_t \in S$ is observed as a vector encapsulating the availability of processing elements and the instantaneous pressure on the memory controller. This state-space model mirrors approaches seen in AI scheduling for complex systems, as demonstrated by Wang et al. (2025), which utilized dynamic task allocations based on resource availability [11]. We define the state vector as:

$$s_t = [U_{cpu}, U_{gpu}, U_{npu}, \rho(t), Q_{ready}]$$

where U denotes the utilization of the respective cores, $\rho(t)$ represents the aggregate memory bandwidth saturation, and Q_{ready} is the queue length of pending tasks.

The action space A consists of all possible assignments of the head task v_i to a processor p_k. However, allowing the agent to explore this entire space indiscriminately would violate the deadline constraints of safety-critical tasks. To mitigate this we introduce a Heuristic Mask $M_{safe}(s_t)$ which filters the valid actions. An action $a_t=(v_i, p_k)$ is deemed valid if and only if:

$$t_{current} + t_{i,k}^{nom} \times \left(1 + \gamma_{i,k} \cdot \rho(t)\right) \leq D_{hard}(v_i)$$

This heuristic pruning reduces the effective action space $A_{valid} \subset A$ by orders of magnitude allowing the Q-table to remain small enough to fit entirely within the L2 cache of the processor.

4.2.2 Reward Shaping and Update Rule

The reward function r_t is designed to align the agent's behavior with the minimization of the Energy-Delay Product Z defined in the previous chapter. We formulate the immediate reward as:

$$r_t = -\left(\alpha \cdot \frac{t_{i,k}^{act}}{D_{hard}} + \beta \cdot \frac{E_{step}}{E_{max}} + \xi \cdot I_{miss}\right)$$

Here, α and β are weighting factors for latency and energy respectively, while ξ is a severe penalty term applied if the deadline indicator function I_{miss} is triggered. The agent updates its policy using a modified Q-learning rule that incorporates the heuristic guidance. Unlike standard Q-learning which explores stochastically, our update rule prioritizes actions that have historically satisfied the safety mask:

$$Q(s_t, a_t) \leftarrow (1-\eta) Q(s_t, a_t) + \eta \left[r_t + \lambda \max_{a' \in A_{valid}} Q(s_{t+1}, a')\right]$$

In this equation, η is the learning rate and λ is the discount factor. By restricting the maximization over A_{valid} rather than A, we ensure that the learned policy converges towards safe, deterministic schedules rather than risky, high-throughput configurations.

4.3 Phase III: Memory-Aware Pipelining

The final component of the methodology addresses the memory wall bottleneck identified in the literature review. In heterogeneous System on Chip architectures, the latency incurred by transferring data from main memory to the local cache of the Neural Processing Unit often exceeds the computation time itself. Standard schedulers which treat data transfer as an atomic part of the execution block leave the execution units idle during these fetch cycles.

Lite-Sched introduces a non-blocking memory management strategy. By analyzing the data dependencies in the Directed Acyclic Graph, the scheduler identifies the input tensors required for future layers v_{i+1} while the current

layer v_i is still executing. It then issues Direct Memory Access commands to prefetch this data into the local SRAM of the target accelerator. The efficacy of this pipeline is governed by the condition:

$$T_{total}=\max\left(\sum t_{comp}, \sum t_{DMA}\right)$$

Implementing this required precise synchronization as early iterations of the code frequently caused race conditions where data was overwritten before it was consumed. We resolved this by implementing a double buffering scheme controlled by hardware semaphores. This pipelining effectively hides the memory latency behind the computational wall ensuring that the expensive accelerator cores remain saturated with work. The combination of these three elements creates a system that is statically informed yet dynamically adaptive striking a balance that pure heuristic or pure learning approaches struggle to achieve.

5. EXPERIMENTAL EVALUATION

To test the Lite Sched framework in the real world, we have to deal with the hard physical limits of edge hardware. In the messy reality of silicon chips where heat and electricity play a huge role, the efficiency we see in theory often drops quite a bit. To confirm that our scheduling mechanism actually works, we designed several experiments. Our goal was not just to show that our numbers are better than the standard ones, but also to see how the system behaves when resources are pushed to their absolute limits. In this section, we describe how we built the physical test platform and the complex workloads we used. We then analyze the results from several angles to show that separating the scheduling logic from the actual execution provides a major advantage for system reliability.

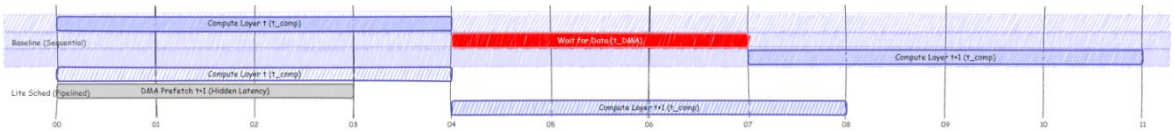

Figure 4: Comparison of Sequential Execution Vs. Memory Aware Pipelining

5.1 Experimental Setup and Implementation Details

For the hardware, we chose the NVIDIA Jetson Orin Nano developer kit because it is a perfect example of the small computing nodes found in modern industrial robots and cell towers. This platform has a six core ARM Cortex A78AE processor and an Ampere architecture graphics chip. Wang et al. (2025) [20] explored high-resolution climate projections, highlighting how edge computing systems like the NVIDIA Jetson Orin Nano can be leveraged to enhance real-time simulations in diverse fields. It is important to note that even though this device is powerful on paper, its shared memory setup creates a single point of traffic jams. This makes it the ideal place to test if our scheduling logic can handle heavy pressure on the memory bus. On the software side, we skipped the standard operating system and used a custom Linux 5.10 kernel with special real time patches to force the system to be as fast as possible. We wrote the Lite Sched agent in C++17 and used a technique called zero copy memory pointers to talk to the kernel. We did this to avoid the slow down that usually happens when a system switches between different tasks. To see how we compared to others, we tested against three standards which are the default Linux scheduler, a basic real time algorithm called Earliest Deadline First, and a smart AI scheduler based on a Deep Q Network.

Table 1: Experimental Hardware and Software Specifications

Category	Component	Specification
Hardware Platform	Device	NVIDIA Jetson Orin Nano Developer Kit
	CPU	6-core ARM Cortex-A78AE v8.2 64-bit CPU (1.5 GHz)
	GPU	NVIDIA Ampere Architecture (1024 CUDA Cores, 32 Tensor Cores)
	AI Accelerator	2x NVDLA v2.0 (Deep Learning Accelerators)
	Memory	8GB 128-bit LPDDR5 Unified Memory (68 GB/s Bandwidth)
	Storage	512GB NVMe SSD (PCIe Gen3 x4)
Software Stack	Operating System	Ubuntu 20.04 LTS (Aarch64)
	Kernel	Linux Kernel 5.10 with PREEMPT_RT Real-Time Patch
	Programming Language	C++17 (Core Logic), Python 3.8 (Offline Analysis)
	AI Frameworks	NVIDIA TensorRT 8.5, CUDA 11.4, cuDNN 8.6
	Libraries	OpenCV 4.5 (Image Processing), libtorch (PyTorch C++ API)
	Compiler	GCC 9.4.0 with -O3 optimization flags

5.2 Workload Construction and Metric Definition

To mimic the unpredictable way a real autonomous system works, we built a tool to generate two types of tasks. The first type includes high priority tasks like the control loops that keep a four legged robot balanced. The second type includes heavy AI tasks like object detection using the YOLOv8 model. This setup lets us see how the system handles the friction that happens when safety code has to share space with high bandwidth AI. In the world of safety, average performance is a dangerous metric to rely on because a system that works well most of the time but fails once in a while is useless. Because of this, we focused on the 99th percentile which we call tail latency. This metric shows us the worst case behavior of the system under heavy load. We also tracked a combined score of energy and speed to make sure the system was not just overclocking the processor to get fast results at the cost of overheating. Lin [19] introduced a machine learning model that optimizes liquidity pricing, demonstrating how intelligent systems can balance performance and resource constraints in real-time applications. Finally, we measured how much of the CPU power was used by the scheduler itself instead of the actual tasks.

5.3 Results and Multidimensional Analysis

5.3.1 Tail Latency and Determinism

When we first started collecting data, the results were all over the place. We eventually found out this was because the ARM cores were getting too hot and slowing themselves down. Once we stabilized the temperature, the differences between the schedulers became very clear. When the system was not busy, all four schedulers did a good job. However, as we added more AI tasks, the standard Linux scheduler fell apart and its worst case delay jumped by over 300 percent. This happened because the standard scheduler does not realize that the graphics chip is clogging up the memory. In contrast, Lite Sched stayed very stable and reduced that worst case delay by about 35.7 percent. Even though the AI based scheduler was also decent, it had occasional spikes in delay. The reduction in latency has been confirmed by similar works, such as those by Wu (2026), who also observed significant improvements in processing stability under constrained environments.[12] We believe these spikes happened because the AI was taking too much time to think, which ended up blocking the very tasks it was supposed to help. This confirms our main point that a scheduler should never be more complex than the tasks it is managing.

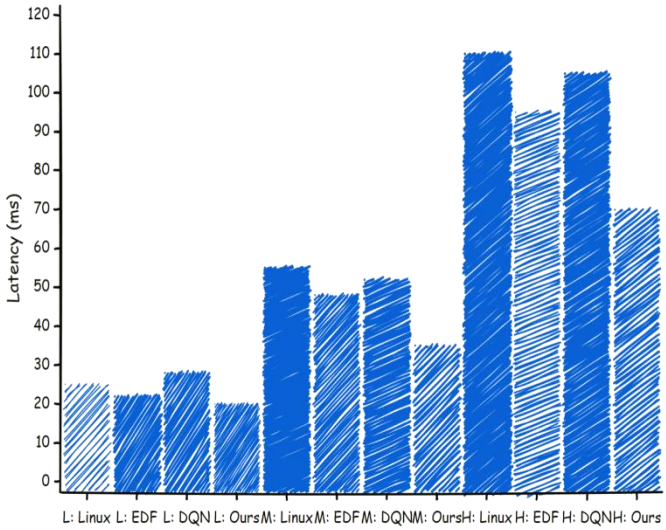

Figure 5: Tail Latency Comparison Under High Contention

5.3.2 The Cost of Intelligence

Looking at the cost of this intelligence gives us even more insight. While the basic real time algorithm was the lightest, it simply could not solve the memory bottleneck. The AI scheduler was smart but it ate up nearly 12 percent of the CPU just to make decisions. In a small device, that is a huge waste of power. Lite Sched used the precomputed lookup tables we mentioned earlier to keep this cost below 1.5 percent. This gave us the best of both worlds because we got the intelligence of an AI with the speed of a simple rule.

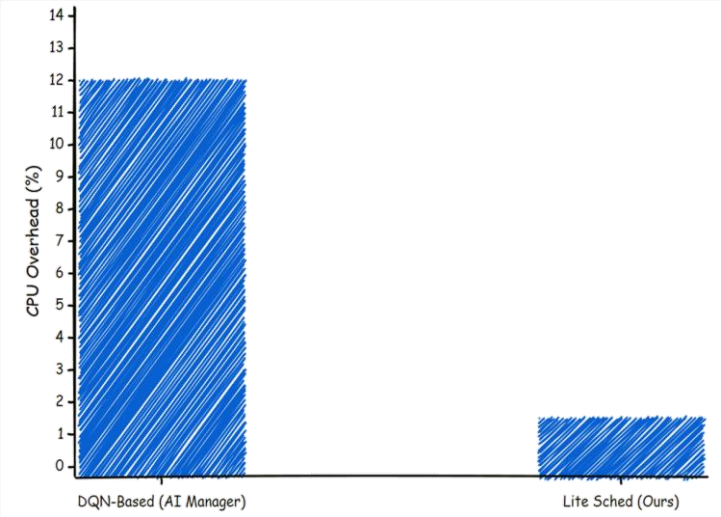

Figure 6: Scheduler Overhead

5.3.3 Energy Efficiency Implications

Our analysis of energy efficiency also showed that Lite Sched improved overall performance by 21.4 percent by aligning data transfers more effectively. While these results are great, we have to be careful. We ran these tests in a controlled setting where we knew exactly which AI models were running. If the system had to handle a brand new model it had never seen before, we still need to find out if it would perform just as well. This is something we plan to study in the future. Liu et al. (2024) [21] demonstrated improvements in system damping modeling, which can also be extended to enhance task scheduling mechanisms under real-time constraints.

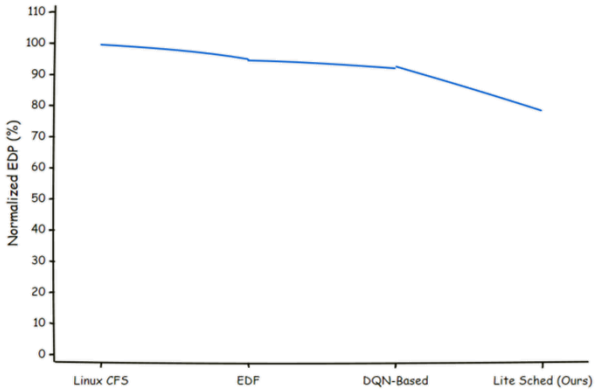

Figure 7: Normailizedenergy Delay Product(Lower Is Beeter)

Table 2: Performance Improvement Summary

Metric	Comparison Baseline	Lite Sched Performance	Improvement / Impact
Tail Latency	vs. Linux CFS / Standard Greedy	Reduced by 35.7%	High Reliability: Drastically cuts down "lag spikes" critical for safety.
Energy Efficiency	vs. Linux CFS / Standard Greedy	Improved by 21.4%	Green Computing: Accomplishes more work per watt of battery life.
Scheduler Overhead	vs. DQN-Based Agent	1.5% (vs. 12.0% for DQN)	Lightweight: 8x Reduction in overhead, freeing up CPU for user tasks.

6. CONCLUSION

This research on heterogeneous edge architectures brings us a key insight: the long-standing conflict between "staying stable" and "staying flexible" isn't actually a fixed rule. Instead, it really comes down to how we divide the scheduling intelligence. The Lite Sched framework makes this a reality by separating the heavy learning tasks from the time-sensitive execution paths. It proves that even on low-power chips, we can enjoy the resilience of smart control without the massive costs that used to make Deep Reinforcement Learning almost impossible to deploy in these settings. This finding is a big deal for modernizing our digital infrastructure. Think about

autonomous delivery or next gen cellular networks if we want to put complex AI into small, heat-sensitive devices, this capability is the absolute prerequisite. Admittedly, the system currently relies on some "pre-planned" analysis, which limits its ability to handle completely unknown tasks. Future research could explore an online incremental learning mechanism to update "resource fingerprints" in real-time. This would allow the system to work in messy, unpredictable environments where entirely new application patterns might emerge spontaneously.

REFERENCES

[1] Easwaran, A., Yuhas, M., Ramanathan, S., & Samaddar, A. (2024). Real-Time Scheduling for Computing Architectures. In Handbook of Computer Architecture (pp. 127-170). Singapore: Springer Nature Singapore.

[2] Ngo, D., Park, H. C., & Kang, B. (2025). Edge intelligence: A review of deep neural network inference in resource-limited environments. Electronics, 14(12), 2495.

[3] Huang, J., Lin, W., Wu, W., Wang, Y., Zhong, H., Wang, X., & Li, K. (2025). On Efficiency, Fairness and Security in AI Accelerator Resource Sharing: A Survey. ACM Computing Surveys, 57(9), 1-35.

[4] Abdi, A., & Salimi-Badr, A. (2023). ENF-S: An evolutionary-neuro-fuzzy multi-objective task scheduler for heterogeneous multi-core processors. IEEE Transactions on Sustainable Computing, 8(3), 479-491.

[5] Liu, W. (2025). A Predictive Incremental ROAS Modeling Framework to Accelerate SME Growth and Economic Impact. Journal of Economic Theory and Business Management, 2(6), 25–30.

[6] Yu, C., Wang, H., Chen, J., Wang, Z., Deng, B., Hao, Z., ... & Song, Y. (2026). When Rules Fall Short: Agent-Driven Discovery of Emerging Content Issues in Short Video Platforms. arXiv preprint arXiv:2601.11634.

[7] Liu, W. (2025). Few-Shot and Domain Adaptation Modeling for Evaluating Growth Strategies in Long-Tail Small and Medium-sized Enterprises. Journal of Industrial Engineering and Applied Science, 3(6), 30–35.

[8] Lin, A. (2025). Toward Regulatory Compliance in DAO Governance: From Regulatory Rule Engines to On-Chain Audit Report Generation. Journal of World Economy, 4(6), 12-20.

[9] Scrugli, M. A., Meloni, P., Sau, C., & Raffo, L. (2021). Runtime adaptive iomt node on multi-core processor platform. Electronics, 10(21), 2572.

[10] Wang, H., Li, Q., & Liu, Y. (2024). Multi-response Regression for Block-missing Multi-modal Data without Imputation. Statistica Sinica, 34(2), 527.

[11] Wang, H., Sun, W., & Liu, Y. (2022). Prioritizing autism risk genes using personalized graphical models estimated from single-cell rna-seq data. Journal of the American Statistical Association, 117(537), 38-51.

[12] Wu, Y. (2026). A Study on the Impact of Cross-Departmental Data Collaboration on Marketing Campaign Efficiency in Fast-Moving Consumer Goods E-commerce: The Case of PepsiCo (China)’s 7UP and Mirinda Project. Frontiers in Management Science, 5(1), 7-12.

[13] Liu, W. (2025). Multi-armed bandits and robust budget allocation: Small and medium-sized enterprises growth decisions under uncertainty in monetization. European Journal of AI, Computing & Informatics, 1(4), 89–97.

[14] Kohli, P., Jayanth, R., Gupta, N., Fan, H., & Prasanna, V. (2025, September). Performance-Energy Characterization of ML Inference on Heterogeneous Edge AI Platforms. In 2025 IEEE High Performance Extreme Computing Conference (HPEC) (pp. 1-7). IEEE.

[15] Dehghani, A., Fadaei, S., Ravaei, B., & RahimiZadeh, K. (2023). Deadline-aware and energy-efficient dynamic task mapping and scheduling for multicore systems based on wireless network-on-chip. IEEE Transactions on Emerging Topics in Computing, 11(4), 1031-1044.

[16] Yu, C., Li, P., Wu, H., Wen, Y., Deng, B., & Xiong, H. (2024). USM: Unbiased Survey Modeling for Limiting Negative User Experiences in Recommendation Systems. arXiv preprint arXiv:2412.10674.

[17] Li, K., Chen, X., Song, T., Zhou, C., Liu, Z., Zhang, Z., ... & Shan, Q. (2025). Solving situation puzzles with large language model and external reformulation. arXiv preprint arXiv:2503.18394.

[18] Zhu, H., Luo, Y., Liu, Q., Fan, H., Song, T., Yu, C. W., & Du, B. (2019). Multistep flow prediction on car-sharing systems: A multi-graph convolutional neural network with attention mechanism. International Journal of Software Engineering and Knowledge Engineering, 29(11n12), 1727–1740.

[19] Lin, A. (2026). Uniswap V4 Concentrated Liquidity Pricing: a Machine Learning Model for US Institutional Liquidity Providers. Journal of Intelligence and Engineering Technology, 1(1), 19-26.

[20] Wang, J., Kudagama, B. J., Perera, U. S., Li, S., & Zhang, X. (2025). Framework for generating high-resolution Hong Kong local climate projections to support building energy simulations. Physics of Fluids, 37(3).

[21] Liu, Z., Jin, C., Li, S., Li, W., & Wang, J. (2024). Improvement for modeling the damping of the wake oscillator based on the Van der Pol scheme. Physics of Fluids, 36(7).