Transactions on Computer Science and Intelligent Systems Research
ISSN: 2960-1800, eISSN: 2960-2238 | Volume 14
TPCEE 2026

Tail-Latency-Aware Execution Configuration for Edge AI under Resource Contention

Fengyuan Zhang *, Jianbo Ding

[1] New York University, New York, NY, USA

[2] SonicWall, CA, USA

* Corresponding Author: fz2240@nyu.edu

Abstract. Edge AI services are increasingly deployed in single-vCPU execution slots—containers and microVMs that share one virtual core with data-collection, compression, and telemetry processes. In this setting, the latency measured on an idle device says little about the latency under contention: co-located programs and request bursts produce occasional long delays that dominate the p95/p99 tail. This paper studies how the execution configuration of an inference runtime—the intra-op thread count, the concurrency cap, and the thread-wait policy—should be selected online to keep response times stable while model weights, precision, and input size remain fixed. A measurement study on ONNX Runtime with MobileNetV2 and ResNet-18 shows that no static configuration is adequate: spin-waiting reduces throughput by 1.6–3.7×, fair-share scheduling cuts the capacity of the framework default to one half or one third under co-runners, and the concurrency that restores capacity under contention inflates the quiet-state service time about fourfold. We present TASC, a tail-aware selector of execution configurations that profiles a Pareto-pruned candidate set offline, estimates the external runnable load from per-thread scheduler statistics with explicit compensation for the service's own threads, predicts the p99 response time of every candidate from backlog, burstiness, and profiled service-time distributions, and switches under a switching-cost-aware asymmetric hysteresis. In open-loop experiments driven by production Azure arrival traces, real ImageNet payloads, and real co-located programs, TASC is the most robust of four policies across eight model–scenario combinations: its p99 latency stays within 1.15× of the per-combination best (best static configuration: 1.44×, threshold rule: 1.52×, framework default: 7.33×), it lowers the p99 latency by up to 6.4× relative to the default, and its controller consumes about 0.25% of the CPU.

Keywords: Edge AI; Tail latency; Resource contention; Execution configuration; ONNX Runtime; Online adaptation.

1. Introduction

Running deep neural network (DNN) inference close to the data source reduces network latency, bandwidth consumption, and privacy exposure, and edge intelligence has become a mainstream deployment paradigm [1]. In practice, an edge inference service rarely owns its device. Edge gateways and serverless edge platforms package applications into containers or lightweight virtual machines with small CPU allocations [2], and the inference service shares that allocation with sensor ingestion, log compression, telemetry, or a second application. A model that completes a request in 11 ms on an idle slot may take several times longer while a co-located program is active, and a burst of requests that arrives during such a period accumulates into a backlog that persists long after the burst has ended. Users experience these events as occasional long delays. Because interactive systems are judged by their slowest responses rather than by their average [3], the relevant metrics are tail percentiles such as p95 and p99 together with the fraction of requests that miss a deadline.

Inference runtimes expose a small set of execution-layer knobs that strongly influence these metrics without modifying the model: the number of intra-operator threads, the number of requests executed concurrently, and the way idle threads wait for work. ONNX Runtime, for example, sizes its intra-op thread pool to the number of physical cores and lets idle threads spin before they sleep [4]. These defaults are tuned for a dedicated machine. On a single shared vCPU they lead to two failure modes

Content from this work may be used under the terms of CC BY-NC 4.0 licence (https://creativecommons.org/licenses/by-nc/4.0/).
Published by Warwick Evans Publishing.

that Section II quantifies: the service loses CPU share to co-runners and saturates at moderate request rates, and attempts to regain throughput with more threads collapse when the threads spin. Conversely, a configuration that is robust under contention—several concurrent single-threaded workers—quadruples the service time when the slot is quiet. The best configuration therefore depends on both the current load and the current contention, and both change on a time scale of seconds.

Selecting the configuration online raises three difficulties. First, the contention level is not directly observable: scheduler signals such as run-queue delay are dominated by the service's own threads, so a naive controller confuses its own parallelism with external interference and reacts to the consequences of its previous decision. Second, the objective is the latency tail, which is driven by transient backlog and arrival burstiness rather than by the mean service time that profiles usually report. Third, reconfiguration has a cost and the signals are noisy, so purely reactive rules oscillate. Datacenter serving systems address related problems by adapting batch sizes, replicas, or model variants [5–8], and colocation managers partition hardware resources between latency-critical and batch tasks [9–11]. These mechanisms require multiple machines or privileged control of isolation hardware, neither of which is available inside an edge execution slot.

This paper presents TASC (Tail-Aware Selection of execution Configurations), a runtime controller that selects the concurrency cap, the intra-op thread count, and the wait policy of an ONNX Runtime service online. The contributions are as follows:

A characterization of execution configurations on a single-vCPU microVM under real co-located programs, showing that spin-waiting, fair-share capacity loss, and the concurrency–latency trade-off make every static configuration fragile (Section II).

A self-interference-compensated contention estimator that infers the external runnable load from per-thread scheduler statistics with a mean absolute error of 0.10 runnable threads, compared with 1.23 for the uncompensated signal (Section III-C).

A tail-latency predictor that combines a backlog-drain term with a G/G/k waiting-time tail and profiled service-time quantiles, together with a switching-cost-aware selector with asymmetric hysteresis (Sections III-D and III-E).

An open-loop evaluation with production Azure arrival traces, ImageNet payloads, and four contention scenarios, in which TASC stays within 1.15× of the per-combination best p99 latency in all eight model–scenario combinations, whereas the best static configuration, a threshold rule, and the framework default reach 1.44×, 1.52×, and 7.33× (Section V).

Section II characterizes the configuration space, Section III presents TASC, Section IV describes the methodology, Section V reports the results, Section VI reviews related work, and Section VII concludes.

2. Background and Motivation

2.1. Execution Configuration in ONNX Runtime

ONNX Runtime executes a model through an inference session. Each session owns an intra-op thread pool whose size is set by intra_op_num_threads; the calling thread participates in the computation, so a session with n threads creates $n - 1$ helper threads, and the session option session.intra_op.allow_spinning (enabled by default) lets idle helpers spin before blocking [4]. Concurrency is obtained by issuing inference calls from several threads; to avoid contention inside a shared pool, each worker owns its own session. We denote a configuration by $c = (k, n, w)$, where k is the number of concurrently active workers (the concurrency cap), n is the intra-op thread count per worker, and $w \in \{b, s\}$ denotes blocking or spinning waits; k2n1b, for instance, denotes two concurrent single-threaded workers with blocking waits. Model weights, numerical precision (FP32), and input size (224 × 224) are fixed, so a configuration changes only how the same computation is scheduled.

2.2. Single-vCPU Edge Slots

The target platform is a lightweight KVM-based microVM of the kind used for serverless functions [2], with one vCPU of an Intel Xeon processor at 2.1 GHz, 4 GB of memory, and Linux 6.18. The kernel scheduler time-slices the CPU among the runnable threads of a scheduling group in proportion to their weights [12]; programs that run in the same container or session as the service therefore compete with it thread by thread. We emulate two common classes of co-located programs with real software: a CPU-bound compression job (gzip -6 over a 64-MB file in a loop) and a memory-bound copy kernel in the style of STREAM [13] over two 128-MB arrays. Five contention states are used throughout: quiet, CPU×1, CPU×2, Mem×1, and Mem×2, where the multiplier is the number of co-runner instances.

2.3. Characterization

For every blocking configuration with $k \cdot n \leq 4$ and two spinning variants, we measured the closed-loop saturation throughput and the per-request service time—JPEG decoding and pre-processing, inference, and post-processing—in each contention state. Fig. 1 shows MobileNetV2 and Table I summarizes both models. Four observations motivate the design.

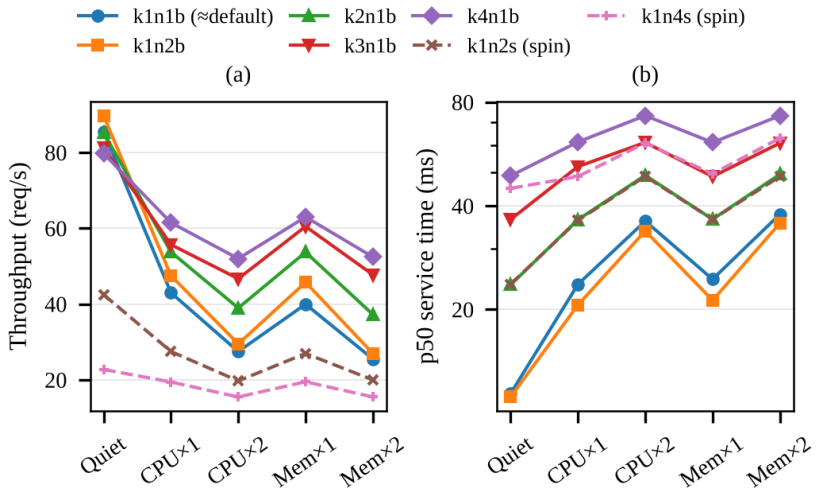

Figure 1. Characterization of MobileNetV2 execution configurations on one vCPU under five contention states: (a) saturation throughput and (b) median per-request service time including pre- and post-processing. Dashed lines denote spin-waiting configurations.

Observation 1 (spin-waiting is harmful on one vCPU). With $n = 2$ and $n = 4$, spinning reduces MobileNetV2 throughput from 89.7 to 42.5 req/s and from 84.2 to 22.7 req/s, and ResNet-18 throughput from 28.5 to 17.3 and 9.7 req/s. A spinning helper occupies the only CPU while the thread it waits for cannot run, which is the regime in which competitive spin-then-block analysis favors blocking [14]. All candidate configurations of TASC therefore block.

Observation 2 (contention removes capacity in proportion to the fair share). The single-worker configuration k1n1b, which matches the framework default on this platform, drops from 85.5 req/s in the quiet state to 43.0 and 27.5 req/s under one and two CPU-bound co-runners; ResNet-18 drops from 31.3 to 15.8 and 10.5 req/s. A request rate that occupies 28% of the quiet capacity thus occupies 87% of the capacity under two co-runners.

Observation 3 (concurrency restores capacity but costs latency when the slot is quiet). Because each runnable worker thread receives a share of the CPU, k4n1b sustains 52.0 req/s under CPU×2 (1.89× k1n1b) for MobileNetV2 and 21.2 req/s (2.02×) for ResNet-18. In the quiet state, however, four concurrent requests share one CPU, and the median service time grows from 11.4 to 48.9 ms (MobileNetV2) and from 31.5 to 127.7 ms (ResNet-18).

Observation 4 (the contention type shapes the tail). Intra-op parallelism with blocking waits shortens a single request under a CPU-bound co-runner (MobileNetV2 under CPU×1: 19.7 ms for k1n3b versus 23.7 ms for k1n1b), but memory-bound co-runners inflate its tail: the p99 service time of k1n4b is 105 ms under Mem×2 versus 45 ms under CPU×2, although the two throughputs are similar.

Table I. Saturation throughput (req/s) of candidate configurations

	MobileNetV2			ResNet-18		
Config.	Quiet	CPU×2	Mem×2	Quiet	CPU×2	Mem×2
k1n1b	85.5	27.5	25.5	31.3	10.5	10.5
k1n2b	89.7	29.5	27.0	28.5	12.0	12.2
k1n3b	89.2	29.5	27.0	28.8	12.0	12.2
k1n4b	84.2	27.2	25.7	28.5	11.7	11.5
k2n1b	85.2	39.0	37.2	31.8	15.7	15.2
k2n2b	84.9	41.2	40.2	30.3	17.3	16.7
k3n1b	81.4	46.7	47.7	32.0	19.0	18.5
k4n1b	79.7	52.0	52.4	31.3	21.2	20.0
k1n2s	42.5	19.7	20.0	17.3	8.2	8.5
k1n4s	22.7	15.5	15.5	9.7	6.2	6.7

Bold: highest value per column among blocking configurations; k1n2s and k1n4s use spin-waiting.

Switching between pre-created sessions is inexpensive: the first request after a switch is at most 2.2 ms (MobileNetV2) and 4.7 ms (ResNet-18) slower than in steady state. Taken together, the configuration that minimizes the tail changes with both the load and the contention state, and it can be changed cheaply—precisely the conditions under which online selection pays off.

3. TASC Design

3.1. Overview

Fig. 2 shows the structure of TASC. Requests enter a FIFO queue and are dispatched to k active workers, each of which owns a session with n intra-op threads and wait policy w drawn from a pre-created session pool. Every 100 ms a controller reads per-thread scheduler statistics, the backlog, and the arrival history, updates the contention estimate, predicts the p99 response time of every candidate configuration, and decides whether to switch. A switch changes only which workers are active and which sessions they use; in-flight requests complete on their original session.

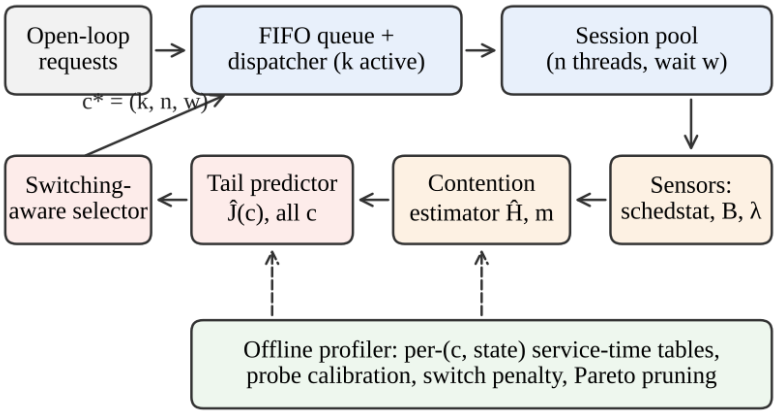

Figure 2. Architecture of TASC. Solid arrows form the online control loop; dashed arrows denote offline profiles.

3.2. Offline Profiling and Candidate Pruning

For each candidate c and contention state s, the profiler runs the service in closed loop for 4–6 s and records the throughput Xc(s), the mean, squared coefficient of variation cs2, and 99th percentile S99, c(s) of the service time, the CPU time per request κc(s), and the ratio rc(s) of run-queue waiting time to running time of the service threads. It also measures the switching penalty πc and calibrates an idle probe (Section III-C). A candidate is pruned if another candidate with an equal or smaller thread footprint k·n has a p99 service time that is no higher and a throughput that is no lower (within 3%) in every state. Pruning keeps seven of the eight blocking candidates for MobileNetV2 and all eight for ResNet-18, so the pools contain seven and eight sessions. Profiling takes about 5 min (MobileNetV2) and 7 min (ResNet-18) per model and platform.

3.3. Self-Interference-Compensated Contention Estimation

The quantity that determines how each candidate behaves is the external runnable load H, the time-averaged number of runnable threads that do not belong to the service. Linux exposes the cumulative running and run-queue waiting time of every thread (/proc/self/task/*/schedstat). Consider one CPU shared fairly among the R(t) runnable service threads and H external runnable threads. A runnable service thread then accumulates running time at rate 1/(R + H) and waiting time at rate (R + H − 1)/(R + H). Summing over the service threads in a window gives

$$W/U = EU[R] + H - 1 \tag{1}$$

Where W and U are the total waiting and running times and EU[R] is the running-time-weighted mean number of runnable service threads. The raw ratio W/U is therefore contaminated by the service's own parallelism—k4n1b alone produces W/U ≈ 3 on an idle slot—and a controller that uses it directly mistakes its own concurrency for interference. TASC removes this term with the quiet-state profile. Let $\bar{\jmath}$ be the time-averaged number of in-service requests during busy periods, which the dispatcher measures, rc0 the quiet-state ratio of c, and νn = 1 + r0(1, n) the number of runnable threads contributed by one request with n intra-op threads. TASC estimates

$$\hat{R}(c, \bar{\jmath}) = \nu n + (rc0 + 1 - \nu n)(\bar{\jmath} - 1)/(k - 1) \tag{2}$$

Which equals νn with one request in service and rc0 + 1 at saturation, and computes $\hat{H} = \max\{0, W/U + 1 - \hat{R}\}$, smoothed by an exponentially weighted moving average. When the service is idle, (1) carries no information; the controller then executes a 4-ms busy probe and uses its own wait-to-run ratio, calibrated offline, to detect whether external load is present. Fig. 3 compares the estimate with the injected load for 80 configuration–state pairs at saturation: the mean absolute error is 0.10 (maximum 0.43) for $\hat{H}$ and 1.23 for the raw ratio. A second signal, the CPU time per completed request divided by κc(quiet), measures memory interference and weights the CPU-type and memory-type profiles where they differ.

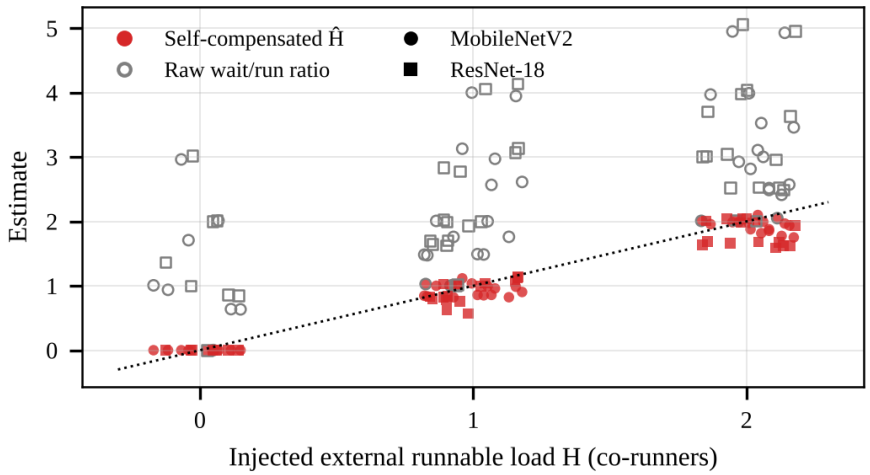

Figure 3. Contention estimation at saturation for all blocking configurations and contention states of both models. The self-compensated estimate $\hat{H}$ follows the injected external runnable load, whereas the raw wait-to-run ratio also counts the service's own parallelism. Points are jittered horizontally.

3.4. Tail-Latency Prediction

For each candidate c, TASC interpolates the profile at $\hat{H}$ (linearly between states, with fair-share scaling beyond two co-runners) to obtain Xc, $\bar{S}c$, S99,c, and cs2. The arrival-rate estimate $\hat{\lambda}$ is the maximum of the 2-s rate and the mean of the 0.5-s and 2-s rates, which reacts to bursts without discarding history, and ca2 is the squared coefficient of variation of the inter-arrival times over the last 2 s. With backlog B (queued plus in-service requests), a newly arriving request first waits for the backlog to drain:

$$Dc = \max\{0, B - k + 1\}/Xc \tag{3}$$

For the steady-state component, TASC treats c as k servers of rate $\mu = Xc/k$ with offered load $a = \hat{\lambda}/\mu$. With the Erlang-C probability of waiting Pw, the Allen–Cunneen approximation [15] gives

$$E[W] = Pw(ca2 + cs2) / (2(k\mu - \hat{\lambda})) \tag{4}$$

And, with an exponential conditional waiting time, the 99th percentile of the waiting time is

$$W99 = (E[W]/Pw) \ln(Pw/0.01) \tag{5}$$

For $Pw > 0.01$ and zero otherwise. If the utilization $\rho = \hat{\lambda}/Xc$ reaches 0.97, the candidate is instead charged the backlog growth over a 2-s horizon. The predicted tail is

$$\hat{J}c = \max\{Dc, W99\} + S99,c. \tag{6}$$

The drain term dominates during bursts and favors high-capacity configurations; the steady-state term dominates at moderate load and favors configurations with short service times.

3.5. Switching-Cost-Aware Selection

TASC chooses $c^* = \text{argmin } \hat{J}c$; among candidates within 3% of the minimum it prefers the smallest thread footprint, which leaves CPU time to co-runners and keeps fewer sessions active. A switch from the current configuration c0 is executed only when two conditions hold. First, the predicted gain $G = \hat{J}c0 - \hat{J}c^*$ must exceed 10% of $\hat{J}c0$ and amortize the switching penalty over the hold time Th = 1 s,

i.e., $G \cdot \hat{\lambda} \cdot Th > \pi c^* \cdot k^*$. Second, at least Th must have elapsed since the last switch, unless c* offers a higher predicted capacity and the backlog exceeds k0 + 1. This asymmetric hysteresis escalates immediately when a backlog builds up and de-escalates only after the evidence has persisted, which suppresses oscillations caused by noisy short-window signals.

3.6. Implementation

TASC is implemented in Python 3.12 on ONNX Runtime 1.31 with the CPU execution provider. For every (n, w) used by a candidate, the session pool contains as many sessions as the largest k among those candidates. The dispatcher, the workers, and the controller are threads of one process; ONNX Runtime releases the interpreter lock during inference and the image decoder releases it during decoding. One decision over seven candidates takes 177 µs of CPU time and reading the scheduler statistics of eleven threads takes 76 µs, i.e., about 0.25% of the CPU at a 100-ms period; the idle probe runs only when the service has no work.

4. Experimental Methodology

4.1. Models, Payloads, and Pipeline

We use MobileNetV2 [16] and ResNet-18 [17] as ONNX graphs (opset 13, FP32, 224 × 224 input, batch normalization folded into convolutions) with fixed weights; the model files occupy 13.3 MB and 44.6 MB. FP32 kernel latency does not depend on the weight values, so the graphs reproduce the execution cost of the reference architectures. Each request carries one of 39 real ImageNet sample images (JPEG, 200–845 pixels per side). A worker decodes the image with DCT-domain downscaling, resizes the short side to 256 pixels, crops the central 224 × 224 region, normalizes it, runs the model, and computes a softmax with top-5 selection. The end-to-end (E2E) latency includes queueing, pre-processing, inference, and post-processing.

4.2. Workloads and Contention Scenarios

Arrivals are generated in open loop. The steady scenarios use Poisson arrivals; the bursty scenarios replay contiguous segments of the Azure LLM inference trace 2023, a production trace of request timestamps released with [18], compressed in time to the target mean rate while preserving the burst structure. The mean rates are $\lambda = 24$ req/s for MobileNetV2 and $\lambda = 8$ req/s for ResNet-18 (28% and 26% of the quiet single-worker capacity); trace scenarios are replayed at 0.6λ because their one-second peaks reach 4.4–7.3 times the mean. The deadlines are D = 150 ms and D = 450 ms. Table II lists the scenarios. The development workload DEV uses a disjoint segment of the conversation trace and a different contention sequence; it is used only to select the best static configuration. Co-runners are started in the service's session and toggled with SIGSTOP/SIGCONT at the scheduled times.

Table II. Workload scenarios

Scen.	Arrivals	ca2 / peak:mean	Contention schedule	Dur.
S1	Poisson, λ	1.0 / 1.7	CPU×2 throughout	45 s
S2	Poisson, λ	1.0 / 1.5	Mem×2 throughout	45 s
S3	Azure code trace, 0.6λ	71.5 / 6.0 (44.0 / 4.4)	quiet / CPU×2, alternating every 6 s	45 s
S4	Azure code trace, 0.6λ	96.9 / 7.3 (12.1 / 5.2)	Mem×2 → Mem×1 → Mem×2 → CPU×1 → Mem×2 → quiet	45 s
DEV	Azure conv. trace, λ	1.0 / 1.5	Mem×2 → quiet → Mem×2 → CPU×2 → Mem×1	30 s

ca2: squared coefficient of variation of inter-arrival times; peak:mean: ratio of the largest 1-s arrival count to the mean (MobileNetV2 replay; ResNet-18 in parentheses). S4 phases last 4–10 s.

4.3. Policies

We compare four policies. ORT default uses one worker and default session options; on this platform ONNX Runtime selects one intra-op thread, so it behaves as k1n1b. Best static runs the configuration with the lowest p99 latency on DEV among {default, k1n2b, k2n1b, k2n2b, k3n1b, k4n1b}: k4n1b for MobileNetV2 (p99 104 ms on DEV versus 1374 ms for the default) and k3n1b for ResNet-18 (183 ms versus 368 ms). Threshold is a reactive rule that uses the same estimator $\hat{H}$ as TASC: it selects k1n1b when $\hat{H} < 0.5$; otherwise it selects k3n1b when the backlog exceeds two requests and a single-request configuration (k1n3b for MobileNetV2, k1n2b for ResNet-18) when it does not. TASC uses the pruned candidate set and the parameters of Section III, which were fixed before the test runs.

4.4. Metrics and Measurement Integrity

E2E latency is measured from the scheduled arrival time, which avoids the coordinated-omission bias of closed-loop load generators [19, 20]. Every arrival is counted: a request that has not completed 10 s after the last arrival counts as a deadline miss with infinite latency (no such request occurred). Percentiles are nearest-rank values over all arrivals. We report p50, p95, and p99 E2E latency, the deadline-miss rate, the number of configuration switches, and the controller cost; peak resident memory is measured in separate 10-s runs in fresh processes. Each run executes in a fresh process, and a watchdog thread invalidates any run in which the clock advances by more than 0.5 s between 20-ms ticks, which detects host-level pauses; no run was invalidated.

5. Results

5.1. Overall Comparison

Tables III and IV list the results, and Fig. 4 compares the p99 latencies. Table V summarizes robustness as the ratio of each policy's p99 latency to the lowest p99 among the four policies in the same model–scenario combination. TASC is never more than 15% above the best policy, achieves the best p99 in 4 of eight combinations, and has the lowest mean deadline-miss rate (5.2%). The best static configuration wins 3 combinations but falls 44% behind in S3, the threshold rule never wins and reaches 1.52×, and the framework default is up to 7.33× worse.

Table III. E2E latency (ms), deadline-miss rate (%), and switches for MobileNetV2 (D = 150 ms)

Scen.	Policy	p50	p95	p99	Miss	Sw.
S1	ORT default	190	489	749	59.8	0
	Best static	49	80	102	0.0	0
	Threshold	61	126	150	1.2	130
	TASC	49	87	117	0.0	4
S2	ORT default	102	307	396	33.8	0
	Best static	47	82	106	0.0	0
	Threshold	60	118	144	0.7	131
	TASC	49	95	121	0.5	8
S3	ORT default	26	153	233	5.1	0
	Best static	43	192	334	9.0	0
	Threshold	37	217	354	9.0	26
	TASC	42	176	268	7.4	22
S4	ORT default	108	1481	1560	45.7	0
	Best static	74	620	733	38.1	0
	Threshold	73	666	829	32.9	37
	TASC	64	556	654	31.2	19

Bold: best p99 and best miss rate per scenario.

Table IV. E2E latency (ms), deadline-miss rate (%), and switches for ResNet-18 (D = 450 ms)

Scen.	Policy	p50	p95	p99	Miss	Sw.
S1	ORT default	190	568	691	10.6	0
	Best static	124	204	260	0.0	0
	Threshold	119	230	279	0.0	59
	TASC	114	188	246	0.0	14
S2	ORT default	163	473	651	6.9	0
	Best static	124	215	302	0.0	0
	Threshold	119	264	331	0.0	37
	TASC	111	217	271	0.0	18
S3	ORT default	88	1578	1699	31.5	0
	Best static	96	357	466	2.3	0
	Threshold	75	433	598	4.2	26
	TASC	78	385	488	2.3	15
S4	ORT default	94	389	524	2.8	0
	Best static	100	195	303	0.0	0
	Threshold	92	254	315	0.0	25
	TASC	101	190	258	0.0	18

Bold: best p99 and best miss rate per scenario.

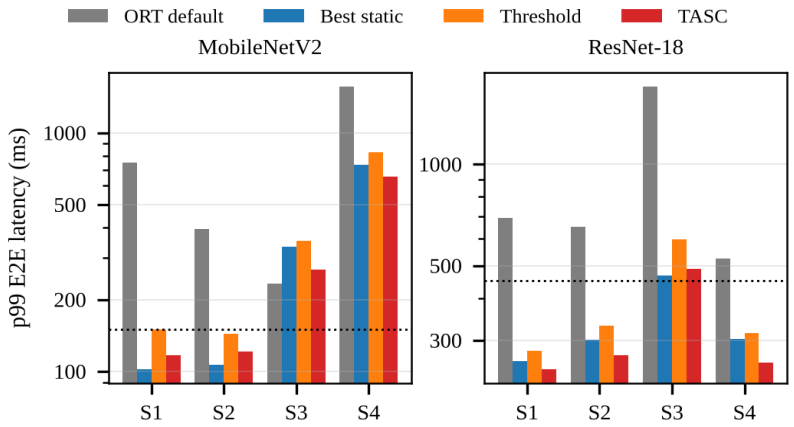

Figure 4. p99 E2E latency of the four policies in the four scenarios (log scale). Dotted lines mark the deadlines.

Table V. Robustness across the eight model–scenario combinations

Policy	Worst p99 ratio	Geo-mean p99 ratio	Wins (of 8)	Mean miss (%)
ORT default	7.33×	2.75×	1	24.5
Best static	1.44×	1.10×	3	6.2
Threshold	1.52×	1.30×	0	6.0
TASC	1.15×	1.06×	4	5.2

Ratio: p99 latency divided by the lowest p99 of the four policies in the same combination.

5.2. Sustained Contention

Under two sustained co-runners (S1 and S2), the framework default operates close to saturation. MobileNetV2 reaches a p99 latency of 749 ms with 59.8% deadline misses in S1 and 396 ms with 33.8% misses in S2; ResNet-18 reaches 691 ms and 650 ms. TASC reduces these values to 117 and

121 ms for MobileNetV2 (6.4× and 3.3× lower) and to 246 and 271 ms for ResNet-18, with at most 0.5% misses. The latency decomposition explains the gain: in S1 the p99 queueing delay of MobileNetV2 falls from 710 ms to 42 ms, while the median inference time rises from 23.7 to 33.3 ms because concurrent workers share the CPU. Trading a moderate increase in execution time for a large reduction in queueing is the central mechanism under contention. The best static configuration k4n1b was selected for heavy contention and is the steady-state optimum for MobileNetV2 in S1 and S2 (102 and 106 ms); TASC reaches 1.14–1.15× of it while switching only 4 and 8 times. For ResNet-18, TASC is better than the best static configuration in both scenarios. The threshold rule reacts to the instantaneous backlog without hysteresis, switches 130 and 131 times within 45 s for MobileNetV2, and ends 35–47% above the best p99.

5.3. Bursty Arrivals and Time-Varying Contention

S4 combines production bursts with phases of different contention types and intensities, the condition that TASC targets. TASC achieves the lowest p99 latency and miss rate for both models: 654 ms and 31.2% for MobileNetV2 (best static 733 ms, threshold 829 ms, default 1560 ms) and 258 ms without misses for ResNet-18 (303, 315, and 524 ms). Most of the remaining MobileNetV2 misses occur in bursts that arrive during Mem×2 phases, when the one-second peak arrival rate (about 105 req/s) exceeds the slot's capacity under any configuration (at most 52 req/s, Table I). Fig. 5 shows how TASC behaves: it switches to k4n1b during contended bursts and returns to the single-request configuration k1n3b when the backlog clears, spending 46.5% of the time in k1n3b and 38.3% in k4n1b. Fig. 6 shows that the advantage extends over the whole tail of the distribution.

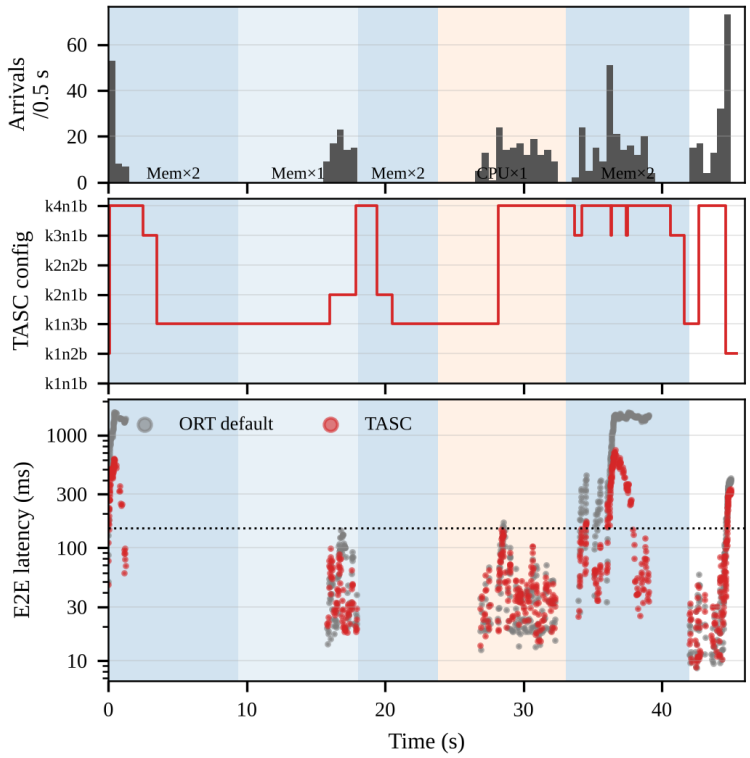

Figure 5. MobileNetV2 in S4: (a) arrivals per 0.5 s, with contention phases shaded (orange: CPU, blue: memory); (b) configuration selected by TASC; (c) E2E latency of every request under the ORT default and TASC. The dotted line marks the 150-ms deadline.

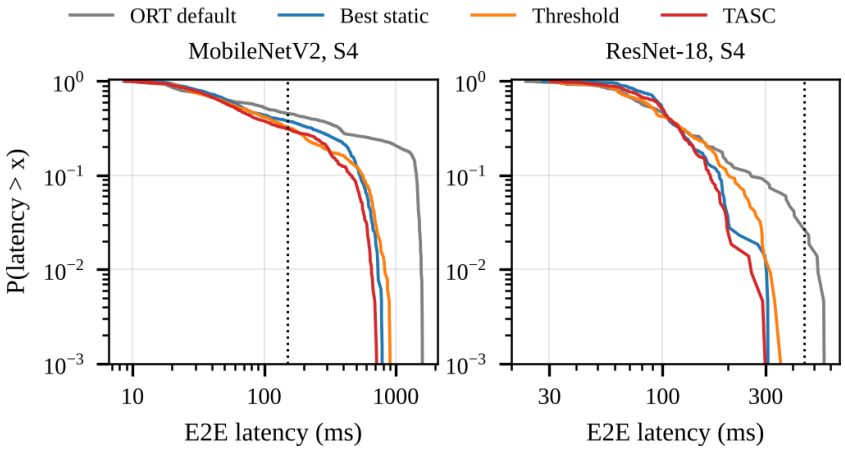

Figure 6. Complementary CDF of the E2E latency in S4. Dotted lines mark the deadlines.

S3 alternates 6-s quiet periods with 6-s periods of two CPU-bound co-runners. Splitting the requests by phase reveals two regimes. During contended periods TASC has the lowest p99 latency (136 ms, versus 200 ms for the default, 151 ms for the best static configuration, and 168 ms for the threshold rule). However, 85% of the requests (554 of 648) arrive in large bursts during quiet periods, where a single FIFO worker completes a burst in arrival order and is difficult to improve upon; this gives the default the best overall p99 for MobileNetV2 (233 ms versus 268 ms for TASC). The best static configuration, which keeps four requests in service, and the threshold rule, which reacts late to the end of contention, reach 334 and 354 ms. For ResNet-18 the default collapses (1699 ms), while TASC and the best static configuration are within 5% of each other.

5.4. Ablation

Table VI isolates the design elements on the two bursty scenarios of MobileNetV2. Hysteresis consistently stabilizes the controller: removing it more than doubles the number of switches in S3 (22 to 47) and triples it in S4 (19 to 62); in S3 this raises the p99 by 16%, whereas in S4 the additional reactivity lowers it by 6%. Self-compensation matters when quiet and contended periods alternate: without it, the controller interprets its own concurrency as interference, remains in high-concurrency configurations after contention ends, and S3 degrades by 26% in p99 latency and from 7.4% to 13.4% in misses. In S4, which is contended for 93% of the time, overestimating contention acts as over-provisioning and lowers the p99 by 17%; for ResNet-18 in S3 the two variants are within 1%. Replacing the p99 objective (6) with the predicted mean response time lowers the p99 in four of six model–scenario combinations (by 5% for ResNet-18 in S4 and by up to 41% for MobileNetV2 in S3) and raises it in two (by 11% for MobileNetV2 in S2 and 18% for ResNet-18 in S3), while switching 3.9–6.5× more often under sustained contention. The saturated S99,c in (6) is conservative for multi-worker configurations that are only partly occupied, so the p99 objective under-weights capacity during bursts; an occupancy-dependent service-time model is the direct refinement.

Table VI. Ablation of TASC on MobileNetV2 (p99 in ms, miss rate in %)

	S3			S4		
Variant	p99	Miss	Sw.	p99	Miss	Sw.
TASC (full)	268	7.4	22	654	31.2	19
w/o hysteresis	310	9.1	47	612	29.8	62
w/o self-compensation	338	13.4	25	544	31.3	20
mean objective	159	1.9	11	593	26.9	16

5.5. Overheads

Table VII lists the memory footprint. The session pool of TASC occupies 279 MB for MobileNetV2 (7 sessions) and 711 MB for ResNet-18 (8 sessions), compared with 113 and 192 MB for the single-

session default and 206 and 337 MB for the best static configuration. Candidate pruning determines the pool size, and sharing the read-only weights among the sessions of a pool reduces the footprint toward one copy of the model per distinct thread count. The controller consumes about 0.25% of the CPU, and switching penalties stay below 5 ms.

Table VII. Session pool size and peak resident memory

	MobileNetV2		ResNet-18	
Policy	Sessions	Peak RSS (MB)	Sessions	Peak RSS (MB)
ORT default	1	113	1	192
Best static	4	206	3	337
Threshold	4	203	4	381
TASC	7	279	8	711

Measured in isolated 10-s runs under two CPU-bound co-runners.

6. Related Work

Inference serving. Clipper [5] introduced adaptive batching and model selection for prediction serving, Clockwork [6] builds predictable DNN serving on the deterministic execution time of individual inferences, INFaaS [7] selects model variants and scales resources automatically, and MArk [8] combines predictive autoscaling with batching across cloud services. These systems act at the cluster or batch level; TASC acts on the execution configuration inside one slot, where neither replicas nor batching are available to absorb contention.

Interference and colocation. Bubble-Up [9] predicts the degradation caused by pressure on the memory subsystem, Heracles [10] co-locates best-effort tasks with a latency-critical service by managing isolation mechanisms in a feedback loop, and PARTIES [11] extends dynamic partitioning to multiple interactive services. Mendoza et al. [21] study interference-aware scheduling for inference serving. These approaches rely on placement or on privileged mechanisms such as cache partitioning, frequency scaling, and core allocation, whereas TASC uses only unprivileged runtime knobs and estimates interference from the service's own scheduler statistics.

Thread count and waiting policy. Thread Reinforcer [22] searches for the thread count of multithreaded programs using operating-system-level monitoring, and competitive spinning [14] analyzes when threads should spin or block. The fair-share and load-balancing behavior of the Linux scheduler [12] determines how such choices interact with co-runners. TASC applies these insights to DNN inference sessions with a tail-latency objective and an explicit model of self-interference.

Edge inference. Neurosurgeon [23] partitions DNN computation between the device and the cloud, nn-Meter [24] predicts kernel-level inference latency on edge devices, and AsyMo [25] and SANI [26] partition intra-op work on asymmetric mobile CPUs; SANI observes that static partitioning does not adapt to runtime interference. TASC is complementary: it keeps the model and kernel library fixed and adapts how requests are executed when the device is shared. Our methodology follows the open-loop, tail-centric practice of TailBench [27] and Treadmill [20].

Learning-based optimization. Deep reinforcement learning has been applied to multi-objective optimization under hard constraints in other computing domains: Zhang et al. [28] coordinate global placement and local refinement in IC physical layout with a hierarchical policy, congestion-aware reward shaping, and adaptive action-space pruning, balancing routing congestion, power, and timing. Such policies are trained offline over many episodes; TASC instead derives its decisions from a few minutes of profiling and an analytical tail model, so it needs no training on the target device and its controller uses about 0.25% of the CPU.

7. Conclusion

On a shared single-vCPU edge slot, the execution configuration of an inference runtime determines whether a model keeps a stable response time or accumulates long queues under contention. The measurements in this paper show that spin-waiting, fair-share capacity loss, and the concurrency–latency trade-off rule out every static configuration, and that the information required to choose well—external runnable load, backlog, and burstiness—can be obtained cheaply from scheduler statistics once the service's own threads are accounted for. TASC turns these observations into a predictive, switching-aware controller that keeps the p99 latency within 1.15× of the best policy in every evaluated combination and lowers it by up to 6.4× relative to the framework default at about 0.25% CPU cost. Natural extensions are an occupancy-dependent service-time model for the tail predictor, multi-core slots in which intra-op parallelism and core sharing interact, and session pools that share model weights.

References

[1] Zhou, Z., Chen, X., Li, E., Zeng, L., Luo, K., & Zhang, J. (2019). Edge intelligence: Paving the last mile of artificial intelligence with edge computing. Proceedings of the IEEE, 107(8), 1738–1762.

[2] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., & Popa, D.-M. (2020). Firecracker: Lightweight virtualization for serverless applications. In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI) (pp. 419–434).

[3] Dean, J., & Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74–80.

[4] ONNX Runtime. (n.d.). Thread management. ONNX Runtime documentation. https://onnxruntime.ai/docs/performance/tune-performance/threading.html

[5] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., & Stoica, I. (2017). Clipper: A low-latency online prediction serving system. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI) (pp. 613–627).

[6] Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., & Mace, J. (2020). Serving DNNs like Clockwork: Performance predictability from the bottom up. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (pp. 443–462).

[7] Romero, F., Li, Q., Yadwadkar, N. J., & Kozyrakis, C. (2021). INFaaS: Automated model-less inference serving. In Proceedings of the USENIX Annual Technical Conference (ATC) (pp. 397–411).

[8] Zhang, C., Yu, M., Wang, W., & Yan, F. (2019). MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In Proceedings of the USENIX Annual Technical Conference (ATC) (pp. 1049–1062).

[9] Mars, J., Tang, L., Hundt, R., Skadron, K., & Soffa, M. L. (2011). Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (MICRO) (pp. 248–259).

[10] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., & Kozyrakis, C. (2015). Heracles: Improving resource efficiency at scale. In Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).

[11] Chen, S., Delimitrou, C., & Martínez, J. F. (2019). PARTIES: QoS-aware resource partitioning for multiple interactive services. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (pp. 107–120).

[12] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., & Fedorova, A. (2016). The Linux scheduler: A decade of wasted cores. In Proceedings of the 11th European Conference on Computer Systems (EuroSys).

[13] McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 19–25.

[14] Karlin, A. R., Li, K., Manasse, M. S., & Owicki, S. (1991). Empirical studies of competitive spinning for a shared-memory multiprocessor. In Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP) (pp. 41–55).

[15] Allen, A. O. (1990). Probability, statistics, and queueing theory with computer science applications (2nd ed.). Academic Press.

[16] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). MobileNetV2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 4510–4520).

[17] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 770–778).

[18] Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., & Bianchini, R. (2024). Splitwise: Efficient generative LLM inference using phase splitting. In Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture (ISCA) (pp. 118–132).

[19] Schroeder, B., Wierman, A., & Harchol-Balter, M. (2006). Open versus closed: A cautionary tale. In Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI) (pp. 239–252).

[20] Zhang, Y., Meisner, D., Mars, J., & Tang, L. (2016). Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA) (pp. 456–468).

[21] Mendoza, D., Romero, F., Li, Q., Yadwadkar, N. J., & Kozyrakis, C. (2021). Interference-aware scheduling for inference serving. In Proceedings of the 1st Workshop on Machine Learning Systems (EuroMLSys) (pp. 80–88).

[22] Pusukuri, K. K., Gupta, R., & Bhuyan, L. N. (2011). Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC) (pp. 116–125).

[23] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., & Tang, L. (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (pp. 615–629).

[24] Zhang, L. L., Han, S., Wei, J., Zheng, N., Cao, T., Yang, Y., & Liu, Y. (2021). nn-Meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys) (pp. 81–93).

[25] Wang, M., Ding, S., Cao, T., Liu, Y., & Xu, F. (2021). AsyMo: Scalable and efficient deep-learning inference on asymmetric mobile CPUs. In Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom) (pp. 215–228).

[26] Sang, Q., He, P., Liang, H., Gong, Y., Hu, C., Zhou, X., & Cheng, D. (2026). Unleash all cores: Asymmetry-aware scalable DNN inference on mobile CPUs. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI).

[27] Kasture, H., & Sanchez, D. (2016). TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC) (pp. 1–10).

[28] Zhang, H., Ge, Y., Zhao, X., & Wang, J. (2025). Hierarchical deep reinforcement learning for multi-objective integrated circuit physical layout optimization with congestion-aware reward shaping. IEEE Access, 13, 162533–162551.