

nn-Live: Predicting Runtime Latency for Concurrent DNN Inference on Mobile Devices

Zilin Ye¹, Qianyi Huang¹, Zhicheng Luo¹, Xu Chen¹, Qian Zhang²

¹Sun Yat-sen University, ²Hong Kong University of Science and Technology

Email: {yezlin5, luozhc23}@mail2.sysu.edu.cn, {huangqy89, chenxu35}@mail.sysu.edu.cn, qianzh@cse.ust.hk

Abstract

Modern mobile applications (e.g., AR/XR, assistant robots) increasingly rely on multiple concurrent DNN models, yet their inference latency becomes highly unpredictable under dynamic resource contention from concurrent workloads. In mobile SoCs with heterogeneous CPU architectures, such as big.LITTLE designs, the latency is affected by a complex interplay of factors, including heterogeneous core performance, cache hierarchies and shared memory bandwidth. Existing latency prediction methods either fail to capture runtime dynamics or overlook the heterogeneity in mobile processors. In this paper, we propose *nn-Live* that can predict live inference latency for DNN models on mobile CPUs under dynamically changing operating conditions. *nn-Live* leverages per-core hardware counter statistics to quantify the interference from concurrent tasks, and utilizes a lightweight linear model to estimate each core's interference contribution to a specific task, enabling robust latency prediction across dynamic execution conditions. Given the large pool of DNN models and huge potential task placement configurations, *nn-Live* further adopts an adaptive hierarchical sampling strategy to reduce profiling cost while preserving configuration diversity and representativeness. We build a large-scale interference analysis dataset¹ across three mainstream SoCs. Evaluation results show that *nn-Live* achieves high prediction accuracy under diverse runtime environments, with a clear advantage over existing methods. We further demonstrate the extensibility of *nn-Live* to mobile accelerators (e.g., GPU/NPU).

CCS Concepts

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Computing methodologies** → **Machine learning**.

¹Publicly available at <https://github.com/ye3084/nn-Live-dataset>.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

MobiCom '26, Austin, TX, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2505-0/2026/10

<https://doi.org/10.1145/3795866.3844468>

Keywords

Mobile SoC, Deep Neural Network, Resource Contention, Hardware Performance Counters

ACM Reference Format:

Zilin Ye, Qianyi Huang, Zhicheng Luo, Xu Chen, Qian Zhang. 2026. *nn-Live*: Predicting Runtime Latency for Concurrent DNN Inference on Mobile Devices. In *The 32nd Annual International Conference on Mobile Computing and Networking (MobiCom '26)*, October 26–30, 2026, Austin, TX, USA. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3795866.3844468>

1 Introduction

The rapid proliferation of Deep Neural Networks (DNNs) on mobile and edge platforms has enabled a new generation of latency-critical applications, ranging from Augmented and Virtual Reality (AR/VR) to robotics and autonomous vehicles [22, 38]. Unlike traditional single-task scenarios, these systems increasingly require the simultaneous execution of multiple DNN models to perceive and interact with complex environments in real time. However, the concurrent execution of these workloads on resource-constrained mobile/edge hardware introduces significant challenges. Specifically, resource contention creates unpredictable interference patterns that severely impact inference latency. Consequently, accurate latency prediction in such volatile environments is essential to guarantee the service quality and user experience of these applications.

In recent years, several studies have attempted to model and predict DNN inference latency on mobile platforms, and these studies can be broadly summarized into two categories. The first category (e.g., BRP-NAS [9], nn-meter [34]) focuses on static latency prediction. They learn the relationship between model architecture and inference latency with offline profiling. However, they implicitly assume exclusive resource usage and an interference-free runtime environment, overlooking resource contention from concurrent tasks. Consequently, their predictions often deviate significantly from real-world behavior. The second category incorporates dynamic runtime conditions. Building upon static predictors, these methods introduce runtime signals such as CPU utilization [10, 29], memory information (usage, capacity, etc.) [5, 10, 26, 29], DVFS operating points [15, 21], or global last-level cache (LLC) miss rates [11, 23] to estimate

the impact of system dynamics on inference latency. While system-level metrics indicate certain aspects of runtime dynamics, they cannot accurately capture the heterogeneity of big.LITTLE architecture on mobile CPU, and thus limit their prediction accuracy.

We conduct an extensive measurement study on commercial mobile platforms and observe that the presence of multiple concurrent DNN inferences leads to substantial increases in latency. Due to the heterogeneous big.LITTLE architecture of mobile CPU, the latency is affected by a complex interplay of factors, including heterogeneous core performance, cache hierarchies, and shared memory bandwidth. Specifically, tasks running on big cores experience more severe slowdown as they issue memory requests more aggressively and are therefore more susceptible to resource contention. Contention intensifies when tasks are running on cores sharing level-2 cache. These effects are further amplified by vendor-specific DVFS and thermal management policies, which introduce additional runtime dynamics and unpredictability. Moreover, different DNN models exhibit varying levels of resilience to resource contention, making accurate prediction even more complicated.

To bridge this gap, we present *nn-Live*, a lightweight latency prediction framework for concurrent DNN inference on mobile devices. Based on the above analysis, in order to achieve accurate latency predictions, *nn-Live* needs to address two major challenges. First, for a given DNN task running on a specific core, how can we quantify the interference contribution from other cores and its own resilience (from both hardware and model perspectives) to such interference? These intricate interference interactions depend on the processing power of cores hosting these tasks and their contention for shared resources. Additionally, system-level strategies including DVFS policies and task scheduling also influence interference behavior.

To tackle this challenge, we observe that the hardware-counter statistics during task execution indicate its computational intensity and resource usage. Thus, we can collect the hardware-counter statistics for the given task and other concurrent tasks. Their element-wise vector product yields an interpretable metric that effectively describes their resource contention intensity and the interference impact of the concurrent workloads on the given task. *nn-Live* involves two stages, offline profiling and online prediction. We identify that the heterogeneous architecture of mobile CPUs requires a per-core analysis. During offline stage, we get the counter profile for each task and every task-core combination for task profiling and hardware profiling. During online deployment, *nn-Live* collects per-core hardware performance counters, which provide low-overhead, online-accessible signals that effectively reflect runtime execution characteristics. It then combines these online counters with

the task's offline counter profile to construct per-core interference fingerprints, which serve as inputs to a lightweight linear model. This model estimates each core's interference contribution and, together with live system-state features (processor frequency, temperature), predicts the real-time inference latency of DNN models.

The second challenge is effective sampling in the large combination space. Given the large pool of DNN tasks, the potential number of concurrent DNN tasks, and heterogeneous core clusters in mobile CPUs, the possible task-core combination space is extremely large. We need to keep profiling overhead manageable while maintaining diversity and representativeness. To reduce profiling cost, *nn-Live* adopts an adaptive hierarchical sampling strategy. We cluster DNN tasks based on their runtime features and treat tasks within the same cluster as homogeneous. Similarly, CPU cores sharing the same cache hierarchy are treated as homogeneous. We then perform hierarchical sampling across these task and core clusters to efficiently cover the large combination space with minimal profiling overhead.

To evaluate the effectiveness of *nn-Live*, we validate our approach on three mainstream mobile devices powered by heterogeneous SoCs—Google Tensor G2, Snapdragon 8+ Gen 1, and Snapdragon 8 Elite—which exhibit distinct CPU organizations. We evaluate *nn-Live* across diverse execution conditions, including light to heavy concurrent workloads and task execution on big/middle/little cores with all possible combinations of placements. The slowdown levels range from -15% (beneficial interference) to over 230% (intensified contention). Results show that *nn-Live* achieves high latency prediction accuracy under diverse runtime environments: 92.5% and 94.2% ($\pm 5\%$ error) on Snapdragon 8+ Gen1 and 8 Elite, and 82.3% ($\pm 10\%$ error) on Google Tensor G2, with negligible runtime overhead, significantly outperforming approaches based solely on global hardware counters or system-level metrics. Furthermore, while *nn-Live* is designed for heterogeneous mobile CPUs, we demonstrate its extensibility to mobile accelerators (e.g., GPUs/NPUs) in Section 9.

The major contributions of this paper are as follows:

- (1) We propose *nn-Live*, a framework for predicting DNN inference latency in real time on mobile CPUs under volatile operating conditions and concurrent workloads, and demonstrate its extensibility to mobile accelerators.
- (2) We leverage core-level hardware counters to analyze asymmetric interference among concurrent workloads and an adaptive hierarchical sampling strategy to reduce profiling cost while preserving configuration diversity and representativeness.
- (3) We build an interference analysis dataset across three mainstream SoCs, covering diverse task types, task

placements, per-core PMU measurements and system-level trace files. Results show that *nn-Live* achieves high latency prediction accuracy under diverse runtime environments, with a clear advantage over existing methods.

2 Measurement and Analysis

In this section, we analyze the factors that influence a task’s runtime latency, including co-running workloads, system-level dynamics, and the task’s own resilience to interference. To quantify interference, we measure the runtime slowdown of a task, defined as:

$$\text{Slowdown} = \frac{L_d - L_{\text{ref}}}{L_{\text{ref}}} \times 100\%,$$

where L_d denotes the task latency under dynamic conditions, and L_{ref} is the reference latency measured in an interference-free environment (i.e., no co-running workloads). A larger slowdown indicates a stronger impact from interference.

We select ten representative deep neural network models² for measurement. For each experiment, we designate one model as the **main task** and denote other concurrent workloads as **background tasks/workloads**. We measure the runtime slowdown for each main task under various background configurations. Our background configurations cover different workload intensities (1–5 background workloads), different combinations of models, and all possible core placements. For each configuration, we randomly select 1 to 5 models as background workloads and pin each background workload to a dedicated CPU core. After the background environment stabilizes, we launch the main task and iterate it across the available CPU cores (without hosting background tasks) to measure its latency slowdown at each core. We repeat this procedure for all models under the same background environment.

All tasks (both main and background tasks) are integrated into an Android application running in the foreground. To ensure representativeness, experiments are conducted on three mobile devices with heterogeneous processor architecture (Table 1). We aggregate the results by configuration type and report the average and 99th percentile (P99) slowdown for the main task.

2.1 Observations

We examine three categories of factors that influence performance: co-running workloads, the task’s inherent resilience to interference, and system-level dynamics.

²Non-DNN background tasks (e.g., UI rendering and camera activities) induce negligible latency variance. Thus, our evaluation focuses on concurrent DNN execution. The tests include ShuffleNetV2, the MobileNet family, EfficientNet-Lite models, SqueezeNet, the ResNet family, and InceptionV3.

Table 1: Specifications of the three mobile SoCs.

SoC / Device	CPU Cluster Configuration
Google Tensor G2 (Pixel Tablet)	Big (cores 6–7): 2× Cortex-X1 @ 2.85 GHz Mid (cores 4–5): 2× Cortex-A76 @ 2.35 GHz Little (cores 0–3): 4× Cortex-A55 @ 1.80 GHz Cache hierarchy: Not publicly disclosed (L1: private per core, L2/L3: unknown)
Snapdragon 8+ Gen 1 (Xiaomi Pad 6 Pro)	Big (core 7): 1× Cortex-X2 @ 3.0 GHz Mid (cores 4–6): 3× Cortex-A710 @ 2.5 GHz Little (cores 0–3): 4× Cortex-A510 @ 1.8 GHz Cache hierarchy: L1: private per core L2: shared between cores 0–1, cores 2–3 L3: shared among all cores (0–7)
Snapdragon 8 Elite (OnePlus Ace 5 Pro)	Big (cores 6–7): 2× Oryon super cores @ 4.32 GHz Little (cores 0–5): 6× Oryon performance cores @ 3.53 GHz Cache hierarchy: L1: private per core L2: shared between cores 0–5, cores 6–7

2.1.1 Co-running workloads. Fig. 1 (a)–(c) illustrate how the latency slowdown of main task changes as the number of background workloads grows from one to five across the three mobile SoCs. Fig. 2 (a)–(c) illustrate the latency slowdown of main tasks on different types of CPU cores under varying background load placements. We summarize two key empirical observations drawn from these experiments.

Observation 1: Background workloads introduce significant slowdown, and the impact intensifies as the workload increases. Across all three SoCs, introducing background workloads leads to noticeable degradation in DNN inference latency, and the slowdown generally becomes more severe as more concurrent workloads are added. On Google Tensor G2, this effect is the most pronounced: the average slowdown on big cores increases from 65.2% (1 background workload) to 184.4% (5 background workloads), while the P99 slowdown rises from 131.5% to 227.9%. On Snapdragon 8+ Gen1, the little cores are most sensitive to interference. Their average slowdown increases from 8.7% to 57.6%, and the P99 slowdown consistently remains above 72%. On Snapdragon 8 Elite, overall slowdown is relatively mild: the average slowdown on big cores increases only from 1.5% to 11.5%. However, the P99 slowdown still shows substantial increase, reaching 28.7% with five concurrent workloads.

Among the three platforms, Tensor G2 exhibits the most pronounced slowdown. On the one hand, this is related to its limited cache capacity. Tensor G2 is equipped with a 4 MB LLC, compared with a 6 MB LLC in Snapdragon 8+ Gen 1 and two 12 MB LLCs (24 MB in total) in Snapdragon 8 Elite. The smaller LLC limits the amount of frequently accessed data that can be retained in the cache, thereby increasing memory access overhead. On the other hand, system-level tracing reveals that under high CPU load caused by concurrent DNN execution, a subset of DNN executions on Tensor G2 experiences a substantial reduction in CPU cycle rate, indicating

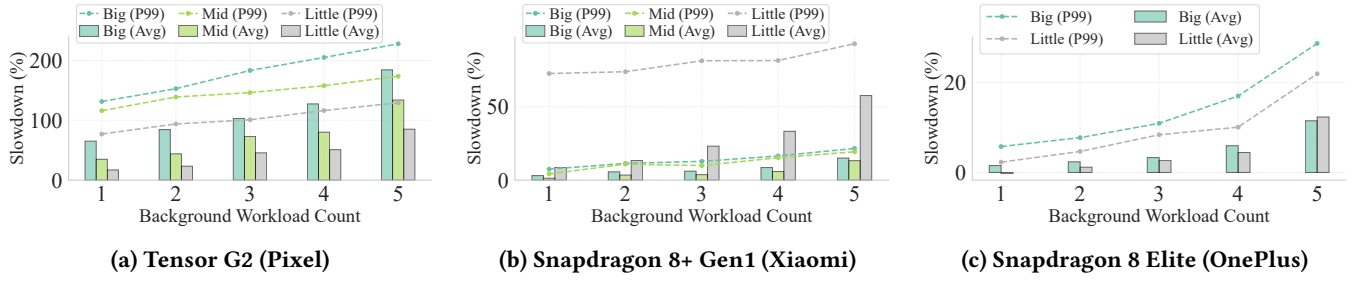


Figure 1: Slowdown under increasing background load conditions (1–5 workloads) on three mobile SoCs.

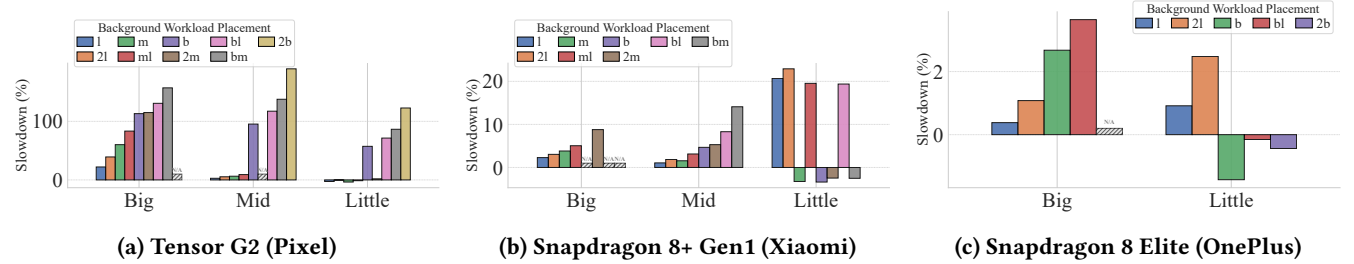


Figure 2: The effect of task placements on inference latency. We show the results for main tasks on big/middle/little cores and all possible task-core combinations for background workloads (1–2 workloads).

transient degradation in CPU execution performance. Such degradation leads to markedly increased inference latency for the affected executions. As shown in Figure 10a in the Appendix, when MobileNet is co-located with a background DNN workload on the big-core cluster, approximately 30% of the executions experience a latency shift from the normal range of approximately 5 ms to approximately 20 ms. Figure 10b further summarizes this behavior under different numbers of background workloads. As the number of concurrent workloads increases, the proportion of affected executions gradually increases. In addition, system-level tracing shows that *kworke* threads, which handle asynchronous kernel tasks, are frequently awakened on Tensor G2 during concurrent DNN execution and compete with user-space inference tasks for CPU time, further amplifying the observed latency fluctuations. We classify these effects as a form of co-running interference.

Observation 2: Interference exhibits significant asymmetry across CPU cores. The interference between the main task and background workloads is influenced by multiple factors. First, task slowdown is not uniform across cores. In general, tasks running on big cores experience more severe slowdown than those on mid and little cores. This effect is particularly evident on Tensor G2 (Fig. 1(a)), which is due to differences in computational power among cores—big cores are more powerful and issue resource requests more frequently when running tasks, making them more susceptible to resource contention.

Meanwhile, we observe an exception that on Snapdragon 8+ Gen1 (Fig. 1(b)), tasks running on little cores experience the most severe slowdown. Further analysis indicates that this is related to the asymmetric cache design (Table 1). Unlike the mid and big cores, which have private L2 caches, the little cores on Snapdragon 8+ Gen1 are organized in pairs and share a small L2 cache. When background workloads run on the sibling core within the same little-core pair, the main task experiences not only L3 contention but also strong L2 contention, resulting in more severe performance degradation.

In Fig. 2, we further illustrate how task placements affect inference latency. We show the results for main tasks on big/middle/little cores and all possible task-core combinations for background workloads (1–2 background workloads in this figure). A notable phenomenon on Tensor G2 is that tasks running on mid and little cores exhibit significant latency slowdown only when background workloads are present on the big cores; in other cases, the interference is relatively minor. On Snapdragon 8+ Gen1 and Snapdragon 8 Elite, tasks running on little cores are noticeably affected only when background workloads are also located on little cores. In other scenarios, tasks actually exhibit slight acceleration, indicating a positive interference effect.

Regarding the positive interference phenomenon, our analysis suggests that it may be related to memory bandwidth activation triggered by background workloads on the big/middle cores. We use a memory bandwidth measurement tool [27]

Table 2: Model characteristics and measured slowdown.

Model	GMACs	Weight (MB)	Slowdown(%)
ShuffleNetV2	0.0438	5.20	19.5027
MobileNetV3Small	0.0582	9.75	18.4479
MobileNetV2	0.3009	13.51	14.1519
SqueezeNet	0.3582	4.71	10.9725
EfficientNetLite0	0.3853	17.91	14.8614
EfficientNetLite2	0.5767	23.47	11.7192
EfficientNetLite4	1.3481	50.05	15.1776
ResNet18	1.8154	44.63	18.9921
InceptionV3	2.8440	90.99	12.8070
ResNet50	3.8670	97.80	13.6815

to evaluate changes in memory bandwidth achieved on little cores under different background workload placements. Figure 11 in the Appendix presents the results measured on the Snapdragon 8+ Gen 1. The memory bandwidth achieved on the little cores increases when the big or middle cores are also active, but decreases when additional workloads are placed on the little cores. This trend is consistent with the positive-interference results shown in Figure 2b.

In summary, when main tasks and background workloads run on different cores, the interference relationship varies. Heterogeneity in computational power, cache sharing structure design, and changes in memory bandwidth are key factors influencing the interference among concurrent workloads. These findings underscore the importance of identifying the hosting cores of both main tasks and background workloads for accurate interference analysis and latency prediction.

2.1.2 Interference Resilience of Different Models. Table 2 reports the average latency slowdown of various models running on the big cores of the Snapdragon 8+ Gen1, under 5 co-running workloads. The results reveal substantial variation in performance degradation across models when exposed to co-running interference, highlighting clear differences in their interference resilience. Notably, this resilience does not follow a simple linear trend with respect to parameter size or model complexity. Among lightweight architectures, we observe that *ShuffleNetV2* and *MobileNetV3-Small*, as well as the mid-sized *ResNet18*, exhibit relatively large slowdowns under background loads, indicating weaker robustness. In contrast, *SqueezeNet* and *EfficientNet-Lite2* experience the smallest latency degradation, demonstrating more stable execution behavior under interference.

These findings suggest that the interference resilience of neural network models is not solely determined by computational complexity or parameter scale. More informative characteristics are required to effectively quantify and predict a model’s robustness under resource contention.

2.1.3 System-level interference. Through system-level trace monitoring, we identify that frequency scaling is the primary source of system-level interference. Frequency regulation in mobile devices mainly relies on Dynamic Voltage and Frequency Scaling (DVFS) and thermal throttling strategies. DVFS [15, 21] dynamically adjusts the frequency based on core workload and power budget, while thermal throttling reduces frequency proactively when the chip temperature exceeds a pre-defined threshold to prevent overheating. Based on our observations, the runtime latency of DNN tasks on the CPU is approximately inversely proportional to the core frequency, which can be described by a simple formula:

$$L_{\text{curr}} = \frac{f_{\text{ref}}}{f_{\text{curr}}} \cdot L_{\text{ref}} + \varepsilon, \quad (1)$$

where L_{curr} is the current latency, L_{ref} is the reference latency, f_{ref} is the core frequency corresponding to the reference latency and f_{curr} is the current core frequency. ε is a disturbance term that arises from execution phases that are weakly correlated with the core frequency. Thus, to predict the inference latency, we should be able to estimate the processor frequency when the task is running.

2.2 Opportunities and Examples

Based on previous observations, interference patterns on mobile platforms are jointly determined by core heterogeneity, cache architecture, cluster topology, and system scheduling policies. Traditional methods usually leverage system-level metrics, such as CPU utilization, last-level cache (LLC) miss rate to reflect system dynamics, which fail to characterize core heterogeneity and complicated interactions across cores. We observe that Linux kernel provides not only aggregated global hardware-counter statistics, but also detailed per-core hardware performance counters (PMUs). It provides us with the opportunities to characterize tasks’ runtime behavior and capture interference from different cores.

On one hand, the PMU statistics of a running task reveal how the task interacts with the system’s resources. For example, instruction counts can show how computationally intensive the task is; L1/L2/L3 cache statistics can help identify how well a task interacts with different levels of cache, which directly impacts execution speed; PMU can also collect memory-related event statistics (e.g., loads/stores and cache misses), which can be used to infer memory read/write activity and estimate memory bandwidth utilization. Tasks that frequently contend for these shared resources will experience higher interference, leading to increased latency. On the other hand, the PMU statistics from other cores also manifest their pressure on shared resources. These hardware event statistics serve as low-overhead, online indicators that effectively reflect real-time system dynamics.

Table 3: Hardware Performance Counters

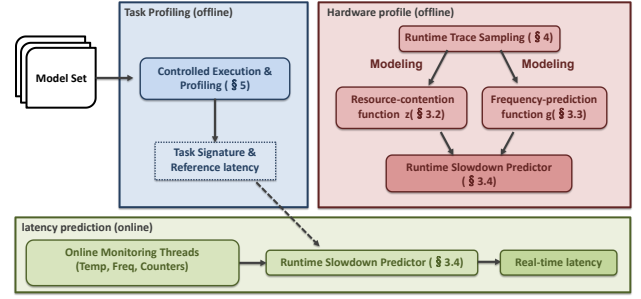
Group	Counter	Description
cpu_exec	cycles	Total core clock cycles.
	instructions	Retired instructions.
	stalled-cycles-frontend	Cycles stalled in the front-end pipeline.
	stalled-cycles-backend	Cycles stalled in the back-end pipeline.
cache	cache-references	Cache hierarchy access count.
	cache-misses	Aggregate cache miss count.
	L1-dcache-loads	L1 data-cache load operations.
	L1-dcache-load-misses	Misses of L1 data-cache loads.
	LLC-loads	Last-level cache (LLC) load accesses.
	LLC-load-misses	Misses of LLC load operations.
branch	branch-loads	Branch prediction or lookup operations.
	branch-load-misses	Missed branch lookups.
	branch-misses	Mispredicted branches.
tlb	dTLB-loads	dTLB translation lookups for data loads.
	dTLB-load-misses	Misses of dTLB translation lookups.

Table 4: Correlation Coefficients for Different Metrics

Group	Pearson (main)	R^2 (per-core)	R^2 (global)
branch-load-misses	0.7563	0.596	0.070
branch-loads	0.7804	0.570	0.143
branch-misses	0.7556	0.571	0.077
cache-misses	0.6340	0.803	0.403
cache-references	0.5992	0.784	0.380
cycles	0.6957	0.882	0.382
dTLB-load-misses	0.6936	0.854	0.318
dTLB-loads	0.6000	0.782	0.379
instructions	0.7523	0.882	0.480
L1-dcache-load-misses	0.6372	0.815	0.409
L1-dcache-loads	0.6000	0.791	0.381
LLC-load-misses	0.5824	0.871	0.501
LLC-loads	0.5692	0.887	0.373
stalled-cycles-backend	0.5568	0.868	0.056
stalled-cycles-frontend	0.7739	0.648	0.016

To validate the relationship between these indicator signals and latency slowdown, we collect different types of hardware counter data and categorize them into four groups based on their semantic meaning, which are shown in Table 3. We also collect the latency slowdown of the 10 DNN models under different concurrency levels (1–2 background tasks) across different devices and task-core combinations.

Table 4 shows the correlation between PMU statistics and inference latency. The first column in Table 4 shows the Pearson correlation coefficient between the main task latency and PMU statistics of the core hosting the main task. We can see that the correlation coefficient is high, mostly above 0.6. It indicates that the PMU statistics of a running task can reveal its interactions with system resources, which is directly related to runtime latency. The second and third columns compare the Coefficient of determination (R^2) between per-core and global PMU. We can see that R^2 coefficients from per-core data are significantly higher than the global data, underlying the importance of per-core analysis.

**Figure 3: *nn-Live* Framework.**

3 *nn-Live* Design

3.1 System Overview

The goal of *nn-Live* is to predict the runtime latency of DNNs in volatile system environments. As shown in Fig. 3, *nn-Live* consists of three stages: *offline task profiling*, *hardware profiling* and *online latency prediction*.

For task profiling, *nn-Live* collects task signatures of DNN models running on different cores, as well as reference latencies L_{ref} , under a specified frequency. These profiles are recorded in a query table for reference. For hardware profiling, *nn-Live* utilizes an adaptive hierarchical sampling technique (Section 4) to automatically generate sampling configurations based on processor structures, cache hierarchies, and model runtime characteristics. The collected data are used to build the resource contention function z (Section 3.2) and the frequency prediction function g (Section 3.3). With these two functions, *nn-Live* builds the runtime latency predictor (Section 3.4). For each device, hardware profiling is required only once.

During the *online latency prediction* stage, *nn-Live* maintains monitoring threads that collect real-time temperature, processor frequency, and hardware counter statistics. The latency predictor utilizes real-time monitoring data along with the task signature to perform the latency slowdown prediction. By combining the predicted slowdown with the reference latency, *nn-Live* outputs the expected per-core inference latency.

3.2 Resource-Contention Function $z(\cdot)$

The goal of $z(\cdot)$ is to capture the resource contention on heterogeneous CPUs. In Section 2, we observed that interference exhibits significant asymmetry across cores, arising from differences in computational capability, cache hierarchy, and shared memory bandwidth. To quantify the contention intensity, we utilize core-level PMU event statistics (Table 3), which characterize both a task's resistance to interference and the interference imposed by background workloads.

We construct task signatures (vector $\mathbf{s}$) from PMU event statistics collected while each task runs on different cores. The task signature $\mathbf{s}^{m,c}$ is extracted offline by analyzing the PMU events of task m running on core c under a reference frequency. It characterizes the task's access patterns to shared resources as well as its runtime execution behavior. Formally,

$$\mathbf{s}^{m,c} = [\chi_{\text{cpu_exec}}^{m,c}, \chi_{\text{cache}}^{m,c}, \chi_{\text{branch}}^{m,c}, \chi_{\text{tlb}}^{m,c}, \dots], \quad (2)$$

where $\chi_r^{m,c}$ represents the resource access pattern of task m on core c for resource r (e.g., CPU, cache). Larger values indicate higher resource demand, making the task more susceptible to contention. Note that the PMU counter value is directly proportional to the CPU frequency. Consequently, the task signature varies under different frequencies. During online prediction, we scale the relevant components of the signature by the factor $f_{\text{curr}}^c / f_{\text{ref}}^c$, where f_{curr}^c is the runtime processor frequency.

For background interference, we collect PMU event statistics for each core $\tilde{c}$, which is hosting background workload, to capture the access patterns of concurrent workloads to shared resources. This yields a background interference fingerprint:

$$\mathbf{b}^{\tilde{c}} = [\tilde{r}_{\text{cpu_exec}}^{\tilde{c}}, \tilde{r}_{\text{cache}}^{\tilde{c}}, \tilde{r}_{\text{branch}}^{\tilde{c}}, \tilde{r}_{\text{tlb}}^{\tilde{c}}, \dots]. \quad (3)$$

The vector $\mathbf{b}^{\tilde{c}}$ has the same dimensions as the task signature, where each component represents the intensity of the core's access to various resources. Higher values indicate more aggressive usage of that resource, corresponding to stronger interference.

Given the task signature and the background interference fingerprint, we next construct a core-aware interference magnitude vector. As both the task signature and background interference fingerprint demonstrate the demand for shared resources, their element-wise product yields their resource contention intensity. By concatenating the element-wise product of the task signature and background interference fingerprint across all background cores, we can effectively capture the competition between the main task and the background workloads. This is expressed as:

$$\boldsymbol{\psi}^{m,c} \triangleq \bigoplus_{\tilde{c} \in C_{\text{bg}}} \mathbf{s}^{m,c} \odot \mathbf{b}^{\tilde{c}}, \quad (4)$$

where $\odot$ denotes element-wise multiplication and $\bigoplus$ denotes concatenation across all background cores $\tilde{c}$. The vector $\boldsymbol{\psi}^{m,c}$ represents the competition across different resource dimensions. It aligns interference penalties with the task's actual demands. High interference in a non-critical resource causes little delay, while the same interference in a critical resource causes major slowdowns.

Finally, we aggregate the contention effects across each resource dimension using a linear combination, which is both lightweight and efficient, making it suitable for real-time

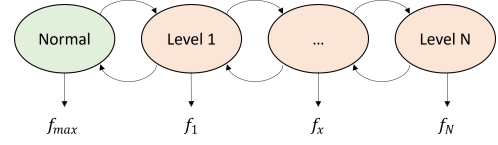


Figure 4: Thermal state transition and CPU frequency.

latency prediction with minimal overhead. Specifically, the resource contention function $z(m, c)$ is given by the following equation:

$$z(m, c) = \boldsymbol{\alpha} \cdot \boldsymbol{\psi}^{m,c}, \quad (5)$$

where the weight vector $\boldsymbol{\alpha}$ is obtained during the offline profiling stage. During the online prediction phase, monitoring threads collect real-time background interference fingerprints, which can reflect the current system resource usage. The resource contention function $z(m, c)$ captures the overall interaction between the task's resource sensitivity and the background interference patterns.

3.3 Frequency Prediction Function $g(\cdot)$

As described in Section 2.1.3, in our analysis, the system-level interference mainly stems from frequency regulation and can be modeled by a simple scaling factor:

$$\Phi^c = \frac{f_{\text{ref}}^c}{f_{\text{exp}}^c}, \quad (6)$$

where f_{ref}^c denotes the reference frequency, which is the frequency during the task profiling phase when measuring the reference latency. f_{exp}^c represents the expected processor frequency when the task is assigned to core c given the current system environment.

We observe that the processor frequency is closely correlated with the DVFS throttling state and exhibits a stage-wise characteristic. As shown in Fig. 4, when the throttling state is normal, the activated core will remain in maximum frequency; but it will reduce to a limited frequency when in throttled states. Fig. 13 in the appendix presents representative long-running traces from three mobile SoCs. Within the same state, the frequency of the activated cores remains relatively stable. This piecewise-stable behavior allows the state to be inferred in a relatively static manner, without tracking short-term variations in temperature or workload³.

Based on the above observations, we employ a threshold-based function $g(\cdot)$ to infer the throttling state:

$$\text{state} = g(f_{\text{curr}}, T_{\text{chip}}, \text{Cycles}), \quad (7)$$

where f_{curr} denotes the frequency of the currently activated cores, T_{chip} denotes the temperature reported by the sensor

³Android provides APIs for estimating thermal headroom on supported devices. However, these APIs provide only device-level thermal information and do not expose per-core or per-cluster frequency transitions. Therefore, they cannot be used to predict the expected frequency of an activated core.

involved in the throttling decision, and $Cycles$ is the number of CPU cycles observed within the current window, serving as an indicator of the processor's workload intensity. Then, the expected core frequency is determined based on the inferred state. It is worth noting that the frequency may fluctuate between adjacent levels within certain stages, and multiple short-term unstable levels may occur between two stable throttling levels; in such cases, we use the average frequency across these transient levels as the representative expected frequency.

3.4 Latency Predictor

Based on the resource-contention function $z(m, c)$ (Section 3.2) and frequency interference factor Φ^c (Section 3.3), we now build the latency predictor:

$$L_d^{m,c} = L_{ref}^{m,c} \cdot \Phi^c(g) \cdot \exp(z(m, c)), \quad (8)$$

where $L_{ref}^{m,c}$ is the reference latency in a contention-free environment (*i.e.*, no concurrent workloads), available from task profile. Φ^c captures the slowdown directly induced by frequency scaling, and $\exp(z)$ models the slowdown caused by resource contention. The exponential form allows for a log-transformation to convert the non-linear multiplicative noise into additive noise, improving stability and accuracy of linear regression. It also guarantees a smooth and strictly positive slowdown factor, consistent with the behavior observed on mobile SoCs.

Note that when the system frequency decreases, the intensity of resource contention also tends to decrease. At this point, the components of the task signature in Eq. (2) and the background interference fingerprint in Eq. (3) decrease accordingly, leading to a smaller z . This allows z to capture the reduction in contention intensity induced by frequency scaling. Eq. (8) integrates the two orthogonal interference mechanisms into a unified predictor for runtime latency. It provides an analytical and interpretable latency prediction model that combines the reference latency with frequency scaling and task-interference interactions.

4 Adaptive Hierarchical Sampling

To construct the latency predictor described in Section 3.4, during the offline profiling stage, *nn-Live* requires runtime samples that capture representative interference patterns on heterogeneous mobile CPUs. Enumerating all possible coexistence scenarios leads to a combinatorial explosion in the sampling space. This is driven by the vast diversity of DNN tasks, varying concurrency levels, and complex placement options across heterogeneous cores (big/middle/little cores, L3-only cache contention or L2+L3 cache contention). Fortunately, unlike learning-based predictors that rely on large-scale training datasets, the linear model in *nn-Live* only

needs a compact yet representative dataset. Therefore, the primary objective of the sampling procedure is to obtain a small but expressive dataset that sufficiently covers the key interference behaviors observed in real systems. In *nn-Live*, we perform model clustering and core aggregation to reduce the sampling space.

Model Clustering. To ensure the sampling dataset captures the diverse runtime behaviors of DNN models, we select 24 representative models, covering mainstream convolutional networks, depthwise separable networks, and Vision Transformer (ViT) variants. Table 9 (in Appendix) summarizes their computational complexity (Giga Multiply-Accumulate Operations, GMACs) and model size.

We run each model under contention-free conditions to obtain its task-signature vectors on different core types. These vectors characterize the model's runtime behavior and resource-access patterns across heterogeneous cores, forming the feature space for model clustering. To obtain a well-conditioned embedding and suppress noise in the high-dimensional PMU space, we apply principal component analysis (PCA) to project all task-signature vectors into a low-dimensional subspace. K-means clustering is then performed in this PCA-transformed space. Since vanilla K-means tends to produce pathological clusters containing only a single model, we adopt a size-constrained K-means clustering algorithm that enforces constraints on the cluster size from $n/8$ to $n/3$, where n is the total number of models. We sweep the number of clusters from 3 to 8 and compute the silhouette coefficient for each configuration, selecting the one that maximizes the silhouette score as the final clustering choice. The resulting clusters define homogeneous model groups $\{\mathcal{M}_j\}$, whose members exhibit highly similar resource sensitivities and runtime characteristics. For example, ShuffleNetV2, MobileNetV3Small, and NASNetMobile are grouped into the same cluster. These models exhibit highly similar runtime behaviors and resource-access patterns, representing lightweight architectures optimized for mobile and resource-constrained environments.

Core Aggregation. In heterogeneous mobile SoCs, processor cores that belong to the same core type (big/mid/little) and share the same cache hierarchy typically exhibit highly homogeneous contention patterns in their PMU feature space. This observation allows us to aggregate processor cores. We dynamically construct homogeneous core groups $\{C_k\}$ based on the actual contention behavior of the task. For example, on the Snapdragon 8+ Gen 1 SoC, when task m runs on a **mid core**, interference originates only from the shared L3 level. Therefore, the homogeneous core groups naturally correspond to the CPU topology: $C_k = \{[7], [6, 5, 4], [3, 2, 1, 0]\}$. When task m runs on a **little core** (e.g., CPU 0), interference comes from both the shared L3 cache and the private L2 cache shared with its sibling core (CPU 1). As a result, the

groups become: $C_k = \{ [7], [6, 5, 4], [3, 2], [1, 0] \}$. Based on the homogeneous groups, the interference vector $\psi^{m,c}$ in Eq. (4) turns out to be:

$$\tilde{\psi}^{m,c} = \bigoplus_{k=1}^K \left(\sum_{\tilde{c} \in C_k} s^{m,c} \odot b^{\tilde{c}} \right). \quad (9)$$

In the above example, assume that there are 3 concurrent tasks. According to the core grouping, we can reduce the possible combinations from $P_3^8 = 336$ to 8. With model clustering and core aggregation, hierarchical sampling generates a representative profiling dataset for *nn-Live*. This dataset is used in the offline stage to build the resource contention function and the latency predictor.

5 Implementation

The implementation of *nn-Live* is based on the Android platform, utilizing Android Studio Koala (2024.1.1 Patch 1) as the development environment, with Gradle version 7.6 and Java compatibility set to version 1.8. TensorFlow (v2.16.1) is used as the deep learning framework for model inference. We enable liteRT's debug tracing mode on the mobile device and use Perfetto [13] and Perf [1] to collect model latency and hardware counter statistics.

In the offline task profiling phase, DNN tasks are executed on different cores to collect task signatures and reference latencies. This information is then recorded in a lookup table, which will be used in the online prediction phase for runtime latency estimation. To build the resource contention function $z(\cdot)$, the processor frequencies are set to the maximum frequency for Google Tensor G2 and Snapdragon 8+ Gen 1. On Snapdragon 8 Elite, its maximum frequency is as high as 4.32 GHz for big cores. When running at its maximum frequency, the device easily overheats and triggers thermal regulation. Thus, we set its processor frequency to 3.072 GHz and 2.7456 GHz for the big and little cores, respectively. During data collection, we sample 4000 data points per device. The configurations span all combinations of core assignments for each device, considering processor cluster topologies and cache structures. To construct the frequency prediction function g , we collected data using the device's default Dynamic Voltage and Frequency Scaling (DVFS) policy. Multiple sets of high-stress tests (with 1 to 5 DNN workload threads activated) were conducted, each lasting 5 minutes. During testing, a C++ thread continuously polls temperature sensor values located under the `/sys/class/thermal/` directory, while CPU cycle counts were recorded using the perf tool.

In the online phase, *nn-Live* runs two background monitoring threads. The first thread monitors PMU counters on a per-core basis, polling every 0.5 seconds and recording the data. The second thread monitors system temperature and frequency, polling relevant system files. When latency

prediction is required, *nn-Live* reads the dynamic environmental data and uses this data, along with the task signature, to predict the runtime latency for the model on all available cores. It also reports the interference contribution from each core.

6 Experiments

6.1 Experimental Setup

To comprehensively evaluate the prediction performance of *nn-Live*, we design three testing protocols and collect testing data for each protocol:

Protocol A (In-Distribution Testing): We collect runtime data under the same sampling strategy as the profiling stage to evaluate the prediction accuracy within the same distribution.

Protocol B (Model-Held-Out Testing): We select six DNN models from the model pool (MobileNetV3 Small [16, 18], EfficientNet Lite 0 [2, 24, 28, 31], EfficientNet Lite 4, InceptionV3 [8, 20, 32, 33], ResNet34 [4, 12, 19], EfficientViT B2 [6, 7, 17]), exclude them from the hardware profiling stage and collect their runtime latencies as testing samples to test the generalization ability of *nn-Live* on unseen tasks.

Protocol C (NAS-Variant Testing): Based on four common model families (MobileNetV3 Small, MobileNetV3 Large, EfficientNet Lite 0, ResNet34), we perform neural architecture search (NAS) and select five variants at different FLOPs percentiles (5%, 25%, 50%, 75%, 95%). This protocol is intended to demonstrate that *nn-Live* can be integrated with NAS to select appropriate model variants according to the dynamic runtime environments and latency requirements.

Baselines. We implemented four baselines for comparison: (1) Static; (2) CPU-Util; (3) CPU-Util + MEMINFO; (4) PMU-Sys data. Baseline 1 is a static analysis baseline that does not consider dynamic environment changes during prediction, serving as an upper bound on prediction performance when dynamic information is unavailable; this baseline is implemented via a lookup table (LUT) scheme. Baselines 2 and 3 rely on traditional system resource usage metrics to reflect device load [10, 29, 30]. Baseline 2 utilizes CPU utilization while baseline 3 combines CPU utilization with MEMINFO. Baseline 4 leverages global hardware performance counters to capture underlying execution characteristics [23, 37]. Specifically, baselines 2-4 all employ linear regression models for training and prediction.

Evaluation Metrics. We report the mean absolute percentage error (MAPE) and Root Mean Square Error (RMSE) of latency prediction. We also report the 99th percentile error (p99 error), providing insight into the worst-case performance of *nn-Live*. Besides, we report the $\pm 5\%$ and $\pm 10\%$ accuracy, which is the percentage of models with predicted latency within the corresponding error bounds.

Table 5: Detailed performance comparison across three testing protocols on all devices.

method	Tensor G2			Snapdragon 8+ Gen1			Snapdragon 8 Elite		
	RMSE(ms)	MAPE(%)	P99(ms)	RMSE(ms)	MAPE(%)	P99(ms)	RMSE(ms)	MAPE(%)	P99(ms)
Protocol A (In-Distribution)									
Static	204.0	34.2	777.2	77.4	9.0	403.9	5.3	3.5	22.4
CPU-Util	116.7	25.5	452.0	65.2	8.7	336.8	3.1	2.6	12.3
CPU-Util+MEM	116.7	25.7	449.4	64.8	8.7	337.2	3.1	2.6	11.4
PMU-Sys	71.5	11.0	300.8	65.7	8.8	340.6	3.0	2.6	12.1
ours	44.7	6.1	183.8	10.2	2.1	43.5	2.1	1.9	8.2
Protocol B (Model-Held-Out)									
Static	232.9	34.0	843.0	91.5	8.9	451.7	6.8	3.5	34.8
CPU-Util	125.9	24.7	442.5	78.5	8.6	369.1	4.0	2.7	18.9
CPU-Util+MEM	126.5	24.9	444.1	78.1	8.6	365.0	3.9	2.6	17.3
PMU-Sys	76.4	10.7	323.4	79.0	8.5	368.7	3.8	2.6	18.2
ours	47.1	6.5	181.5	11.8	2.1	53.3	2.7	1.9	11.9
Protocol C (NAS-Variant)									
Static	161.0	33.0	710.4	59.0	8.5	325.5	3.2	3.4	12.7
CPU-Util	90.2	26.5	349.4	49.0	8.3	278.1	2.0	2.9	7.8
CPU-Util+MEM	91.3	26.6	361.5	47.5	9.1	259.7	2.0	3.1	8.7
PMU-Sys	70.9	11.2	339.3	50.0	8.0	271.1	1.8	2.8	6.4
ours	48.2	6.6	217.9	6.2	2.2	25.4	1.4	1.9	5.1

6.2 Latency Prediction Results

6.2.1 Comparison with Baselines. Table 5 reports the detailed results of our approach and the baseline methods under the three testing protocols, each consisting of 1000 test samples. The results demonstrate that, compared with all baselines, *nn-Live* consistently achieves the best performance across all testing protocols and devices, highlighting the effectiveness of our design. Specifically, across the three devices and protocols, *nn-Live* attains an average MAPE of 3.48%, significantly outperforming PMU-Sys (7.35%), CPU-Util+MEM (12.43%), CPU-Util (12.28%), and the Static scheme (15.34%). *nn-Live* also substantially reduces P99 prediction error.

Compared with Tensor G2, the Snapdragon 8+ Gen1 and Snapdragon 8 Elite exhibit lower prediction errors. This is because these devices deliver higher performance and exhibit more stable interference patterns. In contrast, Tensor G2 experiences considerable latency fluctuation due to transient reductions in CPU cycle rate for a subset of executions, with additional interference from *kworke*r threads.

For the baseline methods, the Static scheme represents the upper bound of prediction performance for approaches that do not consider dynamic environments. Due to runtime variability, it exhibits very high RMSE and P99 errors. Baseline approaches based on system-level metrics partially reduce RMSE and P99 error; however, as shown in Section 2.1.1, heterogeneous-core systems exhibit entirely different interference behaviors depending on the specific task placement. System-level metrics cannot distinguish these scenarios, leading to prediction bias. Fig. 5a shows the fitting results of the system-level PMU-based approach and our method for $z(\cdot)$ on the test data from Protocol A. The results demonstrate that our method achieves a good fit to the data, whereas the system-level PMU-based approach fails to effectively identify the interference patterns on heterogeneous cores. Fig. 5b illustrates the intermediate results of our method for execution samples collected at six different frequencies. $\Phi^c(g)$

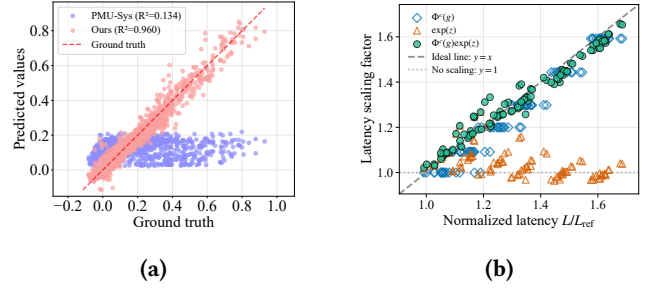


Figure 5: (a) PMU-Sys vs. *nn-Live*. (b) Intermediate components of *nn-Live*.

provides discrete estimates of frequency scaling, whereas $\exp(z)$ models the impact of resource contention. Ideally, $\Phi^c(g) \exp(z) = L/L_{ref}$. As the CPU frequency decreases, the overall execution latency increases, accompanied by an increase in $\Phi^c(g)$ and a decreasing trend in $\exp(z)$.

In Fig. 6, we further present the $\pm 5\%$ prediction accuracy of our method and the baselines. Since latency fluctuations on Tensor G2 are considerably larger, its $\pm 5\%$ accuracy is relatively low (only 50.6%), and thus we report $\pm 10\%$ accuracy. The final results show that our method achieves high prediction accuracy under dynamic interference: 92.5% and 94.2% ($\pm 5\%$ error) on Snapdragon 8+ Gen1 and Snapdragon 8 Elite, and 82.3% ($\pm 10\%$ error) on Tensor G2.

6.2.2 Error Analysis. In this section, we provide a detailed analysis of the prediction errors observed across different devices and examine their underlying causes. We divide the evaluation data of Protocol A into five groups according to the number of background threads and report their prediction errors. As shown in Fig. 7, the Tensor G2 exhibits not only larger prediction errors but also noticeably greater variability compared with Snapdragon 8+ Gen1 and Snapdragon 8 Elite. This is mainly because Tensor G2 experiences stronger interference during concurrent DNN execution, leading to more unstable latency fluctuations that are inherently hard to predict.

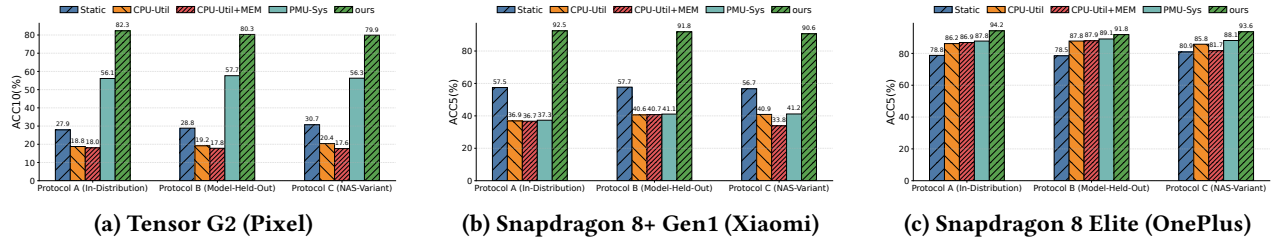


Figure 6: Compared to baseline predictors, *nn-Live* achieves much higher accuracy on all three test protocols.

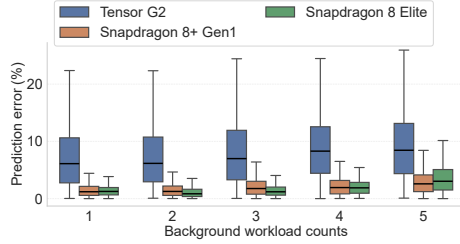


Figure 7: Error analysis.

Across all three SoCs, we also observe a consistent trend: the prediction error and its variation increase as the number of background threads rises. Increasing number of background threads causes the system state to become more unstable and amplifies the slowdown of DNN latency, which in turn leads the resource-contention model to underestimate latency under heavy-load conditions. Fortunately, such scenarios are uncommon in practical mobile deployments, where multiple compute-intensive DNN workloads rarely run concurrently; therefore, their impact on real-world applicability is minimal.

6.3 Microbenchmark Testing

6.3.1 Ablation Study on Core-aware Inference Vector. To validate its effectiveness, we conduct two groups of ablation experiments. In the first group, no feature fusion is applied, and the task signature and background interference fingerprints are used independently as features. In the second group, system-level PMU counter information is fused with the task signature. We test these methods on Protocol A running on the Snapdragon 8+ Gen1 platform. The results are shown in Table 6, where we report the performance with tasks running on big, mid, and little cores, separately. Our proposed method achieves the best performance. Both the no-fusion baseline and the fusion with system-level PMU counters result in degraded performance, especially on the little cores with asymmetric cache architecture. When using system-level PMU counter fusion, the prediction accuracy within $\pm 5\%$ on little cores is only 8.3%. With *nn-Live*, it improves to 91.9%.

Table 6: Ablation Study on Interference Vector

Method	Big core		Mid core		Little core	
	RMSE (ms) ↓	ACC5 (%) ↑	RMSE (ms) ↓	ACC5 (%) ↑	RMSE (ms) ↓	ACC5 (%) ↑
PMU-Sys Fusion	4.5	65.7	8.0	64.9	97.4	8.3
No-Fusion	2.9	88.8	7.4	79.8	34.6	63.9
nn-Live	2.1	94.1	3.8	92.1	15.2	91.9

Table 7: Random vs. hierarchical sampling

Method	RMSE (ms) ↓	MAPE (%) ↓	ACC10 (%) ↑	P99 (ms) ↓
Protocol A (In-Distribution)				
Random Sampling	53.0	8.4	67.7	243.1
Hierarchical Sampling	50.7	7.0	78.2	212.5
Protocol B (Model-Held-Out)				
Random Sampling	57.9	8.3	68.2	245.3
Hierarchical Sampling	47.9	6.7	79.4	197.6
Protocol C (NAS-Variant)				
Random Sampling	48.5	8.0	71.6	270.6
Hierarchical Sampling	52.1	7.3	76.0	259.2

6.3.2 Hierarchical Sampling. To evaluate the effectiveness of hierarchical sampling, we compare it with random sampling. Each method collects 500 samples, and the results are reported under Protocols A, B, and C. We particularly present the results on the more challenging Tensor G2 dataset. As shown in Table 7, our method consistently outperforms random sampling across all three protocols, achieving higher prediction accuracy and lower tail latency prediction error.

We analyzed the samples collected by random sampling and found biases in data samples. As there are usually more little cores (*i.e.*, 4) compared to middle (*i.e.*, 2) and big cores (*i.e.*, 2) on the SoC, random sampling tends to collect more samples with background workloads assigned to little cores. This not only wastes sampling budget but also causes the resource contention function z to over-assign weights to little cores. In contrast, our hierarchical sampling performs proportional sampling according to the number of background threads and task-core combinations for each thread count. This approach ensures that the resource contention function $z(\cdot)$ fairly assigns the weights to each core.

6.4 Cost Analysis

6.4.1 Online prediction phase. *nn-Live* features a lightweight design with extremely low overhead during the online phase.

The resource contention function $z(\cdot)$ employs a linear model that aggregates cores based on processor topology and cache structures. For each group in C_k , we collect 15 PMU counter statistics, as shown in Table 3. Thus, there are 45, 60, and 30 features on Tensor G2, Snapdragon 8+ Gen1, and Snapdragon 8 Elite devices, respectively. The forward inference cost is under 200 MAC operations, making the overall computational overhead of *nn-Live* negligible. The two background threads poll PMU counters and OS files every 0.5 seconds. We further evaluate the monitoring overhead on devices using Perfetto to record CPU active time, voltage, and current. When no DNN is running, they increase CPU active time and power consumption by 0.62% and 0.58%, respectively; during DNN execution, both runtime and power overheads remain below 0.1%. *nn-Live* can suspend polling when the system is idle. While CPU-frequency monitoring can be event-driven, PMU and temperature interfaces currently require periodic sampling. A hybrid event-driven and polling-based implementation is a promising direction and is expected to further reduce the monitoring overhead.

6.4.2 Offline profiling phase. The data collection cost during the profiling phase is also lightweight. For task profiling, we run the DNN model on each CPU core, which can be completed in one minute. For hardware profiling, we show the $\pm 5\%$ ($\pm 10\%$) prediction accuracy with different numbers of samples in Fig. 8. We can see that on all three devices, the performance stabilizes with more than 500 samples. Each runtime sample requires approximately 15 seconds to collect. Thus, data sampling can be completed within roughly 3 hours, including the system cooling time between cycles. For each device, hardware profiling only needs to be performed once, and we can see from Table 5 that *nn-Live* generalizes well to unseen models.

7 Case Study

We show how *nn-Live* guides runtime model selection and scheduling on a Snapdragon 8 Elite using ResNet34 and four pruned variants (v0–v4). Given an 85 ms latency budget, the system triggers a prediction whenever the main task arrives or environment shifts occur, selecting the highest-capacity configuration that satisfies the constraint. Fig. 9 illustrates this adaptation across five stages:

Stage 1 (Arrival): With low interference, *nn-Live* predicts 77.7 ms for v0 on big cores, which is selected for maximum accuracy.

Stage 2 (Contention): Background workloads arrive, pushing v0 latency to 94.9 ms (exceeding budget). The system switches to v1 on big cores (75.1 ms).

Stage 3 (DVFS Throttling): Thermal accumulation triggers frequency scaling. *nn-Live* downgrades the model to v3 on big cores to maintain latency at 73.8 ms.

Stage 4 (Preemption): New workloads preempt big cores. The system migrates the lightweight v4 variant to little cores (76.9 ms) to preserve service continuity.

Stage 5 (Recovery): Background tasks finish, restoring big-core availability. *nn-Live* upgrades back to v3 on big cores (72.6 ms), maximizing capacity as resources permit.

As shown in Fig. 9, *nn-Live* effectively navigates interference, DVFS, and core preemption to guarantee latency while maximizing model performance.

8 Related Work

Static latency prediction. Early work on DNN inference latency prediction primarily assumes a static execution environment and relies on offline profiling or static model features. Representative approaches include GCN-based predictors for NAS benchmarks [9] and kernel-level decomposition methods such as *nn-meter* [34]. Although accurate under controlled settings, these methods fail to capture runtime variability caused by dynamic workloads and resource contention in mobile systems.

Dynamic and PMU-based prediction. Recent studies improve performance prediction by incorporating runtime resource metrics or hardware monitoring signals. Some approaches use CPU utilization, memory throughput, and I/O activity to predict execution latency [10, 29], while *Maple* and *Maple-Edge* construct PMU-based hardware and runtime descriptors for DNN latency prediction [3, 25]. PMU statistics have also been used to estimate co-location interference. For example, *Themis* predicts application slowdown on spatially multitasked GPUs [37], and *VELTAIR* estimates interference pressure to support adaptive compilation and scheduling for multi-tenant CPU inference [23]. Other studies model cross-core interference using shared-cache and memory-bandwidth pressure [35, 36], or quantify execution-efficiency loss through online PMU sampling [14]. However, these approaches primarily target homogeneous processors, GPUs, or server-class platforms, and often rely on coarse-grained global statistics, making them insufficient for capturing the architectural heterogeneity of mobile CPUs. In contrast, *nn-Live* targets heterogeneous mobile CPUs and models placement-dependent contention by combining core-level task signatures with interference fingerprints. It further integrates frequency prediction and hierarchical sampling to support online latency prediction under dynamically changing co-runners, core assignments, and operating frequencies.

9 Discussion

Extensibility to hardware accelerators. While *nn-Live* is designed for heterogeneous mobile CPUs, it can be extended

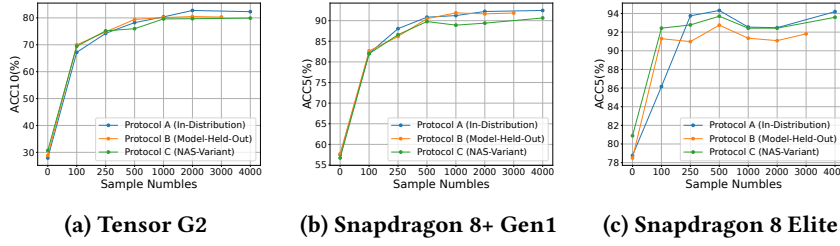


Figure 8: Prediction accuracy vs. sample number

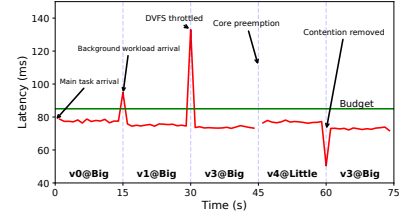


Figure 9: Case study

to accelerators through hardware-specific adaptations. Compared with CPUs, mobile accelerators have limited scheduling capability in multi-DNN concurrent scenarios. When multiple DNNs are executed concurrently, the runtime behavior typically becomes serialized execution, and thus the latency is dominated by queuing delay. Our measurements on GPUs show that, under the same scheduling priority, the observed inference latency of each co-located model satisfies:

$$L^{\text{obs}} \approx \frac{1}{u^{\text{obs}}} \sum_{i=1}^N (L_i^{\text{solo}} \cdot u_i^{\text{solo}}), \quad (10)$$

where L^{obs} and u^{obs} denote the observed latency and GPU utilization during concurrent execution; L_i^{solo} and u_i^{solo} denote the standalone latency and GPU utilization of model i , respectively. Based on this formula, *nn-Live* can be extended to GPUs by collecting the standalone end-to-end latency L^{solo} and average utilization u^{solo} of each model during offline analysis and storing them in a lookup table. During online execution, *nn-Live* predicts latency based on the lookup table entries and the current GPU utilization. Note that this formula-based predictor requires no model training and incurs negligible construction overhead. Fig. 12a (in the Appendix) shows the scenario when multiple MobileNetV3 and ResNet models are executed concurrently, while Table 8 (in the Appendix) reports the quantitative results on three Adreno and Mali GPUs. The predictor achieves MAPE values of 3.21%–7.06%, demonstrating that *nn-Live* can effectively predict multi-DNN latency across different mobile GPU architectures.

Fig. 12b (in the Appendix) shows that concurrent DNN execution on the NPU exhibits the same cumulative latency pattern as GPU execution, as described by Eq. (10), indicating that the latency is also dominated by queuing delays. Therefore, *nn-Live* can be extended to NPUs using the same approach. However, applying *nn-Live* to NPUs remains challenging, as runtime information is not readily accessible on many existing platforms. In addition, DNN models containing dynamic-shape tensors or operators that are unsupported by the accelerator delegate are not suitable for accelerator execution and are therefore kept on the CPU side. Deeper analysis into these hardware accelerators is our future work.

Runtime Adaptation. During offline profiling only, *nn-Live* uses core pinning and controlled task placement to collect reproducible profiling samples for constructing the latency prediction model. Note that *nn-Live* works without core-pinning. During online deployment, task scheduling and migration are dynamically managed by the operating system and runtime framework. To capture the resulting variations in execution conditions, *nn-Live* continuously monitors per-core PMU events, CPU frequency, and temperature. As illustrated in the case study, *nn-Live* can identify runtime workload and execution-state variations from the monitored signals, enabling latency prediction and core-selection decisions under changing system conditions.

10 Conclusion

In this paper, we present *nn-Live*, a latency prediction framework for DNN inference. It considers the resource contention among concurrent tasks and the heterogeneity of mobile CPU. *nn-Live* investigates the underlying factors contributing to inference slowdown and builds a lightweight and interpretable linear model for latency prediction. Evaluation results show that *nn-Live* can achieve accurate latency predictions in dynamic system environments and generalizes well to unseen models. Furthermore, we explore the extensibility of *nn-Live* to mobile accelerators. We believe that in applications where multiple DNNs work concurrently, *nn-Live* can serve as a valuable tool for model selection and task scheduling, ultimately improving service quality and user experience.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China under Grant 62472452; in part by Guangdong Basic and Applied Basic Research Foundation (No. 2025A1515010262). Q. Huang is the corresponding author.

References

- [1] 2025. Linux kernel profiling with perf. <https://perfwiki.github.io/main/tutorial/>. Accessed: 2025-12-06.
- [2] Mohd Nadhir Ab Wahab, Amril Nazir, Anthony Tan Zhen Ren, Mohd Halim Mohd Noor, Muhammad Firdaus Akbar, and Ahmad Sufril Azlan

- Mohamed. 2021. Efficientnet-lite and hybrid CNN-KNN implementation for facial expression recognition on raspberry pi. *IEEE Access* 9 (2021), 134065–134080.
- [3] Saad Abbasi, Alexander Wong, and Mohammad Javad Shafiee. 2022. Maple: Microprocessor a priori for latency estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2747–2756.
 - [4] Noor Mohammed Almoosawi and Raidah Salim Khudeyer. 2021. ResNet-34/DR: a residual convolutional neural network for the diagnosis of diabetic retinopathy. *Informatica* 45, 7 (2021).
 - [5] Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. 2022. Performance modeling of computer vision-based cnn on edge gpus. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 5 (2022), 1–33.
 - [6] Han Cai, Junyan Li, Muyan Hu, Chuang Gan, and Song Han. 2022. Efficientvit: Multi-scale linear attention for high-resolution dense prediction. *arXiv preprint arXiv:2205.14756* (2022).
 - [7] Han Cai, Junyan Li, Muyan Hu, Chuang Gan, and Song Han. 2023. EfficientViT: Lightweight multi-scale attention for on-device semantic segmentation.
 - [8] Na Dong, Li Zhao, Chun-Ho Wu, and Jian-Fang Chang. 2020. Inception v3 based cervical cell classification combined with artificially extracted features. *Applied Soft Computing* 93 (2020), 106311.
 - [9] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. 2020. Brp-nas: Prediction-based nas using gcns. *Advances in neural information processing systems* 33 (2020), 10480–10490.
 - [10] Junquan Fan, Jiahui Hou, and Xiangyang Li. 2023. Accurate Deep Learning Inference Latency Prediction over Dynamic Running Mobile Devices. In *2023 19th International Conference on Mobility, Sensing and Networking (MSN)*. IEEE Computer Society, 293–300.
 - [11] Cheng Fang, Sicong Liu, Zimu Zhou, Bin Guo, Jiaqi Tang, Ke Ma, and Zhiwen Yu. 2024. AdaShadow: Responsive test-time model adaptation in non-stationary mobile environments. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. 295–308.
 - [12] Mingyu Gao, Dawei Qi, Hongbo Mu, and Jianfeng Chen. 2021. A transfer residual neural network based on ResNet-34 for detection of wood knot defects. *Forests* 12, 2 (2021), 212.
 - [13] Google. 2025. Peretto: System profiling, app tracing and trace analysis. <https://peretto.dev/>. Accessed: 2025-12-06.
 - [14] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. 2014. Quality Time: A Simple Online Technique for Quantifying Multicore Execution Efficiency. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 169–179. doi:10.1109/ISPASS.2014.6844481
 - [15] Sebastian Herbert and Diana Marculescu. 2007. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 international symposium on Low power electronics and design*. 38–43.
 - [16] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*. 1314–1324.
 - [17] Derek Jin and Shengyou Zeng. 2024. EfficientViT for Video Action Recognition. In *2024 IEEE International Conference on Big Data (Big-Data)*. IEEE, 7323–7326.
 - [18] Brett Koonce. 2021. MobileNetV3. In *Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization*. Springer, 125–144.
 - [19] Brett Koonce. 2021. ResNet 34. In *Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization*. Springer, 51–61.
 - [20] Chunmian Lin, Lin Li, Wenting Luo, Kelvin CP Wang, and Jiangang Guo. 2019. Transfer learning based traffic sign recognition using inception-v3 model. *Periodica Polytechnica Transportation Engineering* 47, 3 (2019), 242–250.
 - [21] Chengdong Lin, Kun Wang, Zhenjiang Li, and Yu Pu. 2023. A workload-aware DVFS robust to concurrent tasks for mobile devices. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. 1–16.
 - [22] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. 2021. Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In *Proceedings of the 19th ACM conference on embedded networked sensor systems*. 1–14.
 - [23] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 388–401.
 - [24] Marsa Mahasin and Irma Amelia Dewi. 2022. Comparison of csp-darknet53, cspresnext-50, and efficientnet-b0 backbones on yolo v4 as object detector. *International journal of engineering, science and information technology* 2, 3 (2022), 64–72.
 - [25] S Nair, S Abbasi, A Wong, and MJ Shafiee. 2022. Maple-edge: A runtime latency predictor for edge devices. In *2022 IEEE. In CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 3659–3667.
 - [26] Karthick Panner Selvam and Mats Brorsson. 2023. Dippm: a deep learning inference performance predictive model using graph neural networks. In *European Conference on Parallel Processing*. Springer, 3–16.
 - [27] Raas. [n. d.]. mbw: Memory Bandwidth Measurement Tool. <https://github.com/raas/mbw>. Accessed: 2025-12-06.
 - [28] Gopal Sangar and Velswamy Rajasekar. 2025. Optimized classification of potato leaf disease using EfficientNet-LITE and KE-SVM in diverse environments. *Frontiers in Plant Science* 16 (2025), 1499909.
 - [29] Haihong She, Yigui Luo, Zhaohong Xiang, Weiming Liang, and Yin Xie. 2023. Accurate Latency Prediction of Deep Learning Model Inference Under Dynamic Runtime Resource. In *International Conference on Neural Information Processing*. Springer, 495–510.
 - [30] Swetha Srinivasan and Umesh Bellur. 2014. Novel power and completion time models for virtualized environments. *arXiv preprint arXiv:1411.3201* (2014).
 - [31] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.
 - [32] Cheng Wang, Delei Chen, Lin Hao, Xuebo Liu, Yu Zeng, Jianwei Chen, and Guokai Zhang. 2019. Pulmonary image classification based on inception-v3 transfer learning model. *IEEE Access* 7 (2019), 146533–146541.
 - [33] Xiaoling Xia, Cui Xu, and Bing Nan. 2017. Inception-v3 for flower classification. In *2017 2nd international conference on image, vision and computing (ICIVC)*. IEEE, 783–787.
 - [34] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 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*. 81–93.
 - [35] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. 2015. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2015), 1443–1456.
 - [36] Jiacheng Zhao, Xiaobing Feng, Huimin Cui, Youliang Yan, Jingling Xue, and Wensen Yang. 2013. An empirical model for predicting cross-core

performance interference on multicore processors. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 201–212.

- [37] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 653–663.
- [38] Zhihe Zhao, Neiwen Ling, Nan Guan, and Guoliang Xing. 2022. Aaron: Compile-time kernel adaptation for multi-DNN inference acceleration on edge GPU. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*. 802–803.

Appendices

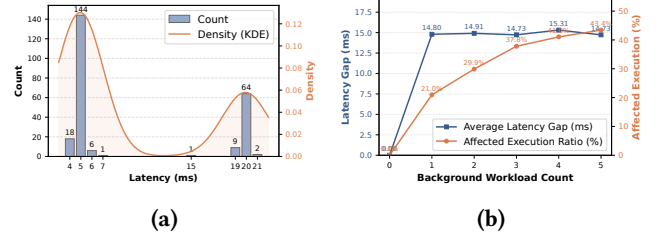


Figure 10: DNN inference latency variation on Tensor G2. (a) Latency distribution of MobileNet with one background DNN workload, using 1-ms bins. (b) Latency gap between affected and unaffected executions, and affected executions (%) under different numbers of background workloads.

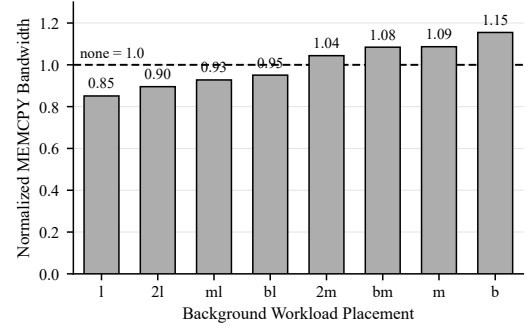


Figure 11: Normalized memory bandwidth achieved on the little cores under different background workload placements on the Snapdragon 8+ Gen 1.

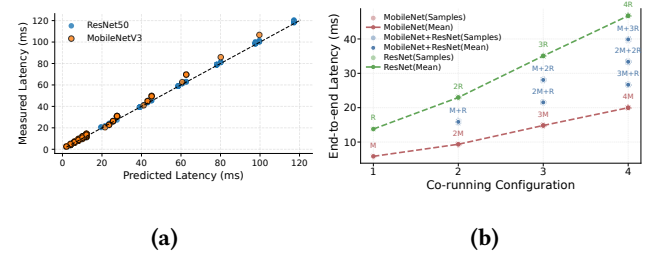
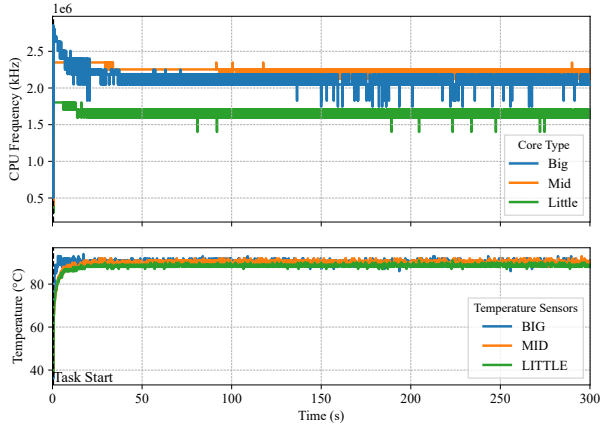


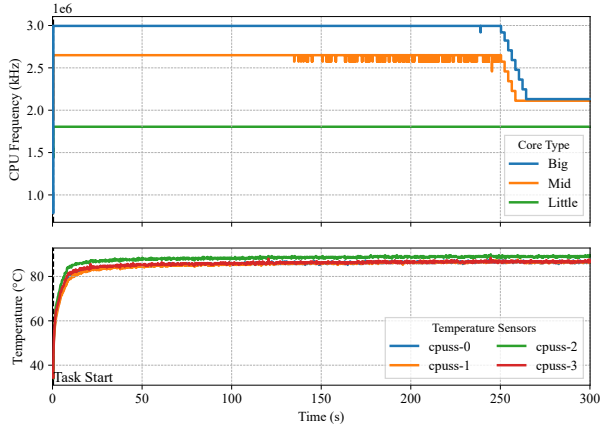
Figure 12: (a) Results on Adreno 830 (b) Results on Qualcomm Hexagon NPU

Table 8: Latency prediction results under concurrent GPU execution.

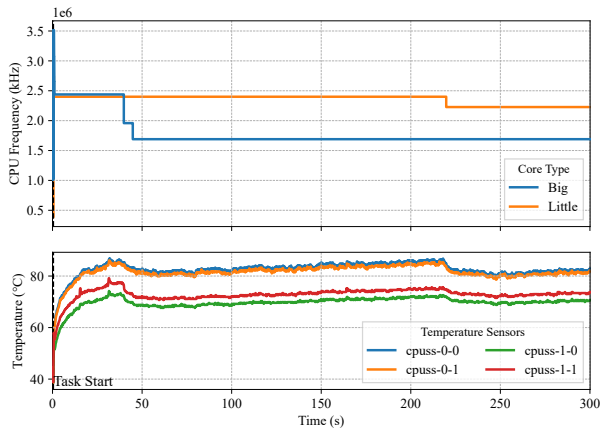
Device	MAE (ms)	MAPE (%)	RMSE (ms)	P99 Error (ms)
Adreno 830	1.26	3.21	2.67	11.46
Adreno 730	2.68	7.06	4.97	17.52
Mali-G710 MP7	71.02	6.49	107.43	288.20



(a) Tensor G2 (Pixel)



(b) Snapdragon 8+ Gen 1 (Xiaomi)



(c) Snapdragon 8 Elite (OnePlus)

Figure 13: CPU-frequency and temperature traces under long-running concurrent DNN execution on different mobile SoCs. CPU frequency is device-specific and evolves through piecewise-stable stages, with short-term fluctuations between adjacent frequency levels on some devices.

Table 9: Overview of computational complexity and model size of the selected models.

Model Name	GMACs	Model Size (MB)
MobileNetV3 Small	0.058	9.75
MobileNetV3 Large	0.220	21.01
MobileNetV2	0.301	13.51
MobileNet	0.569	16.23
ResNet50	3.867	97.80
ResNet34	3.665	83.22
ResNet18	1.815	44.63
EfficientNet Lite 0	0.385	17.91
EfficientNet Lite 1	0.503	20.87
EfficientNet Lite 2	0.577	23.47
EfficientNet Lite 3	0.852	31.57
EfficientNet Lite 4	1.348	50.05
AlexNet	1.140	237.97
DenseNet121	2.835	30.76
DenseNet169	3.361	54.59
InceptionV3	2.844	91.00
NASNet Mobile	0.569	20.32
EfficientFormer	1.312	46.83
EfficientFormer W8A8	1.423	47.20
EfficientViT B2	1.651	92.76
LeViT	0.327	29.81
MobileViT	2.542	21.26
ShuffleNet V2	0.044	5.20
SqueezeNet 1.1	0.358	4.71