SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms

WONIK SEO, KAIST, Republic of Korea
SANGHOON CHA, Samsung Advanced Institute of Technology, Republic of Korea
YEONJAE KIM, JAEHYUK HUH, and JONGSE PARK, KAIST, Republic of Korea

With the proliferation of applications with machine learning (ML), the importance of edge platforms has been growing to process streaming sensor, data locally without resorting to remote servers.Such edge platforms are commonly equipped with heterogeneous computing processors such as GPU, DSP, and other accelerators, but their computational and energy budget are severely constrained compared to the data center servers. However, as an edge platform must perform the processing of multiple machine learning models concurrently for multimodal sensor data, its scheduling problem poses a new challenge to map heterogeneous machine learning computation to heterogeneous computing processors. Furthermore, processing of each input must provide a certain level of bounded response latency, making the scheduling decision critical for the edge platform. This article proposes a set of new heterogeneity-aware ML inference scheduling policies for edge platforms. Based on the regularity of computation in common ML tasks, the scheduler uses the pre-profiled behavior of each ML model and routes requests to the most appropriate processors. It also aims to satisfy the service-level objective (SLO) requirement while reducing the energy consumption for each request. For such SLO supports, the challenge of ML computation on GPUs and DSP is its inflexible preemption capability. To avoid the delay caused by a long task, the proposed scheduler decomposes a large ML task to sub-tasks by its layer in the DNN model.

CCS Concepts: • **Computer systems organization** → **Heterogeneous (hybrid) systems**; **Embedded systems**; • **Software and its engineering** → **Scheduling**; • **Computing methodologies** → *Machine learning;*

Additional Key Words and Phrases: Edge computing, heterogeneous computing, machine learning, inference, task scheduling

ACM Reference format:
Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. 2021. SLO-Aware Inference Scheduler for Heterogeneous Processors in Edge Platforms. *ACM Trans. Archit. Code Optim.* 18, 4, Article 43 (July 2021), 26 pages.
https://doi.org/10.1145/3460352

This work was in part supported by the Institute for Information & Communications Technology Planning & Evaluation (IITP2017-0-00466), Information Technology Research Center (ITRC) support program (IITP-2021-2020-0-01795), and National Research Foundation of Korea (NRF-2020R1A2C1103088). These grants are all funded by the Ministry of Science and ICT, Korea. This work was also partly supported by Samsung Electronics Co., Ltd.
Authors' addresses: W. Seo, Y. Kim, J. Huh, and J. Park (corresponding author), School of Computing, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, Republic of Korea, 34141; emails: {wiseo, yjkim, jhhuh, jspark}@casys.kaist.ac.kr; S. Cha, 130, Samsung-ro, Yeongtong-gu, Suwon-si, Gyeonggi-do, Republic of Korea, 16678; email: s.h.cha@samsung.com.

1544-3566/2021/07-ART43 $15.00
https://doi.org/10.1145/3460352

1 INTRODUCTION

The application areas of **machine learning (ML)** techniques have been growing rapidly, and platforms for ML are expanding from data center servers to mobile devices. One of the emerging computing models for ML processing is edge computing [2, 5, 9, 16, 17, 27, 31, 32, 37, 39–43, 46, 48, 49, 51, 52, 54, 59, 61, 62, 64, 67]. Edge platforms process streams of local sensor data, often using ML algorithms. As edge platforms are widely deployed under the constraints in their costs, their computational capability and energy budget are much more restricted than typical server environments.

Such edge platforms are commonly equipped with multiple heterogeneous computing processors. In addition to conventional CPUs, various accelerators such as GPUs and DSPs are able to process ML computation. With the advancement of hardware acceleration of ML and other common computations, more accelerators can be added to the processors of edge platforms, since such hardware accelerators can provide superior energy-efficient computations for specific algorithms, compared to conventional CPUs [4, 22, 23, 29, 33, 35, 38, 63, 65].

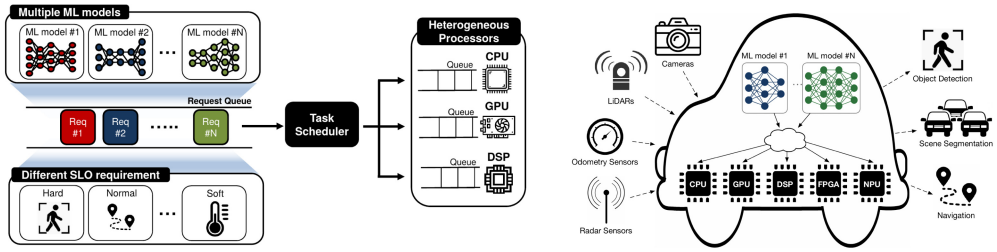
In the edge platforms with heterogeneous computing processors, application requirements for ML computation can differ from the traditional mobile devices. An edge platform constantly processes locally collected data with multiple ML algorithms. With the growing diversification of ML applications, an edge platform must run multiple ML models with different model architectures and parameters. Streams of sensor-generated data are fed to the edge platform, and inference tasks with heterogeneous ML models will be constantly processed for different streams of data.

Furthermore, the edge platform not only performs the computation of multiple ML models but also must provide a certain level of **service-level objectives (SLOs)**. In common use cases, input data are periodically generated, and the ML computation for an input request must be completed within a bounded latency [7, 8, 11, 15, 19, 24]. Supporting SLOs becomes complicated as multiple ML models have different computational requirements.

The aforementioned three factors, heterogeneous computing processors, multiple ML models, and SLO constraints, pose new challenges in the task scheduling of edge platforms. Heterogeneous ML inference computations utilize CPU, GPU, and DSP differently, with high variations of latencies and energy consumption. The scheduler must find the best mapping of incoming requests to the right computing device. In addition, the response latency for each request must be bounded to provide consistent quality of service with the SLO goal. The prior ML inference scheduler assumes either only multiple models on GPUs [50] or a single model on heterogeneous processors [22]. This study investigates the heterogeneity of both ML models and processors.

For effective scheduling considering the three factors, this article proposes new heterogeneity-aware scheduling policies for ML inference tasks. The proposed scheduling framework considers the heterogeneity of ML models as well as the heterogeneity of computing processors. Compared to general computation, an advantage of ML inference tasks is that its overall computation pattern does not dynamically change for each request. Therefore, the resource usage characteristics of the inference computation can be pre-profiled and stored with the model parameters. The proposed schedulers consult the pre-profiled execution behavior of each ML model on different computing processors and makes the scheduling decision to minimize the overall latencies. However, unlike the prior study [50], the proposed scheduler does not assume any fixed arrival rates for ML tasks, and thus the entire scheduling decision is made on-demand to reflect fluctuating request arrivals and resource availability.

In addition to the improvement of the overall average performance, the response latency of each request must be bounded. The scheduler is extended to satisfy such SLO constraints while reducing energy consumption. One of the challenges supporting SLOs is the non-preemptive nature of GPU and DSP computation. Once a big ML task is being processed in GPU or DSP, a much shorter task



(a) Scheduling framework overview.

(b) Example ML inference tasks on heterogeneous processors in an autonomous driving vehicle.

Fig. 1. ML inference for diverse ML tasks in a heterogeneous-platform edge device.

must wait for the completion of the blocking big task. To address such limitation of ML processing in GPUs and DSPs, this study utilizes *model slicing*, which decomposes the inference computation at layer boundaries in DNNs. By decomposing a single DNN processing into multiple smaller tasks, it provides more flexible scheduling of ML tasks.

To examine the effectiveness of proposed scheduling policies, we build an edge platform using the Open-Q 845 HDK Development Kit equipped with CPU, GPU, and DSP and evaluate the various application scenarios where the edge platform serves a thousand ML inference tasks based on six representative DNN models. Compared to naïve scheduling policies, the proposed scheduling policies offer up to 2.84× average performance speedup while reducing the SLO violation rate down to 4.9%. We open-source the software for broader community engagement. The source code is available at the following GitHub repository: https://github.com/casys-kaist/edge-scheduler.

This study is one of the first studies to consider the heterogeneity of both ML models and computing devices for efficient scheduling with SLO supports. The contributions are as follows:

- This study proposes a range of schedulers for heterogeneous processors and heterogeneous ML models, considering different characteristics of various ML models on multiple types of processors.
- It proposes an SLO-aware inference scheduler based on expected latencies using the pre-profiled task behaviors.
- It proposes a model slicing technique to provide coarse-grained preemption for GPU and DSP computation. It resolves the resource blocking by a big ML task, which can lead to SLO violations of following tasks.

The rest of the article is organized as follows. Section 2 presents the background of ML computation at edge devices. Section 3 presents the characteristics of ML workloads on heterogeneous computing processors. Section 4 presents the design space of heterogeneity-aware scheduling for ML tasks. Section 5 presents the experimental methodology, and Section 6 reports the experimental results. Section 7 discusses possible extensions of the proposed scheduler. Section 8 presents the related work, and Section 9 concludes the article.

2 BACKGROUND

This work explores task scheduling mechanisms for multi-model ML inference tasks on edge devices, as shown in Figure 1(a). The schedulers dispatch the various types of ML inference tasks to heterogeneous hardware platforms. In this section, we briefly provide the background details of target application scenarios and target edge device system environment.

2.1 ML Inference in Edge Platforms

In recent years, there have been remarkable advances in the ML algorithms. The community has even started declaring victory in achieving human-level accuracies for certain ML-based tasks (e.g., image recognition and classification) [25, 26]. Built upon such outstanding advances, the industry is now moving toward integrating the ML algorithms into various types of real-world applications and deploying the applications on the edge platforms [4, 22, 23, 29, 33, 35, 38, 47, 63, 65]. In many real-world scenarios, the edge platforms often serve multiple purposes and have to handle different types of inference requests for different ML models at the same time. This trend is inevitable and will intensify since there should be a limited number of computed-enabled edge platforms that can directly interact with humans, while ML algorithms are permeating virtually every application domain. Moreover, even though there exist several ML models that achieve equivalent objectives (e.g., image classification models such as ResNet, MobileNet, SqueezeNet, etc.), all of them are used independently in different real-world application scenarios, depending on the application-specific context and platform-specific constraints.

Consider an example delineated in Figure 1(b). The figure depicts an autonomous driving vehicle, which collects assorted types of sensory data and performs various types of ML inferences to serve multiple applications at runtime. In this example scenario, the autonomous driving vehicle is an edge platform that requires computing capabilities to perform ML inferences, since the nature of the autonomous driving applications requires strictly low latency and high energy efficiency, and hence entirely offloading the ML inferences to the cloud is unlikely to be feasible. An autonomous driving vehicle is not the only edge platform to host more than one ML application. There are many edge platforms used in a wide range of contexts and examples include smart-home hubs (e.g., Google Home and Amazon Echo), fog computing devices, ICU patient monitors, manufacturing robots, and surveillance cameras [23].

2.2 Heterogeneous Processors in Edge Platforms

Unlike conventional embedded devices that usually had a sole purpose or a limited number of features, modern edge platforms often support various capabilities from a wide range of application domains. As different applications need different types of compute operations and are used in unique contexts, the applications have disparate requirements in terms of performance and energy efficiency. To meet the diversified demands, many edge platforms are equipped with *heterogeneous* processors [30, 57].

Take Figure 1(b) for example once again. Autonomous driving vehicles take diverse kinds of sensory data as input and perform inference for assorted ML models while running on a battery, which necessitates a very performant yet power-efficient system equipped with heterogeneous platforms such as CPU, GPU, DSP, FPGA, and NPU, as shown in Figure 1(b). As a real-world example, the Tesla FSD computer consists of three quad-core CPUs, one GPU, two NPUs, one ISP, and a few more ASIC chips [53].

2.3 Service-Level Objectives (SLO) for ML Inference

Most near-real-time applications commonly deployed on edge platforms come with SLOs. Thus, when these applications rely on ML algorithms, achieving controlled latencies from ML inference tasks is important in the perspective of application SLO, since the inference processing time usually takes a significant portion of end-to-end application runtime. Achieving SLOs is particularly challenging in the case of edge platforms compared to the cloud, since the available hardware resource in the system is physically limited and not elastically scalable. Moreover, ML inference

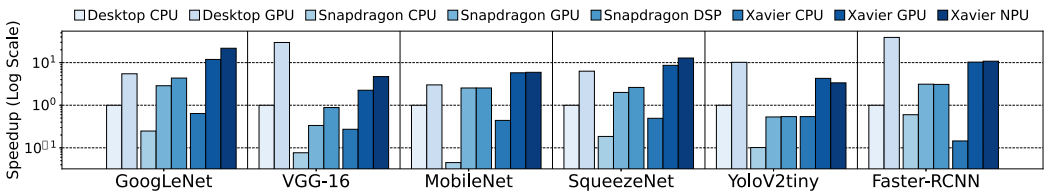


Fig. 2. Speedup of various processors on three different machines compared to the desktop CPU baseline, to run inference for the six evaluated ML models.

tasks on edge are oftentimes a part of mission-critical applications (e.g., pedestrian detection in ADAS), which makes SLOs not just a general guideline, but a must condition. The challenge gets even more difficult when the edge platforms serve diverse ML inference requests at an arbitrary rate on the heterogeneous hardware platforms. In this work, we aim to explore several scheduling policies for multi-model ML inference tasks while navigating the trade-off space of average response time, system throughput, and SLO. From the exploration, we observe how the scheduling decisions affect the system-wide behavior under the given application scenarios.

3 MOTIVATION

3.1 Performance and Energy Efficiency Characterization

We perform preliminary experimental studies to better understand the performance/energy characteristics of edge platforms when hosting ML inference executions. From the results, we observe that there are invariant properties originated from the correlation between (1) the algorithmic characteristics of given ML algorithms and (2) the hardware-specific characteristics per each platform. The insights drive the development of the proposed scheduling policies in this work.

Benchmarks and hardware platforms. We refer to Section 5 for details of the benchmarks. For this preliminary study, we use three different computing machines that come with various types of processors: (1) a desktop machine that comes with Intel Xeon E5-2630 v4 and NVIDIA RTX 2080 Ti; (2) a Qualcomm Snapdragon development board [13] (Open-Q™ 845 Hardware Development Kit) equipped with CPU, GPU, and DSP (see Section 5 for details); and (3) a NVIDIA Xavier development board [14] with ARM Carmel v8.2 CPU, NVIDIA Volta GPU, and NVIDIA NVDLA NPU.

Affinities of ML models to heterogeneous processors. Figure 2 shows the speedup of eight different processors equipped on three different machines when they run the six evaluated ML models. The baseline is the desktop-class CPU (i.e., Intel Xeon). This result demonstrates that the processor *affinity* varies across the ML models depending on diverse model-specific factors such as the number of compute operations, model size, composition of layers, and network topologies. For instance, VGG-16 shows the best performance when it runs on a power-hungry, desktop-class GPU, since it requires a large memory footprint and is able to exploit large data parallelism. On the contrary, for SqueezeNet, the low-power NPU (i.e., NVIDIA NVDLA) performs even better than the desktop GPU (i.e., NVIDIA RTX 2080 Ti), since this model does not provide high enough parallelism to extract high performance from thousands of GPU cores on the desktop GPU, yet the offloading overhead to the NPU is relatively smaller. While we only provide these two examples, Figure 2 shows that there is a significant variety in terms of the processor affinity to different ML models. Interestingly, we observe that the affinities tend to be a fixed property per a given pair of (ML model, processor), so we leverage this property to design our scheduling policies, which will be discussed in Section 4.

Energy efficiency characteristics of hardware platforms. While the ML models have disparate performance characteristics in terms of processor affinities, we observe that the models have

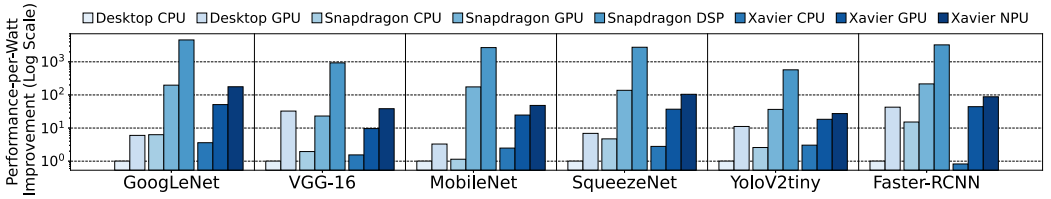


Fig. 3. Performance-per-watt improvement of various processors on the three different machines compared to the desktop CPU baseline, when running inference for the six ML models.

similar energy efficiency characteristics regardless of their algorithmic properties. Figure 3 shows that all ML models have the largest performance-per-watt improvement at low-power DSP (i.e., Snapdragon DSP) and the smallest at the desktop CPU, regardless of which model is evaluated on the platforms. These results imply that in the perspective of schedulers, always mapping all the given inference tasks to the energy-efficient processor is likely to be the best scheduling decision for energy efficiency, even though this strategy can be suboptimal in terms of performance.

3.2 Naïve Scheduling Policies

Affinity-oriented scheduling. A naïve approach we can think of based on the processor affinity results reported in Figure 2 is to always map the ML inference tasks to their best-performing hardware platforms. While this policy is intuitive and straightforward, the policy does not capture (1) the processor availability, (2) the composition of requests incoming to the edge platform at a given time, (3) the request arrival rate per each inference-requesting ML model, and (4) other potentially hidden factors. Therefore, unless the ML models have evenly distributed processor affinities and the inference requests are evenly scattered at a steady arrival rate, a small subset of processors would be bottlenecked while others remain available.

Availability-oriented scheduling. Another naïve approach is to greedily map an incoming inference request to an available processor at the moment. If there is an idle processor, the scheduling is straightforward, but in many cases, the processors are already pre-occupied with other tasks when a scheduling decision needs to be made. To schedule incoming requests while handling the scheduled requests, the scheduler has a waiting queue per each processor where the queue contains the pending requests in the order that the requests arrive to the queue. We use the length of pending requests in this queue as the degree of availability. This way, the scheduler is able to look up the processor with the shortest waiting queue to determine which processor to map the given request. Compared to the affinity-oriented scheduling, this approach tends to make more processors busy. However, such greedy approach does not always produce good decisions in terms of SLO, since the hastily assigned task on a slow processor may require significantly longer response time compared to the task assigned on a fast processor, even though there is a delay due to the in-queue wait time.

Energy-oriented scheduling. The last and simplest approach is to map all requests to the processor that has the highest energy efficiency. This approach could be used in the scenarios where the energy efficiency is the only concern, even though the system throughput and SLO need to be significantly compromised. We use this model as a comparison point for evaluating the energy efficiency of the proposed schedulers.

3.3 Implications of Model Slicing

Model slicing. To schedule an ML inference task, mapping the entire model to a processor as a whole is a straightforward approach. However, mapping a long-lasting inference task to a processor

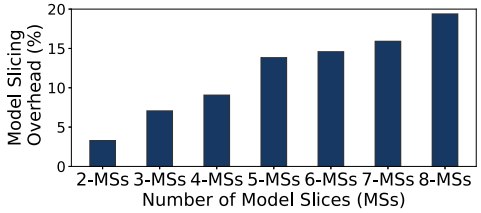


Fig. 4. Execution time overhead due to the number of model slices, normalized to the unsliced model execution time. For this experiment, we select to evaluate VGG-16 on GPU.

in a non-preemptive manner makes the task to fully occupy the processor while preventing the smaller tasks to be scheduled. For instance, when running inference on the Snapdragon DSP, VGG-16, the largest model we evaluated, takes 100 ms, while the smallest one, SqueezeNet, takes 12 ms. This effectively means that if the inference for VGG-16 just started, SqueezeNet potentially waits for about 8.3× of its execution time in the queue. This limitation makes the scheduling problem more challenging, especially when the scheduler aims to meet the SLO requirements.

Model slicing is an algorithmic technique to mitigate the problem, which partitions the large chunk of inference task into smaller subtasks at the boundaries of ML layers (e.g., convolution and fully connected layers). The use of model slicing potentially helps the schedulers balance the load without stalling many small tasks waiting for big tasks to be done and violating SLO significantly. The benefits of model slicing can be maximized when (1) the overhead incurred due to the partitioning is insignificant, and (2) the sliced subtasks take similar execution time on the given processor, because that way, the unit of scheduling can be unified. These conditions are highly dependent on the model slicing strategies.

Methodology. To understand the implications of different model slicing strategies, we first perform an experiment on the Snapdragon DSP for an example model, VGG-16. We select VGG-16 since it is the largest model that could potentially stall other small inference tasks. VGG-16 has 13 Convolution layers and 3 Fully Connected layers, and thus there can be myriad ways to slice the model at the layer boundaries. We explore a small subspace of the search space by limiting the number of slices and the slicing strategies to a few possibilities and discuss the implications of model slicing. In addition, we fix the number of slices to a particular number, four, and investigate the implications of the different slicing strategies for the evaluated ML models.

Overhead due to the number of model slices. Figure 4 shows the overhead on the execution time due to the model slicing as we change the number of model slices from two to eight. The overhead is normalized to the total execution time of the non-sliced model. In this experiment, we slice the model as evenly as possible when it comes to the per-slice execution time. The results are quite intuitive in that the slicing overhead linearly increases as the number of slices increases. For instance, the eight-way slicing strategy imposes almost 20% execution overhead compared to the unsliced vanilla model, which is likely to be too significant to be compromised for potential gains out of possibly better scheduling.

Performance implications of model slicing strategies. Figure 5 presents the overhead on the execution time due to the model slicing when we split the models into *four* slices using different model **slicing strategies (SSs)** There exist many factors that incur the overhead, but the primary factor is the intermediate output data movement at the slice boundaries. Depending on the slicing strategies, the intermediate output data size varies substantially, which explains the large gap between different strategies. The SS-uneven*x* represents the slicing strategies that split the models into unevenly divided slices, while SS-even represents the slicing strategy where we give our best

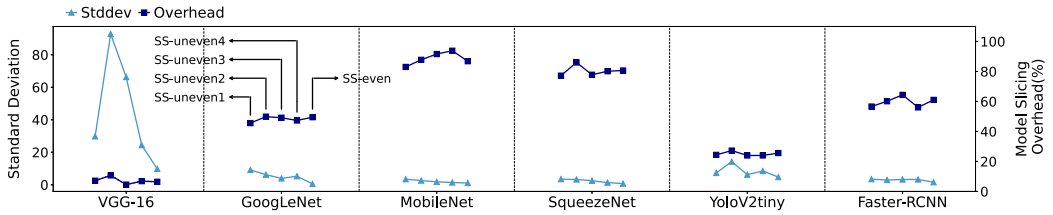


Fig. 5. Standard deviation and execution time overhead due to model slicing strategies, normalized to the unsliced model. For this experiment, we evaluated the six models on Snapdragon GPU.

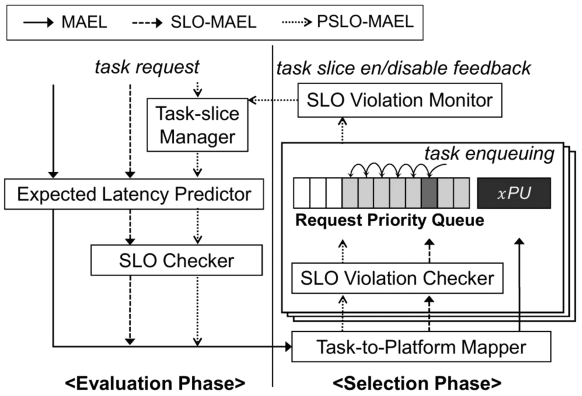


Fig. 6. Overview of the proposed schedulers. The different types of arrows refer to the workflow of the three proposed task scheduling algorithms. *x*PU refers to heterogeneous processors on the edge system, such as CPU, GPU, and DSP.

effort to split the models into slices that have the similar latencies. The figure first presents the unevenness of model slices using the standard deviation delineated with the light blue line graph (triangle marker). For all ML models, the SS-even delivers the lowest standard deviation as expected. The figure also reports the execution time overhead per each slicing strategy depicted with the dark blue line graph (square marker). For all models, the evenness of the slices has a marginal effect on the overall slicing overhead. As mentioned earlier, the balanced slicing strategy that produces slices with similar execution time is helpful for scheduling purposes, so we decided to take the evenly slicing strategy in the proposed scheduling policy, which we will discuss in Section 4.

4 SLO-AWARE INFERENCE SCHEDULERS

While the naïve scheduling algorithms discussed in Section 3 help us explore the trade-off space, none of the algorithms succeeds to effectively achieve the conflicting objectives: (1) minimizing the inference turnaround time (i.e., maximizing the system throughput) and (2) satisfying the inference SLO requirements.

4.1 Overview

We introduce three scheduling policies, each of which has a different set of optimization objectives. We start from a baseline scheduling policy exclusively optimized for the minimal inference turnaround time. The two following SLO-aware schedulers are designed such that they tend to try satisfying the SLO requirements. The workflows of the three schedulers are visually depicted in Figure 6.

Minimum-Average-Expected-Latency (MAEL). Our foundational scheduling policy is ***Minimum-Average-Expected-Latency* (MAEL)**, which is an SLO-agnostic, time-window-based scheduling policy. The sole goal of the MAEL scheduler is to minimize the average turnaround time of all inference tasks, requested and accumulated during the given scheduling window, by predicting their expected inference latencies at the scheduling point. For prediction, we rely on the unique property of ML models where the inference latency for a given model on a particular processor exhibits a very limited variance and is therefore very predictable. Hence, the scheduler at offline collects the profiled information of mappings from the pair of (model, processor) to the associated latency and, at runtime, uses this information to calculate the expected latencies. Since our target system aims to schedule multi-model inference tasks on heterogeneous processors, the search space of scheduling decisions is huge, which makes it infeasible for the runtime scheduler to visit all possible scenarios of task insertions on the per-processor request queue. Therefore, the MAEL scheduling policy and its two variants perform scheduling in two steps, (1) the former of which is the *evaluation phase*, which determines on which processor each inference task should be located, and (2) the latter of which is the *selection phase*, which decides where the task should be located within the per-processor request queue.

SLO-aware MAEL (SLO-MAEL). Since the MAEL scheduling algorithm lacks the SLO awareness, even though there are urgent inference tasks requested in the scheduling window, it dismisses the urgency and schedules based on the average expected latency. To overcome this limitation, we propose the *SLO* scheduler, the main goal of which is to take the avoidance and minimization of SLO violations at the first priority and put the system throughput at the next.

The scheduler is composed of two phases similar to MAEL. In the first phase, the scheduler evaluates if SLO violations are expected to exist. If the SLO violations are not expected to arise, the scheduler simply falls back to the MAEL scheduler. Otherwise, the scheduler aims to minimize the total summation of SLO violation degrees (i.e., how much longer the inferences take beyond the given SLO). Note that this algorithm concerns the SLO violation degrees, not the violation rate (i.e., how much ratio of inferences violate the given SLO), by which the algorithm tries to not only reduce the SLO violation rate but also eliminate the potential starvation issues. This design choice is from the observation that the scheduler optimized for minimizing the SLO violation rate tends to always prioritize scheduling small tasks over long ones. In fact, this phenomenon is intuitive, since saving the many by sacrificing the few is indeed superior in terms of rate, which instead harms the fairness between the scheduled models. To handle the starvation issues on long tasks and provide the inter-model fairness in the system, we seek an insight from a rather conventional scheduling mechanism in operating systems, *aging*, and leverage the idea in our scheduling scheme by using the SLO violation degrees. This way, as time passes, the starved tasks will be expected to have an increasing degree of violation, which pushes them to be prioritized in scheduling.

Preempting SLO-aware MAEL (PSLO-MAEL). Although the SLO-aware MAEL scheduler strikes the balance between the two optimization objectives, system throughput (i.e., the inverse of average inference turnaround time) and SLO, the algorithm still dispatches the inference tasks in a non-preemptive way, which in certain cases significantly limits the scheduling capabilities. For instance, a few long tasks already occupy all the hardware platforms and are expected to complete the computation in quite a while, and the hands of the SLO-aware MAEL scheduler are tied, which would in turn engender significant SLO violation in terms of both rate and degree.

To address the large SLO violation problem, we leverage the inherent algorithmic property of ML models that they consist of multiple layers of which computations can be represented as a series of inference tasks. To this end, we propose the *Preempting* SLO-aware MAEL scheduler, which leverages model slicing techniques [22] that split the large models into smaller yet evenly sized

ALGORITHM 1: Minimum-Average-Expected-Latency (MAEL)

Input : T: Inference tasks
P: Hardware platforms
$L(T,P)$: Inference latency of T on P
Output : Scheduling decision

1: **procedure** MAEL(T, P, L)
2: // Evaluation Phase
3: $candidates \leftarrow \emptyset$
4: **while** new_candidate_exists() **do**
5: $candidates \leftarrow candidates \cup \{(t_i, p_j) \mid \forall\, t_i \in T, \exists\, p_j \in P\}$
6: **for** $c \in candidates$ **do**
7: $score(c) = 0$
8: **for** $(t,p) \in c$ **do**
9: $score(c)$ += $\frac{1}{L(t,p)+wait\_time(p)}$
10: // Selection Phase
11: Find c where $score(c)$ is the max score
12: **for** $(t,p) \in c$ **do**
13: Insert t into $RequestPriorityQueue(p)$

sub-models and populate a sub-task per a sub-model to achieve the *preemption-like* effect for the scheduling purpose. Apparently, as we discuss in Section 3.3, there is overhead associated with the model slicing to initiate multiple small inference runs instead of a single large one. Thus, this scheduling algorithm is not always active and instead only turned on when the reduction in SLO violation degrees due to the preemption is expected to be significantly larger than the overhead.

4.2 Minimum-Average-Expected-Latency (MAEL)

Algorithm 1 elaborates the MAEL scheduling algorithm. This algorithm takes three sets of inputs: (1) T: a set of inference tasks given to the scheduler at a certain point in the runtime, which is periodically invoked. The scheduling window is a configurable parameter, which is empirically determined. (2) P: a set of processors available on a given edge platform. Modern edge platforms are increasingly equipped with a diverse set of hardware processors, including not only conventional processors such as CPU but also various accelerators such as GPU, DSP, and NPU [12, 18]. (3) $L(T,P)$: a set of mappings from pairs of (inference task, hardware platform) to the associated latency. The inference latency is heavily dependent on the algorithmic property of model and computing capabilities of hardware platforms, which are deterministic and make the accurate latency prediction possible. As discussed earlier, this information is collected in priori though offline profiling, and the runtime scheduler simply looks up the mapping table to get the latency. The output of the algorithm is scheduling decisions, which include the mapped hardware platform and the scheduled location in the request queue.

The scheduling algorithm constitutes two phases. In the evaluation phase, the scheduler first finds all possible task-to-platform mappings and collects them into a set, called *candidates* (Line 5). Then, iterating over the candidates, it calculates per-candidate scores (Lines 6∼9). To calculate the per-candidate score, the scheduler estimates the expected latency, which is the sum of (1) the profiled latency of task t on the platform p and (2) the current wait time due to the pending tasks already scheduled on the platform p (Line 9). As we would like to prioritize the candidate that delivers the *minimum* expected latency, we invert the calculated expected latency and accumulate for all tasks. After the evaluation phase, the scheduler obtains the per-candidate score, $score(c)$, which is in turn used in the selection phase.

In the selection phase, the scheduler sweeps over the collected scores and figures out which task-to-platform mapping produces the minimum average expected latency (Line 11). Once the

mappings are determined, the task is assigned to each platform (Line 13). Each platform has a request priority queue, which stacks up the scheduled tasks in an order based on past scheduling decisions. The scheduled task t is inserted in a place between the pending tasks in the request priority queue in such a way that the average expected latency is minimized. This way, the proposed two-phase scheduling mechanism effectively reduces the cost of search space exploration, while the scheduler is not capable of finding the globally optimal solutions.

Time complexity. To further analyze the scheduling overhead, we use the big O notation to mathematically represent the time complexity of the scheduling algorithm. The MAEL algorithm is fundamentally a brute-force approach, which (1) explores all possible candidates of task-to-platform mappings and (2) finds the best candidate mappings that minimize the expected latency. Therefore, the time complexity scales as the number of possible candidates increases. The number of the entire set of possible mappings is m^n, where n is the number of ML inference tasks to schedule, and m is the number of hardware platforms to be scheduled on. There is only a constant time required to get the latency for each mapping, since the latencies for possible task-to-platform mappings are profiled offline and can be obtained by a simple table lookup at runtime. Given the mappings, we now need to sort the collected latencies and find a candidate with the minimal latency. Since we need to sort the m^n possible candidates in terms of latency, the final time complexity is $O(m^n log(m^n))$. Therefore, as the n and m increase, the time complexity scales exponentially. However, in practical settings, (1) n (number of tasks) is mostly under at most 10 since the scheduling window of this algorithm is set to a few tens of milliseconds (e.g., 10 ms in our experiments), and (2) m (number of hardware platforms) on a machine is in most cases a small value (e.g., 3 in case of Snapdragon board). In fact, the small values of n and m effectively make the scheduling overhead substantially smaller compared to the execution time of ML tasks. In our experiments, the overhead was almost negligible (avg: 30 us, min: 10 us, and max: 140 us).

4.3 SLO-Aware MAEL (SLO-MAEL)

Algorithm 2 depicts the **SLO-aware MAEL (SLO-MAEL)** scheduling algorithm, which is built upon the MAEL algorithm while being designed such that it chases another objective, SLO. In addition to the algorithm inputs discussed in Section 4.2, the algorithm gets one more input, $SLO(T)$, which contains the task-specific SLO requirements.

Essentially, the score-based priority scheduling method is identical to that of the MAEL algorithm; however, at the evaluation phase, the scheduler calculates two independent scores, *score_ael* and *score_slo*, which represent scores for average expected latency and SLO, respectively. Before calculating the scores, the scheduler first checks if the expected latency is larger than the required SLO (Line 10). If yes, which means the task is expected to violate the SLO, instead of calculating the expected latency, the scheduler calculates the *degrees* of SLO violation, which can be calculated by normalizing the expected latency with the SLO requirement, and then accumulates the negated values of the SLO violation degrees (Line 11). This way, the *score_slo* contains a negative value *if* there is at least an SLO violation among the to-be-scheduled tasks.

When the scheduler reaches the selection phase, the scheduling decision can be simply made since the score is designed in such a way that the larger *score* value a candidate has, the better scheduling decision the candidate is.

4.4 Preempting SLO-aware MAEL (PSLO-MAEL)

Algorithm 3 details our last scheduling algorithm, the **Preempting SLO-aware MAEL (PSLO-MAEL)** algorithm. This algorithm leverages the *model slicing* to effectively enable preemption even on non-preemptive hardware and software inference frameworks, in order to further reduce

ALGORITHM 2: SLO-aware MAEL (SLO-MAEL)

Input : T: Inference tasks
P: Hardware platforms
$L(T,P)$: Inference latency of T on P
$SLO(T)$: SLO of T
Output : Scheduling decision

```
1:  procedure SLO-MAEL(T, P, L, SLO)
2:      // Evaluation Phase
3:      candidates ← ∅
4:      while new_candidate_exists() do
5:          candidates ← candidates ∪ {(t_i, p_j) | ∀ t_i ∈ T, ∃ p_j ∈ P}
6:      for c ∈ candidates do
7:          score_ael(c) = score_slo(c) = 0
8:          for (t, p) ∈ c do
9:              exp_l = L(t, p) + wait_time(p)
10:             if exp_l > SLO(t) then
11:                 score_slo(c) -= exp_l / SLO(t)
12:             else
13:                 score_ael(c) += 1 / exp_l
14:         score(c) = score_slo = 0 ? score_ael : score_slo
15:     // Selection Phase
16:     Find c where score(c) is the max score
17:     for (t, p) ∈ c do
18:         Insert t into RequestPriorityQueue(p)
```

the SLO violations. The inputs and outputs of this algorithm are commensurate with the ones in the SLO-aware MAEL algorithm. However, unlike the prior two algorithms, the PSLO-MAEL algorithm maintains a stateful variable, *SliceMode*, which is a flag switch that enables and disables the model slicing. To set this flag on or off, the scheduler monitors how beneficial the model slicing is to reduce the adverse effects of SLO violations and prudently decide to turn on/off. The reason we choose to have this algorithm stateful is that the scheduler needs to speculatively turn on the slicing so that the already-sliced small tasks can prevent the potential SLO violations of the unseen large tasks, which will arrive in the future. Note that model slicing is only helpful when the "short" incoming tasks preempt the "long" already-scheduled task. Moreover, the model slicing comes with a significant amount of latency overhead. Therefore, turning on model slicing without obtaining the SLO violation reduction only imposes the performance degradation, which can be eschewed via the speculative model slicing mechanism.

In the evaluation phase, the scheduler has a conditional block that checks whether the model slicing is on (Lines 3∼7). If yes, the scheduler slices the inference task (t) into a set (T') of sliced sub-tasks (sub_t) and puts all the sub-tasks into the task set (T) while removing the original large task (t) from the task set (T) to prevent computing duplicated tasks (Lines 5∼6). Not all of the models are subject to be sliced; only the large ones are. The threshold for determining whether to slice is empirically chosen and the slicing mechanism tends to produce evenly balanced sub-tasks so that the scheduling using the sub-tasks is more manageable. For brevity, in Algorithm 3, we omit the above details with respect to how the scheduler selects the slicing-target models. The rest of the evaluation phase does not have to be updated since the slicing replaces large tasks with the functionally identical sub-tasks in the set of inference-requested tasks; therefore, the SLO-aware MAEL scheduler smoothly identifies the optimized scheduling decision.

In the selection phase, there exists a slight difference at the end of the algorithm, which updates the *SliceMode* flag (Lines 24∼27). While the scheduler inserts the task into its request priority queue,

ALGORITHM 3: Preempting SLO-aware MAEL (PSLO-MAEL)

Input : T: Inference tasks
P: Hardware platforms
$L(T,P)$: Inference latency of T on P
$SLO(T)$: SLO of T
Output : Scheduling decision

procedure PSLO-MAEL(T, P, L, SLO)
  // Evaluation Phase
  if SliceMode is on **then**
    for $t \in T$ **do**
      Slice $t \in T$ into $T' = \{sub\_t : \sum sub\_t = t\}$
      Insert $t' \in T'$ into T
    Remove t from T
  $candidates \leftarrow \emptyset$
  while new_candidate_exists() **do**
    $candidates \leftarrow candidates \cup \{(t_i, p_j) \mid \forall\, t_i \in T, \exists\, p_j \in P\}$
  for $c \in candidates$ **do**
    $score\_ael(c) = score\_slo(c) = 0$
    for $(t,p) \in c$ **do**
      $exp\_l = L(t,p) + wait\_time(p)$
      if $exp\_l > SLO(t)$ **then**
        $score\_slo(c)$ -= $\frac{exp\_l}{SLO(t)}$
      else
        $score\_ael(c)$ += $\frac{1}{exp\_l}$
    $score(c) = score\_slo = 0\ ?\ score\_ael : score\_slo$
  // Selection Phase
  Find c where $score(c)$ is the max score
  for $(t,p) \in c$ **do**
    Insert t into $RequestPriorityQueue(p)$
    if Slicing helps SLO violation reduction **then**
      SliceMode=True
    else
      SliceMode=False

if the *SliceMode* is off, it checks if the given task is expected to violate the SLO requirement due to the pending long-latency tasks. If the condition is met, the *SliceMode* is turned on; otherwise, the mode remains switched off. If the *SliceMode* is already on, the scheduler sees if the sliced pieces of model help eliminate the potential SLO violations. If yes, the *SliceMode* remains True; otherwise, it is turned off. The scheduler ensures the dependencies among the sliced sub-tasks by halting the dispatch of the following sub-tasks until the executions of dependent sub-tasks are completed.

5 METHODOLOGY

5.1 Benchmarks

To examine the performance and energy consumption of ML inference tasks on edge platforms in diverse circumstances, we select six DNN models that have disparate application domains, model sizes, and network topologies, as shown in Table 1. We select these DNN models since they are representative and widely used as the core DNN structure in real-world edge applications such as lane detection [3], object tracking [45], image segmentation [55], object detection [6, 44], and pedestrian detection [56]. While many of them fall into the equivalent application domain, their performance and efficiency properties are unique since their network topologies and the employed operators are disparate, which necessitates their independent use in a single edge platform.

Table 1. Evaluated Machine Learning Models

Name	Model Size	Layers	Domain	Latency (ms)		
				CPU	GPU	DSP
GoogLeNet	96 MB	57 Conv + 1 FC	Image Classification	330	28	18
VGG-16	528 MB	13 Conv + 3 FC	Image Classification	1,150	263	100
MobileNet	16 MB	26 Conv + 1 FC	Image Classification	742	13	13
SqueezeNet	3 MB	13 Conv + 3 FC	Image Classification	182	16	12
Yolo-v2-tiny	5 MB	9 Conv	Object Detection	253	48	47
Faster-RCNN	167 MB	5 Conv + 3 FC	Object Detection	105	20	20

5.2 Hardware Platforms

We use Open-Q 845 HDK Development Kit, which is equipped with three hardware platforms: (1) a Qualcomm Snapdragon SDA845 CPU, (2) a Qualcomm Adreno 630 GPU, and (3) a Qualcomm Hexagon 685 DSP. The board vendor provides **Snapdragon Neural Processing Engine (SNPE)** SDK that is compatible with Caffe and Tensorflow so that the users can readily deploy the pre-trained DNN models onto the desired platform on the board. The current version of SNPE SDK supports a limited set of DNN operators. For instance, it lacks the support for RNN operators such as LSTM and GRU. However, such narrow coverage is the limitation of the particular platform, not a fundamental limitation of proposed scheduling schemes. In fact, the proposed schemes are not bound by any specific domain of ML algorithms either. Moreover, the SNPE SDK is closed source and a only offers restricted hardware-software interface that prevents the users from investigating the internal architecture and datapath of the SoC board, which did not allow us to perform detailed performance characterization.

For the experiments, we use Caffe and the Caffe-provided pre-trained models without making any algorithmic modifications (e.g., quantization and pruning). We measure the energy consumption using the device's current and voltage information recorded from Linux's event recording mechanism, `uevent`. The `uevent` produces the current and voltage information of platform batteries to the /sys/class/power_supply/battery/ directory, and we use this number to estimate the energy consumption. While we prototype our proposals and build an edge system on the Snapdragon development board, the proposed scheduling algorithms can be readily ported on any ML-serving systems that are equipped with heterogeneous processors, such as NVIDIA Jetson Xavier and server environments that we used to perform the preliminary study discussed in Section 3.1.

5.3 Request Generation Scenarios

Emulating the real-world request generation scenarios is difficult yet important to evaluate the proposed scheduling policies. However, the research in the field of edge computing is in its initial phase and there is not much resource publicly available to use for the experimental purposes. Therefore, we make a set of assumptions and synthetically generate the requests in various scenarios on a virtual edge platform. To emulate the task generation rate, we use the Poisson distribution for the models, similar to several prior works [10, 15, 19, 34, 50, 66]. For all scenarios, we set a fixed period of time at which the requests arrive to the system, which is 2 seconds. The 2 seconds is a long enough length of wall-clock runtime that we can evaluate the effectiveness of the scheduling policies for diverse application scenarios. During the runtime, the frequency and distributions of request arrivals depend on the given scenarios.

Table 2 reports the list of scenarios that we evaluate. The "Name" column shows the code name of each scenario that we will refer to in Section 6. The "Scenario" column shows the composition of long (L), medium (M), and short (S) inference runs, and the "Model" column denotes the

Table 2. Scenarios That Assume Multiple Edge Applications Running Together on an Edge System, Which Constitute Many Inference Tasks from a Diverse Set of Evaluated ML Models Reported in Table 1

Name	Scenario	Model (L, M, S)		% Requests	
Scenario1	LS (2)	VGG (L)	FAS (S)	33.3%	66.6%
Scenario2	MS (2)	YOL (M)	MOB (S)	33.3%	66.6%
Scenario3	MSS (3)	YOL (M)	MOB (S)	33.3%	33.3%
		SQU (S)		33.3%	
Scenario4	MMS (3)	YOL (M)	GOO (M)	25.0%	25.0%
		FAS (S)		50.0%	
Scenario5	LMSS (4)	VGG (L)	GOO (M)	25.0%	25.0%
		MOB (S)	FAS (S)	25.0%	25.0%
Scenario6	LMMS (4)	VGG (L)	GOO (M)	16.6%	33.3%
		YOL (M)	FAS (S)	33.3%	16.6%
Scenario7	LMMSSS (6)	VGG (L)	GOO (M)	16.6%	16.6%
		YOL (M)	MOB (S)	16.6%	16.6%
		SQU (S)	FAS (S)	16.6%	16.6%
Scenario8	LMMSSS (6)	VGG (L)	GOO (M)	10.0%	50.0%
		YOL (M)	MOB (S)	10.0%	10.0%
		SQU (S)	FAS (S)	10.0%	10.0%
Scenario9	LMMSSS (6)	VGG (L)	GOO (M)	12.5%	25.0%
		YOL (M)	MOB (S)	25.0%	12.5%
		SQU (S)	FAS (S)	12.5%	12.5%

Scenarios 1 and 2 constitute two types of ML inference tasks, while Scenarios 3 and 4 have three, and Scenarios 5 and 6 have four, and finally Scenarios 7 through 9 have six. *L*, *M*, and *S* refer to the long-, medium-, and short-taking inference tasks. The rightmost column presents the distributions of ML inference requests per the corresponding model. The ratio is set to a fixed distribution, but the request generation is based on the Poisson distribution.

corresponding DNN models for each scenario. The last column, "% Requests," represents the fractions of incoming requests, which have a one-to-one mapping from each model presented in the "Model" column. The total number of requests dispatched on each scenario is determined depending on the workload level that the edge platform is targeted to. In cases where the workload is overly lightweight or overly heavyweight, the system would be underutilized (i.e., most processors are idle for most of the runtime) and overloaded (i.e., the system cannot keep up with the incoming request rate and is always occupied with a full of work), regardless of the scheduling decisions. To evaluate diverse workload characteristics, we sweep through the workload intensity from lightweight to heavyweight and show how the scheduling algorithms affect the performance, energy efficiency, and SLO on the evaluated edge platform.

5.4 Metrics

To evaluate the performance and energy consumption of the proposed scheduling policies, we use various metrics that include **Averaged Normalized Turnaround Time (ANTT)**, energy consumption, energy delay product (EDP), energy delay squared product (ED^2P), and SLO. **Normalized Turnaround Time (NTT)** represents the performance slowdown (ML_i^{policy}) compared to its ideal case ($ML_i^{solo\_best}$), where the given task runs alone on the best-performing processor. ANTT

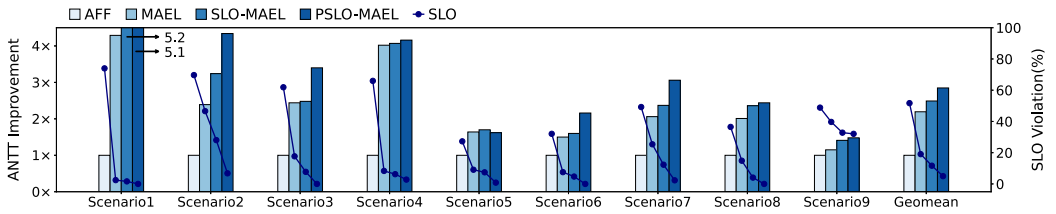


Fig. 7. ANTT improvement and the SLO violation rate for four different scheduling algorithms including the affinity-oriented scheduling algorithm (AFF) and the proposed three scheduling algorithms. The baseline is the case where AFF is used.

is the average normalized turnaround time of all ML inference tasks, as denoted in Equation (1). Note that the turnaround time not only includes the processing time but also the wait time in the ready queue. In other words, the turnaround time refers to the response time in the perspective of application users, rather than the pure execution time that goes to the ML inference.

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{ML_i^{policy}}{ML_i^{solo\_best}} \tag{1}$$

Defining the SLO target for ML inference tasks on the edge platform is a challenging task due to the lack of clearly defined response time guideline. Moreover, the SLO can vary significantly as the context of the application changes. For instance, while the pedestrian detection on an autonomous driving car has a very strict SLO requirement, monitoring traffic through object detection algorithms running on multiple cameras may not be on the critical path and can be performed asynchronously in the background. Therefore, in this work, we adopt a methodology used in a prior work, PREMA [11], which aims to achieve the SLO target for ML inference tasks on cloud. PREMA sets the SLO target as ($ML_i^{solo\_best} \times N$). We empirically choose to use 10 for N. Since our evaluation target is an edge platform equipped with a set of heterogeneous processors (i.e., CPU, GPU, and DSP), we pick a processor that best-performs per each DNN model and set the SLO targets as 10 times longer than the latency we measure on the best-performing processor. For instance, for SqueezeNet, since the inference time on the best-performing processor is 12 ms, we choose to set the SLO target to 120 ms.

6 EVALUATION

To empirically investigate the effectiveness of our three schedulers, we evaluate them under various application scenarios based on various Machine Learning models. We show how the three schedulers navigate the trade-off space of ANTT improvement (i.e., a proxy for system throughput) and SLO.

6.1 ANTT Improvement

Figure 7 presents the ANTT improvement of our three schedulers when their results are normalized to those of the affinity-oriented scheduling algorithm, introduced in Section 3.2. We call the affinity-oriented scheduling AFF for conciseness. We use the AFF as our baseline scheduling algorithm since AFF always shows better ANTT results than the possible counterparts such as the availability-oriented scheduling algorithm. We notice that the availability-oriented scheduling algorithm tends to schedule a lot of ML tasks to the slowest processor, CPU, when it is available, but this scheduling decision often incurs significant latency overhead for the tasks. The evaluated turnaround time includes the waiting delay in the request queue. On average, MAEL, SLO-MAEL, and PSLO-MAEL offer 2.19×, 2.48×, and 2.84× speedup, respectively. The results show that the MAEL, SLO-MAEL, and PSLO-MAEL algorithms offer a substantial performance improvement in

comparison with the naïve AFF algorithm, while suppressing the SLO violation to lower rates as we employ the SLO-oriented schemes.

Note that in most cases, SLO-MAEL and PSLO-MAEL offer even higher performance gains compared to MAEL since maximizing SLO not only helps avoid the SLO violation but also improves the throughput for small tasks, which effectively improve the system-wide "average" throughput. This phenomenon can be attributed to the short-task preference of SLO-aware scheduling algorithms and the definition of the ANTT metric. Our SLO targets are set proportionally to the execution latency of each model, similar to a prior work [11], which means the small models have tighter SLOs. As SLO-aware scheduling algorithms try to avoid the SLO violations as much as possible, they prefer to schedule the tasks with tight SLOs, which are the small ones. Since the definition of ANTT calculates the average *normalized* turnaround time, such short-task preference makes the ANTT improvement higher, which explains the increasing ANTT improvement when the algorithms try to satisfy SLOs. In fact, such scheduling schemes are likely to hurt the fairness between short and long tasks, though they would improve the system-wide throughput, which is an interesting challenge for multi-tenant systems, yet we leave it for future work.

Figure 7 also shows that depending on the scenarios, the algorithms behave differently in terms of turnaround time and SLO. For instance, in the case of Scenario 7, the SLO-MAEL and PSLO-MAEL algorithms make a significant difference in SLO violation results. In this particular case, the SLO-MAEL and PSLO-MAEL schedulers effectively exploit the room that the non-urgent inference tasks have, to serve urgent tasks promptly, which ends up significantly reducing the SLO violation. On the other hand, in the case of Scenario 1, none of our schedulers violates the SLO substantially from the beginning, so the PSLO-MAEL scheduling policy does not greatly help on the SLO violation reduction, but it improves the overall system throughput. The results demonstrate that in all cases, the schedulers are able to schedule the tasks in the direction that simultaneously achieves the conflicting goals, low turnaround time and SLO satisfaction, without compromising either.

6.2 Tail Latency and SLO

Figure 8 delineates the 99% tail latencies for the six evaluated models. The tail latencies are normalized to the SLO target, which means when a model sees the result of higher than 1.0, it implies the SLO violation exists. For Scenario 1 through Scenario 8, we observe that the baseline scheduling algorithm, AFF, exhibits a substantial imbalance among the tail latencies of different ML models. Moreover, for all the scenarios, the AFF algorithm produces the tall bars that go far beyond the SLO target and violate the SLO requirements. However, as we employ the expected latency-oriented scheduling (MAEL) and the SLO-aware scheduling algorithms (SLO-MAEL and PSLO-MAEL), the heights of SLO-violating bars are significantly shortened and squeezed under the SLO target (i.e., the horizontal 1× line). At the same time, at the expense of such gains, the heights of other bars that were initially smaller than the SLO target at AFF enlarge yet doenot go beyond the SLO target, which shows that the SLO-aware scheduling algorithms effectively prioritize the deadline-approaching, urgent tasks *without* newly introducing the SLO violations of other tasks. Scenario 9 is an exceptional case where the request generation rate is too high and the load is beyond the capabilities of given computing resources, which represents the state that even an oracle scheduler is unable to satisfy the SLO. For such a thrashed situation, the effectiveness of our scheduling algorithms is largely harmed, and yet rather, due to the prolonged tasks scheduled on the undesirable processors and model slicing overhead, we see the performance degradation and more SLO violations compared to the simple AFF baseline.

6.3 Energy Efficiency

While the energy efficiency is not one of the primary objectives of the proposed scheduling policies, we examine the implications on the energy efficiency since the policies can be used for

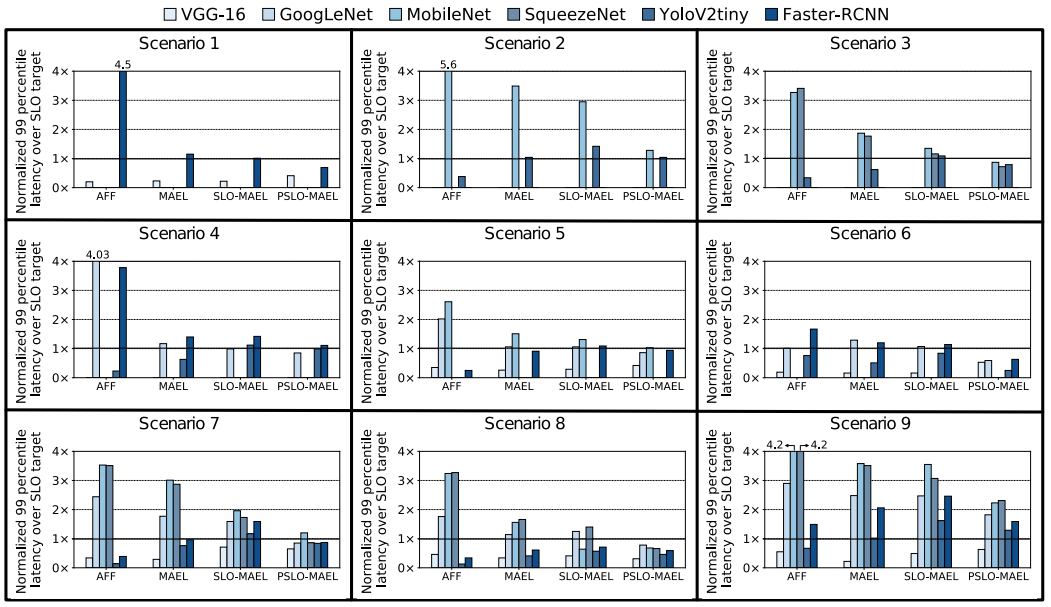


Fig. 8. Ninety-nine percent tail latencies when the proposed scheduling policies are used for all the scenarios. The horizontal lines at 1× refer to the point where the tail latencies meet the SLO target. If the bar is larger than 1×, it means an SLO violation.

potentially energy-constrained edge devices[1] We use (1) performance-per-watt, (2) energy delay product, and (3) ED^{2}P improvements to investigate the trade-off between system throughput and energy efficiency. Figure 9 shows the results for the five scheduling algorithms including the MAEL, SLO-MAEL, and PSLO-MAEL algorithms plus the two naïve ones, affinity-oriented and energy-oriented scheduling algorithms. We refer to the energy-oriented scheduling as ENR for brevity. The results are normalized to those of the AFF algorithm.

For all scenarios, the ENR algorithm is always superb in terms of performance-per-watt improvement, since this algorithm simply redirects all the inference tasks to the most energy-efficient processor, DSP, which dissipates significantly less energy for the same amount of computation, in comparison with the alternative processors such as CPU and GPU. Compared to the AFF algorithm, the ENR algorithm offers 2.74× performance-per-watt improvement, while the MAEL, SLO-MAEL, and PSLO-MAEL algorithms show 51%, 52%, and 43% degradation, respectively. However, when we emphasize more on the performance aspect of the system, which means we employ EDP and ED^{2}P instead of performance-per-watt improvement, the results are reversed. For EDP results, the MAEL, SLO-MAEL, and PSLO-MAEL algorithms offer 1.05×, 1.19×, and 1.59× EDP improvements, respectively. For ED^{2}P results, the MAEL, SLO-MAEL, and PSLO-MAEL offer 2.32×, 2.96×, and 4.53× ED^{2}P improvements, respectively, while the ENR algorithm experiences 77% ED^{2}P degradation. These results imply that the proposed scheduling algorithms not only concern the system throughput and SLO satisfaction rate but also implicitly push the scheduling decision in a direction that improves energy efficiency as well.

[1]The edge platforms we target are not the severely energy-constrained, battery-run edge platforms such as IoT devices, but rather power-connected local machines such as SoC computers for autonomous driving vehicles or smart home hubs. For this reason, we take the energy efficiency less significantly than the turnaround time and SLO.

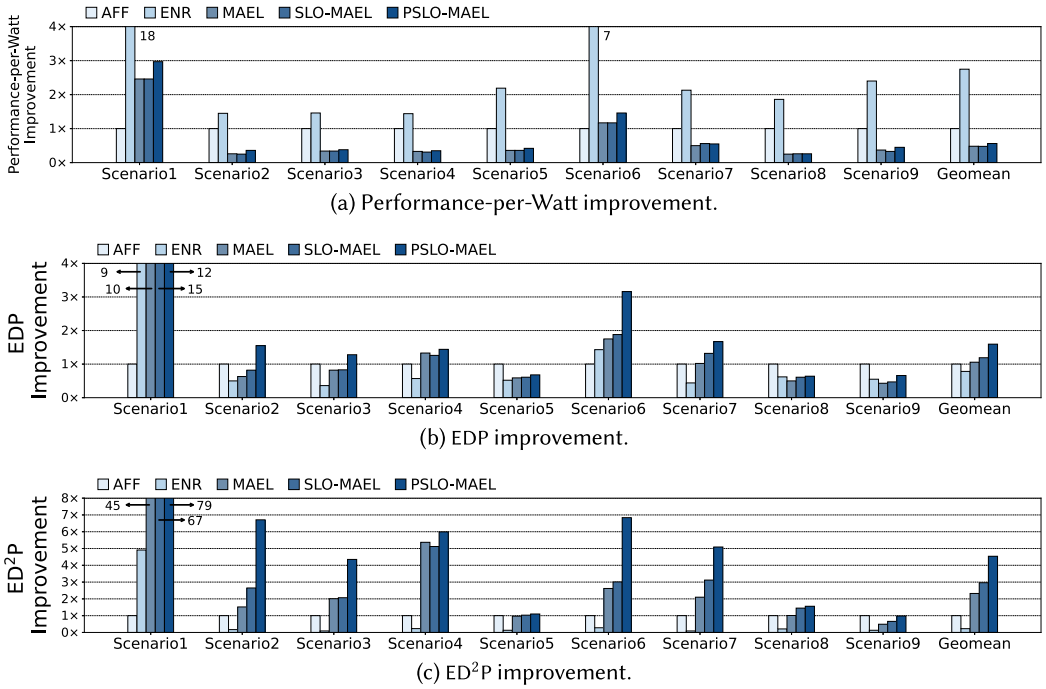


Fig. 9. Performance-per-watt, EDP, and ED^{2}P improvement results compared and normalized to the baseline scheduling algorithm, AFF.

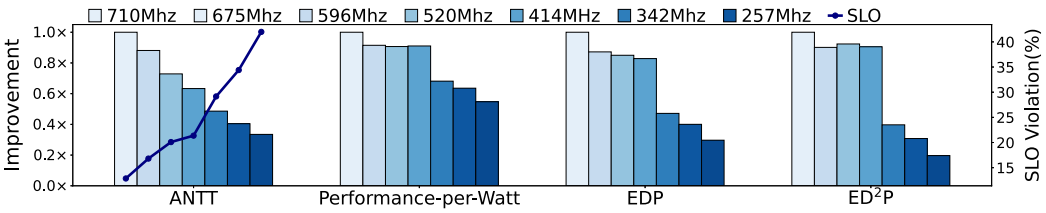


Fig. 10. ANTT, Energy, EDP, and ED^{2}P improvement plus SLO violation rate as we sweep through the GPU frequency from 710 MHz to 257 MHz. For this experiment, we use the SLO-MAEL scheduling algorithm. The results are normalized to the case where we use 710 MHz for GPU.

6.4 Implications of Dynamic Voltage Frequency Scaling

To navigate the performance-energy tradeoff, one possible knob for fine-grained scheduling is to use the **dynamic voltage frequency scaling (DVFS)**. Intuitively, it makes sense to dynamically alter the frequency and trade off the performance for efficiency if the lowered performance does not lead to the SLO violations. However, our empirical study corroborates that the use of this knob is virtually ineffective since lowering frequency prolongs the execution latency, which ends up imposing higher energy consumption and, in turn, exacerbates the energy efficiency. In our experiment, we use the GPU's DVFS feature on the evaluated Snapdragon board. As the CPU lacks the "per core" DVFS feature, the use of DVFS on CPU affects the "system-wide" performance; thus, we did not use it for the experiment. DSP lacks the DVFS feature. Figure 10 reports the results. For all metrics (i.e., ANTT, performance-per-watt, EDP, and ED^{2}P improvement), the highest frequency (i.e., 710 MHz) delivers the best performance and efficiency. Moreover, the SLO violation is minimal when we use

the 710 MHz. We believe that such ineffectiveness is attributed to the generally high parallelism and little control flow divergence of ML inference execution. Based on the empirical insights, we decided not to employ the DVFS feature as one of the knobs for our scheduling algorithms.

7 DISCUSSION

7.1 Predictability of ML Inference Latency

The proposed scheduling policies are built upon an insight that the ML inference latency tends to be deterministically predictable if the ML inference is performed for a particular model on a particular hardware platform. Note that such property tends not to hold if multiple ML inference tasks are co-located and batched on a single platform, as discussed in a prior work [21]. However, we are able to avoid the condition to be met and leverage this deterministic property since in our scheduling schemes, a single hardware platform always runs inference for a single model at a given time, and the batch size is always set to one, which not only minimizes the possibilities of nondeterministic system behaviors but also facilitates the scoring-based optimization by simplifying the objective function. CPU is an exception to this enforcement since as a host processor, CPU takes care of diverse processes including non-inference tasks such as platform control and management software. However, our schedulers barely map the inference tasks to CPU since its inference latency is significantly longer than the alternative processors, and thus, the variance in CPU performance had a limited influence on the end-to-end system performance. Moreover, the goal of proposed scheduling algorithms is to serve almost exclusively ML inference tasks, assuming that the non-inference workload is either rare or lightweight.

Another condition that makes the ML inference latency possibly unpredictable is the input data dependency of ML models. As the ML models tend not to have control flow in their computations, such unpredictability often comes from the size of input data that a model inference takes. A representative example algorithm is RNN for language translation, which takes as input a sentence composed of multiple words where the number of words is unknown a priori. For such cases, we enforce the models to have a fixed input data size by inserting dummy data so that the inference latency becomes deterministic and predictable. While such constraints require some extent of compromise on the throughput, the constraints enable improving the latency predictability and in turn the schedulability with minimal SLO violations, which is the highest priority in this work.

A great body of works [11, 15, 20, 22, 34, 66] have leveraged the aforementioned property to develop various sorts of system design solutions such as the SLO management and resource optimization. PREMA [11] proposes a predictive multi-task ML inference scheduling algorithm exploiting the deterministic predictability of ML inference. To motivate the work, the authors perform preliminary studies. First, they investigate whether the ML inference latency is deterministic by executing 1,000 inference runs on GPU for models with 50 different layer types and report that the latencies are off less than 4% from the average. They also report that the ML inference latencies obtained by Google Cloud TPUv2 also fall within 0.2% standard deviation of the mean [11]. Clockwork [20] is an ML serving distributed system, which leverages the deterministic predictability of ML inference. Not only the aforementioned examples but also many others [15, 34, 66] have leveraged the unique property of ML inferences, which largely inspired this work.

7.2 Queuing Theory

The goal of this work is not to devise a scheduling mechanism, which hits the very sweet spot in the trade-off space placed with the dimensions of performance, energy efficiency, and quality of service. It is rather to look into a wide spectrum of possible scheduling policies while exploring the trade-off space. One key observation we made from the empirical studies is that the request arrival

scenarios make a significant impact on the effectiveness of the scheduling policies. The request arrival scenarios are driven by not only the general workload of incoming requests but also the arrival rate, the compositions of requesting models, the periodic patterns of request bursts, and many others. Therefore, it is critical to accurately model the request generation scenarios in such a way that they are very close to the real-world system environments of ML-serving edge platforms. A natural future direction of this study is to answer this question: how do we more accurately model edge devices with heterogeneous processors, serving multi-model ML tasks, *without* actually building the real system? Queuing theory [1] is a line of research in which researchers have been trying to model and solve similar problems as ours. The scheduling problem this work aims to address is a particular case of the problems that queuing theory tries to solve, in that (1) we have a specific number of clients (i.e., ML inference tasks), and (2) there are a fixed number of servers (i.e., heterogeneous processors for a given edge platform). We believe that not only would this modeling be a novel contribution by itself, but also it would help identify undiscovered issues and challenges in designing more mature scheduling policies in the future effort.

7.3 Theoretical Guarantees

This work aims to explore the trade-off space between the SLO guarantees and system throughput and seeks to strike a careful balance between the two, which (1) pushes the system throughput to the maximum (2) while the violations are minimized, but the violations could still possibly happen. In other words, few SLO violations are an expense for higher throughput, which we compromise in this work. In case of real-time systems that require strict latency constraints, such compromise must not be an option. The real-time systems would require the theoretical guarantees, and thus the systems must employ a scheduling policy that exclusively accepts the requests when the target latency can be safely met with sufficient headroom for runtime volatility and conservatively discards excessive requests. Devising such scheduling policy is on its own is a novel research problem that we aim to explore as future work. This work has the similar goal to the scheduling mechanisms largely explored in the realm of ML inference schedulers for datacenters [10, 11, 15, 19, 20, 34, 50, 66], which seek to offer the best-effort SLO guarantees.

7.4 Edge-Cloud Continuum

While this work focuses on on-device ML scheduling, offloading ML inferences to the cloud is another alternative option. As a matter of fact, the modern mobile and edge devices are currently offloading most of their ML inference tasks to the cloud (e.g., Apple Siri), rather than hosting on the edge. Therefore, it is natural to consider the cloud as a co-processing engine in the perspective of edge devices and deem the edge-cloud continuum as a virtual ML serving system. In our experiment, the desktop GPU (NVIDIA RTX 2080 Ti) performed on average 6.8x faster than the Snapdragon's GPU counterpart. However, the challenge is the network communication cost, which is often not affordable for real-world applications. To estimate the cost, we performed a preliminary experiment using a VM hosted by Microsoft Azure. The average latency to exchange the input and output data was 32 ms, which effectively canceled out the acceleration benefits. Nevertheless, we still believe that offloading to the cloud makes sense, especially when the edge devices are excessively overloaded and the SLO is a primary concern.

7.5 Requests with Different Priorities

Although the proposed SLO-aware schedulers aim to meet the SLO target for all requests *equally* without having any precedence, the schedulers can be readily modified to support the precedence-based scheduling by assigning the priority to critical tasks in the queue if they should be executed preferentially. In fact, our PSLO-MAEL algorithm already employs the notion of prioritization

through the preemption for relatively shorter tasks. Hence, enabling such customized prioritization on top of our solution should be fundamentally straightforward.

8 RELATED WORK

8.1 ML-Based Applications on Edge Devices

There has been a great body of work that leverages ML algorithms in the real-time edge applications [2, 5, 9, 16, 17, 27, 31, 32, 37, 39–43, 46, 48, 49, 51, 52, 54, 59, 61, 62, 64, 67]. The applications are from a wide range of application domains and the examples are autonomous driving [5, 17, 39, 46, 49, 51, 52, 67], mobile computing [31, 40, 48], robotics [37, 43, 59, 62], smart home technologies [2, 9, 16, 32, 64], smart city technologies [27, 41, 42, 54], and surveillance systems [61]. While ML is already a pivotal component in modern real-time application domains, we can readily expect the boundary of this scope to be significantly expanded in the near future, as the community is pushing forward to practicalize ML algorithms and deploy them in a wider spectrum of application scenarios. This trend will lead a growing number of edge platforms to serve more than one ML inference request since the number of applications that every edge platform needs to host keeps increasing, while the number of compute-enabled edge platforms that can directly interact with humans is limited. Therefore, without effective scheduling mechanisms, the edge platforms would be readily swamped with numerous ML inference tasks, which is an important problem to solve. To the best of our knowledge, this work is the first effort to investigate the trade-off space of scheduling policies that handle multiple ML inference tasks on heterogeneous-processor edge platforms.

8.2 Inference Task Scheduling for Heterogeneous-Platform Edge Device

There have been several efforts that make use of heterogeneous processors for efficient ML inference [4, 22, 23, 29, 33, 35, 38, 63, 65]. Facebook [63] shares their hardware-software stack for ML inference at heterogeneous mobile systems. Neurosurgeon [33] provides DNN model partitioning techniques and intelligently maps the model shards on to the cloud and mobile device. Similar to Neurosurgeon, MOSAIC [22] and DeepX [38] propose the DNN model partitioning techniques, but these works aim to map the sliced model shards onto heterogeneous processors at edge. DeepMon [29] takes computing for Convolution operators offloaded to GPU and utilizes both CPU and GPU to minimize the inference latency. DeepIoT [65] proposes model compression and sparsification techniques to compact DNN models so that the compressed models are better fit for power-constrained IoT devices. μLayer [35] intelligently maps ML inference tasks to CPU and GPU on mobile devices by leveraging layer distribution and processor-specific quantization techniques. Cao et al. [4] propose to schedule a single ML inference on the heterogeneous platforms by splitting the given model into pieces and distributing over the CPU/GPU platforms. Pipe-it [58] proposes a pipelined design to split convolutional layers in asymmetric big.LITTLE multicore clusters to improve throughput. Wang et al. [60] navigate the performance-power tradeoff space of mobile SoCs equipped with heterogeneous processors when they perform ML inferences. Heo et al. [28] propose an ML inference latency prediction model for GPU and devises multipath neural networks, which enable the runtime to choose which path to take to meet real-time latency constraints. AutoScale [36] is an execution scaling engine that leverages Reinforcement Learning to adaptively determine which platform to pick for performing inference to improve energy efficiency in edge-cloud systems. While the prior work has made the initial efforts to enable ML inference on the edge system, none has taken into consideration the scheduling problem for multiple ML inference tasks running on the heterogeneous-processor edge system, on which this work focuses.

8.3 Service-Level Objectives (SLO) for ML Inference

There have been prior works that aim to offer SLO for ML inference services, but most works solely target the cloud [7, 8, 11, 15, 19, 24]. Baymax [8] and Prophet [7] provide runtime solutions that offer SLO on latencies when non-preemptive accelerators are used for ML inference at the cloud. PREMA [11] has a similar goal with Baymax and Prophet but aims to solve the problem for preemptive **neural processing units (NPUs)**. Swayam [19] is a distributed autoscaling framework on the cloud, which offers SLO while maximizing the resource utilizing of a scaling system. Clipper [15] is a pioneering work that proposes an ML inference serving system on the cloud. These works have pioneered to meet the SLO requirements for ML inference jobs on the cloud, but few works have been done to provide SLO for ML inference at the edge device.

9 CONCLUSION

With the advent of Edge Computing and the Internet of Things (IoT), the industry is building compute-enabled edge platforms equipped with heterogeneous processors. These platforms are being actively deployed to places where humans can directly interact—from mobile devices to autonomous vehicles, smart home/city devices, and robots. In a rather disjoint effort, the industry is accelerating the integration of ever-advancing ML technologies and real-time edge applications. Thus, efficiently mapping the multi-model ML tasks to heterogeneous processors at edge is a crucial challenge to address. This work is a timely effort that sets out to address the scheduling problem on ML-enabled edge platforms and takes an effective initial step toward exploring the trade-off space of performance, quality of service, and energy efficiency. This work identifies the limitations of naïve scheduling policies, proposes novel scheduling policies that navigate the dimensions of the trade-off space, and evaluates its implications on system throughput, SLO, and energy efficiency.

REFERENCES

[1] Arnold O. Allen. 2014. *Probability, Statistics, and Queueing Theory*. Academic Press.

[2] Debraj Basu, Giovanni Moretti, Gourab Sen Gupta, and Stephen Marsland. 2013. Wireless sensor network based smart home: Sensor selection, deployment and monitoring. In *2013 IEEE Sensors Applications Symposium Proceedings*. IEEE, 49–54.

[3] Ravi Bhandari, Akshay Uttama Nambi, Venkata N. Padmanabhan, and Bhaskaran Raman. 2018. DeepLane: Camera-assisted GPS for driving lane detection. In *Proceedings of the 5th Conference on Systems for Built Environments*. 73–82.

[4] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2019. Work-in-Progress: Video analytics from edge to server. In *2019 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'19)*. IEEE, 1–2.

[5] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*. 2722–2730.

[6] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. 2017. Learning efficient object detection models with knowledge distillation. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 742–751.

[7] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 17–32.

[8] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices* 51, 4 (2016), 681–696.

[9] Jonghwa Choi, Dongkyoo Shin, and Dongil Shin. 2005. Research and implementation of the context-aware middleware for controlling home appliances. *IEEE Transactions on Consumer Electronics* 51, 1 (2005), 301–306.

[10] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2020. LazyBatching: An SLA-aware batching system for cloud machine learning inference. *arXiv preprint arXiv:2010.13103* (2020).

[11] Yujeong Choi and Minsoo Rhu. 2020. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 220–233.

[12] Google Cloud. 2019. Edge TPU. https://cloud.google.com/edge-tpu.

[13] Intrinsyc Technologies Corporation. 2021. Qualcomm Snapdragon development board. https://www.intrinsyc.com.

[14] NVIDIA Corporation. 2019. Jetson AGX Xavier Developer Kit. https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.
[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 613–627.
[16] Prafulla N. Dawadi, Diane J. Cook, and Maureen Schmitter-Edgecombe. 2013. Automated cognitive health assessment using smart home monitoring of complex tasks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 43, 6 (2013), 1302–1313.
[17] Ürün Dogan, Johann Edelbrunner, and Ioannis Iossifidis. 2011. Autonomous driving: A comparison of machine learning techniques by means of the prediction of lane change behavior. In *2011 IEEE International Conference on Robotics and Biomimetics*. IEEE, 1837–1843.
[18] Samsung Electronics. 2019. Samsung NPU. https://news.samsung.com/global/samsung-electronics-introduces-a-high-speed-low-power-npu-solution-for-ai-deep-learning.
[19] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 109–120.
[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 443–462.
[21] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of Facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, 488–501.
[22] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. 2019. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19)*. IEEE, 165–177.
[23] Tianshu Hao, Yunyou Huang, Xu Wen, Wanling Gao, Fan Zhang, Chen Zheng, Lei Wang, Hainan Ye, Kai Hwang, Zujie Ren, et al. 2018. Edge AIBench: Towards comprehensive end-to-end edge computing benchmarking. In *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 23–30.
[24] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 620–629.
[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*. 1026–1034.
[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
[27] Ying He, F. Richard Yu, Nan Zhao, Victor C. M. Leung, and Hongxi Yin. 2017. Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach. *IEEE Communications Magazine* 55, 12 (2017), 31–37.
[28] Seonyeong Heo, Sungjun Cho, Youngsok Kim, and Hanjun Kim. 2020. Real-time object detection system with multi-path neural networks. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'20)*. IEEE, 174–187.
[29] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. Deepmon: Mobile GPU-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 82–95.
[30] ADLINK Technology Inc. 2019. Heterogeneous Computing for AI at the Edge. https://www.adlinktech.com.
[31] Arash Jahangiri and Hesham A. Rakha. 2015. Applying machine learning techniques to transportation mode recognition using mobile phone sensor data. *IEEE Transactions on Intelligent Transportation Systems* 16, 5 (2015), 2406–2417.
[32] Vikramaditya R. Jakkula and Diane J. Cook. 2011. Detecting anomalous sensor events in smart home data for enhancing the living experience. *Artificial Intelligence and Smarter Living* 11, 201 (2011), 1.
[33] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.
[34] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Grandslam: Guaranteeing SLAs for jobs in microservices execution frameworks. In *Proceedings of the 14th EuroSys Conference 2019*. 1–16.

[35] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the 14th EuroSys Conference 2019*. 1–15.

[36] Young Geun Kim and Carole-Jean Wu. 2020. AutoScale: Optimizing energy efficiency of end-to-end edge inference under stochastic variance. *arXiv preprint arXiv:2005.02544* (2020).

[37] Jens Kober, Erhan Oztop, and Jan Peters. 2011. Reinforcement learning to adjust robot movements to new situations. *Robotics: Science and Systems, MIT Press Journal* 6 (2011), 33–40.

[38] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'16)*. IEEE, 1–12.

[39] Jaeyoung Lee, Aravind Balakrishnan, Ashish Gaurav, Krzysztof Czarnecki, and Sean Sedwards. 2019. Wisemove: A framework for safe deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1902.04118* (2019).

[40] Emiliano Miluzzo, Tianyu Wang, and Andrew T. Campbell. 2010. Eyephone: Activating mobile phones with your eyes. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Networking, Systems, and Applications on Mobile Handhelds*. 15–20.

[41] Mehdi Mohammadi and Ala Al-Fuqaha. 2018. Enabling cognitive smart cities using big data and machine learning: Approaches and challenges. *IEEE Communications Magazine* 56, 2 (2018), 94–101.

[42] Mehdi Mohammadi, Ala Al-Fuqaha, Mohsen Guizani, and Jun-Seok Oh. 2017. Semisupervised deep reinforcement learning in support of IoT and smart city services. *IEEE Internet of Things Journal* 5, 2 (2017), 624–635.

[43] Amir Mosavi and Annamaria R. Varkonyi-Koczy. 2017. Integration of machine learning and optimization for robot learning. In *Recent Global Research and Education: Technological Challenges*. Springer, 349–355.

[44] Mahyar Najibi, Mohammad Rastegari, and Larry S. Davis. 2016. G-CNN: An iterative grid based object detector. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2369–2377.

[45] Nick Nordlund, Heesung Kwon, Geeth Ranmal De Mel, and Leandros Tassiulas. 2018. Image classification on the edge for fast multi-camera object tracking. In *2018 IEEE Military Communications Conference (MILCOM'18)*. IEEE, 1–5.

[46] Gennaro Notomista and Michael Botsch. 2017. A machine learning approach for the segmentation of driving maneuvers and its application in autonomous parking. *Journal of Artificial Intelligence and Soft Computing Research* 7 (2017), 243—255. https://www.sciendo.com/article/10.1515/jaiscr-2017-0017.

[47] Jihong Park, Sumudu Samarakoon, Mehdi Bennis, and Mérouane Debbah. 2019. Wireless network intelligence at the edge. *Proceedings of the IEEE* 107, 11 (2019), 2204–2239.

[48] Naser Peiravian and Xingquan Zhu. 2013. Machine learning for Android malware detection using permission and API calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE, 300–305.

[49] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. 2017. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging* 2017, 19 (2017), 70–76.

[50] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.

[51] David Stavens and Sebastian Thrun. 2012. A self-supervised terrain roughness estimator for off-road autonomous driving. *arXiv preprint arXiv:1206.6872* (2012).

[52] David Michael Stavens. 2011. *Learning to Drive: Perception for Autonomous Cars*. Stanford University.

[53] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, et al. 2020. Compute solution for Tesla's full self-driving computer. *IEEE Micro* 40, 2 (2020), 25–35.

[54] Bo Tang, Zhen Chen, Gerald Hefferman, Tao Wei, Haibo He, and Qing Yang. 2015. A hierarchical distributed fog computing architecture for big data analysis in smart cities. In *Proceedings of the ASE BigData & SocialInformatics 2015*. 1–6.

[55] Hu Tao, Weihua Li, Xianxiang Qin, and Dan Jia. 2018. Image semantic segmentation based on convolutional neural network and conditional random field. In *2018 10th International Conference on Advanced Computational Intelligence (ICACI'18)*. IEEE, 568–572.

[56] Denis Tomè, Federico Monti, Luca Baroffio, Luca Bondi, Marco Tagliasacchi, and Stefano Tubaro. 2016. Deep convolutional neural networks for pedestrian detection. *Signal Processing: Image Communication* 47 (2016), 482–489.

[57] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. 2016. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud'16)*. IEEE, 20–26.

[58] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. 2019. High-throughput CNN inference on embedded ARM Big. LITTLE multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2254–2267.

[59] Shouyi Wang, Wanpracha Chaovalitwongse, and Robert Babuska. 2012. Machine learning algorithms in bipedal robot control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 5 (2012), 728–743.

[60] Siqi Wang, Anuj Pathania, and Tulika Mitra. 2020. Neural network inference on mobile SOCs. *IEEE Design & Test* 37, 5 (2020), 50–57.

[61] Shibo Wang, Shusen Yang, and Cong Zhao. 2020. SurveilEdge: Real-time video query based on collaborative cloud-edge deep learning. In *IEEE Conference on Computer Communications (IEEE INFOCOM'20)*. IEEE, 2519–2528.
[62] Tianyu Wang, Giuseppe Cardone, Antonio Corradi, Lorenzo Torresani, and Andrew T. Campbell. 2012. Walksafe: A pedestrian safety app for mobile phone users who walk and talk while crossing roads. In *Proceedings of the 12th Workshop on Mobile Computing Systems & Applications*. 1–6.
[63] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. 2019. Machine learning at Facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE, 331–344.
[64] Rayoung Yang and Mark W. Newman. 2013. Learning from a learning thermostat: Lessons for intelligent systems for the home. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 93–102.
[65] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. 2017. Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–14.
[66] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 1049–1062.
[67] Quanwen Zhu, Long Chen, Qingquan Li, Ming Li, Andreas Nüchter, and Jian Wang. 2012. 3D Lidar point cloud based intersection recognition for autonomous driving. In *2012 IEEE Intelligent Vehicles Symposium*. IEEE, 456–461.

Received June 2020; revised March 2021; accepted April 2021