Dělen: Enabling Flexible and Adaptive Model-serving for Multi-tenant Edge AI

Qianlin Liang
University of Massachusetts Amherst
Amherst, MA, USA
qliang@cs.umass.edu

Walid A. Hanafy
University of Massachusetts Amherst
Amherst, MA, USA
whanafy@cs.umass.edu

Noman Bashir
University of Massachusetts Amherst
Amherst, MA, USA
nbashir@umass.edu

Ahmed Ali-Eldin
University of Massachusetts Amherst
Amherst, MA, USA
ahmeda@cs.umass.edu

David Irwin
University of Massachusetts Amherst
Amherst, MA, USA
irwin@ecs.umass.edu

Prashant Shenoy
University of Massachusetts Amherst
Amherst, MA, USA
shenoy@cs.umass.edu

ABSTRACT

Model-serving systems expose machine learning (ML) models to applications programmatically via a high-level API. Cloud platforms use these systems to mask the complexities of optimally managing resources and servicing inference requests across multiple applications. Model serving at the edge is now also becoming increasingly important to support inference workloads with tight latency requirements. However, edge model serving differs substantially from cloud model serving in its latency, energy, and accuracy constraints: these systems must support multiple applications with widely different latency and accuracy requirements on embedded edge accelerators with limited computational and energy resources.

To address the problem, this paper presents Dělen,[1] a flexible and adaptive model-serving system for multi-tenant edge AI. Dělen exposes a high-level API that enables individual edge applications to specify a bound at runtime on the latency, accuracy, or energy of their inference requests. We efficiently implement Dělen using conditional execution in multi-exit deep neural networks (DNNs), which enables granular control over inference requests, and evaluate it on a resource-constrained Jetson Nano edge accelerator. We evaluate Dělen flexibility by implementing state-of-the-art adaptation policies using Dělen's API, and evaluate its adaptability under different workload dynamics and goals when running single and multiple applications.

ACM Reference Format:
Qianlin Liang, Walid A. Hanafy, Noman Bashir, Ahmed Ali-Eldin, David Irwin, and Prashant Shenoy. 2023. Dělen: Enabling Flexible and Adaptive Model-serving for Multi-tenant Edge AI. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '23), May 9–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3576842.3582375

1 INTRODUCTION

Model-serving systems expose machine learning (ML) models to applications programmatically via a high-level API [55]. These systems typically leverage one or more previously-trained deep learning models on cloud servers, and mask the complexities of optimally selecting models, managing resources, and servicing inference requests across many applications [8, 16, 27, 55, 56]. Cloud-based model serving systems are generally multi-tenant with each server hosting many deep learning models that serve multiple inference applications [39]. Multi-tenancy increases efficiency by multiplexing limited resources e.g., in GPUs, across applications.

Model serving at the edge is now also becoming increasingly important to support inference workloads for Internet of Things (IoT) applications with tight latency requirements, including in smart homes, mobile health, and wearable devices [40, 48]. Such IoT applications are becoming pervasive due to continuing advances in hardware miniaturization and improvements in the energy-efficiency of sensing and communication. Traditionally, edge devices, such as smart assistants (e.g., Siri, Alexa, etc.), smart cameras, and emerging household robots, that sense data for IoT applications have sent the data over the network to remote servers, often in the cloud, for ML inference processing. However, in recent years, a new generation of IoT devices has emerged that enable sophisticated low-latency processing of data at the edge. As one example, smart speakers with voice assistants that respond to spoken commands require processing and responses in real-time with low-latency to provide an adequate interactive experience for users.

Thus, edge computing – where processing is done at the edge of the network close to users – has emerged as the preferred architecture for enabling low-latency IoT applications. To support edge computing, a new class of low-power embedded hardware accelerators, called neural accelerators, has emerged that is tailored to, and highly energy-efficient at, executing ML inference tasks [6, 21, 31, 42, 45, 50, 52–54]. These accelerators range from the ultra-low-power Arduino Nano and low-power Jetson Nano GPUs to Apple's Neural Engine for its iPhones, and have enabled the rise of edge AI, where embedded edge devices, rather than the cloud, are capable of providing low-latency ML inference tasks for applications. Since IoT devices often have many data sources from numerous sensors and support multiple edge applications, the cloud model-serving paradigm above, which was originally designed for cloud servers, is still applicable at the edge. As with cloud model

[1] Dělen means "to share" in the Dutch language.

IoTDI '23, May 9–12, 2023, San Antonio, TX, USA

ACM ISBN 979-8-4007-0037-8/23/05…$15.00
https://doi.org/10.1145/3576842.3582375

serving, edge accelerators may also service inference requests from many different applications, e.g., using different subsets of sensors.

Importantly, edge model serving differs substantially from cloud model serving in its latency, energy, and accuracy requirements: these systems must support multiple applications with widely different latency and accuracy requirements on embedded edge accelerators with limited computational and energy resources. In particular, unlike cloud servers, edge computing platforms are often embedded devices with severe resource constraints. As a result, supporting *multi-tenancy* by efficiently sharing server and accelerator resources across many IoT applications and multiple models is paramount for edge model serving. While there has been significant prior research on optimizing edge AI inference, enabling support for multi-tenancy and efficient resource sharing, which arise in model serving systems, has not been a focus of prior research. In addition, edge model serving must be *adaptive* at runtime to handle both potentially dynamic and bursty workloads that vary over time, and energy constraints due to limited battery power. Finally, edge model serving must be *flexible* enough to satisfy widely different latency and accuracy requirements from multiple applications.

To address the problem, this paper presents Dělen, a flexible, adaptive, and multi-tenant model-serving system for supporting low-latency IoT applications on edge AI platforms. Dělen is i) multi-tenant in its support for multiple concurrent applications, ii) flexible in enabling applications to specify their own latency, energy, or accuracy requirements; and iii) adaptive in enabling applications to specify policies that adapt their operation under workload variations and energy constraints. In particular, unlike cloud model serving systems, Dělen implements model serving using multi-exit DNNs rather than model selection, since they require much less memory and permit granular control over inference requests using conditional run-time execution based on application-specific latency, energy, or accuracy constraints. In designing, implementing, and evaluating Dělen, we make the following contributions.

- **Dělen Design.** We design an edge model serving platform that enables flexible and adaptive inference request execution for multi-tenant applications. The system's core leverages *conditional run-time execution* in multi-exit DNNs to expose a configurable execution criteria on high-level objectives, including energy, latency, or accuracy. Dělen's tenant applications can leverage these configurations to implement application-specific policies.
- **Adaptation Policies.** Dělen's mechanisms enable edge applications to implement a wide range of adaptive policies, including multiple recently-proposed state-of-the-art policies, and make trade-offs between energy, latency, and accuracy under both workload dynamics and energy/resource constraints. We show how adaptive policies from prior work that use the notion of Pareto frontier, max-min fairness, and energy-awareness can be implemented using Dělen's mechanisms. In addition to application adaptation, we also show how Dělen enables cross-application adaptation of resource allocations to achieve system-wide goals. Our application-specific and multi-tenant policies leverage Dělen's conditional execution framework to handle workload and system dynamics.
- **Implementation and Evaluation.** We implement a prototype of Dělen on a battery-powered Jetson Nano node and use it to conduct a detailed experimental evaluation. We evaluate Dělen by implementing multiple state-of-the-art adaptation policies using Dělen's API, and evaluating Dělen's flexibility and adaptability under different energy, workload and multi-tenancy conditions. For example, we show how using a Pareto Frontier policy prolongs 1.59× battery life while maintaining the desired accuracy.

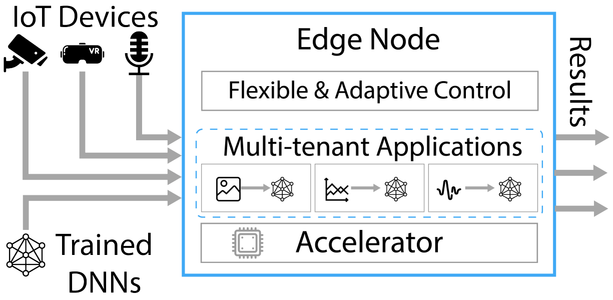


Figure 1: ***Model serving at the edge.***

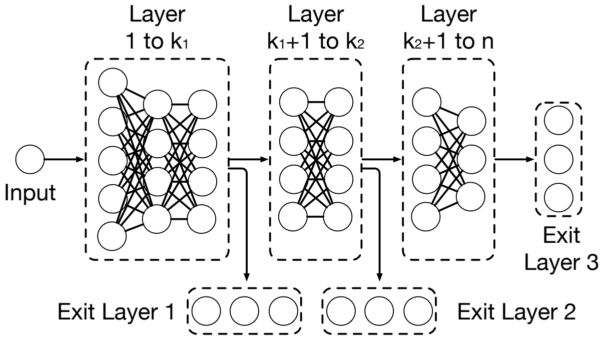


Figure 2: ***Multi-exit DNN architecture***

2 BACKGROUND

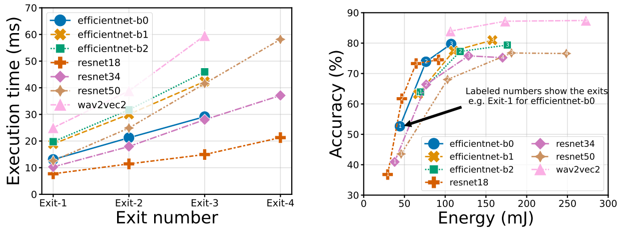
This section provides background on model serving, edge AI, edge accelerators and multi-exit deep neural networks (DNNs).

2.1 Model Serving

Model serving [19, 51] is a computing paradigm for hosting trained machine learning models and providing inference services on these models through a well-defined interface so that applications can easily incorporate ML functionality. Figure 1 shows a generic framework for model serving where IoT devices use various ML models to run the inference workloads. Model serving has become popular in cloud settings due to the growing availability of pre-trained models in domains, such as computer vision (e.g., object detection models, such as YOLO [38]) and natural language processing (e.g., Bert [9]). Cloud-based model serving involves deploying such models on GPU-equipped servers and providing inference-as-a-service to multiple applications. Model serving platforms are inherently multi-tenant in nature, since each server and GPU can host and execute multiple DNN models concurrently.

2.2 Edge AI

As IoT devices have proliferated, edge AI has emerged as a means to provide low latency processing to latency-sensitive IoT applications. The emergence of low-power DNN accelerators has been a key contributor to the rise of edge AI, where DNN inference



(a) Latency vs. Exit

(b) Error Rate vs. Energy

Figure 3: *The relationship between latency, accuracy, energy for a few of the popular multi-exit DNNs.*

can be performed at the edge without relying on cloud servers. While smartphones were the early adopters of neural accelerators, they have become ubiquitous with many options emerging in recent years [3, 6, 21, 31, 42, 45, 50, 52–54]. These options vary in their energy needs and processing capabilities, allowing application-specific selection. For example, GAP8 [12] and Arduino Nano 33 [30] are suitable for single sensor lightweight DNN applications as they consume only ∼100mW power, and offer limited memory and accelerator capabilities. In contrast, the Google Coral edgeTPU [7] and the Jetson Nano GPU [35] can support multiple sensor streams and larger DNN models, as they have greater processing capabilities and larger RAM (∼4GB memory) but also consume more power (e.g., 0.5W for the EdgeTPU and 5-10W for the Jetson Nano).

Traditionally, edge AI applications use dedicated accelerators such that accelerator resources are dedicated to providing inference service to a single application. In our work, we consider edge model serving, where we seek to bring the resource-sharing benefits of cloud model sharing to edge AI environments. In this case, similar to cloud model sharing, edge server and accelerator resources are shared across multi-tenant edge applications. Since edge environments are resource-constrained, such multi-tenancy has the potential to increase the utilization of scarce accelerator resources.

2.3 Adaptive Model Serving via Multi-exit DNN

While the multi-tenant nature of edge model sharing provides multiplexing benefits, it also raises challenges due to the workload and system dynamics that arise in edge environments. Hence, there is a need for flexible and adaptive model sharing in such settings [11, 13–15, 17, 22, 33, 34, 43]. Adaptive cloud-based model sharing is typically achieved through model selection [39, 47]. Model selection involves loading a set of pre-trained models of various sizes and complexity and then choosing a specific model at execution time depending on the application’s latency, accuracy, and energy requirements. Adaptation can be performed by dynamically switching to a different model when these requirements change. Model selection can be easily incorporated into cloud model serving by hosting and serving various model families, where each family includes a set of models of different sizes.

While model selection can be incorporated into edge model serving as well, the resource and memory constrained nature of edge accelerators limit the number of models that can be hosted concurrently. Hence, our work focuses on multi-exit DNNs, which have the potential to mimic multiple models of a family using a smaller memory footprint, while allowing similar adaptive execution.

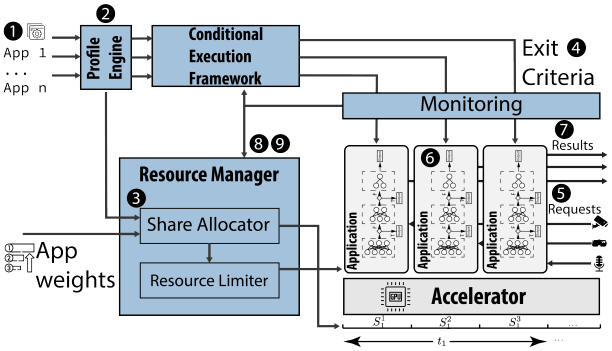


Figure 4: *Dělen’s architecture includes 1) a conditional runtime execution framework, 2) a resource manager, and 3) a profiling and monitoring engine.*

Multi-exit DNNs, originally proposed in [44], are a class of DNN models with multiple exit points. While traditional DNN models execute all layers to produce a result, multi-exit DNNs allow exits at intermediate layers and can produce a less accurate, but faster result, by taking an earlier exit. Figure 2 shows the general architecture of multi-exit DNNs. Each successive exit involves executing more layers than the previous exit and thus produces an incrementally “better” result, at the cost of increasing latency and energy consumption. Executing all layers to take the final exit produces the same result as a traditional model. Figure 3 shows the trade-off between latency, energy, and accuracy at different exits for 7 multi-exit DNNs. By dynamically deciding on which exit to use, multi-exit DNNs enable application-level adaptation [4, 23, 28, 32, 36], while consuming fewer memory resources than model selection approaches. Our work considers model serving mechanisms and policies to make adaptive trade-offs when deciding on exit points.

3 DĚLEN: DESIGN AND OVERVIEW

Our main focus in designing Dělen as an edge model-serving platform is to enable *flexible* and *adaptive* execution of inference requests for *multi-tenant* applications. We begin by describing Dělen’s key design components and subsequently provide an overview of its *flexible* and *adaptive* runtime execution workflow.

3.1 Dělen Design Components

Figure 4 shows the overall architecture for Dělen. The core of the system is its conditional runtime execution framework, which allows applications to configure various runtime execution criteria. Dělen has two additional components that facilitate the operation of its conditional runtime execution framework: a resource manager, and a profiling and monitoring module. We next describe the design and functionality of each component.

3.1.1 ***Conditional Runtime Execution Framework.*** We design our conditional execution framework as a runtime mechanism to provide applications with a configurable execution criteria across high-level objectives, such as energy, latency, or accuracy. The *flexibility* of our execution criteria enables devising application-specific policies that depend on the specific needs and high-level objectives of individual applications. Our framework also enables applications to *adapt* to changing workload dynamics by configuring their execution criteria at runtime. The *flexibility* and *adaptability* of Dělen

Parameter	Criteria, where $OP \in \{<, ==, >\}$
Response time	(response_time OP threshold)
Confidence	(confidence OP threshold)
Accuracy	(accuracy OP threshold)
Energy	(energy_used OP threshold)
Computation	(flops_used OP threshold)
Exit	(exit_number OP threshold)
Null	false /*forces full execution*/

Table 1: *Dělen exit-selection criteria.*

is embedded into the conditional runtime execution and is enabled by two key design decisions that we next outline in detail.

First, we allow applications to define criteria for different parameters that specify either resources, such as energy and computation, or high-level objectives, such as accuracy, response time, and confidence. The different parameters and relevant criteria are shown in Table 1. We expose two methods `specify_criteria()` and `combine_criteria()` to register one or more exit selection criteria and combine them using boolean operators, respectively. As an example, an application can specify (response_time > $20ms$) || (confidence > 0.7) as its criteria, which will trigger an exit-selection if the partial response time exceeds the 20ms threshold or if the result confidence exceeds 0.7. An energy cap on the execution can also be specified using the criteria, such as energy_used > $100mJ$. Similarly, an application can determine the exit number that will yield the desired accuracy using an external policy. For example, exit_number == 2, causes the second exit to be taken. The exit criteria is stored in a per-application table.

Second, we enable applications to update their exit criteria at runtime to handle changing request and workload dynamics. For example, an application-specific policy may decide to choose a later exit for a complex inference request to ensure high accuracy at the cost of energy. Similarly, another application may instead decide to choose earlier exits under high workload intensity to ensure low latency at the cost of accuracy. To enable *adaptability*, the runtime system leverages a *monitoring module* to track the current state of execution for each inference request that an adaptation policy can leverage to update the exit criteria. We note that the exit-criteria *can be selected on a per-request basis*, as well as for a given workload, and is informed by the adaptation policies presented in §4.

The *flexibility* of our framework lies in enabling a wide range of exit criteria for a given application and across applications to handle their different needs and objectives. The *adaptability* of our framework lies in allowing applications to change their exit criteria on a per-request, as well as a per-workload basis, to handle the dynamics at the inference and workload levels.

3.1.2 *Resource Manager*.

Dělen enables multiple applications to share accelerator resources, while configuring their application-specific exit criteria. It enables support for *multi-tenancy* through its resource manager component. The resource manager in-turn has two key components: the *share allocator* and the *resource limiter*.

The *share allocator*'s job is to assign shares to each application based on the initial application configuration that includes application priority, application characteristics, and the choice of fairness mechanism to use for multi-tenancy. Dělen's resource manager is work-conserving and distributes the shares of applications that are not using them to all applications. The applications can reclaim their assigned share if they start using them. We also allow applications to cooperate with other applications and allow each other to use their share when not in use. To facilitate this, we provide a mechanism to update the initially assigned shares at runtime based on the current status of the cooperative applications.

The *resource limiter*'s job is to ensure that each application respects its share and does not exceed it over a specified time window. To do so, the resource manager uses a standard token bucket algorithm for rate limiting [46]. In Dělen, the shares represent the limits for each application, while the number of time-slices assigned to each application represents the tokens. The token capacity for each application is refilled after each limiting interval. While a hard limit on resource usage can also be used, we use a token bucket algorithm to allow short bursts of workloads for individual applications. Dělen's support for *multi-tenancy* is enabled by the resource manager that allows application-specific resource limits. The resource manager also enables an additional notion of *adaptability* where the shares assigned to each application can be updated at runtime.

3.1.3 *Profiling and Monitoring Engine*.

The profiling and monitoring engine component of Dělen performs two key tasks: a one-time *profiling* of an application's characteristics and the continuous *monitoring* of applications' runtime behaviour.

The application *profiling* step allows Dělen to get information about the resource usage of the DNN model. The profiling engine executes inference requests using the model several times to gather the following profile data at the granularity of an exit: (i) FLOPs, which indicate the number of floating point operations performed by the exit, (ii) energy consumed when executing the exit at full GPU speed, and (iii) the latency to execute the exit. Each request is repeated several times, and for each exit, to gather information about each execution path. This information is used by the resource manager to assign the initial shares to each application as well as provided to the applications that leverage it to decide on the exit criteria using an application-specific adaptation policy.

The continuous *monitoring* process monitors the status of each application at runtime, which includes the current exit number, FLOPs performed, energy consumed, total time taken, and other metrics, such as confidence value. This information is used by the runtime adaptation of the exit-selection criteria and runtime adaptation of per-application shares for the multi-tenancy scenario.

3.2 Overview of Dělen Workflow

In this section, we discuss the typical steps taken by an application using Dělen at startup and at runtime. At startup, Dělen receives an application and its initial configurations (Step ❶). The profile engine uses the application code and its initial configuration to generate the profile data (Step ❷). The *share allocator* component of the resource manager then allocates shares to the application based on the profile data and the initial configurations (Step ❸). At the same time, it configures the *resource limiter* component with the share for the given application. The adaptation policy used by the application then uses the conditional execution framework to determine the initial exit criteria for the inference requests (Step ❹). At this point, the start time configuration process completes and the application starts executing on the accelerator (Step ❺).

Algorithm 1: *Per-workload adaptation with target error.*

```
Input: A list of profiled error rate Err; target error rate $e_{target}$.
n ← total number of exits ;
for $i = 1 \dots n$ do
    if Err[i] < $e_{target}$ then
        specify_criteria(exit_number == i) ;
        return
specify_criteria(exit_number == n) ;
```

Algorithm 2: *Pareto adaptation with error constraint.*

```
Input: A list of Pareto-optimal confidence criteria $\pi$; target error
       rate $e_{target}$; error profile for running policy $\pi$ with model
       $\alpha$ and dataset $\omega$, $\gamma_\alpha^\pi(\omega)$, and energy profile $\eta_\alpha^\pi(\omega)$.
$\pi^* \leftarrow \arg\min_{\pi \in \{\pi | \gamma_\alpha^\pi(\omega) < e_{target}\}} \eta_\alpha^\pi(\omega)$ ;
specify_criteria(c$_i$ > $\pi_i^*$)
```

The workflow at runtime depends on the adaptation policy used by the application. If the application uses a single exit number as a criteria for all the inference requests in a given workload, the runtime execution framework does not require active monitoring of resources. However, if the application specifies a certain criteria to be met for each inference request, such as a confidence threshold for the output, the execution framework continuously monitors the status of the application (Step ❻)). The criteria is evaluated after each exit and the output is generated once the criteria is met (Step ❼). It is possible that the criteria is not met at any of the exits; in this case, the output is generated after the last exit.

In addition to these per-application steps, Dělen's workflow has additional steps that ensure work-conserving and cooperative multi-tenancy among tenant applications. By default, Dělen monitors the status of each application and redistributes the unused shares to all the other applications to be work conserving (Step ❽). Additionally, if an application specifies their willingness to cooperate, the *share allocator* module will also recompute the weights assigned to each application to allow one application's share to be used by another application (Step ❾).

4 ADAPTATION POLICIES

In this section, we highlight Dělen's flexibility to implement a wide range of policies. To do so, we use Dělen to implement several adaptation policies from prior work. We broadly classify these policies into two categories based on their function: application-specific and multi-tenant policies. Application-specific policies work on individual applications and adapt the runtime execution based on a per-request or per-workload basis. Multi-tenant policies ensure that each of multiple concurrent applications receives their fair share of resources and can individually optimize their application-specific objectives.

4.1 Application-specific Policies

Our application-specific policies target either per-request or per-workload runtime execution adaptation. Per-request policies adapt the runtime execution for each request depending on its resource needs. Per-workload policies adapt to different workload dynamics, but apply the same criteria to all the requests in a given workload. In addition, we present additional policies that are applicable to both the individual requests and the workloads.

4.1.1 ***Per-request Adaptation Policies.*** Applications that specify high level objective, e.g., on latency, may need to adjust the amount of computation to achieve this objective. Such applications can use Dělen's conditional execution framework to define arbitrary per-request policies. An application can define one, or more, criteria for each request and the execution continues until these criteria are met. For example, prior work demonstrates that all inference requests do not have the same complexity; in this case, we can use a confidence threshold to avoid wasting computing cycles and energy [23, 36, 44]. As another example, if an application specifies a target latency, the execution greedily continues as long as the execution can go through the next exit within the latency limit.

The application can also use logical operators to combine multiple criteria. For example, an application can specify the confidence threshold in addition to the response time, which can be used to achieve higher-level secondary objectives, such as accuracy maximization, while satisfying the response time requirements [23, 36, 44]. In this case, the execution will stop once either the response time or the confidence criteria are met.

4.1.2 ***Per-workload Adaptation Policies.*** Dělen allows an application to specify criteria that consider the workload characteristics, e.g., request arrival rate, and application objectives, e.g., target error rate, and adapt the criteria as the workload dynamics changes.

The per-workload adaptation policies from prior work specify one or more targets that determine the exit layer for all the requests over some duration [4, 17, 32]. For example, an application can specify a target error rate that is used to determine the exit layer for all the requests that meet that target error rate. As shown in Algorithm 1, given a target error rate and error profile for the used model from the profiling engine, this algorithm selects the first exit that meets the target error rate. If none of the exits can satisfy the target, it specifies the last exit (i.e., executing the entire model). It then uses the `exit_number` criteria to configure the runtime execution for all the incoming inference requests. We call this adaptation policy the *per-workload static* policy.

The static nature of this policy is well-suited to a scenario where the workload dynamics, such as the request rate, do not significantly change. However, for many edge applications, the workload can experience sudden spikes of requests. In this case, the per-workload policy can adapt the criteria to handle the change in the request rate. For example, under high load, an application may decide to take a hit on the error rate to serve all requests within a target response time. On the other hand, under low load, the application can opportunistically decrease its target error rate while serving all requests within a target response time. An implementation of this policy leverages the monitoring module of Dělen to continuously monitor the incoming workload characteristics and configures the *per-workload static* policy to handle the changing dynamics. To do so, it inputs the target latency and latency profile to determine the exit number. The configuration of target latency based on the workload characteristics is application-specific. We call this variant of per-workload adaptation policy the *per-workload dynamic* policy.

4.1.3 ***Pareto Adaptation Policies.*** In this section, we describe policies from prior work that combine the desired properties of both the per-request and the per-workload adaptation policies.

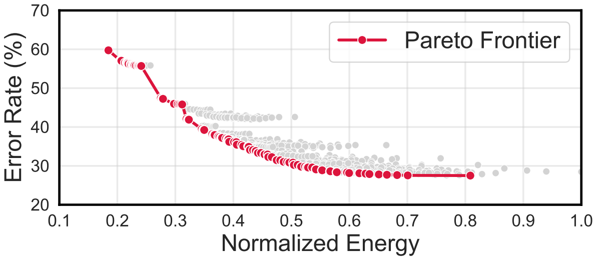


Figure 5: *Pareto Frontier of energy and accuracy for a 4-exit ResNet50. Each circle is 3-tuple containing the confidence threshold for each of the 3 possible exits.*

The per-request policies satisfy criteria for individual requests, but are oblivious to workload-level trade-offs and requirements. The per-workload policies treat all requests equally and may not meet the objectives for each individual request. The Pareto policies aim to achieve the best of both policies. They enable the high-level trade-offs of per-workload policies, while also meeting the desired criteria on a per-request basis. We note that Pareto policies are not necessarily better than either of the individual policies.

An example of such policies is the *Pareto adaptation* policy, which is inspired by prior work on the Pareto Frontier approach [26, 29]. In our *Pareto-adaptation* policy, we use the confidence as the criteria for our per-request policy. Lets assume that the confidence c_i is the maximum value of the output Softmax layer at exit i and π_i is the confidence threshold for exit i. The exit criteria for our per-request policy is defined as $c_i > \pi_i$, where i is the exit number.

To optimize the energy-error trade-off, offered by the per-workload policies, we sample a set of confidence criteria Π and apply the Pareto frontier approach [26, 29]. Specifically, given a multi-exit DNN model α, a dataset ω, and a confidence criteria $\pi \in \Pi$, we can compute the error rate, $\gamma_\alpha^\pi(\omega)$ and energy $\eta_\alpha^\pi(\omega)$. We then compute a set of Pareto optimal criteria Π^*, such that $\pi^* \in \Pi^*$ if and only if for all $\pi \neq \pi^*$, either $\gamma_\alpha^\pi(\omega) > \gamma_\alpha^{\pi^*}(\omega)$ or $\eta_\alpha^\pi(\omega) > \eta_\alpha^{\pi^*}(\omega)$.

Figure 5 shows an example of the Pareto Frontier using a 4-exit `ResNet50` DNN for energy and error rate tradeoff. We employ grid search method and sample 1000 confidence criteria. Each point represents a confidence criteria and the Pareto optimal criteria set is shown in red. As shown, the Pareto frontier represents the optimal combinations of energy-error that is achievable for a given accelerator and a DNN. With the Pareto optimal criteria set and its corresponding error and energy profile, we search a criteria in the set, which consumes the minimum energy while meeting the error constraint. This can be done efficiently in $O(\log n)$ time. The pseudocode for Pareto Adaptation is shown in algorithm 2.

Note that while we focus on the energy-error trade-off in Algorithm 2, we can generalize it to define a trade-off between any two metrics, such as latency and error. To do so, we need only replace the input profile with the profile of target metric.

Algorithm 3: *Cross-application adaptation.*

```
Input: $\lambda_i \in \Lambda$ - request rate for each application $i$; $w_i \in \mathcal{W}$ -
       weights for each application; $\mathbf{L_i} \in \mathcal{L}$ - latency profile for
       application $i$; $\beta_i \in \mathbf{F}$ - a Boolean variable indicating if an
       application $i$ participates in cooperative adaptation.
Output: S - a set specifying share for each application.
while $T$ do
    $\zeta \leftarrow$ set of cooperative applications ($\beta_i == 1$);
    if ($\zeta \neq \emptyset$) then
        A $\leftarrow$ status of each application in $\zeta$;
        $\mathcal{W} \leftarrow$ recompute_weight(A, $\Lambda$) ;
    S $\leftarrow$ max_min_fairness($\mathcal{W}, \Lambda, \mathcal{L}$);
    $l_i \leftarrow$ current service time constraint for application $i$;
    # Other policies such as Pareto Front can also be applied here;
    specify_criteria(confidence> S[$i$]) ;
    # Send S to resource limiter;
```

4.2 Multi-tenant Policies

As mentioned in §3, Dělen supports multi-tenant applications by assigning shares of available resources to applications, and employing multi-tenant adaptation policies to ensure a fair and work-conserving access of resources to each application. In our multi-tenant policy, we use the notion of max-min fairness but any other notion of fairness can be used as well [24]. Max-min fairness assigns shares of accelerator time to different applications based on their workload and weights specifying their priority. For example, if two applications get 5 and 10 requests per second, their fair shares are 33% and 66%, respectively. In a cooperative sharing scenario, the unused portions of a share can be redistributed among applications.

Our multi-tenant adaptation policy can concurrently handle both non-cooperative and cooperative applications, as shown in Algorithm 3. The parameter β indicates an application's willingness to participate in cooperative sharing and its value is set to 0 for all non-cooperative applications. The resource manager computes the shares for each application at the start based on the initially specified request rates. The value of the share for non-cooperative applications only changes, for work conservation, when some of the applications are not using their share, while other needs them. For cooperative applications, we monitor their status and periodically recompute their weights based on their current workload. In this case, the sensor that sees a higher workload is given a higher share to process its increased demands. There is a down-call from the multi-tenant adaptation policy to all the applications to update their criteria. These application can use their share to implement any of the per-application adaptation policies from §4.1.

5 DĚLEN IMPLEMENTATION

In this section, we outline how we implement the key design components of Dělen, described in §3, and additional supporting modules. We implement Dělen on a Jetson Nano that runs CUDA 10.2, cuDNN 8, and TensorRT 7.1.3. However, it can support other deep learning frameworks, such as Tensorflow [1] and Pytorch [37]. Our prototype is implemented using around 3000 lines of code (the source code for our system is publicly available at *URL blinded*). We only provide implementation details of the multi-exit DNN training engine, conditional runtime execution framework, and profiling engine as they require adapting the implementation for the model serving framework and underlying hardware.

Multi-exit DNN Training Engine. While a large number of pretrained DNNs are available for common tasks, such as image classification, object detection, and speech recognition through various model repositories [10], multi-exit versions of the standard DNN models are generally not available. Since we leverage multi-exit DNNs, we also design and implement a PyTorch-based training engine to convert a standard DNN model into a multi-exit version.

The state-of-the-art DNN architectures [18, 41] use the concept of a neural block, which is a pattern of layers that are used repeatedly to build the DNN. To build a multi-exit DNN from a standard DNN, our training engine adds classification layers (e.g., linear layers) on top of some intermediate blocks and uses the intermediate output of the block for classification as an early exit. In the training process, the application specifies the desired number of exits for the conversion. A lower number of exits has a small training overhead but it provides less options for adaptive execution based on high-level objectives and resource availability. On the other hand, a large number of exits adds a significant training overhead but offers a significantly higher flexibility for adaptive execution.

Furthermore, unlike standard DNNs, the training engine needs to compute the loss and train all exits in the multi-exit case. Instead of training the exits one by one, our training engine uses weighted cumulative loss similar to MSDNet [20] and trains all exits at once,

$$L_{total} = \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},y)\in\mathcal{D}} \sum_{i=1}^{N} w_i L(f_i(\mathbf{x}), y).$$

where $\mathcal{B}$ is the mini-batch, $\mathbf{x}$ is the input data and y is the ground truth label. L is the loss function for each exit. In this work, we use a cross-entropy loss function for all exits. $f_i(\mathbf{x})$ is the output of exit i and $w_i > 0$ is the weight of that exit. Although prior work [20] suggests using the same weight for all exits, we empirically found that this can hurt the performance of later exits and makes them *less* accurate than earlier exits. Since our goal is to incrementally improve accuracy with each successive exit, our training engine gives higher weights to later exits to ensure monotonic improvements.

Conditional Runtime Execution Framework. TensorRT's runtime does not support multi-exit DNN execution and we implement runtime support for multi-exit DNNs and conditional execution. After training a model, the runtime system converts each n-exit DNN into n-sub-networks. The inputs and outputs of these sub-networks share the same buffer. To load a DNN model on the GPU, the runtime system loads all of its sub-networks independently in TensorRT and then runs each of them on demand. The conditional execution allows the exit criteria to be evaluated at the exit block boundaries, which enables an early exit to be taken.

Profiling Engine. As discussed in §3, the profiling engine gathers the resource usage information for the application DNN. In our implementation, we first compile the multi-exit DNN model using the TensorRT compiler in the native format to run it using TensorRT—Nvidia's low-level runtime framework for Jetson Nano GPUs. The profiling engine runs a configurable number of inference requests, termed M, at each exit i for N times. The value of M depends on the dataset used, while we configure the value of N to be one as our data set is large enough to get reliable profile data.

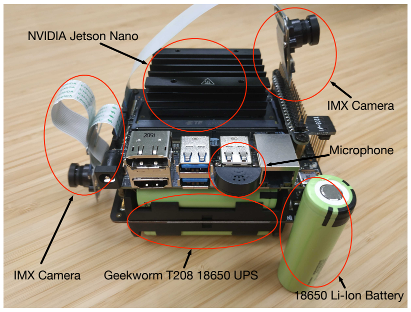


Figure 6: *Dělen prototype implemented on Jetson Nano in battery-powered configuration.*

Models	# Exits	GFLOPS	Energy (mJ)	Accuracy
ResNet18	4	0.42	107	0.7450
ResNet34	4	3.68	158	0.7521
ResNet50	4	4.14	176	0.7657
EfficientNet-B0	3	0.42	92	0.7971
EfficientNet-B1	3	0.61	170	0.8097
EfficientNet-B2	3	0.71	249	0.7936
wav2vec2	3	2.51	273	0.8744

Table 2: Characteristics of our multi-exit DNNs.

6 EXPERIMENTAL EVALUATION

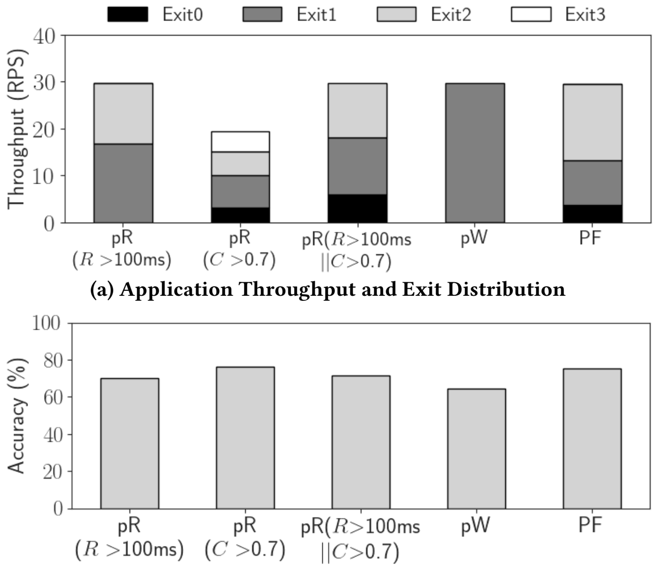
In this section, we describe our experimental setup and present the results for our various adaptation policies using our prototype.

6.1 Experimental Setup

Dělen Prototype. Our experimental setup comprises the Jetson Nano node in Figure 6 running Dělen's implementation described in §5. The Jetson Nano is equipped with a Quad-core ARM A57 CPU, a 128-core Maxwell GPU, and a 4GB RAM shared between the CPU and the GPU. The node runs Ubuntu 18.04, CUDA 10.2, CuDNN 8, and TensorRT 7.1.3. As Figure 6 shows, it is battery powered using 6 rechargeable batteries with a capacity of 3400MAh each.

DNN Models. We use seven DNN models from three popular DNN families to process image and speech data, as these comprise most of the edge applications. Specifically, we use `EfficientNet`[41] and `ResNet`[18] for image classification tasks and `wav2vec2`[2] for speech recognition tasks. For `ResNet` and `EfficientNet`, we choose `ResNet18`, `34`, `50` and `EfficientNet-B0`, `B1`, `B2` to represent a small, medium and large model, respectively. For the `wav2vec2` model, we reduce the convolutional dimension from 512 to 128 and the number of encoder layers from 12 to 6 to improve execution efficiency. We used our custom DNN training engine to train and create multi-exit versions of these DNN models. We used the profiling engine to gather detailed resource usage information about each model, which is summarized in Table 2.

Training Details: We used our custom DNN training engine to create multi-exit versions of all seven DNN models. We used the Food-101 dataset [5] for training the `EfficientNet` and `ResNet` models, and the Speech Commands dataset [49] for training the `wav2vec2` model. We applied transfer learning and loaded pre-trained weights for ImageNet before training. All models are trained using the Adam optimizer [25] with a learning rate of 10^{-3}. We trained the models on an off-the-edge device using two NVIDIA GeForce GTX 1080 Ti GPUs with a mini-batch size of 128 for 50 epochs.



(a) Application Throughput and Exit Distribution

(b) Achievable Accuracy

Figure 7: *The application throughput, exit distribution, and accuracy for* `per-request` *(pR),* `per-workload` *(pW), and* `Pareto-frontier` *(PF) adaptation policies. Dělen runs* `ResNet50` *and has a workload of 30 requests/second.*

IoT workloads: The workload for our experiments is generated by multiple concurrent sensing applications that generate sensor data either from the hardware sensors or from a pre-generated sensor trace stored on the file system. The IoT inference workload used in each experiment is described in the corresponding sections.

6.2 Dělen's Flexibility

In this section, we demonstrate the flexibility of Dělen in implementing a wide range of application-specific policies that configure the exit-selection criteria based on either high-level objectives, such as latency, confidence, and accuracy (§6.2.1), or resource constraints, such as energy-efficiency (§6.2.2).

6.2.1 ***Adaptation policies in action.*** In Figure 7, we demonstrate Dělen's flexibility by implementing and evaluating multiple adaptation policies from §4 using a 4-exit `ResNet50` fed at 30 requests/second. We run three different variants of the `per-request` policy that use response time (>100ms), confidence threshold (>0.7), and both as the exit selection criteria. We also evaluated our `per-workload` and `Pareto-frontier` policies, which try to maximize the accuracy for a given inference request rate.

We observe in Figure 7a that policy performance changes between various exits in different ways. For instance, the `per-workload` policy selects a single exit that fulfills the request rate. The `per-request`(C>0.7) policy is always attempts to achieve the confidence threshold and often opts for higher exits which requires more time and limits application throughput. This issue is mitigated when the confidence criteria is combined with the response time or using the `Pareto-frontier` policy that changes the thresholds according to the rate.

We next highlight the impact of decisions made by different policies on the accuracy of inference requests in Figure 7b. As expected, the fixed selection of an exit for the `per-workload` yields the lowest accuracy, while the `per-request` prioritizing only confidence achieves the highest accuracy at the cost of low throughput. In contrast, the `per-request` that prioritizes response time either sacrifices accuracy to achieve lower response time or unnecessarily achieves higher accuracy for a handful of requests. Interestingly, the combination of these policies avoids their individual pitfalls and processes all requests with a slight accuracy hit. Finally, we show that the `Pareto-frontier` is able to use the confidence criteria knowledge to find the best configurations that adhere to time constraints as well as increases accuracy.

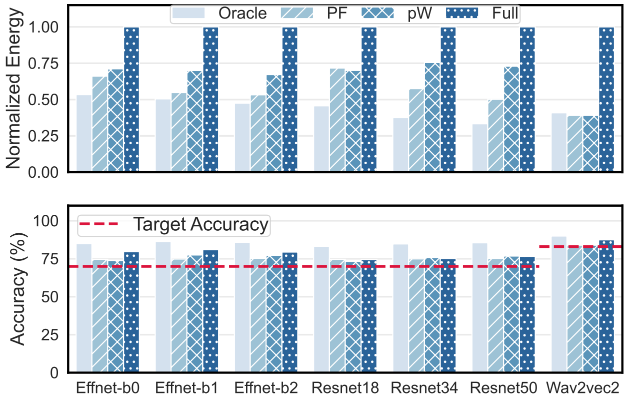


Figure 8: *The performance of optimal exit (oracle), Pareto frontier (PF), per-workload (pW), and last exit (Full) policies in optimizing energy-efficiency under accuracy constraint.*

6.2.2 ***Optimizing for energy efficiency.*** We next demonstrate the flexibility of Dělen in optimizing for energy-efficiency, while also respecting accuracy constraints. In addition to the `per-workload` and `Pareto-frontier` policies, we implement two new policies: the *Oracle* policy and the *Full* policy. The *Oracle* is a `per-request` policy that takes the first exit which produces the correct answer on a per-request basis. If the correct answer is not found, it chooses the first exit in the model and produces the output. This is the optimal policy that maximizes accuracy while minimizing energy consumption. The *Full* policy is implemented using the `per-workload` framework with the last exit as the exit number for all the inference requests.

Figure 8 shows the energy usage and accuracy for different policies under the accuracy constraint of 70% for image classification (e.g., `EfficientNet` and `ResNet`) and 83% for speech recognition (e.g., `wav2vec2`) tasks. We run the models with the entire test datasets, with a batch size of 1, while measuring the energy consumption and accuracy. As expected, the *Oracle* policy achieves the highest accuracy for all DNNs and the lowest energy for all DNNs except `Wav2Vec2`, where it has a comparable energy usage with other policies.

The *Full* policy achieves the second highest accuracy, which comes at the cost of having the highest energy footprint. The `per-workload` policy, on average, uses 29.94% less energy than *Full* policy. Finally, the `Pareto-frontier` policy consumes up to 38.7% less energy than the `per-workload`, with a mean reduction of 20.1%. Also, it consumes up to 60.9% less energy than *Full*, with a mean reduction of 43.9%. For `ResNet18`, the energy consumption of the `Pareto-frontier` policy is slightly higher than that of the

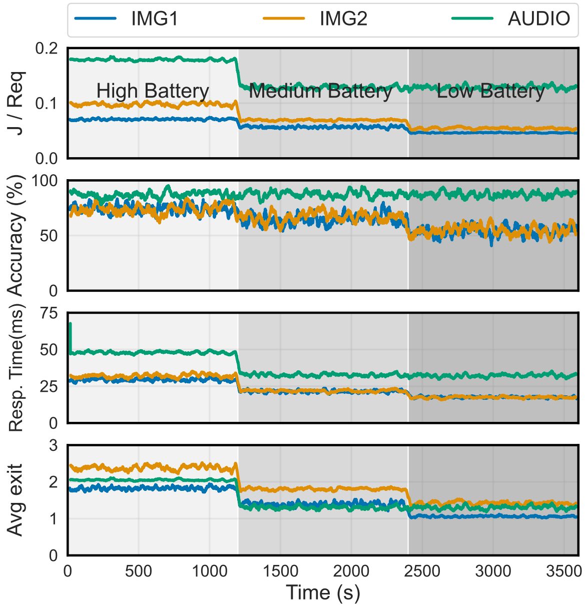


Figure 9: *The performance of* `Pareto-frontier` *policy in adapting to different battery dynamics by minimizing energy usage under accuracy constraints.*

`per-workload` policy. This is because the model is too small to yield a high confidence output at early exits, and thus more inputs are passed to later exits to achieve high accuracy, which increases energy consumption. All policies achieve higher accuracy than the constraints, as specified by the red dash lines.

Key Takeaways. *The flexibility of Dělen's conditional runtime execution framework allows applications to define a wide range of per-request and per-workload execution criteria including accuracy, response time, confidence, and energy-efficiency.*

6.3 Dělen's Adaptability

We next demonstrate the adaptability of Dělen in handling varying resource dynamics, such as battery levels (§6.3.1), and workload characteristics, such as handling changing request rates (§6.3.2).

6.3.1 ***Adapting to battery dynamics.*** In this experiment, we demonstrate how applications can leverage Dělen to optimize energy efficiency with accuracy constraints under varying battery dynamics that are ever-present at the edge. This adaptation is useful for controlling the depletion rate of battery and vital for ensuring a long battery life. We run three applications, `IMG1`, `IMG2`, and `AUDIO`, that use `EfficientNet-B0`, `ResNet34`, and `wav2vec2`, respectively. We change the battery level from high to medium to low. We leverage Dělen's flexibility to configure the `Pareto-frontier` policy accuracy constraints of 70%, 60%, and 50% for the two `IMG` tasks and 89%, 86% and 83% for the `AUDIO` task, under high, medium, and low battery conditions, respectively.

Figure 9 shows the runtime adaptation of the `Pareto-frontier`, where all three DNN models use more energy per request to achieve higher accuracy during the high battery period. As the battery level drops to medium, applications adapt to using less energy and lower accuracy by opting for earlier exits. At the lowest battery level, applications conserve energy by further reducing energy usage at the cost of further decreasing accuracy. We note that the `AUDIO` application sees less degradation than the two `IMG` because the accuracy at different exits for `AUDIO` is very similar (see Figure 3).

Figure 10 illustrates the benefits of Dělen on energy-efficiency and the node's battery life by showing the battery's state of charge over time. We use the experimental setup from the previous figure, but set the request rate to 10 RPS for all tasks and allow the battery to discharge until a 10% state of charge. In addition to the `Pareto-frontier`, we evaluate the performance of the `per-workload` and *Full* policies from §6.2.2. As shown, the battery life for the *Full*, `per-workload`, and `Pareto-frontier` policies are 4.02, 4.56, and 6.42 hours, respectively. The `Pareto-frontier` policy allows the battery to last 1.41× longer than `per-workload` and 1.59× longer than *Full*.

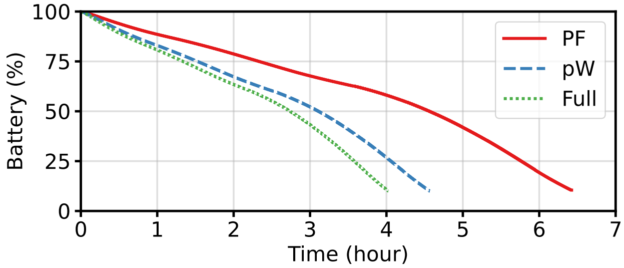


Figure 10: *Battery levels for different policies over time.*

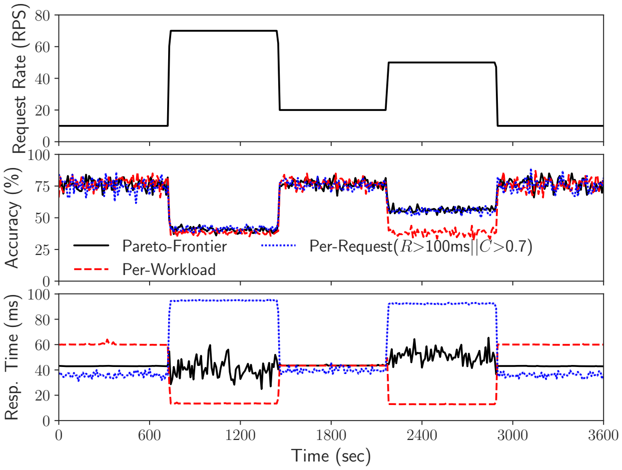


Figure 11: *Adapting to workload dynamics at runtime using combined* `per-request`*,* `per-workload`*, and* `Pareto-frontier` *policies while using* `ResNet50` *for image classification tasks.*

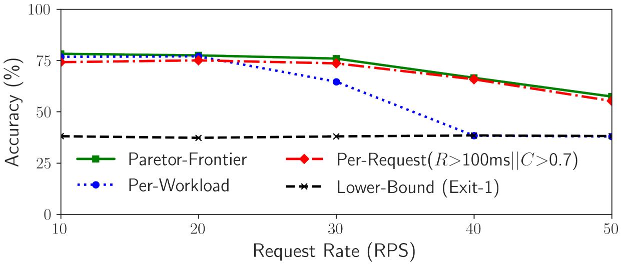


Figure 12: *The accuracy of inference for various runtime adaptation policies shown in Figure 11.*

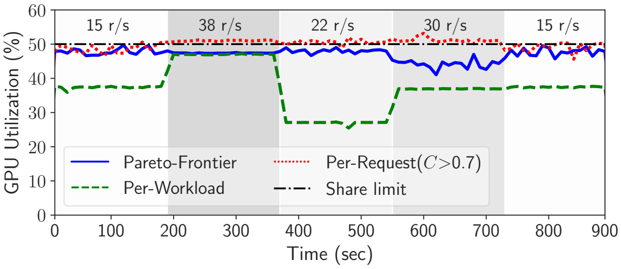


Figure 13: *The effect of limiting resources to an individual application across different request rates.*

6.3.2 ***Adapting to workload dynamics.*** We next evaluate the efficacy of Dělen in adapting to changing workload dynamics that are inherent to many edge IoT applications. Figure 11 shows the effects of changes in the request rate on accuracy and response time. As shown, irrespective of the policy, Dělen was able to instantly adapt to the workload changes as it updates the execution criteria every second. The overhead of such frequent updates is negligible and discussed in §6.5. However, despite adaptation, the impact on accuracy and response time is not the same across all policies. For instance, the conservative nature of the per-workload policy decreases the accuracy equally for both medium and high request rates, but other policies were able to better cope with the medium request rates. Figure 11(bottom) shows the average response time. The behaviour of the combined per-request policy adapts to different request rates. At low rates, it relies more on the confidence threshold, but at high rates it uses the response-time threshold as an exit strategy.

To further quantify the accuracy implications of different policies with workload changes, we gradually increase the workload in steps from 10 RPS to 50 RPS with the objective of maximizing accuracy. Dělen monitors the input rate and notifies the application to adapt its behavior. Figure 12 shows the policies' accuracy and input rate trade-off in executing a 4-exit ResNet50 workload. Here, we use the policies from the previous experiment. We also add another per-workload (Exit==1) as a lower-bound on accuracy. As shown, the conservative nature of the per-workload policy resulted in lower accuracy than other policies and it ends up choosing the first exit at 40 RPS. The other policies overcome this limitation by choosing the current exit if the confidence is above the threshold or the response time limit is hit. The Pareto-frontier policy makes better decisions due to its prior knowledge of the relationship between confidence thresholds, processing time, and accuracy.

Key Takeaway. *The adaptive runtime execution of Dělen allows applications to adapt to changing resource and workload dynamics while achieving high-level objectives.*

6.4 Delen's Multi-tenancy

In this section, we demonstrate Dělen's ability to allocate and restrict shares assigned to individual (§6.4.1) or multiple concurrent applications (§6.4.2) in a fair and work-conserving manner.

6.4.1 ***Restricting per-application resources.*** As mentioned in §3 Dělen supports limiting per-application resources to ensure that applications adhere to their assigned resource share. Resource limitations are an integral part of how Dělen enables multi-tenancy of applications. To reiterate, each application is assigned a certain number of tokens depending on its share. Once an application exhausts its tokens, Dělen will either delay or drop the requests depending on the applications' configuration. While Dělen limits the resources, policies themselves can be oblivious or aware of their assigned share. For example, both the Pareto-frontier and the per-workload policies consider the request rate and time assigned per request when configuring the exit-selection criteria. However, the per-request policies do not consider the workload characteristics[2].

Figure 13 shows the resource usage of a single application when limited to using only 50% of the system-level resources. We vary the request rate to evaluate the ability of Dělen's resource limiter in limiting the usage of applications that inherently respect their shares and for those which do not. As shown, the per-workload policy and the Pareto-frontier, which are mindful of the resource limit, do not reach the limit. However, the per-request(C>0.7) in this case, attempts to surpass the limit, but the resource limiter restricts the application's usage to below the limit.

6.4.2 ***Ensuring fair multi-tenancy of applications.*** We next demonstrate Dělen's ability to support multi-tenancy, while ensuring fairness, in scenarios where applications operate in a cooperative or a non-cooperative manner using MAX-MIN fairness. Figures 14 and 15 shows how conditional execution can be used to adapt the application's behavior for cooperative and non-cooperative sharing. We use three applications IMG1, IMG2, and AUDIO. Initially, the three applications use Resnet 34 at 15 FPS, Efficientnet-B0 at 15 FPS, and wav2vec2 at 5 samples/sec, respectively. All three applications use the Pareto-frontier policy and are assigned a weighted share based on their initial workload. At $t = 20$ min, the IMG1 application sees a workload burst where its request rate rises to 30 FPS for the next 20 minutes and then returns back to 15 FPS.

Non-cooperative sharing uses local adaptation where the assigned shares do not change in the face of workload changes, and each application makes local adaptation decisions to react to changes. In this case, IMG1 application adapts to the workload increases at $t = 20$ by decreasing its accuracy and request time (to maintain the same time-share). As shown in Figure 14, IMG1 rate doubling is translated to halving the response time and energy and a decrease in accuracy by 30%. Also, the other two applications do not see any impact of these burst seen by IMG1.

In contrast, cooperative sharing is a global adaptation approach where both the MAX-MIN shares and the conditional execution criteria are modified to react to workload dynamics. In this case, when IMG1 sees a higher workload, the weighted shares of all three applications are recomputed. IMG1 gets a relatively higher share while the relative share of IMG2 and AUDIO falls. Further, all three applications react to the change in their share according to their policy, Pareto-frontier in this case, and recomputes request time. As seen in Figure 15, this causes a small degradation in response time and accuracy for all three, while IMG1 is better able to absorb the workload spike by a smaller drop in the accuracy. The two policies show how MAX-MIN fairness and conditional execution enable different behaviors depending on system's goals.

[2] A malicious application can surpass its limit by issuing many requests. However, analysis of security aspects and mitigation strategies is out of scope for this paper.

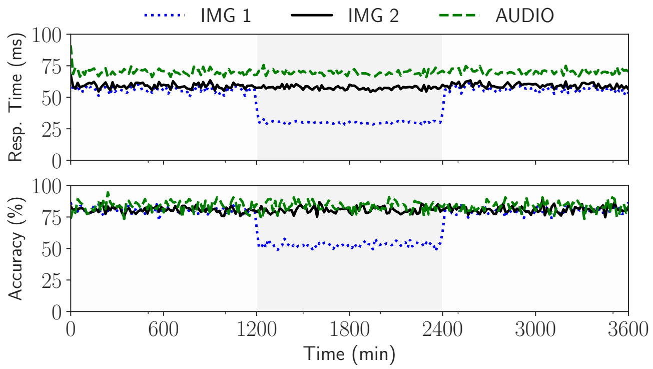


Figure 14: ***Non-Cooperative local MAX-MIN adaptation.***

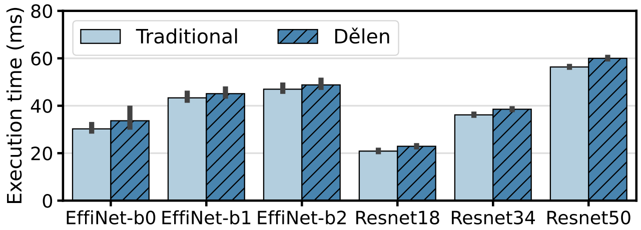


Figure 16: ***The overhead of Dělen's conditional execution framework compared to traditional (single exit) approach.***

Key Takeaways. *Dělen can restrict the resource usage of individual applications and leverages that mechanism to enable multi-tenancy while allowing each application to flexibly configure their exit selection criteria and adapt to workload dynamics at runtime.*

6.5 Delen's Overheads

Finally, we evaluate overheads imposed by the components of our systems. Dělen's system overhead comes from two operations: 1) updating the exit criteria and 2) evaluating the exit criteria. Table 3 depicts these overheads. As can be seen, updating overhead depends on the policy. For some simple policies, such as Algorithm 1, the updating overhead is negligible. For complex policies, such as `Pareto-frontier`, it takes up to a few milliseconds. However, the results can be cached as the pareto-frontier is assumed to fixed throughout the application's lifetime. In this case, the overall updating overhead is negligible. Dělen's main overhead comes from evaluating the exit criteria. To enable early exits, extra classification layers are inserted into the DNN. Each of these classification layers takes ~ 1ms to execute and 100μs to evaluate the specified boolean criteria. The more exits we go through, the higher the evaluation overhead. Figure 16 shows the full model execution time comparison between Dělen multi-exit and traditional (single-exit) models. The figure shows the *worst* case evaluating overhead for each request. As shown, the mean *worst* case overhead is 6.9%.

7 CONCLUSIONS

In this paper, we presented the design, implementation, and evaluation of Dělen, which is a flexible, adaptive, and multi-tenant model-serving system for supporting low-latency IoT applications on edge AI platforms. Dělen exposes a high-level API that enables individual edge applications to specify a bound at runtime on the latency, accuracy, or energy of their inference requests. We evaluated Dělen's flexibility by implementing state-of-the-art adaptation policies using its API, and evaluated its adaptability under different workload dynamics and goals when running single and multiple concurrent applications.

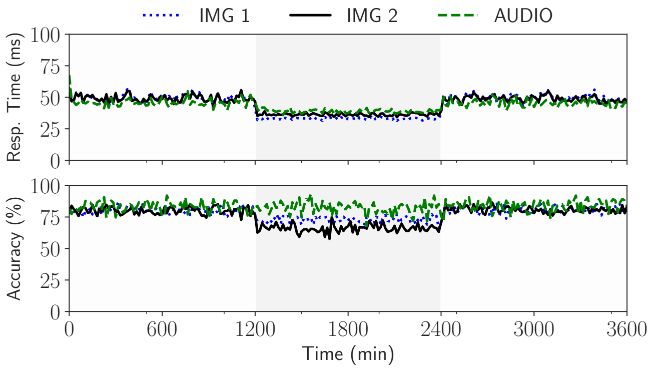


Figure 15: ***Cooperative global MAX-MIN adaptation.***

Evaluation Metric	Time Overhead
Classifier Network Evaluation	1 ms
Exit Criteria Evaluation	100 μs
Update Criteria (PF)	2.6 ms
Update Criteria (others)	70 μs

Table 3: ***Dělen's overheads.***

ACKNOWLEDGEMENTS

We thank the IoTDI reviewers for their valuable comments, which improved the quality of this paper. This research is supported by NSF grants 2211302, 2211888, 2213636, 2105494, US Army contract W911NF-17-2-0196, Adobe and Amazon Web Services.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Alexei Baevski, H. Zhou, Abdel rahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. *ArXiv* abs/2006.11477 (2020).

[3] Brendan Barry, Cormac Brick, F. Connor, David Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma. 2015. Always-on Vision Processing Unit for Mobile Applications. *IEEE Micro* 35 (2015), 56–66.

[4] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. 2017. Adaptive Neural Networks for Efficient Inference. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 527–536.

[5] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. 2014. Food-101 – Mining Discriminative Components with Random Forests. In *European Conference on Computer Vision*.

[6] Qingqing Cao, Noah Weber, Niranjan Balasubramanian, and Aruna Balasubramanian. 2019. DeQA: On-Device Question Answering. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. 27–40. https://doi.org/10.1145/3307334.3326071

[7] S. Cass. 2019. Taking AI to the edge: Google's TPU now comes in a maker-friendly package. *IEEE Spectrum* 56 (2019), 16–17.

[8] D. Crankshaw, G. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov. 2020. Inferline: Latency-aware Provisioning and Scaling for Prediction Serving Pipelines. In *SoCC*.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[10] Open Neural Network Exchange. 2021. *ONNX model zoo.* https://github.com/onnx/models

[11] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) *(MobiCom '18)*. 115–127. https://doi.org/10.1145/3241539.3241559

[12] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. 2018. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 1–4. https://doi.org/10.1109/ASAP.2018.8445101

[13] Jason Flinn, Soyoung Park, and Mahadev Satyanarayanan. 2002. Balancing performance, energy, and quality in pervasive computing. *Proceedings 22nd International Conference on Distributed Computing Systems* (2002), 217–226.

[14] Jason Flinn and M. Satyanarayanan. 1999. Energy-Aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 48–63. https://doi.org/10.1145/319151.319155

[15] Peizhen Guo, Bo Hu, and Wenjun Hu. 2021. Mistify: Automating DNN Model Porting for On-Device Inference at the Edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 705–719. https://www.usenix.org/conference/nsdi21/presentation/guo

[16] M. Halpern, B. Boroujerdian, T. Mummert, E. Duesterwald, and V. Reddi. 2019. One Size Does Not Fit All: Quantifying and Exposing the Accuracy-latency Trade-off in Machine Learning Cloud Service APIs via Tolerance Tiers. In *ISPASS*.

[17] Walid A. Hanafy, Tergel Molom-Ochir, and Rohan Shenoy. 2021. Design Considerations for Energy-efficient Inference on Edge Devices. In *The Twelfth ACM International Conference on Future Energy Systems (e-Energy '21)* (Virtual Event, Italy). 7 pages. https://doi.org/10.1145/3447555.3465326

[18] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 770–778.

[19] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. 2021. Rim: Offloading Inference to the Edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation* (Charlottesvle, VA, USA) *(IoTDI '21)*. Association for Computing Machinery, New York, NY, USA, 80–92. https://doi.org/10.1145/3450268.3453521

[20] Gao Huang, Danlu Chen, T. Li, Felix Wu, L. V. D. Maaten, and Kilian Q. Weinberger. 2017. Multi-Scale Dense Convolutional Networks for Efficient Prediction. *ArXiv* abs/1703.09844 (2017).

[21] 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* (Niagara Falls, New York, USA) *(MobiSys '17)*. Association for Computing Machinery, New York, NY, USA, 82–95. https://doi.org/10.1145/3081333.3081360

[22] Nitthilan Kanappan Jayakodi, Syrine Belakaria, Aryan Deshwal, and Janardhan Rao Doppa. 2020. Design and Optimization of Energy-Accuracy Tradeoff Networks for Mobile Platforms via Pretrained Deep Models. *ACM Trans. Embed. Comput. Syst.* 19, 1, Article 4 (Feb. 2020), 24 pages. https://doi.org/10.1145/3366636

[23] Nitthilan Kannappan Jayakodi, Anwesha Chatterjee, Wonje Choi, Janardhan Rao Doppa, and Partha Pratim Pande. 2018. Trading-Off Accuracy and Energy of Deep Inference on Embedded Systems: A Co-Design Approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2881–2893. https://doi.org/10.1109/TCAD.2018.2857338

[24] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. 1998. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *The Journal of the Operational Research Society* 49, 3 (1998), 237–252. http://www.jstor.org/stable/3010473

[25] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2015).

[26] Stefanos Laskaridis, Stylianos I. Venieris, Hyeji Kim, and Nicholas D. Lane. 2020. HAPI: Hardware-Aware Progressive Inference. *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (2020), 1–9.

[27] Y. Lee, A. Scolari, B. Chun, M. Santambrogio, M. Weimer, and M. Interlandi. 2018. PETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *OSDI*.

[28] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. 2020. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Transactions on Wireless Communications* 19, 1 (2020), 447–457. https://doi.org/10.1109/TWC.2019.2946140

[29] Christopher A. Mattson and Achille Messac. 2005. Pareto Frontier Based Concept Selection Under Uncertainty, with Visualization. *Optimization and Engineering* 6, 1 (2005), 85–115. https://doi.org/10.1023/B:OPTE.0000048538.35456.45

[30] David Mellis, Massimo Banzi, David Cuartielles, and Tom Igoe. 2007. Arduino: An open electronic prototyping platform. In *Proc. Chi*, Vol. 2007. 1–11.

[31] Niluthpol Chowdhury Mithun, Sirajum Munir, Karen Guo, and Charles Shelton. 2018. ODDS: Real-Time Object Detection Using Depth Sensors on Embedded GPUs. In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 230–241. https://doi.org/10.1109/IPSN.2018.00051

[32] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. 2020. EPerceptive: Energy Reactive Embedded Intelligence for Batteryless Sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems* (Virtual Event, Japan) *(SenSys '20)*. 382–394. https://doi.org/10.1145/3384419.3430782

[33] Dushyanth Narayanan and Mahadev Satyanarayanan. 2003. Predictive Resource Management for Wearable Computing. In *MobiSys '03*.

[34] Brian D. Noble, Mahadev Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. 1997. Agile application-aware adaptation for mobility. *Proceedings of the sixteenth ACM symposium on Operating systems principles* (1997).

[35] Nvidia. 2020. *NVIDIA Jetson Modules.* Retrieved October 19, 2020 from https://developer.nvidia.com/embedded/jetson-modules

[36] Priyadarshini Panda, Abhronil Sengupta, and Kaushik Roy. 2016. Conditional Deep Learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 475–480.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035.

[38] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 779–788.

[39] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[40] Mahadev Satyanarayanan and Nigel Davies. 2019. Augmenting Cognition Through Edge Computing. *Computer* 52, 7 (2019), 37–46. https://doi.org/10.1109/MC.2019.2911878

[41] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *ArXiv* abs/1905.11946 (2019).

[42] Tianxiang Tan and Guohong Cao. 2021. Efficient Execution of Deep Neural Networks on Mobile Devices with NPU. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (Co-Located with CPS-IoT Week 2021)* (Nashville, TN, USA) *(IPSN '21)*. 283–298. https://doi.org/10.1145/3412382.3458272

[43] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive Deep Learning Model Selection on Embedded Systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Philadelphia, PA, USA) *(LCTES 2018)*. 31–43.

[44] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*. 2464–2469. https://doi.org/10.1109/ICPR.2016.7900006

[45] Camill Trueeb, Carmelo Sferrazza, and Raffaello D'Andrea. 2020. Towards vision-based robotic skins: a data-driven, multi-camera tactile sensor. In *2020 3rd IEEE International Conference on Soft Robotics (RoboSoft)*. 333–338. https://doi.org/10.1109/RoboSoft48309.2020.9116060

[46] J. Turner. 1986. New directions in communications (or which way to the information age?). *IEEE Communications Magazine* 24, 10 (1986), 8–15.

[47] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. ALERT: Accurate Learning for Energy and Timeliness. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 353–369.

[48] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2019. Towards Scalable Edge-Native Applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing* (Arlington,

Virginia) *(SEC '19)*. Association for Computing Machinery, New York, NY, USA, 152–165. https://doi.org/10.1145/3318216.3363308

[49] Pete Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *ArXiv* abs/1804.03209 (2018).

[50] Hao Wu, Jinghao Feng, Xuejin Tian, Edward Sun, Yunxin Liu, Bo Dong, Fengyuan Xu, and Sheng Zhong. 2020. EMO: Real-Time Emotion Recognition from Single-Eye Images for Resource-Constrained Eyewear Devices. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) *(MobiSys '20)*. 448–461. https://doi.org/10.1145/3386901.3388917

[51] Xiaorui Wu, Hong Xu, and Yi Wang. 2020. Irina: Accelerating DNN Inference with Efficient Online Scheduling *(APNet '20)*. Association for Computing Machinery, New York, NY, USA, 36–43. https://doi.org/10.1145/3411029.3411035

[52] Mengwei Xu, Xiwen Zhang, Yunxin Liu, Gang Huang, Xuanzhe Liu, and Felix Xiaozhu Lin. 2020. Approximate Query Service on Autonomous IoT Cameras. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) *(MobiSys '20)*. 191–205. https://doi.org/10.1145/3386901.3388948

[53] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. 2020. NEMO: Enabling Neural-Enhanced Video Streaming on Commodity Mobile Devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom) *(MobiCom '20)*. Article 28, 14 pages. https://doi.org/10.1145/3372224.3419185

[54] Juheon Yi, Sunghyun Choi, and Youngki Lee. 2020. EagleEye: Wearable Camera-Based Person Identification in Crowded Urban Spaces. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom) *(MobiCom '20)*. Article 4, 14 pages. https://doi.org/10.1145/3372224.3380881

[55] C. Zhang, M. Yu, W. Wang, and F. Yan. 2019. Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference. In *USENIX ATC*.

[56] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *HotCloud*.