Elastic Execution of Multi-Tenant DNNs on Heterogeneous Edge MPSoCs

Soroush Heidari
Arizona State University
sheidar1@asu.edu

Mehdi Ghasemi
Southern Illinois University
mehdi.g@siu.edu

Young Geun Kim
Korea University
younggeun_kim@korea.ac.kr

Carole-Jean Wu
Meta Inc.
carolejean.wu@gmail.com

Sarma Vrudhula
Arizona State University
vrudhula@asu.edu

2024 IEEE/ACM Symposium on Edge Computing (SEC) | 979-8-3503-7828-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SEC62691.2024.00029

***Abstract*—The growing complexity of machine learning (ML) tasks drives the rapid deployment of *multi-tenant* ML workloads at the edge presenting unique challenges due to the variable computational demands and strict latency requirements. This paper introduces a holistic elastic scheduler, *EMERALD*, designed to optimize the execution of multi-tenant machine learning (ML) workloads on heterogeneous edge (Multiprocessor System on Chip) MPSoCs under strict runtime constraints. *EMERALD* employs input resolution scaling to dynamically adjust the computational demands of deep neural networks (DNNs), thereby enhancing the ability to meet stringent latency requirements while maintaining high accuracy. The scheduler consists of two main components: a local greedy scheduler and a global scheduler. The local scheduler actively manipulates input resolution in response to deadline violations, selecting the resolutions that minimally impact accuracy and maximally reduce response time. The global scheduler, an Integer Linear Programming (ILP)-based scheduler, fine-tunes the decisions of the local scheduler by considering factors such as DNN dependencies, scene complexity, hardware heterogeneity, and the trade-offs between accuracy and makespan associated with input scaling adjustments. This hierarchical approach allows *EMERALD* to effectively balance computational efficiency and accuracy, significantly reducing missed deadlines—achieving 11x and 12.3x fewer missed deadlines compared to *CAMDNN* and *HEFT*, respectively, in scenarios demanding 30 frames per second. The results underscore the critical role of adaptive input scaling in managing the complexities of edge-based ML deployments.**

***Index Terms*—DNN Serving, Multi-tenant DNN, Elastic Scheduling, Edge Processing, Heterogeneous Edge**

I. Introduction

The growing complexity of applications utilizing machine learning techniques is leading to the emergence of *multi-tenant* workloads. These are networks of customized deep neural networks (DNNs) where models are executed both serially and concurrently. They are frequently utilized in emerging applications like autonomous vehicles (AV), which have strict deadlines, as well as in others with more flexible deadlines, such as augmented and virtual reality (AR/VR), smart retail, and recommendation systems [10], [18].

Figure 1 shows an example of a network of DNNs for autonomous vehicle applications. After pre-processing, each input image is subjected to different detection and tracking

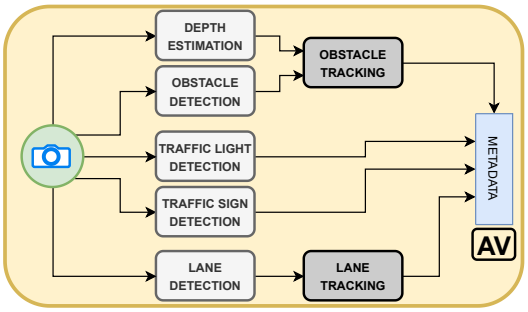


Fig. 1: Network of DNNs. In AV, several DNN models can be connected to facilitate a more complicated ML task.

operations performed by different DNNs. These multi-tenant ML workloads are complex due to the integration of various specialized models and having to operate under strict latency constraints.

A typical approach for processing DNN networks involves utilizing cloud servers, where the user device transmits input data to the cloud and receives the processed results in return [4], [26], [44]. However, this approach suffers from significant communication delays and poses security risks.

Modern MPSoCs (Multiprocessor System-on-Chip) are heterogeneous multi-core processors comprising CPUs, GPUs, DSPs, and, in some cases, custom accelerators, each offering distinct performance and power characteristics. Advances in their design have enabled running increasingly compute and data-intensive ML workloads entirely at the edge [6], [17], [23], [38]. However, efficiently mapping DNN networks onto such heterogeneous MPSoCs while adhering to real-time processing constraints presents a number of challenges, as discussed below.

1) The scheduling and allocation have to be performed frame-by-frame, as each frame varies in the number and types of objects, which define scene complexity. This requires accounting for workload fluctuations due to varying scene complexity and usage scenarios at runtime. Figure 2 shows an example of such a scene.
2) Inputs have to be *batched* and the optimal batch sizes

2837-4827/24/$31.00 ©2024 IEEE
DOI 10.1109/SEC62691.2024.00029

Fig. 2: An example of an input frame for an autonomous vehicle application, where the objects of interest are defined as people, cars, traffic lights, street lanes, etc

have to be determined accounting for the heterogeneity of the MPSoC and the dynamically changing scenes.

3) The required runtime scheduling is a sequential decision problem because the complexity of the present scene and the present state of resources have to be considered, making static scheduling solutions sub-optimal.
4) Deadlines have to be met while maintaining optimal accuracy during runtime workload fluctuations caused by varying scene complexities and usage scenarios.

A. Realistic Use Cases

Autonomous Vehicles (AV): These present good examples of complex multi-tenant, real-time ML workloads at the edge. In AV, multiple DNNs work in parallel utilizing data from various sensors to perform critical functions such as navigation and obstacle detection in real-time. For instance, Figure 1 loosely resembles the NVIDIA DRIVE Perception pipeline which includes a set of DNNs [1], [7], [36] that are essential for detecting driving paths, wait conditions, and other objects in the vehicle's environment. The workload can vary substantially based on the number and types of objects encountered in a scene. The system may need to process more data and make quicker decisions in complex environments, such as busy urban streets, as depicted in Figure 2, in contrast to simpler scenarios such as a clear highway.

Augmented/Virtual (AR/VR): These applications, shown in Figure 3, are good examples of real-time, multi-tenant ML workloads where the processing demands are heavily influenced by the specific usage scenario and the complexity of the scene [18], [38]. Hand-based interactions in AR may use cascaded models, such as hand detection followed by hand tracking. If the initial hand detection model does not detect a hand, the subsequent hand tracking model does not need to be executed. Similarly, the Distream [41] workflow exemplifies another scenario where workload fluctuations are driven by the count and type of objects in live video analytics. In scenarios like the EagleEye [40], the workload fluctuates drastically as the number of faces in a video changes every frame, significantly affecting the computational demand [14].

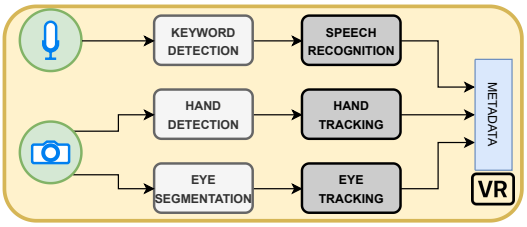


Fig. 3: Network of DNNs pipelines. A VR application pipeline.

B. Need for Elastic Scheduling

Traditional scheduling algorithms often fail to adapt quickly to rapid fluctuations in the demand for computation time and resources. The focus of existing methods has been mostly on minimizing the latency of inference. However, in scenarios where processing deadlines are crucial—such as in safety-critical functions of AVs or the real-time interactive nature of AR/VR applications—ensuring that these delay constraints are not violated becomes paramount. Missing a deadline in such applications can lead to outdated or irrelevant data being processed, which might compromise system performance and user's quality of experience. In these scenarios, dropping frames might seem like a straightforward solution but that can lead to significant information loss and disrupt continuity, which is critical in applications requiring temporal context. These are just a few of numerous examples that justify the need for elastic scheduling to handle varying workloads and ensure maximum accuracy within deadlines. This approach would adapt dynamically by adjusting the input resolution, using quantized models, or switching to smaller models.

C. Main Contributions of Paper

This paper introduces a holistic *elastic scheduler*, called *EMERALD*[1], for multi-tenant DNN workloads on edge MPSoCs, that aims to meet real-time deadlines while maintaining high end-to-end accuracy. In the presence of resource constraints and deadlines, execution latency and model accuracy become interdependent quantities, that present trade-offs. Unfortunately, they are not comparable quantities. For this reason, we define two dimensionless *utility* functions, associated with latency and accuracy. Then a *combined* utility function is defined, which serves as the objective function to maximize.

EMERALD is a hierarchical scheduling framework with two main components: a *local* scheduler, which is a greedy algorithm that maps batches of each DNN's inference requests to the *currently* best execution target, and an infrequently invoked *global* scheduler that adjusts the existing allocation and scaling of the inputs. The local scheduler responds to deadline violations by adjusting the input scales of the model. The model with the lowest impact on overall accuracy and the highest impact on makespan reduction is selected for scaling. *EMERALD* identifies the model with the greatest impact on makespan, calculating the critical path using execution times,

[1]**E**lastic **M**ulti-DNN **E**dge **R**esource **A**llocation and **L**atency-aware **D**eployment

scene complexity, and hardware compatibility of the models. When the overall makespan falls below the deadline, the local scheduler will increase the input resolution of each model, starting with one that results in the highest increase in the utility function of accuracy.

The global scheduler uses integer linear programming (ILP) to determine the optimal allocations and input scales to maximize a combined utility. It considers the dependencies among the network of DNNs, scene complexity, specific features of each DNN, hardware heterogeneity, and the balance between accuracy and response time. The global scheduler is also valuable when there is a need to respond to significant changes in scene complexity, because incremental adjustments to the input resolutions, if done by the local scheduler, could lead to highly sub-optimal solutions.

Compared to two state-of-art algorithms, *CAMDNN* [10] and *Heterogeneous Earliest Finish Time (HEFT)* [34], *EMERALD* reduces the number of deadline misses by more than a factor of 10. Also demonstrated is how a constraint on the frames-per-second (FPS) leads to a change in performance and accuracy.

D. Organization of the Paper

The rest of this paper is organized as follows: Section II defines elastic scheduling, and its use for multi-tenant DNN workloads, and includes a motivational example showing the differences between elastic and non-elastic solutions. Section III discusses the prior work on scheduling the execution of single and multi-DNN workloads and shows the gap in scheduling solutions for a network of DNNs within deadline constraints. The optimization problem and the proposed two-stage elastic scheduler are described in Section IV. Results and conclusions are presented in Section V.

II. Background and Motivation

In this section, we define the elastic task model and then describe how elastic scheduling can be used for deadline-constrained, multi-tenant DNN workloads. This is followed by a motivational example to show the need for an elastic scheduler for managing such workloads.

A. Elastic Scheduling

Elastic scheduling, as introduced by Buttazzo et al. [2], represents a flexible framework designed to manage adaptive tasks that might require dynamic adjustment of their operating periods in response to changing workloads or user requirements. A sporadic task i is represented by a tuple $\tau_i = (C_i, T_i, D_i$, where C_i is the worst-case execution time (WCET), T_i is the period and D_i is the deadline. The elastic task model generalizes the sporadic task model with $\tau_i = (C_i, T_i^{\min}, T_i^{\max}, E_i)$, where T_i is allowed to vary within $[T_i^{\min}, T_i^{\max}]$. The elastic coefficient E_i is a user-defined, application-dependent parameter. The optimization problem solved in [2] is expressed as follows.

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{T}|\mathbf{C},\mathbf{E}) = \sum_{i=1}^{n} \frac{C_i}{E_i}\left(\frac{1}{T_i^{\min}} - \frac{1}{T_i}\right)^2 \\ \text{such that:} \quad & \frac{1}{T_i^{\max}} \le \frac{1}{T_i} \le \frac{1}{T_i^{\min}} \\ & T_i \ge C_i \quad \forall \quad i \end{aligned} \tag{1}$$

The decision variables are the task periods T_i, and the goal is to find the optimal periods that minimize the total deviation from the preferred (minimum) periods while ensuring schedulability. The smaller value of E_i essentially assigns greater importance to keeping the period of task i close to T_i^{min}, thus preserving the highest quality of service for that task.

B. Elastic Scheduling for Multi-tenant DNNs

In this paper, we show how elastic scheduling can be adapted to scheduling multi-tenant DNNs. Instead of adjusting the periods of tasks as in traditional elastic scheduling, we adjust the worst-case execution times of DNN models to meet latency requirements. In multi-tenant DNN workloads, we dynamically configure DNN models to balance execution time and accuracy. Our goal is to maximize a combined utility function that considers both the accuracy of the models and the latency of execution. By doing so, we aim to meet real-time deadlines while maintaining maximum end-to-end accuracy. To achieve this, two key adjustment strategies: *model adaptation* and *data adaptation* can be considered.

Model adaptation involves real-time switching between models of different complexities and computational demands. It can switch between models in the same family, like ResNet18 and ResNet152, which have significantly different execution times and insignificant differences in accuracy. This allows systems to select the most suitable models for efficient responsiveness [42].

Data adaptation involves modifying input data properties, such as the size of input features, to effectively manage computational loads. Decreasing input size can help reduce processing times, which is essential for handling workload peaks without sacrificing system responsiveness. However, both approaches come with the unavoidable cost of reduced accuracy.

Balancing the trade-offs between accuracy and efficiency is crucial when applying these optimization techniques, as adjustments can significantly affect the downstream accuracy of the model. *EMERALD* exploits model elasticity through the scaling of input feature maps. It does not include runtime switching of models as that incurs excessive overhead due to the limited memory capacity of edge devices. In addition to input scaling, *EMERALD* accounts for DNN execution times, kernel loading times, optimal batch size configuration, and variations in scene complexity. Each of these factors plays a critical role in the effective deployment and operation of DNNs, influencing everything from startup delays and throughput to computational efficiency and the accuracy-speed trade-off under varying input content and workload conditions.

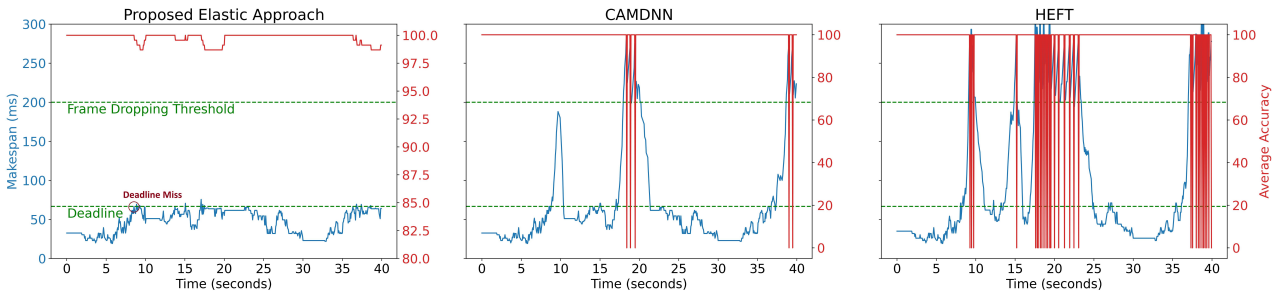


Fig. 4: The proposed approach, *EMERALD*, does not drop any frame while other approaches drop the frames after $3T_{max} = 200ms$. *CAMDNN* and *HEFT* do not incorporate input data scaling. Therefore, they either operate with maximum accuracy or drop frames and do not extract any information from the scene. The deadline for processing a frame is approximately $T_{max} = 66.67ms$.

C. Elastic Scheduling Example

In this section, we present a small example to highlight the differences between *EMERALD* and non-elastic scheduling. The details of *EMERALD* are presented in Section IV and are not important at this point.

Figure 4 shows a comparison of *EMERALD* with two state-of-the-art non-elastic schedulers, *CAMDNN* [10] and *HEFT* [34]. The workloads consist of a network of DNNs with two object detection models followed by three parallel classification models (see Figure 8d). The workloads were profiled on a Qualcomm RB5 development kit. Each figure shows two graphs over a 40-second span, showing makespan and total accuracy. Makespan, which is the maximum completion time of the application on all computing units, is indicated in blue and is the left ordinate. End-to-end model accuracy is shown on the right ordinate and represents the average accuracy across all models. Each plot also shows two dashed lines – the lower dashed line indicates a deadline of 66.67 ms based on an incoming frame rate of 15 FPS. The upper dashed line represents a threshold, which indicates that if the makespan exceeds 200ms, then the input frame loses its real-time relevance and is dropped.

Scene complexity varies over time, leading to fluctuations in workloads. *EMERALD* reacts to deadline violations by gradually changing the scale of the input to maintain the makespan below the required deadline. As soon as the makespan exceeds the deadline, indicated by a red circle, the elastic scheduler begins to downscale the input image size, while meeting the deadline. As shown in the top plot, the impact of this downscaling is a slight reduction in accuracy (less than 3%).

In contrast, both *CAMDNN* and *HEFT* lack workload adjustment strategies, leading to the accumulation of workloads on processing units until the 200 ms threshold is reached. After this point, the frame loses its relevance, and dropping the frame is necessary to prevent an unbounded increase in total makespan. As a result, the utility function of accuracy fluctuates between zero and its maximum, by either processing the frame at the maximum input scale or dropping it entirely. Section IV-D includes examples comparing *EMERALD*, *CAMDNN*, and *HEFT* for two scenarios: bursts of high complexity scenes, and consistently high complexity scenes.

III. Related Work

The related work relevant to EMERALD can be categorized by their approach to serving DNN workloads. These are discussed below.

A. Single DNN on a Single Processor

Runtime DNN inference optimization includes a set of techniques designed to improve the inference latency of DNN models, particularly on resource-limited edge devices. Common methods include adaptive inference [19], [33], [39], quantization [3], [13], [21], [29], and pruning [11], [15], [22], [43]. However, when managing networks of DNN models in near real-time applications that experience fluctuating workloads due to variations in scene complexity and usage scenarios, optimization strategies must extend beyond individual model adjustments.

B. Single DNN on Multiple Processors

1) Partitioning on the Same Device: The performance analysis of executing DNNs on several mobile devices from different hardware vendors (chipsets from Qualcomm, HiSilicon, Samsung, MediaTek, and Unisoc) is presented in [12]. There has been a body of work for mapping a single DNN onto different hardware units of mobile devices. *Pipe-it* [35] is an example of such a framework, which partitions the computation layers between big and LITTLE cores on a multi-core mobile device.

2) Partitioning and Offloading on Multiple Devices: Other approaches extend beyond using a single device, distributing DNN computation across multiple, often resource-constrained, devices. Frameworks such as *MoDNN* [27] and *DeepThings* [45] partition the computation of DNNs between resource-constrained mobile devices using input data partitioning. Additionally, [8], [9] aims to reduce energy consumption by offloading DNN segment computation to cloudlets or connected edge devices, but they do not consider scene complexity or input data variations.

C. Multiple DNNs on Multiple Processors

1) Non-Elastic Approaches: Various non-elastic methods have been proposed to address scheduling multiple DNNs on heterogeneous hardware. For example, *AutoScale* [17] uses

reinforcement learning to select suitable processing units for DNN inference, optimizing energy consumption. Although reinforcement learning (RL) approaches can be effective, they often entail significant computational overhead due to extensive training requirements and slow convergence rates. This makes them impractical for runtime scheduling on resource-constrained devices [25], [26]. Other solutions, such as cloud-based frameworks like Inferline [4], Clipper [5], and Infaas [30], focus on serving DNN workloads on cloud servers by mapping workloads across multiple processors, but they don't address edge-specific constraints. Further, *Band* [14] explores heterogeneous scheduling by partitioning independent DNN models into subgraphs, scheduled based on the least slack time while considering hardware heterogeneity. However, Band fails to handle networks of DNNs with dependencies and inaccurately calculates slack without considering scene complexity.

2) Elastic Approaches: Elastic methods, designed for dynamic environments, enable adaptability to fluctuating workloads. Approaches like *Model Switching* [42] switch between DNN models of varying complexity depending on workload intensity, sacrificing accuracy during load spikes. Similarly, *Jellyfish* [28] exploits both data adaptation and model adaptation techniques to meet inference deadlines. However, both methods are impractical for edge devices due to storage constraints and significant model loading overheads, making frequent model switching infeasible.

D. Network of DNNs on Multiple Processors

1) Non-Elastic Approaches: When it comes to networks of DNNs, frameworks like *CAMDNN* [10] and *DREAM* [16] aim to schedule DNN models on heterogeneous hardware platforms efficiently. *CAMDNN* [10] generates a minimum latency schedule for executing a network of DNN models on a heterogeneous hardware platform, taking into account the scene complexity. *DREAM* [16] addresses the unique challenges of executing DNN networks in augmented reality applications on energy-constrained devices. It combines urgency and latency-preference scores to schedule tasks similarly to the *HEFT* algorithm. However, *DREAM* overlooks crucial factors such as batching opportunities and model duplication, limiting its effectiveness for heterogeneous edge devices.

2) Elastic Approach: Our proposed work, *EMERALD*, fills the gaps left by previous approaches by considering scene complexity, batching inference requests, and accounting for hardware heterogeneity and model loading overheads. We optimize performance by distributing batched requests across processors and adjusting input scaling rather than switching models. EMERALD also addresses networks of DNNs with inter-model dependencies, which are common in edge applications but often overlooked by existing solutions.

IV. Optimization Problem and Proposed Approach

A. System Model

The notations used in this paper are shown in Table I. An application is modeled using a data flow graph $\mathcal{G} = (\mathcal{M}, \mathcal{E})$ where $\mathcal{M}$ is the set of all DNN models and $\mathcal{E}$ represents the dependency between the DNNs. Based on the content of the input data, the number of times a model is called (r_i) is changing. This means that an application cannot be simply modeled as a directed acyclic graph (DAG). Synchronous dataflow graph (SDF) [20] is used to model such a content change. Then the SDF is transformed to a DAG in polynomial time [32], by replicating each model based on the input.

TABLE I: List of notations.

Notation	Description
p_j	processor j
m_i	model i
n_p	number of processors
n_m	number of models
n_c	number of scales
n_b	number of batch sizes
$\mathcal{M}$	set of all models in application graph $\{M_i \vert 1 \leq i \leq n_m\}$
$\mathcal{P}$	set of all available processing units $\{p_j \vert 1 \leq j \leq n_p\}$
$wait_i$	the number of waiting requests for m_i
ρ_i	priority of model i
$\bar{\mathcal{P}}$	set of all available processing units $\{p_j \vert 1 \leq j \leq n_p\}$
rem_j	remaining time for processor j to become idle
k	batch size $(1 \leq k \leq n_b)$
r_i	total number of invocations for model i
$a_{i,j}$	set to 1, if $\sum_k b_{i,j,k} \neq 0$ for a given i, j
$b_{i,j,k}$	number of instances of m_i running on p_j with batch k
$l_{i,j}$	loading time of model i on processor j
$c_{i,l}$	binary scaling factor for model i with model scale l
$e_{i,j,k}$	execution time for model i on processor j with batch k
$E_{i,j}$	total execution time for model i on processor j for all allocated batches
τ	total completion time or makespan
T_{max}	total completion time or makespan
w_i	importance coefficient of model i
$\text{acc}_{i,l}$	accuracy score of model i with model scale l
$Max(U_A)$	maximum accuracy utility across all models
U_A	weighted average accuracy utility of all models
U_L	utility based on latency relative to T_{max}
U_C	weighted sum of normalized accuracy and latency utils
$\text{scaled\_exec}_{i,j,k,l}$	scaled execution time for model i on processor j with batch k and scale l

The objective of *EMERALD* is to obtain the mapping of DNN models onto the hardware units such that the combined utility function is maximized. The combined utility function depends on the completion time, deadline, and end-to-end accuracy. The control knobs are batch variables $b_{i,j,k}$, model scaling factors $c_{i,l}$ and start and finish times $s_{i,j}$ and $f_{i,j}$.

The overall structure of *EMERALD* is shown in Figure 5. The time horizon is divided into intervals of fixed duration call $epochs$. Initially, the scheduler starts with a mapping provided by the global scheduler, assuming that only one instance of each DNN model is needed. As *EMERALD* processes incoming frames, it counts them based on the number of occurrences of a dummy destination (dst) node. When this count reaches the predefined epoch length, the global scheduler is invoked again to re-calibrate the allocation and model scaling factors using the collected scene complexity (r_i) data, hardware availability (rem_j), and current allocations ($a_{i,j}$) reported by the local scheduler. If the epoch length is not reached, the local scheduler continues to operate at runtime, serving inference requests by heuristically adjusting the allocation and scaling factors to adapt to immediate workload changes.

B. Global Scheduler

The objective of the global scheduler is to achieve optimal scheduling and scaling of DNNs across computing resources, with the knowledge of all the DNNs inside a frame to maximize a combined utility including both accuracy and latency. The problem can be formulated precisely as an Integer

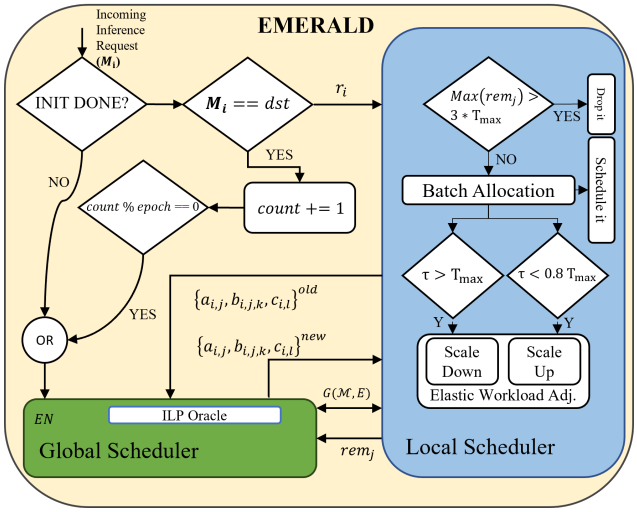


Fig. 5: The overall scheme of our proposed approach (*EMERALD*) showing how the global and local schedulers work together.

Linear Programming (ILP) problem. However, the execution overhead of the ILP solver becomes prohibitive at runtime. To overcome this, a local scheduler is designed to schedule and scale inference requests between infrequent invocations of the global scheduler.

The ILP formulation incorporates DNN-specific profiling data, scene complexity, accuracy information specific to each DNN, and the current residual workloads on the available hardware units. The profiling data for each DNN includes the execution time $scaled\_exec_{i,j,k,l}$ for model i on processor j with batch size k and model scale l, and the accuracy score for model i at scale l. The ILP formulation is given below.

Decision Variables:

The *decision variables* consist of the start and finish times ($\boldsymbol{s_{i,j}}$, $\boldsymbol{f_{i,j}}$) for model i on processor j, $\boldsymbol{b_{i,j,k}}$ and $\boldsymbol{c_{i,l}}$. The total number of decision variables in the ILP formulation is $((n_m \cdot n_p) \times (2 + n_b) + (n_m \cdot n_c))$, derived as follows: Each model has two timing variables (start and finish times) and n_b batch size variables for each of the n_p hardware units, contributing $((n_m \cdot n_p) \times (2 + n_b))$. Additionally, there are n_c scaling factors per model, resulting in $(n_m \cdot n_c)$ additional variables.

Objective Function:

The objective function is the total combined utility that is dependent on both latency and accuracy utilities. In soft real-time systems, the highest utility is achieved when data is processed within a certain time frame. If the makespan is higher than the threshold, the data becomes irrelevant or useless, and both its accuracy and latency utility will drop to zero [18]. The latency utility is defined as:

$$U_L = \begin{cases} U_{\max} & \tau \leq T_{max}, \\ U_{\max} - m(\tau - T_{max}) & T_{max} < \tau \leq 3T_{max}, \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

Makespan τ is the maximum finish time of all nodes on all processors.

The overall pipeline accuracy is a complex function of the individual model's contribution to the final output. For this reason, we use *accuracy scores* [18] — $acc_{j,l}$, which is a dimensionless quantity in [0,1], with larger values denoting better or preferred performance. This allows us to add the scores when models are organized as a pipeline. In fact, the accuracy utility U_A is a weighted sum of individual scores (Eq. 3), which is also in [0,1]. The elasticity coefficient w_i acts as a weighting factor, with lower values of w_i giving more importance to that model's accuracy in the overall utility calculation. This allows for prioritizing certain models based on their criticality or sensitivity to input changes. The maximum accuracy utility ($Max(U_A)$) is achieved when no input scaling is applied.

$$U_A = \sum_{i=1}^{n_m} \sum_{l=1}^{n_c} \frac{1}{w_i} \times c_{i,l} \times \text{acc}_{i,l} \quad (3)$$

$$Max(U_A) = \sum_{i=1}^{n_m} \frac{1}{w_i} \times \text{acc}_{i,0} \quad (4)$$

The combined utility function (Eq. 5) is a convex combination of latency and accuracy utilities, with the weights α and β allowing the scheduler to adapt based on the use-case and scene content. These weights, where $(\alpha + \beta = 1)$, prioritize either accuracy or latency as needed.

$$U_C = \left(\alpha \frac{U_A}{Max(U_A)} + \beta \frac{U_L}{Max(U_L)} \right) \times 100 \quad (5)$$

Constraints:

• The finish time of model i on processor j is equal to its start time plus execution time based on mapped batching plus the loading time if the model is not loaded:

$$\boldsymbol{f_{i,j}} = \boldsymbol{s_{i,j}} + E_{i,j} + \text{loading time}, \quad (6)$$

$$E_{i,j} = \sum_k \left(\boldsymbol{b_{i,j,k}} \cdot \sum_l (\boldsymbol{c_{i,l}} \cdot scaled\_exec_{i,j,k,l}) \right), \quad (7)$$

$$\text{loading time} = \frac{diff(i,j) \times l_{i,j}}{epoch}, \quad (8)$$

$$diff(i,j) = ([a_{i,j}]^{new} . \overline{[a_{i,j}]^{old}}), \quad (9)$$

$$\boldsymbol{b_{i,j,k}} \in \mathbb{Z}^+, \quad \boldsymbol{f_{i,j}}, \boldsymbol{s_{i,j}} \in \mathbb{R}^+, \quad \boldsymbol{c_{i,l}} \in \{0,1\}. \quad (10)$$

$diff(i,j) = 1$ if model i is required to be allocated on processor j. Note that Equation 7 is non-linear. In the final implementation, the product of integer and binary decision variables is linearized using the Big-M method [37].

• The start times of successors $\geq$ finish times of predecessors:

$$i_1 \prec i_2 : \boldsymbol{s_{i_2,j}} \geq \boldsymbol{f_{i_1,j}} \quad \forall j. \quad (11)$$

• The priority constraints based on the critical path determine the order in which DNN models must be executed:

$$\rho_{i_1} > \rho_{i_2} \Rightarrow \boldsymbol{s_{i_2,j}} \geq \boldsymbol{s_{i_1,j}} + E_{i_1,j}. \quad (12)$$

• The sum of batch size for each model over all processors equals to the total number of instances of that model:

$$\sum_{j=1}^{n_p}\sum_{k=1}^{n_b} k \times \boldsymbol{b_{i,j,k}} = r_i. \tag{13}$$

• Set allocation variables based on batch size variables. Ω to be a large integer number (e.g., maximum execution time):

$$\sum_{k=1}^{n_b} \boldsymbol{b_{i,j,k}} \geq (1 - \Omega \times (1 - a_{i,j})) \quad \forall i, j, \tag{14}$$

$$\sum_{k=1}^{n_b} \boldsymbol{b_{i,j,k}} \leq \Omega \times a_{i,j} \quad \forall i, j. \tag{15}$$

• Limit the maximum number of models that are allocated to processor j:

$$\sum_{i=1}^{n_m} a_{i,j} \leq N_j \quad \forall j. \tag{16}$$

C. Local Scheduler

The local scheduler consists of two parts: (1) A dynamic batch allocation mechanism, and (2) An elastic workload adjustment mechanism. Upon the arrival of each new frame, the scheduler first examines the remaining workload on all processors. If this workload exceeds three times the maximum allowed latency ($3 \times T_{max}$), the new frame is dropped to prevent system overload. Otherwise, the dynamic batch allocation mechanism schedules all incoming DNN inference requests onto the available hardware units for the received frame. After scheduling, the scheduler evaluates the makespan for the frame. If the makespan exceeds the deadline, the elastic workload adjustment mechanism is activated to scale down the input feature maps, reducing execution times to meet the deadline. Conversely, if the makespan is less than 80% of the deadline, the elastic workload adjustment mechanism may scale up any previously scaled-down models to enhance accuracy while still satisfying timing constraints.

*1) **Dynamic Batch Allocation**:* The Dynamic Batch Allocation is a four-step heuristic devised to calculate the optimal batch size of inference requests for each DNN model and efficiently allocate them across available heterogeneous hardware units at runtime. The algorithm starts by computing the preferred batch sizes for each model and hardware unit, followed by a cost score calculation that takes into account the hardware units' current workload and execution efficiency. The next step allocates an initial distribution of model instances based on the calculated cost score. Finally, the balancing stage ensures the sum of allocated batches matches the repetition number for a given model.

Step 1.1: Initial Batch Allocation Computation

- *Preferred Batch Size Calculation:* For each DNN model i on every hardware unit j, calculate the preferred batch size based on the sorted list of the average execution time per request for different batch sizes.
- *Batch Combination Determination:* Determine the optimal combination of batch sizes for each hardware unit, aiming to find the most efficient batch configuration for processing model instances.

Step 1.2: Cost Score Calculation

- *Computing Cost Score:* For each hardware unit j, calculate the Cost Score $\mathrm{T}_j(b)$, considering the residual load (rem_j) and the execution time for the best batch combinations.

$$\mathrm{T}_j(b) = \mathrm{rem}_j + \sum_k (b_{i,j,k} \times e_{i,j,k}) \tag{17}$$

Step 1.3: Initial Repetition Distribution

- *Distribution Based on Cost Score:* Allocate initial model i instances to each hardware unit j proportional to the inverse of the unit's Cost Score.

$$b_{i,j} = \left\lfloor \frac{(1/\mathrm{T}_j)}{\sum_j (1/\mathrm{T}_j)} \times r_i \right\rfloor \tag{18}$$

Step 1.4: Balancing Repetitions Across Hardware Units

- *Ensuring Total Repetition Match:* Verify if the sum of allocated repetitions matches the total required repetitions r_i. Adjust if necessary.
- *Load Balancing:* Incrementally adjust repetitions assigned to each hardware unit, focusing on minimal additional cost.
  - Identify the hardware unit j that minimizes $j' = argmin_j(\text{Cost Score}(b_{i,j,k=1} + 1))$.
  - Increase the $b_{i,j',k=1}$ by one for the selected hardware unit.
  - Repeat until $\sum_{j=1}^{n_p}\sum_{k=1}^{n_b} k \times b_{i,j,k} = r_i$.

*2) **Elastic Workload Adjustment**:* The second stage of the local scheduler handles deadline misses. The scheduler starts by computing the critical path of the graph. Models on the critical path are sorted based on their elasticity coefficient where higher priority is given to those with higher elasticity for scaling. This is repeated until the required deadline is met.

The calculation of the critical path is complicated because of the different execution times, varying scene complexity, and hardware compatibility of models. These are accounted for in the definitions of earliest start time (EST), latest finish time (LFT), and slack, as follows.

$$\mathbf{EST}(m_i) = \max_{j \in \mathrm{pred}(i)} (\mathrm{EST}(m_j) + \mathrm{AVG}(e_j|r_j)) \tag{19}$$

$$\mathbf{LFT}(m_i) = \min_{j \in \mathrm{succ}(i)} (\mathrm{LFT}(m_j) - \mathrm{AVG}(e_j|r_j)) \tag{20}$$

$$\mathbf{Slack}(m_i) = \mathrm{LFT}(m_i) - \mathrm{EST}(m_i) - \mathrm{AVG}(e_i|r_i) \tag{21}$$

$\mathrm{AVG}(e_i|r_i)$ is the average execution time of r_i instances of model i, distributed across all heterogeneous processing units. This average is estimated by assuming that the r_i instances are distributed equally across compatible processing units and solving a Min-Max Optimization problem as follows.

$$\begin{aligned} &\mathrm{AVG}(e_i \mid r_i) = \min\left(\max_j (r_{i,j} \times e_{i,j})\right) \\ &\text{subject to} \quad \sum_j r_{i,j} = r_i, \quad r_{i,j} \geq 0 \end{aligned} \tag{22}$$

An elastic workload adjustment algorithm is designed to scale models iteratively, by adjusting the input size to reduce the execution time until the required deadline is met. Additionally, if the makespan falls below a predefined threshold (e.g., 80% of the required deadline), the input size will be recursively increased back to its original size to maintain the highest accuracy possible. The algorithm steps are as follows:

Step 2.1: Compute Critical Path

- For each model m_i in the application graph, calculate the EST, LFT, and Slack.
- Identify the models on the critical path, i.e. Slack$(m_i) \leq 0$.
- Find m_i will smallest slack.

Step 2.2: Sort Models Based on Elasticity

- Sort the models on the critical path in descending order of their elasticity coefficient w_i:

$$\text{sorted\_models} = \text{sort\_desc}(\text{critical\_path\_models}, w_i)$$

Step 2.3: Scaling Down Models

- Decrease the current input size for the most elastic model to reduce the total makespan.
- Update model scales in the *BatchAllocation* algorithm for scheduling the next batch of incoming requests.

This module iteratively modifies the model scales until the deadline constraint is met.

Step 2.4: Scaling Up Models

- If the current makespan falls below a predefined threshold, start reverting the scaling decisions:
  - Revert to lower input scales for the least elastic model where reducing the input size yields the greatest gain in accuracy.
- This method iteratively adjusts the model scales while makespan stays under the threshold and all the models return to the original input size.

D. Examples of EMERALD vs. non-Elastic Schedulers

In this section, we illustrate the benefits and distinctions of elastic scheduling using *EMERALD* over non-elastic methods such as *CAMDNN* and *HEFT* executing a network of DNNs. The example network is shown in Figure 8d. It consists of two object detection models followed by three parallel classification models. The data consists of a sequence of images with two types of objects: cars and people. We explore two different scenarios which are shown in Figures 6 and 7. Note that the required FPS is 15 and frames must be processed within $3T_{max}$, where $T_{max} = 66.6\ ms$ represents the maximum allowable processing time for maintaining a 15 FPS input rate.

Scenario 1: Burst of High Complexity Scenes: Figure 6 shows a situation where there is a temporary surge in scene complexity: 8 people and 13 cars in frame 132 to 10 people and 13 cars in frame 134. At frame 134 both *EMERALD* and *CAMDNN* violate the 66.6 ms deadline necessary to maintain a 15 FPS rate, with *HEFT* lagging significantly behind. In response to this deadline violation, *EMERALD* quickly adjusts by scaling down inputs, minimally affecting the combined utility, which is shown in the score changing from 99.9% to 99.1%. Both *CAMDNN* and *HEFT* violate the deadline by substantial amounts resulting in the accumulation of unprocessed workloads from previous frames.

As the scene becomes less complex from frames 148 to 156, with 3 people and 11 cars, *EMERALD* returns to an optimal normalized utility of 100%, demonstrating its dynamic adaptability and quick recovery. In contrast, *CAMDNN* struggles with the backlog of delayed frames, taking 15 frames to stabilize and resume normal processing. *HEFT* shows even poorer performance, failing to process within $3T_{max}$ and displaying difficulty in coping with the increased computational requirements. This example shows the key advantage of *EMERALD*'s elastic scheduling over *CAMDNN* and *HEFT*. It rapidly adjusts to high computational demand and recovers quickly, ensuring higher combined utility.

Scenario 2: Consistent High Complexity Scenes: This scenario is shown in Figure 7. There is a continuous influx of scenes with high complexity challenging the scheduling capabilities of *EMERALD*, *CAMDNN*, and *HEFT*. At frame 256, the scene complexity peaks with 15 people and 11 cars, pushing the makespan of *EMERALD* and *CAMDNN* above the 66.6 ms threshold while *HEFT*'s makespan significantly exceeds this limit already. *EMERALD* reacts promptly by scaling down inputs, slightly affecting the utility score of accuracy but efficiently preventing a buildup of unprocessed

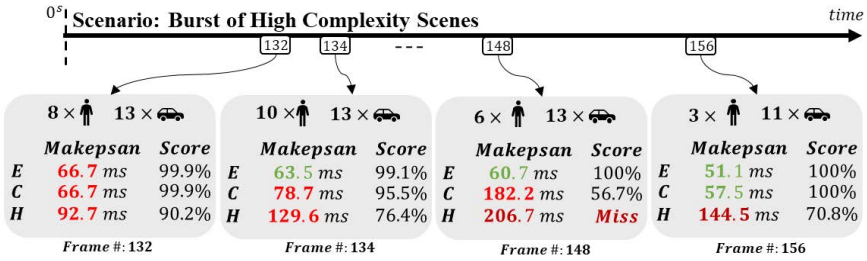


Fig. 6: This figure shows a comparative analysis between our proposed elastic scheduling approach (**E**MERALD) and traditional non-elastic methods (**C**AMDNN and **H**EFT). *EMERALD* demonstrates rapid adjustment to workload changes, swiftly returning to optimal performance. *CAMDNN* also recovers to normal processing speeds after a delay, while *HEFT* exhibits the slowest recovery to meet the required deadlines. Notably, both *CAMDNN* and *EMERALD* leverage batch processing effectively, minimizing the accumulation of residual processing times from previous frames, which is more pronounced in *HEFT*.

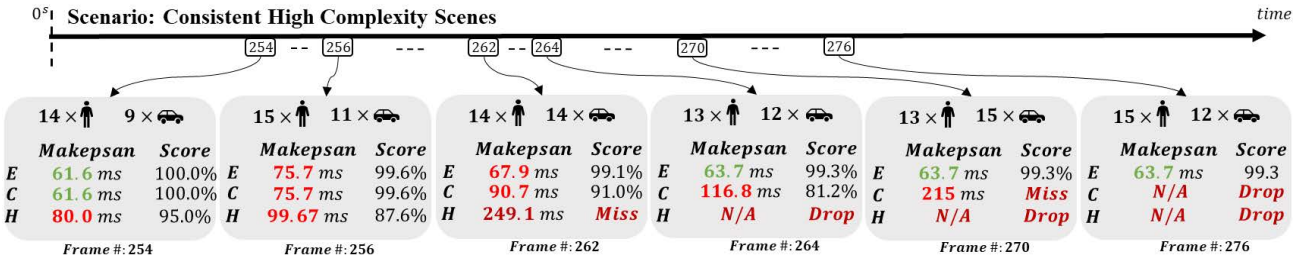


Fig. 7: In this example, we compare our proposed elastic approach (**E**MERALD) to non-elastic solutions (**C**AMDNN and **H**EFT). In non-elastic solutions, when the scene complexity consistently remains high, they fail to respond in real-time to incoming frames.

workloads from previous frames. In contrast, *CAMDNN* begins to struggle significantly by frame 270, where its makespan soars to 215 ms, effectively dropping to a 0% normalized combined utility due to the severe delay. This deterioration highlights *CAMDNN*'s inability to cope with prolonged high demands, taking several frames to recover and ultimately failing to process subsequent frames within $3T_{max}$. *HEFT* displays the worst performance, with its makespan jumping to 300 ms by frame 264, leading to a complete failure with minimum utility. The high makespan shows that *HEFT* cannot manage such high computational loads, and results in the scheduler failing to process frames within $3T_{max}$.

This example emphasizes the importance of using an elastic scheduler that can effectively handle both sudden and sustained increases in demand for computing power in real-time processing pipelines. The adaptive and dynamic nature of the elastic scheduler ensures that systems can maintain performance standards and process tasks on time, even when facing fluctuating workloads over extended periods. This example clearly shows that while traditional static scheduling methods may perform adequately under stable conditions, dynamic environments with varying input rates and computational demands necessitate an elastic approach like *EMERALD* to maintain system performance and output relevance.

V. Experimental Results

A. Hardware Setup

The Qualcomm RB5 development kit was used as the experimental platform. It is equipped with a Kyro 585 ARM CPU, a Qualcomm Adreno 650, and a Qualcomm Hexagon DSP. Extensive profiling of the various DNNs was performed to measure their execution and loading times.

B. Workloads

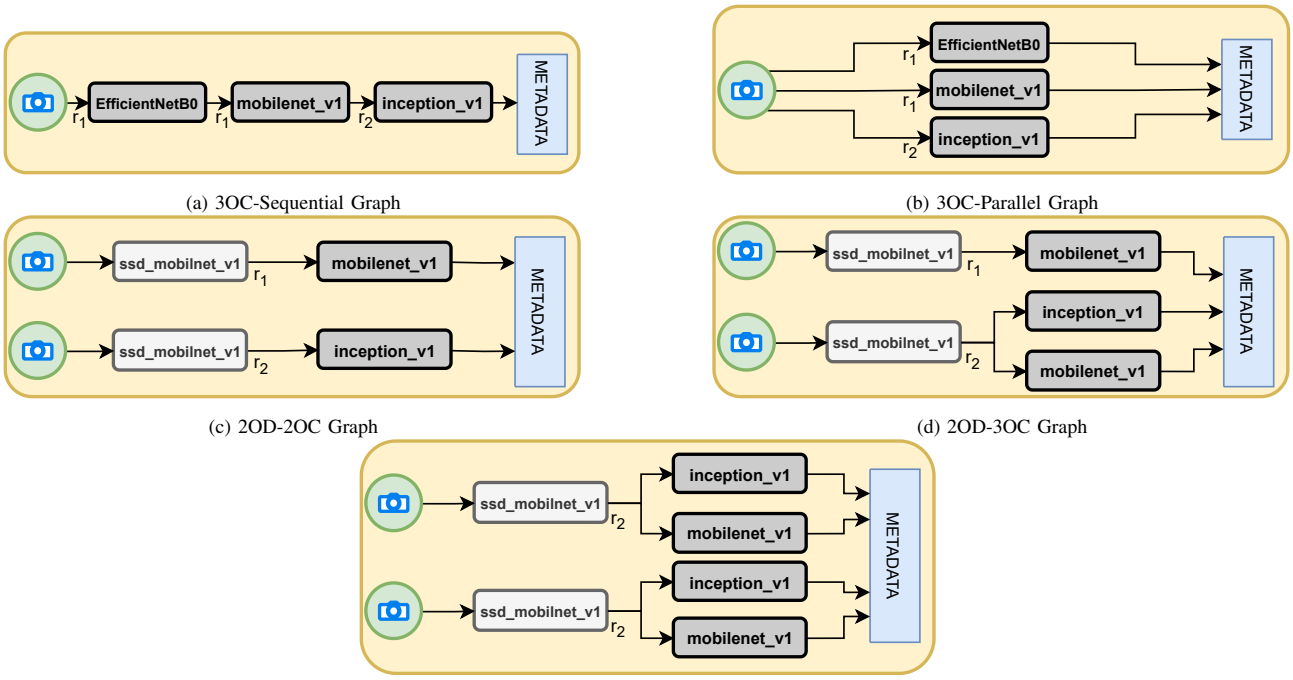
The different networks of DNNs used in our experiments are shown in Figure 8. They consist of instances of a single object detection model, $ssd\_mobilenet\_v1$, and three object classification models, $mobilenet\_v1$, $inception\_v1$, and $EfficientNetB0$. These DNNs were executed on the MPSoC using TFLite. We considered four levels of input scaling based on the original image size of 224×224 pixels (with $l = 0$ showing no downscaling). The input images were downscaled to create three additional scales: $l = 1$ at 192×192 pixels, $l = 2$ at 160×160 pixels, and $l = 3$ at 128×128 pixels. The elasticity coefficient for all models is assumed to be one, which means that no priority is given to any model for scaling. Models are scaled based solely on their impact on the overall makespan. However, depending on the application, the system designer can assign different coefficients to indicate the importance of individual models on end-to-end accuracy.

C. Comparison with Previous Work

EMERALD was compared with state-of-the-art solutions *HEFT* and *CAMDNN* in terms of the number of dropped frames and combined utility. Despite its earlier introduction, *HEFT* [34] remains a widely recognized baseline due to its efficiency and near-optimal performance in makespan minimization [24], [31]. *HEFT* is a greedy algorithm that is not sensitive to the content of the data, batching opportunities, or the deadline. The scheduling decisions are based solely on the execution time of DNNs on the hardware units and the remaining work on each hardware unit. *CAMDNN* is aware of the input content and tries to minimize the delay. However, it does not consider meeting the deadline by lowering the input data size. *EMERALD* appears to outperform both *CAMDNN* and *HEFT* in terms of deadline misses, dropped frames, and normalized utility. The dropping of frames happens when the execution time exceeds $3T_{max}$. Table II shows that *CAMDNN* and *HEFT* miss a significant number of frames whereas *EMERALD* misses deadlines less than 13% of the time, even in the worst-case scenario of a 30 FPS for the 2OD-4OC Figure 8e application graph.

Table III shows *EMERALD*'s reactive strategy of gradually adapting to workload changes results in the least number of missed frames. In contrast, *CAMDNN* and *HEFT*, the rate at which frames are dropped exceeds 40%.

Figure 9 shows that *EMERALD* maintains a consistent combined utility above 95%, while the values of the same for *CAMDNN* and *HEFT* fall below 11% and 9%, respectively. Additionally, the performance of *CAMDNN* and *HEFT* decreases noticeably at higher FPS rates. Their performance predictably worsens as the number of parallel and sequential models increases. In contrast, *EMERALD* adapts to changes in the application graph and input FPS. It is also evident in both Figures 9 and 10 that both the normalized combined utility U_c and average accuracy reduce significantly for *CAMDNN* and *HEFT* as FPS increases. Despite optimal *CAMDNN* allocation,



(a) 3OC-Sequential Graph

(b) 3OC-Parallel Graph

(c) 2OD-2OC Graph

(d) 2OD-3OC Graph

(e) 2OD-4OC Graph

Fig. 8: The application includes object detection and classification models, incorporating a mix of parallel and sequential models to represent different multi-tenant DNN applications with various dependency structures. All of these graphs are meant to show arbitrary multi-DNN graphs rather than a specific real application.

TABLE II: Comparison of deadline misses between *EMERALD*, *CAMDNN*, and *HEFT* over 30000 random scenes for different applications.

	FPS	EMERALD	CAMDNN	HEFT
3OC-Sequential	7	0 (0.0%)	0 (0.0%)	0 (0.0%)
	15	559 (1.9%)	5657 (18.9%)	11925 (39.7%)
	30	3844 (12.8%)	23616 (78.7%)	23749 (79.2%)
3OC-Parallel	7	0 (0.0%)	0 (0.0%)	0 (0.0%)
	15	291 (1.0%)	1243 (4.1%)	7909 (26.4%)
	30	3513 (11.7%)	19840 (66.1%)	20468 (68.2%)
2OC-2OD	7	0 (0.0%)	0 (0.0%)	0 (0.0%)
	15	0 (0.0%)	0 (0.0%)	595 (2.0%)
	30	1912 (6.4%)	21154 (70.5%)	23462 (78.2%)
2OD-3OC	7	0 (0.0%)	0 (0.0%)	0 (0%)
	15	690 (2.0%)	6575 (21.9%)	12185 (40.6%)
	30	3457 (11.5%)	25449 (84.8%)	26676 (88.9%)
2OD-4OC	7	0 (0.0%)	0 (0.0%)	241 (0.8%)
	15	3092 (10.3%)	14866 (49.6%)	19121 (63.7%)
	30	3902 (13.00%)	28208 (94.0%)	28376 (94.6%)

the total workload exceeds the capacity of the edge device, making non-elastic schedulers such as *CAMDNN* impractical. Note that the average overhead of decision-making for 30,000 random scenes for *EMERALD* is close to that of *CAMDNN* and *HEFT*, which makes it suitable for run-time decision-making.

Figure 11 shows that as the required input FPS increases and enforces tighter deadlines, *EMERALD* can consistently reduce the average makespan with optimal input resolution adjustments, irrespective of the graph structure. However, both *CAMDNN* and *HEFT* perform with significantly higher average latency for complex graphs like 2OD-4OC, especially at higher FPS inputs, where the rate of missed deadlines and frame drops also increases. For simpler graphs and lower FPS,

TABLE III: Comparison of number of dropped frames between *EMERALD*, *CAMDNN*, and *HEFT* over 30000 random scenes for different applications.

	FPS	EMERALD	CAMDNN	HEFT
3OC-Sequential	7	0 (0.0%)	0 (0.0%)	0 (0.0%)
	15	0 (0.0%)	123 (0.4%)	1323 (4.4%)
	30	0 (0.0%)	7346 (24.5%)	8519 (28.4%)
3OC-Parallel	7	0 (0.0%)	0 (0.0%)	0 (0.0%)
	15	0 (0.0%)	0 (0.0%)	336 (3.4%)
	30	0 (0.0%)	4826 (16.1%)	6527 (21.8%)
2OD-2OC	7	0 (0.0%)	0 (0.0%)	0 (0.0%)
	15	0 (0.0%)	0 (0.0%)	0 (0.0%)
	30	0 (0.0%)	3403 (11.3%)	5564 (18.5%)
2OD-3OC	7	0 (0.0%)	0 (0%)	0 (0%)
	15	0 (0.0%)	238 (0.8%)	1470 (4.9%)
	30	0 (0.0%)	8321 (27.7%)	10365 (34.6%)
2OD-4OC	7	0 (0.0%)	0 (0.0%)	0 (0.0%)
	15	8 (0.03%)	1844 (6.1%)	3810 (12.7%)
	30	8 (0.03%)	12098 (40.3%)	13839 (46.1%)

such as 3OC-Parallel shown in Figure 8b at 7 FPS, all schedulers show an average makespan below required deadlines with *CAMDNN* and *EMERALD* outperforming *HEFT* due to their optimal batch allocation.

D. EMERALD Performance Across Different Elasticity Coefficients

The performance of the *EMERALD* is evaluated by applying two different sets of elasticity coefficients to the 2OD-3OC graph. Previously, our experiments assumed equal elasticity across all models, allowing the scheduler to select models for scaling on the critical path that most significantly in-

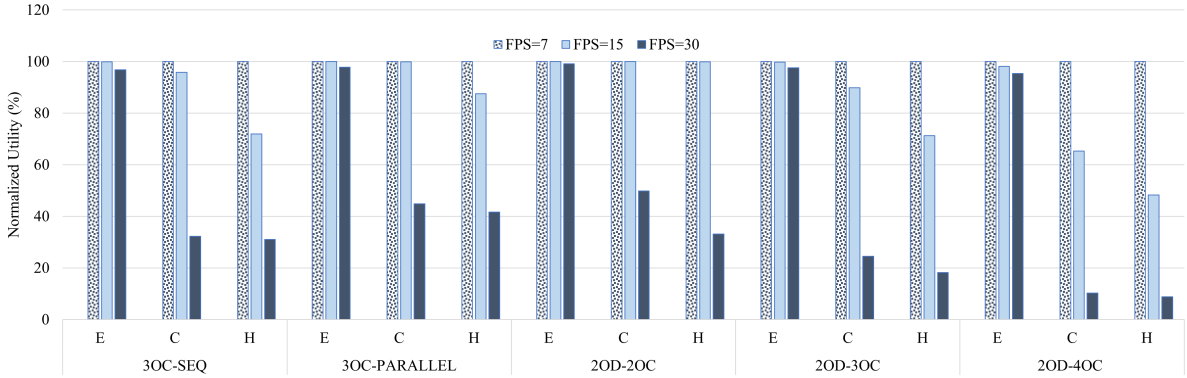


Fig. 9: Combined Utility differences between ***EMERALD***, ***CAMDNN***, and ***HEFT*** over 30000 random scenes for different applications.

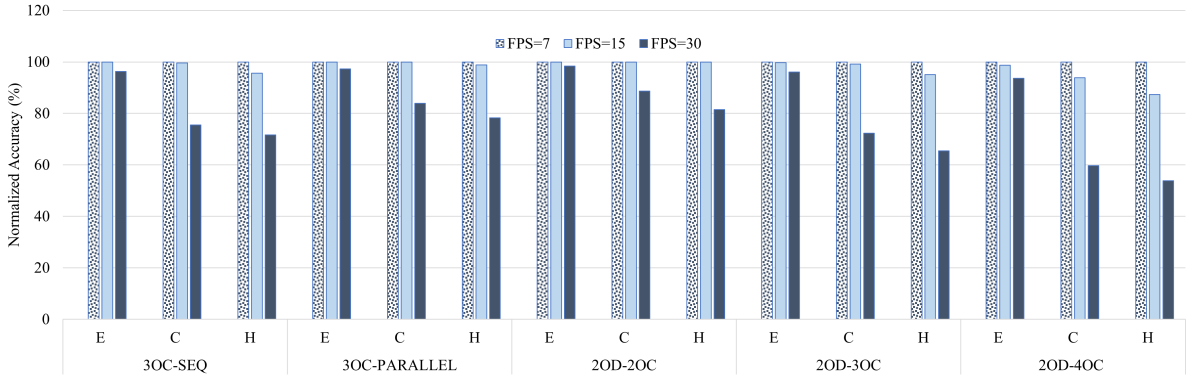


Fig. 10: Accuracy Utility differences between ***EMERALD***, ***CAMDNN***, and ***HEFT*** over 30000 random scenes for different applications.

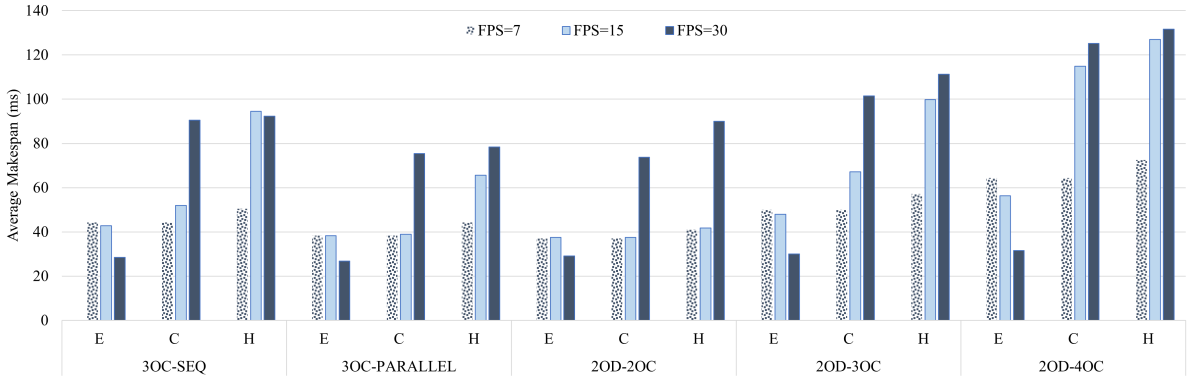


Fig. 11: Average makespan differences between ***EMERALD***, ***CAMDNN***, and ***HEFT*** over 30000 random scenes for different applications.

fluenced the makespan while minimally affecting accuracy. This approach aimed to optimize overall utility. However, assigning distinct elasticity coefficients to nodes modifies the accuracy utility as detailed in Equation 3, resulting in different *EMERALD* behavior.

Figure 12(a) shows that when object detection models with a repetition count of one are assigned lower elasticity, *EMERALD*'s behavior aligns closely with scenarios involving uniform elasticity. This occurs because models with higher repetition rates, which significantly affect the makespan, are prioritized for scaling.

In contrast, Figure 12(b) shows that assigning lower elasticity to object classification models with high repetition requires the *EMERALD* to make more effort to meet the required deadline. This strategy involves initially scaling object detection models before object classification models, which leads to an increase in deadline misses and dropped frames shown in Table IV to maintain an acceptable makespan. These experiments also show the significance of scene complexity in scheduling decisions.

TABLE IV: Comparison of two different sets of elasticity coefficients for the 2OD-3OC graph.

	(a)	(b)
Deadline Misses	2622 (8.74%)	4036 (13.45%)
Dropped Frames	0 (0.00%)	11 (0.04%)
Avg. Makespan (ms)	29.68	3045
Norm. Accuracy (%)	96.11%	93.97%
Norm. Utility (%)	97.82	95.88

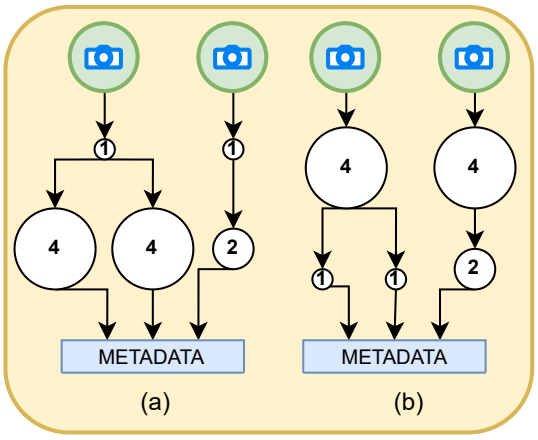


Fig. 12: Comparison of two sets of elasticity coefficients for the 2OD-3OC graph. Node size and value indicate elasticity coefficients, with larger nodes representing higher elasticity. (a) shows lower elasticity in object detection models, while (b) shows lower elasticity in object classification models.

E. Performance of Local and Global Schedulers

The local scheduler uses a greedy heuristic approach with a runtime overhead of approximately 2 ms, which means that it needs time to gradually adjust the workload imbalances across different hardware units. This adjustment process involves scaling the model effectively to meet the deadline. Consequently, this leads to significant fluctuations in both the makespan and combined utility, as shown in Figure 13. The global scheduler which is based on the optimal ILP solution, maintains the accuracy and the performance requirement for 30 FPS. However, the overhead of ILP is not included in the makespan. The overhead is approximately 100 ms which makes it infeasible to be used at every frame.

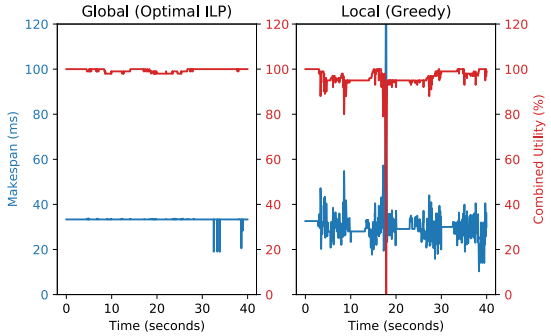


Fig. 13: Although the global scheduler is optimal, it is infeasible to be used for decision-making at every frame due to high overhead.

TABLE V: Comparison between *EMERALD* and local scheduler over 30000 random scenes.

	EMERALD	Local
Deadline Misses	1747 (5.83%)	2837 (9.46%)
Dropped Frames	0 (0.00%)	1 (0.01%)
Avg. Makespan	25.83	29.48
Norm. Utility (%)	97%	97%
Overhead (ms)	3.5	2.6

F. Performance of EMERALD vs The Local Scheduler

As shown in Table V, *EMERALD* outperforms the local scheduler in terms of deadline misses and average makespan, while maintaining similar scheduling overhead. This efficiency is achieved because the global scheduler is invoked infrequently, avoiding significant overhead increases. Furthermore, the global scheduler helps *EMERALD* to quickly revert to higher input scales without waiting for a scale-up threshold, leading to better accuracy utility.

VI. Conclusions

Existing non-elastic and elastic methods offer valuable scheduling frameworks for scheduling DNN workloads, but they fail to address the specific challenges of resource-constrained, multi-DNN workloads on heterogeneous MPSoCs. This paper introduced a novel elastic scheduling framework that fills this gap by integrating considerations for scene complexity, batching, input scaling, and hardware heterogeneity. Our framework effectively enhances both efficiency and performance under dynamic, real-time constraints.

The proposed scheduler is tailored for multi-tenant DNN workloads on edge MPSoCs, prioritizing real-time deadlines while preserving end-to-end accuracy. By dynamically adjusting input resolutions in response to deadline violations, it achieves significant reductions in response times without compromising accuracy. The framework demonstrates notable performance improvements, achieving 11x and 12.3x reductions in missed deadlines, alongside 34% and 40% improvements in average accuracy compared to *CAMDNN* and *HEFT*, respectively.

This work highlights the critical need for an elastic scheduler capable of adapting to fluctuating workloads at runtime, providing a robust solution for real-time DNN execution in resource-constrained environments.

Acknowledgments

This research was supported in part by NSF Grant Numbers 2008244, CCF-1652132, and CCF-1618039, by the Center for Embedded Systems, NSF Grant 1361926, and by the Center for Intelligent, Distributed, Embedded, Applications and Systems, NSF Grant 2231620.

References

[1] M. Bojarski, C. Chen, J. Daw, A. Değirmenci, J. Deri, B. Firner, B. Flepp, S. Gogri, J. Hong, L. Jackel, et al. The NVIDIA PilotNet experiments. *arXiv preprint arXiv:2010.08776*, 2020.

[2] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings 19th IEEE Real-Time Systems Symposium*, pages 286–295. IEEE, 1998.

[3] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.

[4] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[5] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[6] M. Farhadi, M. Ghasemi, S. Vrudhula, and Y. Yang. Enabling incremental knowledge transfer for object detection at the edge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 396–397, 2020.

[7] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[8] M. Ghasemi, S. Heidari, Y. G. Kim, A. Lamb, C.-J. Wu, and S. Vrudhula. Energy-efficient mapping for a network of DNN models at the edge. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 25–30. IEEE, 2021.

[9] M. Ghasemi, D. Rakhmatov, C.-J. Wu, and S. Vrudhula. Edgewise: Energy-efficient CNN computation on edge devices under stochastic communication delays. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(5):1–27, 2022.

[10] S. Heidari, M. Ghasemi, Y. G. Kim, C.-J. Wu, and S. Vrudhula. CAMDNN: Content-aware mapping of a network of deep neural networks on edge MPSoCs. *IEEE Transactions on Computers*, 71(12):3191–3202, 2022.

[11] Y. Hu, S. Sun, J. Li, X. Wang, and Q. Gu. A novel channel pruning method for deep neural network compression. *arXiv preprint arXiv:1805.11394*, 2018.

[12] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool. Ai benchmark: All about deep learning on smartphones in 2019. In *CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3617–3635.

[13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[14] J. S. Jeong, J. Lee, D. Kim, C. Jeon, C. Jeong, Y. Lee, and B.-G. Chun. Band: coordinated multi-DNN inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 235–247, 2022.

[15] Y. Ji, L. Liang, L. Deng, Y. Zhang, Y. Zhang, and Y. Xie. Tetris: Tile-matching the tremendous irregular sparsity. *Advances in neural information processing systems*, 31, 2018.

[16] S. Kim, H. Kwon, J. Song, J. Jo, Y.-H. Chen, L. Lai, and V. Chandra. Dream: A dynamic scheduler for dynamic real-time multi-model ml workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 73–86, 2023.

[17] Y. G. Kim and C.-J. Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *MICRO*, 2020.

[18] H. Kwon, K. Nair, J. Seo, J. Yik, D. Mohapatra, D. Zhan, J. Song, P. Capak, P. Zhang, P. Vajda, et al. Xrbench: An extended reality (XR) machine learning benchmark suite for the metaverse. *Proceedings of Machine Learning and Systems*, 5:1–20, 2023.

[19] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th annual international conference on mobile computing and networking*, pages 1–15, 2020.

[20] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[21] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[22] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient ConvNets. *arXiv preprint arXiv:1608.08710*, 2016.

[23] L. Liu, J. Tang, S. Liu, B. Yu, Y. Xie, and J.-L. Gaudiot. π-rt: A runtime framework to enable energy-efficient real-time robotic vision applications on heterogeneous architectures. *Computer*, 54(4):14–25, 2021.

[24] J. Mack, S. E. Arda, U. Y. Ogras, and A. Akoglu. Performant, multi-objective scheduling of highly interleaved task graphs on heterogeneous system on chip devices. *IEEE Transactions on Parallel and Distributed Systems*, 33(9):2148–2162, 2021.

[25] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *HotNets*, pages 50–56, 2016.

[26] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288. 2019.

[27] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen. MoDNN: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1396–1401. IEEE, 2017.

[28] V. Nigade, P. Bauszat, H. Bal, and L. Wang. Jellyfish: Timely inference serving for dynamic edge networks. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 277–290. IEEE, 2022.

[29] E. Park, S. Yoo, and P. Vajda. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 580–595, 2018.

[30] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[31] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 111. IEEE, 2004.

[32] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, 2018.

[33] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[34] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[35] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra. High-throughput CNN inference on embedded ARM big. little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2019.

[36] Z. Wang, W. Ren, and Q. Qiu. LaneNet: Real-time lane detection networks for autonomous driving. *arXiv preprint arXiv:1807.01726*, 2018.

[37] L. A. Wolsey. *Integer Programming*. John Wiley & Sons, 2020.

[38] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[39] F. Xue, V. Likhosherstov, A. Arnab, N. Houlsby, M. Dehghani, and Y. You. Adaptive computation with elastic input sequence. *arXiv preprint arXiv:2301.13195*, 2023.

[40] J. Yi, S. Choi, and Y. Lee. EagleEye: Wearable camera-based person identification in crowded urban spaces. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[41] X. Zeng, B. Fang, H. Shen, and M. Zhang. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 409–421, 2020.

[42] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg. Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[43] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang. A systematic DNN weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European conference on computer vision (ECCV)*, pages 184–199, 2018.

[44] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *INFOCOM*. IEEE, 2019.

[45] Z. Zhao, K. M. Barijough, and A. Gerstlauer. DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.