Orchestration of Perception Systems for Reliable Performance in Heterogeneous Platforms

Anirban Ghose, Srijeeta Maity, Arijit Kar, Soumyajit Dey
Indian Institute of Technology, Kharagpur
Email: anirban.ghose@cse.iitkgp.ac.in,{srijeeta.maity, arijit.kar14}@iitkgp.ac.in, soumya@cse.iitkgp.ac.in

***Abstract*—Delivering driving comfort in this age of connected mobility is one of the primary goals of semi-autonomous perception systems increasingly being used in modern automotives. The performance of such perception systems is a function of execution rate which demands on-board platform-level support. With the advent of GPGPU compute support in automobiles, there exists an opportunity to adaptively enable higher execution rates for such Advanced Driver Assistant System tasks (ADAS tasks) subject to different vehicular driving contexts. This can be achieved through a combination of program level locality optimizations such as kernel fusion, thread coarsening and core-level DVFS techniques while keeping in mind their effects on task-level deadline requirements and platform-level thermal reliability. In this communication, we present a future-proof, learning-based adaptive scheduling framework that strives to deliver reliable and predictable performance of ADAS tasks while accommodating for increased task-level throughput requirements.**

***Index Terms*—ADAS, OpenCL, Machine Learning, Control Theory, Heterogeneous Multicore, Real Time Scheduling**

I. Introduction

Perception systems in the context of modern automotives present a myriad of challenges for delivering the best possible inference results in a timely manner for diverse vehicular driving contexts. Specifically, for Advanced Driver-Assistance Systems (ADAS), there exist inherent scenario dependent performance requirements in the form of desired levels of detection accuracy for different vision pipelines executing on automotive electronic control units (ECUs). Increase in accuracy requirements for ADAS tasks translate to increase in execution rates at which frames obtained from automobile camera feeds are processed by individual pipelines. For example, we would want a pedestrian detection system to process input frames at a higher rate for increased object detection accuracy when the on-board GPS suggests that the vehicle is approaching a congested area. Similarly, different ADAS tasks impose different frames per second (FPS) requirements for different detection tasks subject to different driving contexts. Sustaining such time-varying FPS requirements for different vision pipelines require designing solutions that allocate resources intelligently in the underlying hardware so as to ensure predictable performance guarantees in terms of real time mapping and scheduling.

While static scheduling based task-core mapping schemes guarantee predictable latency of detection tasks, such approaches are inherently conservative in nature since they consider the worst case execution time estimates of tasks. Scheduling decisions in this context lead to over-provisioning resources and bandwidth which would not be able to handle extremal performance requirements.

The authors acknowledge Intel's support through a PhD fellowship.

Existing works in ADAS scheduling optimize the execution of vision pipelines on GPUs by i) decreasing the overall latency of detection jobs via software pipelining [1], ii) leveraging algorithmic fusion techniques for processing multiple frames concurrently [1] or iii) opting for a combination of fusion and concurrent kernel execution [2]. However, such approaches are GPU-only whereas modern automotive SoCs are heterogeneous with on-board CPU, GPU, FPGA and neural accelerators. This increases mapping options but also makes the design space complex to explore. Such works are also not equipped to alter task-mapping decisions subject to the time varying dynamic FPS requirements dictated by driving contexts as discussed earlier. Fusing and processing frames concurrently on hardware impose a high memory footprint for selected pipelines and may not be a feasible approach always. In such cases, kernel level optimizations such as coarsening and task fusion for controlling the amount of work per thread and overall memory demands become increasingly important. Thus, even for the same set of co-executing pipelines, different choice and degree of optimizations are likely to be useful under different load scenarios.

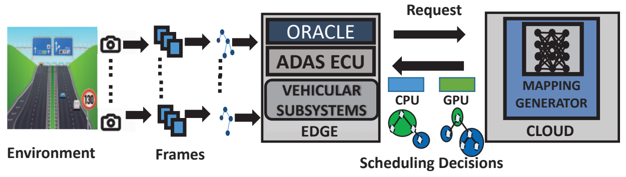


Fig. 1. Runtime System Overview

In this work, we propose an intelligent runtime system that can efficiently schedule ADAS vision pipelines in next-generation automotive embedded platforms in a self-learning fashion so that i) the time varying dynamic detection requirements for each pipeline, ii) platform-level resource constraints and iii) task-level deadline requirements are respected. An overview of the proposed software architecture for our runtime system is depicted in Fig. 1.

We assume the existence of an oracle operating as a service in the vehicle software stack which is i) continuously monitoring environmental parameters such as vehicle terrain, ii) observing the detection results of existing vision pipelines, iii) interacting with on-board sensors (e.g. GPS) and iv) generating requests characterizing FPS requirements for corresponding vision pipelines. These requests are communicated to a mapping generator executing as a service on a cloud server which leverages a learned model trained with Reinforcement Learning (RL) techniques. The mapping generator in our context is meta-level in the sense that it uses this trained model for obtaining a set of task-device mapping decisions that may be admissible at runtime. The learned model suggests on-the-fly mappings which exploit a combination of program-level and device-level optimizations for heterogeneous architectures such as - i) remapping threads of a data parallel task from one device to another ii) reducing the number of threads while increasing the work per thread of a data parallel task (coarsening), iii) merging the computation involved in two different tasks (fusion), iv) altering core-level frequencies of devices by using DVFS techniques and v) a combination of the above. The Reinforcement Learning framework learns the mutual interactions between such optimizations through experience. The mapping decisions are communicated to the ADAS ECU. We note that the decisions reported by the mapping generator are based on ground-truth learned models which assume that task-level execution times are within pre-observed ranges for all invocations on the device. Realistically speaking, this will not be true due to memory contention arising from shared resource access by concurrent tasks. Such mispredicted mappings can lead to high deadline miss rates. For run-time performance recovery in such cases, our low-level scheduler employs control-theoretic tuning of core level DVFS. In summary, we stake claim to the following contributions.

• We characterize and implement an RL based mapping generator for real time ADAS tasks which admits dynamic perception requirements and generates mapping decisions with predictable execution time for heterogeneous embedded platforms.

• We implement a low level runtime environment which leverages a control-theoretic scheduling scheme that mitigates potential deadline misses, thus achieving performance guarantees even for bad quality mapping decisions.

• We provide extensive validation results justifying the usefulness of our scheme on state-of-the-art CNN pipelines executing on an Odroid XU4 platform.

II. Problem Formulation

Let $\mathcal{J} = \{G_1, G_2, \cdots, G_N\}$ be the set of N ADAS pipelines to be scheduled. Each such pipeline is represented by a directed acyclic graph (DAG) $G_k = \langle T_k, E_k \rangle$ where $T_k = \{t_1^k, t_2^k, \cdots, t_n^k\}$ denotes the set of tasks, $E_k \subseteq T_k \times T_k$ denotes the set of edges where each edge (t_i^k, t_j^k) denotes that task t_j^k cannot start execution until and unless t_i^k has finished for a DAG G_k. In the context of such vision pipelines, each task is a computational kernel representing some data parallel computation [3]. Given $\mathcal{J}$, we represent an oracle request as $\mathcal{R}$ = $\{\langle G_1, w_1, p_1\rangle, \langle G_2, w_2, p_2\rangle, \cdots, \langle G_N, w_N, p_N\rangle\}$ where each tuple dictates that each job $G_k \in \mathcal{J}$ arrives periodically with period p_k and processes w_k frames in each execution instance. The set $\mathcal{R}$ specifying such FPS requirements is determined by the oracle subject to observations of the current driving context and remains fixed as long as the context remains unchanged.

Given $\mathcal{R}$, let us denote $\mathcal{G}_k = \{G_k^1, \cdots, G_k^h\}$ as the set of all execution instances of DAG G_k where $h = H/p_k$ and H is the hyper-period which is the l.c.m of the periods (inverse of rate) of DAGs as specified. We denote the j^{th} execution instance of a DAG G_k as G_k^j and the i^{th} task belonging to it as $t_i^{j,k}$. We define the set $\mathcal{G} = \mathcal{G}_1 \cup \mathcal{G}_2 \cup \cdots \cup \mathcal{G}_N$ to be the set of all execution instances of all DAGs in the job set $\mathcal{J}$ executing in the hyper-period H.

Given $\mathcal{G}$, we denote a hyper-period snapshot $\mathcal{H}$ to be the set of tasks of each DAG execution instance $G_k^j \in \mathcal{G}$ waiting for a dispatch decision. We say that every DAG $G_k^j \in \mathcal{G}$ has finished execution *iff* $\mathcal{H}$ becomes empty. We denote $\mathcal{F}(\mathcal{H})$ as the head of $\mathcal{H}$ comprising tasks that are ready to execute and define it as follows.

Given a hyper-period snapshot $\mathcal{H}$, the **frontier** $\mathcal{F}(\mathcal{H})$ is a set of independent tasks belonging to a subset of DAGs $\mathcal{G}' \subseteq \mathcal{G}$ such that the following precedence constraints hold: i) each DAG $G_k^j = \langle T_k^j, E_k^j \rangle \in \mathcal{G}'$ must finish execution before G_k^{j+1} and ii) for each DAG $G_k^j \in \mathcal{G}'$, predecessors of each task $t_i^{j,k} \in T_k^j$ belonging to $\mathcal{F}(\mathcal{H})$ i.e tasks $t_l^{j,k}$ such that $(t_l^{j,k}, t_i^{j,k}) \in E_k^j$ have finished execution. The frontier $\mathcal{F}$ is ordered by the following ranking measures.

1. The rank measure **blevel** of a task t in DAG G=$\langle T, E\rangle$ represents the best case execution time estimate to finish tasks in the longest path starting from t to a task that has no successors in G, assuming all resources are available and is computed as $blevel(t)$=e_t+$\max_{t' \in succ(t)} blevel(t')$, where e_t is the worst case execution time (WCET) of task t and $succ(t) = \{t' | (t, t') \in E\}$.

2. The rank measure **local deadline** of a task t in DAG instance G=$\langle T, E\rangle$ represents the absolute deadline of t and is computed as $local\_deadline = d - blevel(t) + e_t$, where d is the absolute deadline of G, $blevel$ is the aforementioned rank measure and e_t is the WCET of t.

Task execution flow: Considering a frontier of tasks $\mathcal{F}(\mathcal{H})$ for some hyper-period snapshot $\mathcal{H}$ sorted by the *local deadline* measure, the set of available devices in a heterogeneous platform $\mathcal{P}$, and the task $t_{min} \in \mathcal{F}(\mathcal{H})$ with the minimum *local deadline* as inputs, let us define the mapping function $\mathcal{M}$. The function returns a task-device mapping $m = \langle T, P\rangle$ where T comprises a set of tasks comprising the task t_{min} and its descendants. The quantity $P \in \mathcal{P}$ represents one particular device. The choice of constituent tasks in T is motivated by the fact that in certain runtime contexts, multiple tasks/kernels fused and mapped to P offer better speedups and reduced memory usage. In addition to fusion, coarsening schemes can be applied to alter the thread-data mapping layout for the entire fused kernel, leading to further improvement in terms of resource utilization, register/cache usage and overall decrease in launch overhead of individual tasks. Related works have been proposed over the years which investigate the efficacy of kernel fusion [4] and thread coarsening [5] separately on heterogeneous CPU-GPU architectures by considering different runtime contexts. The focal objective of the proposed work is in learning these contexts in the form of a policy function π that can be subsequently used to design the function $\mathcal{M}$. The reason for leveraging a learning based approach can be attributed to the large space of scheduling decisions that are possible using kernel fusion alone. Considering a DAG G of depth D, we assume only vertical fusion i.e. all tasks selected upto a particular depth in the mapping decision $m = \langle T, P \rangle$ are fused and mapped to P. For the fused task T, coarsening is applied to each constituent kernel such that they execute in the minimum execution time possible. The total number of possible mapping configurations for a DAG G of depth D is equal to the number of integer compositions of D which is 2^D. Furthermore, since each fusion based mapping configuration has the option of getting mapped to a CPU or GPU device, the total number of possible scheduling decisions for a single DAG is actually $\omega(2^D)$. Considering a total of $M = |\mathcal{G}|$ DAG instances for an oracle request $\mathcal{R}$, the total space of scheduling decisions is therefore $\omega(2^{MD})$.

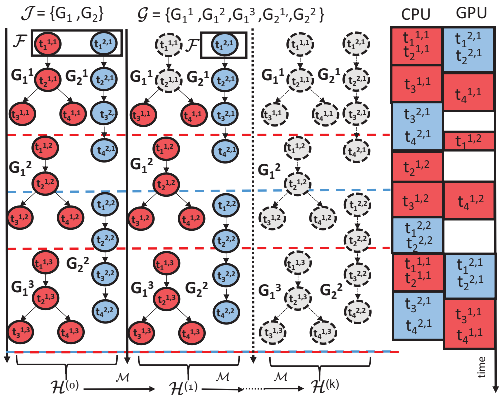


Fig. 2. RL Assisted Task Mapping

We next elaborate with an illustrative example how $\mathcal{M}$ is leveraged to ascertain mapping decisions in this exponential search space. Given $t_{min} \in \mathcal{F}(\mathcal{H})$, the function $\mathcal{M}$ is applied on $\mathcal{H}$ to yield a new hyper-period snapshot $\mathcal{H}'$ and a new frontier $\mathcal{F}(\mathcal{H}')$. This process of applying $\mathcal{M}$ and generating subsequent hyper-period snapshots is continued until each DAG in $\mathcal{G}$ has been scheduled. We present a representative example depicting a complete schedule of task-device mappings for a job set $\mathcal{J} = \{G_1, G_2\}$ in Fig. 2.
Given a hyper-period H, there exists three instances of job G_1 (deadlines are shown using dashed red lines) and two instances of job G_2 (deadlines are shown using dotted blue lines). The set $\mathcal{G}$ is therefore $\{G_1^1, G_1^2, G_1^3, G_2^1, G_2^2\}$. Initially the frontier contains tasks G_1^1 and G_2^1 which have no predecessors i.e. $\mathcal{F} = \{t_1^{1,1}, t_1^{2,1}\}$. Tasks are selected from $\mathcal{F}$ based on the *local deadline* rank measure. The mapping function $\mathcal{M}$ generates a task-device mapping decision $m = \langle \{t_1^{1,1}, t_2^{1,1}\}, CPU \rangle$. This is shown in the Gantt chart in the right hand side of Fig. 2. This in turn creates the hyper-period snapshot $\mathcal{H}^{(1)}$ as depicted in the figure. The corresponding list of mapping decisions is depicted in the Gantt chart schedule in Fig. 2. We summarize our formulation of learning based dispatch as follows.

Given a set $\mathcal{G}$ constructed from a job set $\mathcal{J}$ of ADAS detection pipelines for a given oracle request $\mathcal{R}$ to be executed on the heterogeneous platform $\mathcal{P}$, the objective of the proposed scheduling scheme is to leverage a policy function π trained using RL which dictates the choice of task-device mapping function $\mathcal{M}$.

We note that the set of task-mapping decisions reported by π are platform agnostic and does not take into account variations in latency arising from platform-level interference factors such as thermal throttling, memory thrashing, shared memory contention by other tasks etc. This variation may potentially lead to scenarios where the observed deadline miss rate exceeds the predicted deadline miss rate. For handling cases where the actual behaviour differs from that modeled by the AI component, several control-theoretic scheduling solutions [6] have been proposed in the recent past. These solutions employ lightweight controllers that dynamically tune core-level frequencies of the heterogeneous platform based on differences between actual and expected execution times of tasks and provide formal guarantees with respect to meeting task-level latency goals. In the proposed work, the low level scheduler leverages a state-of-the-art pole-based control-theoretic DVFS scheme [6] while mapping tasks following the decisions reported by the policy π. The latency goals in our context constitute permissible slack constraints (discussed later) for each task which ensure that deadline requirements for individual pipelines are met.

To summarize, the proposed system-level solution for real time ADAS scheduling therefore operates in two distinct phases- i) an AI enabled approach which searches through the exponential space of scheduling decisions and intelligently selects a global set of task-mapping decisions for each ADAS pipeline and ii) a control-theoretic scheme which performs locally for a particular pipeline. Given the requirement of processing an exponential search space for the possibly best global scheduling decisions, we execute the first phase on a cloud server while reserving resources on the ECU for exclusively executing the second phase. This combined approach ensures opportunistic switching to high frequency mode only when required thus reducing thermal induced degradation of lifetime reliability for the overall platform; something which can happen with pure frequency scaling based task scheduling techniques.

III. Methodology

In the recent past, several works [7], [8] advocate usage of deep reinforcement learning methods to learn an optimal policy function for solving scheduling problems. Given the exponential search space of decisions, we employ sample efficient RL methods such as *Deep Q-learning* with experience replay that would learn a state-action value function $\mathcal{Q}(s, a)$ characterizing the goodness of choosing an action a given a state s. The overall methodology depicting the training process for ascertaining $\mathcal{Q}(s, a)$, followed by subsequent usage of the trained model in the deployment phase is illustrated in Fig. 3.
Training Phase: Given a set of oracle requests pertaining to a set of ADAS jobs $\mathcal{J}$ and the set of target platform devices $\mathcal{P}$, the output of the training phase is a state-action value function $\mathcal{Q}$. For each such oracle request, the training process involves the following sequence of steps for updating the network weights of $\mathcal{Q}(s, a)$.
(i) State Extraction: Given an oracle request $\mathcal{R}$, the hyper-period snapshot $\mathcal{H}$ is first constructed. The observed state vector s for $\mathcal{H}$ at any point of time is given by $[dt_1, dt_2, \cdots, dt_{|\mathcal{P}|}, rt_1, rt_2, \cdots, rt_{|\mathcal{J}|}, dr_1, dr_2, \cdots, dr_{|\mathcal{J}|}]$

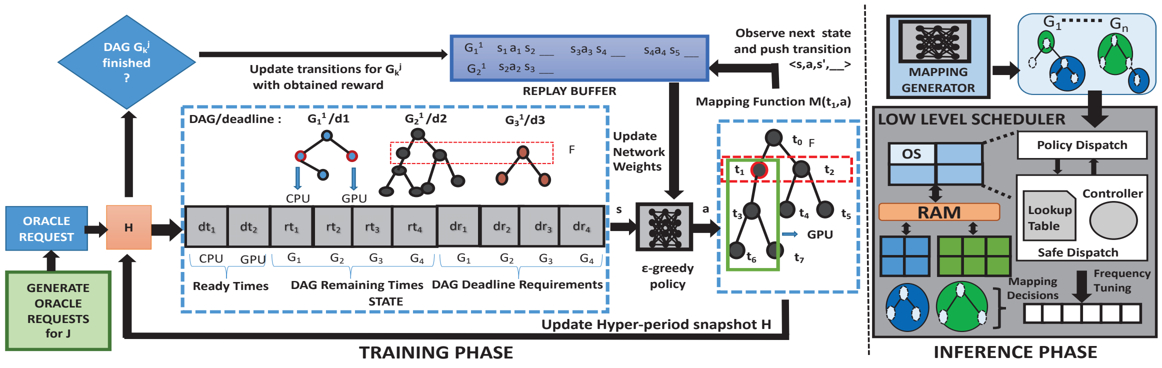


Fig. 3. Training and Inference Methodology Overview

where dt_i represents the time left for the i^{th} device $\in \mathcal{P}$ to become free. The quantity rt_i denotes the 'best case' time estimate remaining for the currently executing instance of DAG in $G_i = \langle T_i, E_i \rangle \in \mathcal{J}$ to finish. This is calculated by the *blevel* estimate of the current task $t \in T_i \in \mathcal{F}$. The quantity dr_i represents the 'time to deadline' estimate for the same instance of the i^{th} DAG in $\mathcal{J}$ to finish so that the deadline of G_i is respected. This is calculated from the difference between the time elapsed since the beginning of the hyper-period and the absolute deadline of the DAG instance G_i in that hyper-period. These three quantities together capture resource availability (dt_i), estimate of the 'best case' time for a DAG to finish completely (rt_i), and estimate of time by which the DAG must finish to meet deadline constraints (determined by dr_i). In Fig. 3, considering a system with 1 CPU and 1 GPU device and a total of 4 periodic DAGs, a sample state vector is depicted.

(ii) Action Selection: Given the state vector s from the previous step, the action a is determined using traditional $\epsilon-$greedy policy schemes. Given the action a and the task with the earliest local deadline $t_{min} \in \mathcal{F}(\mathcal{H})$, we define the mapping function $\mathcal{M}(t_{min}, a) = \langle T, P \rangle$ where T represents constituent tasks comprising t_{min} and its descendants up to a depth d and $P \in \mathcal{P}$ is a device. In our setting, the value of the action a can be used to infer both d and P as follows. An action a is represented as an integer $a \in [0, \cdots, n(D+1)-1]$ where D denotes the maximum height of a DAG $G_i \in \mathcal{J}$ and $|\mathcal{P}| = n$ is the total number of devices/cores in the target platform. An action value $a = i(D+1)+d$ represents fusing a task with descendants up to depth d and mapping to the i-th device so that given a, we have fusion depth $d = a\%(D+1)$ and device id $i = a/(D+1)$ considering $[0, D]$ as domain of depth values and $[0, n-1]$ as domain of device ids. We note that for each task for $t \in T$, selected as a result of action a, the best coarsening factors are used that would return the minimum possible execution time.

(iii) Reward Assignment: After the application of $\mathcal{M}(t_{min}, a)$, the current hyper-period snapshot $\mathcal{H}$ is updated and the subsequent next state s' is observed. An incomplete transition tuple of the form $\langle s, a, s'\_ \rangle$ is pushed to a replay buffer memory $\mathcal{B}$ which is used during training updates. The set $\mathcal{B}$ is populated by tuples for each action mapping decision of some task $t_i^{j,k}$ belonging to DAG instance G_k^j. The final empty element of the tuple represents the future reward r for this transition which is updated once G_k^j finishes execution. A reward of value of -1 is assigned if it is observed that the finishing timestamp of G_k^j exceeds its absolute deadline and a reward of $+1$ for the alternate case. The same reward value r is used to update every incomplete transition tuple in $\mathcal{B}$ pertaining to each task action mapping decision taken during the lifetime of G_k^j.

(iv) Training Updates: Training updates are done by sampling the replay memory $\mathcal{B}$ randomly, constructing a set $B \subseteq \mathcal{B}$ of complete transition tuples and calculating the average loss function $\mathcal{L} = \frac{1}{||B||} \left\{ \sum_{(s,a,r,s') \in \mathcal{B}} L(\delta(s,a,r,s')) \right\}$ where L is the Huber Loss function [9] of the error term $\delta(s,a,r,s')$. The quantity $\delta(s,a,r,s')$ represents the temporal difference (TD) error for a transition tuple (s,a,r,s'). We leverage the standard Double DQN (DDQN) [9] approach where $\delta(s,a,r,s') = \mathcal{Q}(s,a) - (r + \gamma \mathcal{Q}(s', argmax_a \mathcal{Q}(s',a)))$, where γ represents the discount factor and is set to 1 given the episodic setting of our problem. Given the oracle request $\mathcal{R}$, the network is updated multiple times by mini-batch gradient descent using the aforementioned loss function.

The four steps mentioned above and used for updating the network $\mathcal{Q}$ is repeated for each oracle request $\mathcal{R}_i$, for *num_runs* number of times. This entire process is further repeated for a number of training epochs until the average reward observed for the oracle requests converge. The trained network $\mathcal{Q}$ is used to obtain the corresponding optimal policy $\pi^*(s) = argmax_a \mathcal{Q}(s,a)$ which is leveraged in the deployment phase.

Deployment Phase: Given an oracle request $\mathcal{R}_i$ and resulting set of DAG instances $\mathcal{G}$, the mapping generator considers the hyper-period snapshot $\mathcal{H}$ corresponding to $\mathcal{R}_i$ and iteratively performs the following steps - i) observes state s, ii) uses $\pi^*(s)$ to select action a, iii) applies mapping function $\mathcal{M}$ using selected action a on $\mathcal{H}$. The generator invokes multiple inference passes over the learned network until $\mathcal{H}$ becomes empty, finally yielding a sequence of task-device mapping decisions $\Pi(\mathcal{G})$, essentially a task schedule. The mapping generator uses the available WCET estimates of tasks, simulates a run of the task schedule and assesses the percentage of deadline misses. The sequence $\Pi(\mathcal{G})$ is communicated to the low-level scheduler if the miss percentage is below a permissible threshold. Otherwise, the oracle request is neglected.

For every mapping decision $m = \langle T, P \rangle$ pertaining to some DAG instance G_k^j in the task schedule $\Pi(\mathcal{G})$, the low-level scheduler executes T (a single task or a fused variant) on device P and then computes a slack measure $sl(G_k^j)$ = $D(T) - wrt(T)$, where $D(T)$ denotes the current time remaining to meet the deadline for the DAG instance G_k^j and $wrt(T) = \sum_{t \in SUCC(T)} (e_t)$ denotes the worst case remaining time for executing tasks belonging to the set $SUCC(T)$ which comprises all descendants of tasks in T in G_k^j. For ensuring that deadline requirements are met at runtime, the scheduler checks whether $sl(G_k^j) \geq \delta sl$ is satisfied. Here we use δsl to model the overall uncertainty associated with the WCET estimates of the constituent tasks of the DAG. If it is observed that the inequality is not satisfied, the scheduling scheme engages into a safe mode of dispatch and increases the core frequency of P to some frequency f for the subsequent mapping decision $m' = \langle T', P' \rangle$ in the schedule. The value of f is obtained using the speedup equation [6], $sp(n) = sp(n-1) + \rho * err / b(T')$, where $sp(n-1)$ denotes the last i.e. $(n-1)^{th}$ speedup requirement imposed on T' when the slack constraint was violated. The quantity $sp(n)$ denotes the current speedup requirement such that the error term $err = sl(G_k^j) - \delta sl$ is rendered positive. The quantity $b(T') = \sum_{t \in T'} e_t$ represents the WCET estimate for executing T' at the baseline frequency and ρ represents the pole value of the controller updating f. Given the required speedup, the scheme uses lookup tables computed offline that map speedup values of task T' to core frequency values of the device P' to obtain the required operating frequency f. We note that the scheduling scheme continues to operate in this safe mode for each successive mapping decision in the schedule until it is ensured for G_k^j that $sl(G_k^j) \geq \delta sl$.

IV. Experimental Results

We consider the Odroid XU4 embedded heterogeneous platform comprising two quad-core ARM CPUs (big and LITTLE), and one Mali GPU. We leverage the OpenCL framework [3] for collaboratively executing object detection pipelines on the big CPU and Mali GPU with the OS mapped to the LITTLE CPU. We have created four OpenCL based representative object detection pipelines where two pipelines (G_1 and G_2) are based on LeNet architecture [10] each comprising 9 tasks and the remaining two (G_3 and G_4) are based on YOLO-Lite [11] each comprising 12 tasks. The pipelines are implemented using optimized data parallel kernels (such as convolution, general matrix multiplication, pooling, softmax etc.) available in the ARM OpenCL SDK [12]. Our experiments require profiling data for each task as well as each fused task variant on the target platform for setting up our training environment. In this context, a code template generator is implemented that automatically generates coarsened OpenCL code for all possible fused task variants for each pipeline ($9 \times (9-1)/2 = 36$ for the LeNet architecture and 66 for the Yolo-Lite architecture.).

Environment Setup The WCET estimates of each task and each fused task variant in each of the pipelines are obtained by leveraging a co-run degradation based profiling approach outlined in [13]. While profiling each benchmark on a particular device (big CPU or Mali GPU), a micro-kernel benchmark is executed continuously on the other device in parallel for ensuring maximum shared memory interference. The WCET estimates $\tau_{CPU}(t_i)$ and $\tau_{GPU}(t_i)$ represent the time taken (averaged over 10 profiling runs) to execute the task t_i on the CPU or GPU device respectively in the worst possible scenario when the system memory bandwidth is completely exploited. The uncertainty parameter δsl for t_i is computed by taking the variance of execution times in these profiling runs. Using the individual WCET estimates, the overall WCET of the DAG $G_i = \langle T_i, E_i \rangle$ is given by $\tau(G_i) = \sum_{t \in T_i} max(\tau_{CPU}(t_i), \tau_{GPU}(t_i))$.

In our experiments, the oracle request $\mathcal{R}$ is represented by $\{\langle G_1, w_1, p_1 \rangle, \langle G_2, w_2, p_2 \rangle, \langle G_3, w_3, p_3 \rangle, \langle G_4, w_4, p_4 \rangle\}$, with $w_i = 1$ for all DAGs. We vary p_i for each G_i with values from the set $\{\tau(G_i), 1.5 * \tau(G_i), 2 * \tau(G_i)\}$ for generating multiple oracle requests. Since each DAG processes one frame at a time and can arrive using one of the three period values, the total number of oracle requests possible for the job set comprising 4 DAGs is $3^4 = 81$. We train our DDQN using these 81 oracle requests and by setting the number of training runs per epoch i.e. num_runs to 100.

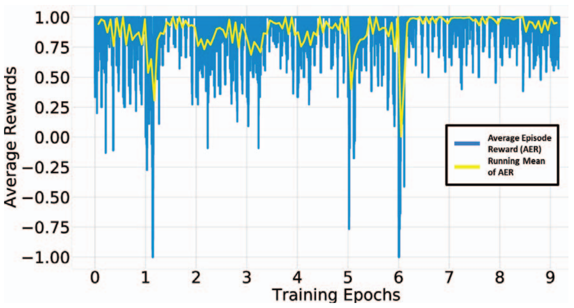
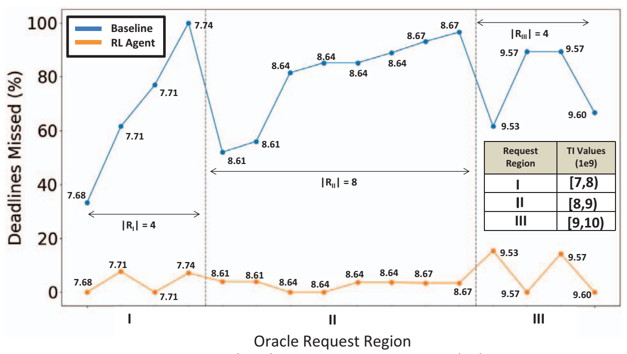


Fig. 4. RL Training Results

Training Results The training algorithm processes the same oracle request $\mathcal{R}$, i.e. it explores schedules for the same resultant set of DAGs $\mathcal{G}$ for a total of 100 episodes before processing the next oracle request $\mathcal{R}'$. Additionally, when the training algorithm moves from processing one request $\mathcal{R}$ to the next request $\mathcal{R}'$, it is ensured that only one period value in the request $\mathcal{R}$ is changed to yield $\mathcal{R}'$. This is done so that during the training phase, after learning the Q-Network for a given oracle request $\mathcal{R}$, the RL environment is not drastically changed during processing of request $\mathcal{R}'$. The neural network architectures employed for $\mathcal{Q}$ contains 1 input layer of size 18, 2 hidden layers of size 30 each with ReLU activation and one output layer of size 24 equipped with a softmax function for predicting action probabilities. The corresponding training results are summarized in Fig. 4. The entire training procedure is repeated for a total of 9 epochs where each epoch consists of a total of $81 * 100 = 8100$ episodes. In the figure, the x-axis is labelled with the training epoch number. Each point of the blue line plot represents the average reward obtained for the decisions taken in each episode. The yellow line plot presents a general trend for the reward where each point represents the running mean of the average episode reward calculated over a consecutive set of 500 episodes. We can observe that average



Request Region	TI Values (1e9)
I	[7,8)
II	[8,9)
III	[9,10)

Fig. 5. Baseline vs. RL $|R_x|$ = # requests with $TI(\mathcal{R})$ in region x.

episode rewards continue to remain positive after epoch 6.

Testing Results: For comparing the schedules generated by our DDQN based RL scheme, we employ the global dynamic EDF scheduling algorithm for executing DAGs on multicore processors outlined in [14] as our baseline scheme. At any point of time, the algorithm considers a task $t_i \in \mathcal{F}$ with the minimum local deadline and simply dispatches it to the device on which the WCET of the task t i.e. $\tau(t_i)$ is minimum. For testing, we leverage the same DAGs but with different oracle request configurations that were not used at the time of training. We consider for each job G_i, period values p_i from the set $\{0.85 * \tau(G_i), 1.7 * \tau(G_i)\}$ and generate a testing set comprising $2^4 = 16$ oracle requests. For each such oracle request $\mathcal{R}$ we define a throughput index (TI) measure which represents the throughput requirement in terms of Floating Point Operations per second (FLOPs). This is calculated as $TI(\mathcal{R}) = \sum_i FLOPs(G_i) \times w_i/p_i$ where $FLOPs(G_i)$ represents the total number of floating point operations required for a detection pipeline, w_i and p_i are as discussed earlier. A comparative evaluation between the schedules observed using the baseline algorithm and the RL scheme for each of the 16 oracle requests is elaborated using Fig. 5. The blue line plot in Fig. 5, represents percentage of deadline misses observed in a hyper-period for schedules determined by the baseline algorithm and the orange line plot represents the same as obtained using RL. The y-axis represents the deadline miss percentage observed. The entire plot is sub-divided into three regions where the points in each region represent the deadline misses for a group of oracle requests as reported by both the schemes. The oracle requests in each region are characterized by a range of Throughput Index (TI) values as depicted in the table in the right hand side of Fig. 5. Inside each region, the oracle requests are sorted in order of T.I. values which are depicted in the line plot itself. We also note that multiple oracle requests in a region may possess the same T.I. value, since we are considering two instances of the same CNN architecture that have the same FLOPs values. It may be observed that in general, for the given set of oracle requests in all regions, the percentage of deadline misses for baseline varies from 40% upto 90% whereas it remains below 20% for the RL scheme.

Target Platform Results: We consider the permissible deadline miss percentage threshold th set by the mapping generator to be $th = 15\%$. For establishing the efficacy of our low level scheduler, we select admissible schedules for oracle requests belonging to region III and present our findings in Table I. Each row represents results for some request $\mathcal{R}$, with the resulting DAG set $\mathcal{G}$ suffering miss percentage d (characterized by $\langle|\mathcal{G}|, d\rangle$) as reported by the mapping generator. In each case, column 2/3 reports the actual deadline misses when the inferred schedule is deployed without/with the safe dispatch mode of the low level scheduler being engaged. One may observe for $\mathcal{G}$ in the first row, 2 out of 6 DAGs miss their deadlines when deployed without safe mode enabled, incurring a miss% of 33. This is reduced to 0 misses when safe mode is engaged.

TABLE I
LOW LEVEL SCHEDULING RESULTS

$\langle\lvert\mathcal{G}\rvert, d\%\rangle$	Deployed System %misses	Safe mode %misses
$\langle 6, 0\%\rangle$	$33\% > th$	$0\% < th$
$\langle 25, 4\%\rangle$	$20\% > th$	$12\% < th$
$\langle 26, 15\%\rangle$	$27\% > th$	$15\% < th$

In general, for each schedule suffering from high miss% under actual deployment without safe mode, the scheduler improves its performance with safe mode engaged. In this way, such dynamic fine-tuning handles the runtime uncertainty associated with AI based scheduling thus ensuring predictable latency of the perception systems.

V. Conclusion

Our proposed combination of RL and low-level performance recovery technique is possibly the first approach that synergizes AI techniques with real time control-theoretic scheduling techniques towards generating kernel-fusion and coarsening based runtime mapping decisions for real-time heterogeneous platforms accelerating ADAS workloads. Future work entails incorporating an edge to cloud feedback mechanism so that on-board mapping decision observations can be leveraged to refine the cloud based inference model using periodic updates.

References

[1] M. Yang and et al., “Re-thinking CNN Frameworks for Time-Sensitive Autonomous-Driving Applications: Addressing an Industrial Challenge,” in *RTAS 2019*.
[2] H. Zhou and et al., “Sˆ 3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads,” in *RTAS 2018*.
[3] J. Stone and et al., “Opencl: A parallel programming standard for heterogeneous computing systems,” *CiSE*, 2010.
[4] Y. Xing and et al., “DNNVM: End-to-end Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators,” *TCAD*, 2019.
[5] N. Stawinoga and T. Field, “Predictable thread coarsening,” *TACO*, 2018.
[6] N. Mishra and et al., “Caloree: Learning control for predictable latency and low energy,” in *SIGPLAN Notices 2018*.
[7] Z. Fang and et al., “Qos-aware Scheduling of Heterogeneous Servers for Inference in Deep Neural Networks,” in *CIKM 2017*.
[8] H. Mao and et al, “Learning scheduling algorithms for data processing clusters,” in *SIGCOMM*, 2019.
[9] H. Van Hasselt and et al., “Deep reinforcement learning with double q-learning,” in *AAAI 2016*.
[10] P. Sermanet and et al, “Traffic sign recognition with multi-scale convolutional networks,” in *IJCNN*, 2011.
[11] R. Huang and et al., “Yolo-lite: A real-time object detection algorithm optimized for non-gpu computers,” in *IEEE BigData*, 2018.
[12] ARM, “Mali OpenCL Compute SDK,” https://developer.arm.com/ip-products/processors/machine-learning/compute-library, 2013.
[13] Q. Zhu and et al., “Co-run Scheduling with Power Cap on Integrated CPU-GPU Systems,” in *IPDPS 2017*.
[14] M. Qamhieh and et al., “Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems,” in *RTNS 2013*.