Prophet: Realizing a Predictable Real-time Perception Pipeline for Autonomous Vehicles

Liangkai Liu*, Zheng Dong*, Yanzhi Wang†, and Weisong Shi‡
*Department of Computer Science, Wayne State University
†Department of Electrical & Computer Engineering, Northeastern University
‡Department of Computer and Information Sciences, University of Delaware

***Abstract*—We have witnessed the broad adoption of Deep Neural Networks (DNNs) in autonomous vehicles (AV). As a safety-critical system, deadline-based scheduling is used to guarantee the predictability of the AV system. However, non-negligible time variations exist for most DNN models in an AV system, even when the whole system is just running one model. The fact that multiple DNNs are running on the same platform makes the time variations issue even more severe. However, none of the existing works have thoroughly studied the root cause of the time variation issue. In the first part of the paper, we conducted a comprehensive empirical study. We found that the inference time variations for a single DNN model are mainly caused by the DNN's multi-stage/multi-branch structure, which has a dynamic number of proposals or raw points. In addition, we found that the uncoordinated contention and cooperation are the roots of the time variations for multi-tenant DNNs inference.**

Second, based on these insights, we proposed the *Prophet* system that addresses the time variations in the AV perception system in two steps. The first step is to predict the time variations based on the intermediate results like proposals and raw points. The second step is coordinating the multi-tenant DNNs to ensure the execution progress is close to each other. From the evaluation results on the KITTI dataset, the time prediction of a single model all achieve higher than 91% accuracy for Faster R-CNN, LaneNet, and PINet. Besides, the perception fusion delay is bounded to 150ms, and the fusion drop ratio is reduced from 5.4% to less than 1 percent.

***Index Terms*—deep neural networks, predictability, autonomous driving, end-to-end system**

I. Introduction

Owing to their high safety and efficiency, autonomous vehicles (AV) have attracted colossal investment and studies from industrial and academic communities [1]. Since the primary goal is to understand the driving environment and drive the vehicle safely, building a reliable and efficient computing stack becomes one of the fundamental challenges for autonomous driving [2]. Given the considerable success in many scenarios, Deep Neural Networks (DNNs) are widely deployed in the autonomous driving system for sensing and perception [3]. Typical examples include YOLOv3, SSD, and Faster R-CNN for object detection [4]–[6]; Deeplabv3 for image semantic segmentation [7], [8]; LaneNet and PINet for lane detection [9], [10]. There are two main reasons for the success of DNNs in autonomous driving systems. The first is the higher accuracy compared with traditional computer vision-based approaches [3]. The other is that DNNs can process raw data, making them suitable for autonomous driving vehicles since it generates terabytes of raw sensor data daily [11].

As a safety-critical system, autonomous driving sets high requirements in accuracy and latency [1]. The high accuracy of DNN-based algorithms promotes the development of autonomous vehicles. However, satisfying the real-time requirements of the sensing, perception, and planning tasks are still significant challenges. As DNN models are widely deployed in perception applications, guaranteeing the real-time execution of the DNN inference becomes the key to satisfying the real-time requirements of autonomous driving [3]. According to [12], when the vehicle drives at 40 km per hour in urban areas, autonomous functions should be effective every 1 meter with the task execution time less than 100ms.

Current AV systems rely on deadline-based scheduling to guarantee predictability, where the deadline is mostly set as the worst-case observed time. However, although the trained model has a fixed structure and weights, non-negligible time variations exist for most DNN models in AV systems, even in silo mode (the whole system is just running one DNN model) [13]–[15]. These inference time variations bring a big challenge for setting deadlines since enormous resources could be wasted if the scheduler manages resources based on the worst-case observed inference time. Prior work observed the time variations for DNN inference in mobile devices and found that inference time follows an approximately Gaussian distribution [13]. Another work on the anytime DNN system also observed the time variations issue and provided a Kalman Filter-based estimation for latency distribution [14]. D^3 is a work that addresses the time variations in AV systems with dynamic deadlines rather than static deadlines [16]. However, none of the existing approaches could handle huge time variations since they do not consider the roots causing DNN inference time variations. In addition to the time variations in silo mode, a real AV system usually consists of hundreds of tasks simultaneously (multi-tenant mode). Multi-tenant DNNs on the same platform compete for resources like GPU, memory, etc., making the time variations issue even more severe. However, none of the existing works have addressed this issue before.

Based on the comprehensive study in this paper, we derive two important insights. *In silo mode, the time variations of DNN inference have strong relationships with the DNN's structure and the runtime configurations.* If we can narrow

the inference time variations to a small range, the resource scheduling will be more efficient. From an empirical study on real AV perception systems, we found that the model structure mainly causes the inference time variations in the object and lane detection models. The multi-stage/multi-branch structure generates a dynamic number of object proposals/raw points, which causes the variances of DNN inference time [6], [9], [10], [17]. For multi-tenant DNN inference, we found that these tasks are competing for resources and collaborating with each other since the results of these DNN tasks need to be merged for the following planning and control modules [1]. This characteristic makes the multi-tenant DNNs inference show even higher time variations than the silo mode. *In multi-tenant mode, proper task coordination to manage the contention and cooperation between multi-tenant DNNs is the key to addressing the time variations issue.* Besides, understanding the DNN model's inference time variations brings some guidance for the multi-tenant DNN coordination.

In this paper, driven by the two insights from the empirical study, we propose *Prophet* solve the time variations issue in the AV perception system in two steps. The first step is to predict the time variations based on the intermediate results like proposals and raw points. The second step is coordinating the multi-tenant DNNs inference to ensure the inference progress is close to each other. *Prophet* realizes a predictable perception pipeline for the AV system in both silo mode and multi-tenant mode. The *Prophet* comprises four main modules: the profiler, the time predictor, the timeline analyzer, and the coordinator. First, the profiler collects real-time process status messages from all running tasks. Next, the time predictor predicts the time needed for the following stage of the current frame based on intermediate results like the number of proposals and raw points. Next, the timeline analyzer generates a delay map based on the time prediction results and the inference progress from all the tasks. Finally, the coordinator leverages the time prediction results and early exits the inference if it is predicted to miss the deadline. Besides, the coordinator uses the delay map to adjust the inference progress by asking some tasks to yield while others skip frames to achieve fairness among multi-tenant tasks. We implement *Prophet* on a GPU workstation and use deadline-based coordination as the baseline. From the experiment results on the KITTI dataset, the time prediction of DNN models achieves higher than 91% accuracy. For the multi-tenant DNNs scenario, the *Prophet* makes the fusion delay time bounded by 150ms, reducing the fusion drop ratio to less than 1 percent. The AV perception system becomes safer and more predictable with Prophet.

In summary, this paper makes the following contributions:

- We get two insights from an empirical study on the roots causing inference time variations for a single DNN and multi-tenant DNN models.
- We propose *Prophet*, which solves the time variations in a predictable AV perception pipeline in two steps: predict the time variations based on the intermediates results like proposals and raw points, and coordinate the multi-tenant DNNs to ensure the inference progress is close to each other.
- We implement *Prophet* in a ROS-based system and evaluate it with the KITTI dataset. The time prediction of a single model all achieves higher than 91% accuracy for Faster R-CNN, LaneNet, and PINet. The fusion delay time is bounded to 150ms, and the fusion drop ratio is reduced from 5.4% to less than 1 percent.

The rest of the paper is organized as follows. Section II and Section III presents the background and motivation of this work. Section IV discusses the insights from empirical study. Section V presents the proposed *Prophet*. Section VI presents the implementation of the *Prophet*, while Section VII presents the evaluation. Section IX describes the related work. Section X concludes the paper.

II. Background

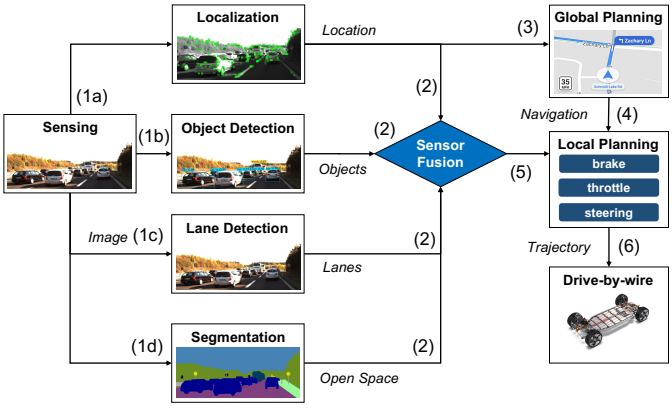

Fig. 1. A general end-to-end pipeline for modular-based autonomous driving.

Autonomous vehicles (AV) are equipped with a variety of sensors to capture and understand the surrounding environment of the vehicles [3], [18]. A typical driving system is based on a modular design, which consists of sensing, perception, planning, and control [2], [12], [19]. Figure 1 shows a generalized pipeline for modular-based autonomous driving.

A sensing node publishes the captured sensor data to all the perception nodes for localization (step 1a), object detection (step 1b), lane detection (step 1c), and segmentation (1d). Next, the perception results are submitted to a sensor fusion node (step 2), which combines the information of the vehicle's location, object, lanes, and open spaces. The location is also published to the global planning node to calculate a navigation route to the destination. The navigation route (step 4) and sensor fusion results (step 5) are both published to the local planning stage, which constructs a local driving space cost map and generates vehicle trajectories and publishes it to the vehicle's drive-by-wire system (step 6). Finally, the drive-by-wire system will send control messages to ECUs through the Controller Area Network (CAN bus) to make the vehicle drive.

III. Motivation

To guarantee the vehicle's safety, the end-to-end pipeline (from sensing to control) requires both functional and tem-

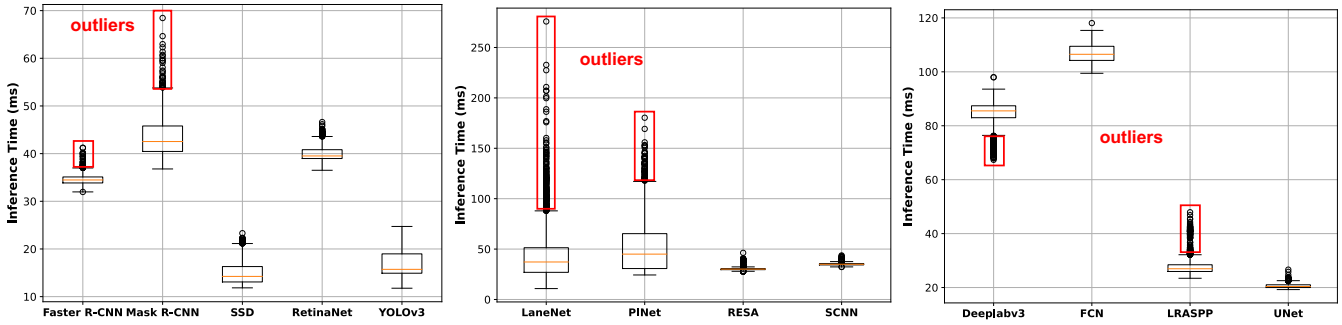

Fig. 2. The Inference latency for state-of-the-art DNN models for (a) object detection, (b) lane detection, and (c) semantic segmentation.

poral correctness. The operations must be performed in real-time. With the huge development in Deep Neural Networks (DNN) and hardware accelerators like Graphics Processing Unit (GPU), these requirements are satisfied in most cases [2], [3]. However, the execution time variability of DNN tasks in perception brings a huge challenge for making the end-to-end system real-time [13], [14], [20]. Figure 2 shows the DNN inference time variability in silo mode, where only one task runs on the platform. We choose state-of-the-art models for object detection (Faster R-CNN [6], Mask R-CNN [17], SSD [5], RetinaNet [21], and YOLOv3 [4]), lane detection (LaneNet [9], PINet [10], RESA [22], and SCNN [23]), and segmentation (UNet [24], Deeplabv3 [8], [25], FCN [26], and LRASPP [27]). Faster R-CNN and Mask R-CNN have more outliers and a wider range than SSD, RetinaNet, and YOLOv3. Two lane detection models (LaneNet, PINet) show huge inference time variations. Two segmentation models also show non-negligible inference time variations.

Since non-negligible inference time variations exist in silo mode, when multiple DNNs run simultaneously on the same platform, the predictability of the perception pipeline becomes even more challenging. Therefore, this paper will focus on realizing a predictable AV perception pipeline.

IV. Key Insights from Empirical Study

DNN inference time variations bring a significant challenge to achieving the predictability of the AV's perception system. In this section, we present in-depth profiling and analysis for DNN inference time variations and coordination of multi-tenant DNNs. Two insights are summarized for the design of a predictable AV perception pipeline.

A. Inference Time Variations for DNN Tasks

The root cause of inference time variations for object and lane detection models is the design of the DNN model. Generalized object and lane detection pipelines are shown in Figure 3. Generally, the detection pipeline starts with the sensor data access and pre-processing of the raw data. For image-based object and lane detection, the pre-processing contains *resize* and *transform* operations pre-defined by the model training process. The output of pre-processed sensor data will be fed into the inference engine for feature extraction and post-processing for final detection results. With the breakdown of inference latency, we found that the majority of time variations are on post-processing.

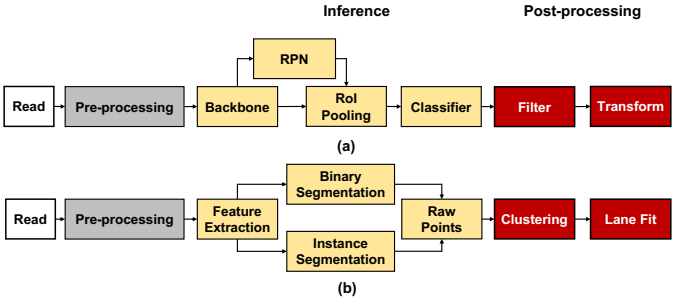

Fig. 3. Generalized two-stage object detection (a) and lane detection (b).

Object Detection: For object detection algorithms, the backbone like ResNet [28] or VGG [29] extracts features from the pre-processed image data and creates feature maps. For one-stage models, the feature maps will be used to generate a static number of anchors or default boxes [4], [5]. For two-stage models, the first stage goes through the Region Proposal Network (RPN), which creates region proposals based on a softmax layer to classify anchors as objects or backgrounds and uses bounding box regression to fix the proposals [6]. The second stage starts with Region of Interest (RoI) pooling which collects features maps and region proposals to extract proposal feature maps. Then the proposals are sent to the classifier, which has fully connected layers to get the proposal's object class and fixed bounding box. Post-processing operations include bounding box filter and transformation to the original image.

Therefore, we can find the difference in network design: two-stage models generate a dynamic number of proposals in the first stage. *The dynamic number of proposals give two-stage models more time variations in post-processing.* In contrast, one-stage models generate a static number of anchors/default boxes for detection.

Lane Detection: The network in lane detection models starts

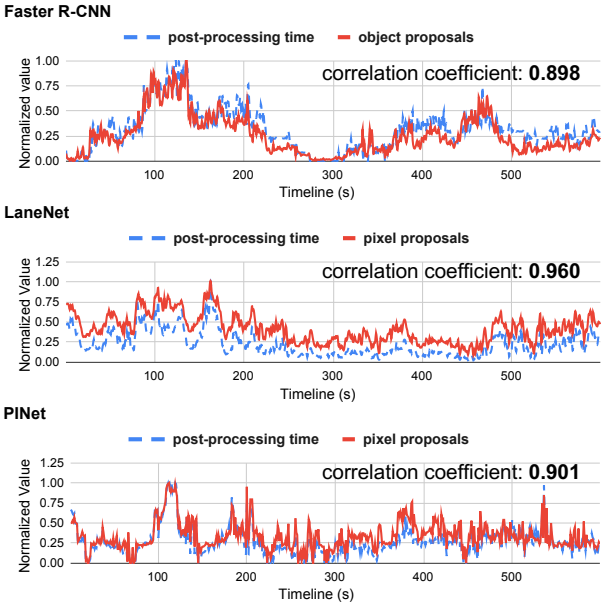

Fig. 4. The time sequence and correlation coefficients of proposals and DNN inference post-processing time.

with feature extraction, which contains an encoder-decoder structure like the hourglass block to create feature maps. For models like RESA and SCNN, the output of the multi-layer encoder-decoder is directly the lanes. Two segmentation branches are leveraged for models like LaneNet and PINet to get the pixel-wise lane proposals. Binary segmentation outputs the pixels belonging to lanes, while the instance segmentation branch outputs which lane each pixel belongs to. Raw lane points are generated by combining the results from these two branches. Post-processing contains the clustering of raw lane points and lane fitting results.

Lane detection models show similar relationships between the structure and the inference time variations. From the inference latency in Figure 3(b), LaneNet and PINet are based on two branches to generate binary and instance-level pixel proposals. The generated raw points (pixel proposals) are a dynamic value, contributing to the time variations in post-processing.

To validate our object/lane detection model structure analysis, we collected the intermediate results, including proposals and raw points from Faster R-CNN, LaneNet, and PINet. We include the second stage (RoI pooling and classifier) into the post-processing for Faster R-CNN. From Figure 4, we found that the number of proposals/pixels and post-processing time is highly correlated. Since the post-processing mainly causes the inference time variations, we can conclude that the dynamic number of object proposals and raw points caused the inference time variations for object/lane detection.

Insight 1: *Non-negligible time variations exist in the runtime of object and lane detection models, mainly caused by post-processing. A dynamic number of object proposals and raw points caused the inference time variations for two-stage/two-branch-based object/lane detection.*

B. Uncoordinated Execution of Multi-Tenant DNN Tasks

Since AV is composed of hundreds of tasks in several modules, the impact of a single model's time variations could be even worse when multiple models run simultaneously. From Figure 1, the perception module comprises four tasks: localization, object detection, lane detection, and segmentation. Figure 7 shows a simplified timeline of multi-tenant DNN inference generated based on the traces collected by *nvprof* from multiple processes [30].

Four tasks are considered for perception: ORB-SLAM2 [31] for monocular camera-based localization, executed on CPU; Faster R-CNN, LaneNet, and Deeplabv3 are executed with the same GPU device. All the tasks are assigned with the same priority for CPU and GPU resources. The results for the latency of all four tasks under multi-tenant mode are shown in Figure 5. ORB-SLAM2 shows the lowest time variations, while all three DNN tasks show drastic variations. Compared with Figure 2, which runs in silo mode, we found that all three DNN models show much higher inference time variations, mainly caused by the competition of CPU and GPU resources. As a result, driven by the wooden barrel effect, the capacity of a barrel is determined not by the longest wooden bars, but by the shortest. The fusion task is expected to show higher time variations than any individual task. Figure 6 shows the results for sensor fusion delay, which is the time interval between the current and former sensor fusion message. Although the inference time of each task is mostly less than 200ms, the fusion delay is larger than 250ms for most of the time, with the highest fusion delay larger than 800ms, which is unacceptable for safety-critical applications like autonomous vehicles.

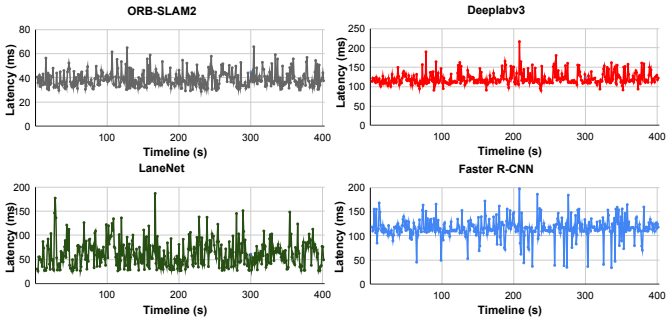

Fig. 5. The latency of multi-tenant DNNs and the delay of fusion messages.

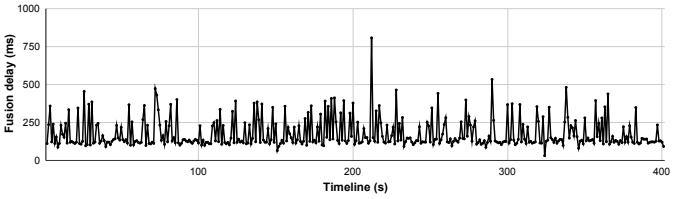

Fig. 6. The delay of fusion message.

The root cause for these huge time variations is the uncoordinated execution of multi-tenant DNN tasks. Figure 7

shows a timeline analysis of multi-tenant tasks. Although images are published from the camera with a static frame per second (FPS), each task subscribes to the communication bus separately, making some nodes run faster than others. Since ORB-SLAM2 is not competing for GPU resources with other tasks and its execution latency is less than the communication delay, it processes all the images published by the sensor. The number in each time frame represents the image's sequence number. Owing to the contention for GPU resources, Deeplabv3's inference on the n^{th} frame is much slower than other tasks, so it missed the $(n+1)^{th}$ frame and the $(n+3)^{th}$ frame. Similarly, Faster R-CNN has a longer inference time on the $(n+1)^{th}$ frame, so it missed the $(n+2)^{th}$ frame and the $(n+3)^{th}$ frame. As a result of the interference, sensor fusion happens only on the n^{th} and the $(n+4)^{th}$ frame. Suppose the scheduler or controller knows each task's execution speed. It could coordinate the execution to ensure all tasks are running at a similar speed.

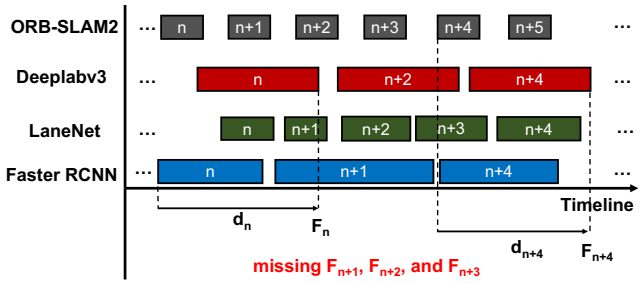

Fig. 7. The timeline analysis of uncoordinated execution of multi-tenant DNN tasks.

Furthermore, the current GPU runtime library CUDA is not open-sourced, and the execution on GPU is not preemptable, which makes it waste lots of GPU resources for some tasks because they miss the deadline [32]. Since the single model's inference variation is determined by the number of proposals/raw points, the coordinate could terminate some tasks early if the prediction latency is longer than the deadline, yielding CPU/GPU resources to other slower tasks.

Insight 2: *Uncoordinated execution of multi-tenant DNN tasks causes huge time variations for sensor fusion, and it also wastes lots of GPU resources since the GPU stream is non-preemptable. The coordinator could terminate some tasks early based on the predicted time variations to make it yield resources for other tasks.*

V. Prophet Design

The time variations in DNN inference bring an essential challenge for building a predictable AV perception pipeline. This section presents the *Prophet* system, which contains a proposal-based inference time modeling to predict time variations for a single task and a coordinator to reduce the time variations when multiple tasks run simultaneously.

A. Prophet Overview

Based on two insights from an empirical study, *Prophet* is designed to build a predictable AV perception pipeline in two steps. The first step is to predict the inference time of a single DNN based on Insight 1, and the second is to coordinate the execution of multi-tenant tasks based on their inference progress. As is shown in Figure 8, sensor data is fed into N number of perception tasks, and then the fusion task combines all the results for the planning module.

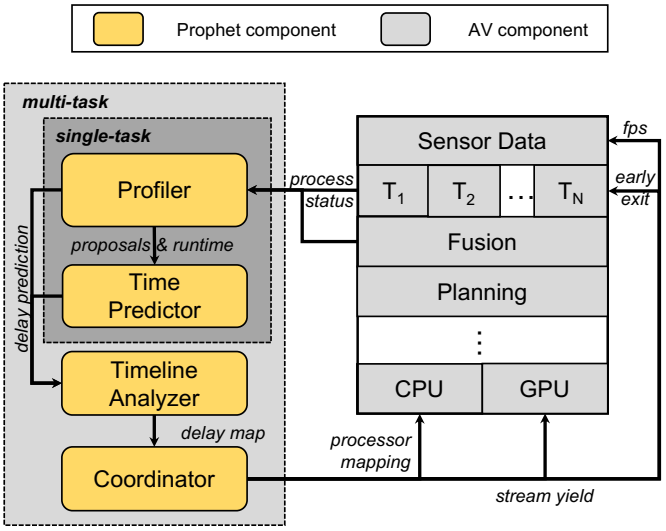

Fig. 8. High-level design of Prophet system.

The *Profiler* is designed to monitor the real-time execution of every task in perception and fusion. `process_status` is published by the tasks after inference on each frame, and it consists of information like process ID, scheduling policy, priority, image sequence number, proposals (for object detection)/raw points (for lane detection), object/lane probabilities, and the execution time of the current frame. Next, the proposal and execution time will be combined and sent to the time predictor to predict the inference time. As a result, all tasks have a predicted response time for the current frame. Together with the image sequence number, this delay prediction is sent to the timeline analysis to coordinate multi-task execution. If the response time for the current frame is larger than the task's deadline, the post-processing would be skipped. Besides, the timeline analyzer constructs a `delay_map` based on each task's image sequence number and the sensor driver's newest image sequence number. The `delay_map` represents the progress of each task, which is used to calculate how many frames to skip for each task. Besides, the `delay_map` also determines whether a task should yield GPU resources to other tasks and whether the sensor driver should reduce the publishing frequency.

B. Proposal-based Inference Time Modeling

Profiler. Monitoring the execution status and intermediate results from all the tasks is the key to inference time modeling. The profiler is designed to collect and analyze `process_status`, which contains all the information, including the image timestamp, app name, process ID, scheduling policy, priority, image sequence number, etc. Figure 9

shows an example of the `process_status` message. The runtime means the execution time for the current frame. For object detection, the proposals represent the number of object proposals from the first stage. For lane detection, the proposals represent the number of raw points generated for clustering. The object represents the number of detected objects in the current frame with probabilities higher than a threshold (0.5 for Faster R-CNN), and the probability contains all the probabilities for detection. The object and probability are filled with several detected lanes/segments and their probabilities for lane detection and segmentation.

```
stamp_secs: 1645386023
stamp_nsecs: 1935958
app: faster-rcnn
pid: 6355
scheduling_policy: "SCHED_OTHER"
priority: 0
image_seq: 186
runtime: 0.0346
proposals: 172
objects: 7
probability: [0.9986, 0.9985, 0.9984, 0.9965, 0.9953,
0.9163, 0.7939, 0.2788, 0.1954, 0.1857, 0.0886]
```

Fig. 9. An example of the `process_status` message for Faster R-CNN.

Time Predictor. For each perception task, the time predictor consists of two parts: static time modeling for prediction of time variations brought by a different number of object/lane proposals and dynamic time modeling for prediction of time variations caused by the devices' runtime conditions (temperatures, etc.) [14]. Based on the observations in Section IV, we know that the DNN inference time for one-stage object detection, one-branch lane detection, and segmentation models are within a limited range. The real challenge is the time prediction of two-stage object detection and two-branch lane detection models, which show huge inference time variations caused by post-processing. In contrast, time variations of the other processes are negligible. Besides, since the operations in post-processing are closely related to the intermediate results in inference, finding the mathematical relationship between post-processing and inference becomes the key to predicting model DNN inference time.

Static time modeling. The static time modeling is based on collected logs to find the distribution and relationship for each component within the DNN inference. To begin with, we collected logs of DNN models with fine-grained breakdowns, and some intermediate inference results, including the proposal. Typically, the inference time can be divided into four parts: reading, pre-processing, inference, and post-processing. As is shown in Equation 1 and Equation 2, subscripts represent the breakdown of each part: $_{read}$ for reading, $_{pre}$ for pre-processing, $_{infer}$ for inference, and $_{post}$ for post-processing. We use $t^{(0)}$ to represent the total inference time and $t_r^{(0)}$ to represent the remaining time (total time without post-processing) for collected logs. To distinguish the collected logs and current measurement, superscript $^{(0)}$ represents the collected logs (history) while superscript $^{(n)}$ represents the current measurement.

$$t^{(0)} = t^{(0)}_{read} + t^{(0)}_{pre} + t^{(0)}_{infer} + t^{(0)}_{post} \tag{1}$$

$$t^{(0)}_r = t^{(0)}_{read} + t^{(0)}_{pre} + t^{(0)}_{infer} \tag{2}$$

Based on the time profiling, the time variations in reading, pre-processing, and inference are limited to a small range. Therefore, we use the average value from collected logs for prediction of the current remaining time $t_r^{(n)}$. As shown in Figure 4, the post-processing time correlates with the number of proposals. For object detection, we leverage the number of proposals from the RPN to predict the post-processing time. We found a linear relationship between the number of proposals and post-processing time for object detection based on the correlation analysis and code computation complexity analysis. The prediction of the current post-processing time $t^{(n)}_{post_{OD}}$ is shown in Equation 3, where α_1 and α_0 are coefficients learned from logs. $p^{(n)}$ is the current number of proposals. λ is the calibration factor for dynamic time profiling [14]. Similarly, in lane detection, the raw points generated from two segmentation branches can be seen as lane point proposals, which can be leveraged to predict the follow-up post-processing time that contains huge time variations. Since the clustering process requires calculating all the neighbors of the pixels, the relationship is expected to be polynomial rather than linear. By analysis of the code time complexity and testing different orders of polynomial regression of the lane points with post-processing time, we found a second-order polynomial relationship between lane points and clustering time. The prediction of the current post-processing time $t^{(n)}_{post_{LD}}$ is shown in Equation 4, where β_2, β_1, and β_0 are also coefficients learned from logs. $l^{(n)}$ is the current number of lane points.

$$t^{(n)}_{post_{OD}} = (\alpha_1 p^{(n)} + \alpha_0)\lambda \tag{3}$$

$$t^{(n)}_{post_{LD}} = \left[\beta_2 (l^{(n)})^2 + \beta_1 l^{(n)} + \beta_0\right] \lambda \tag{4}$$

Dynamic time modeling. In addition to static time modeling, we introduce a calibration factor λ which reflects the dynamic time modeling caused by the devices' runtime conditions (temperatures, etc.) [14]. The calibration factor reflects the relative execution time performance compared with when logs are collected. Equation 5 shows the calculation of λ, using the average current remaining time $\overline{t_r^{(n)}}$ divided by the average remaining time in collected logs $\overline{t_r^{(0)}}$. With the adjustment of prediction time from the static time modeling, the final prediction for DNN inference time is generated.

$$\lambda = \frac{\overline{t_r^{(n)}}}{\overline{t_r^{(0)}}} \tag{5}$$

With the calculation of the remaining time $t_r^{(0)}$, the post-processing time $t_{post}^{(0)}$, and calibration factor λ, we can predict the current inference time as shown in Equation 6 and Equation 7. Since all the regression coefficients are trained from collected logs, we first add the average history remaining time ($\overline{t_{r_{OD}}^{(0)}}$ and $\overline{t_{r_{LD}}^{(0)}}$) with the predicted post-processing time $(\alpha_1 p^{(n)}+\alpha_0)\lambda$ and $\beta_2 \left[l^{(n)}\right]^2+\beta_1 l^{(n)}+\beta_0$). Then we multiply it with the calibration factor λ.

$$t_{OD}^{(n)} = (\overline{t_{r_{OD}}^{(0)}} + \alpha_1 p^{(n)} + \alpha_0)\lambda \tag{6}$$

$$t_{LD}^{(n)} = \left\{\overline{t_{r_{LD}}^{(0)}} + \beta_2 \left[l^{(n)}\right]^2 + \beta_1 l^{(n)} + \beta_0\right\}\lambda \tag{7}$$

C. Multi-Tenant DNN Execution Coordination

With the inference time prediction of each perception task, coordinating their executions to guarantee real-time is the second step. To begin with, all prediction results and execution progress will be sent to the timeline analyzer, which constructs a delay map and sends it to the coordinator. The coordinator will generate task/system management policies based on the delay map to achieve real-time and fairness.

Timeline Analyzer. With the `process_status` message sent from each task, the timeline analyzer can create a global `delay_map` to represent the execution progress of each task compared with the sensor data publisher. Since inference time variations exist for most DNN models, the processing image sequence number varies significantly among them. We can calculate the delay based on the delta of current publishing and processing image sequence numbers for each task. The `delay_map` is constructed by combining all the delay values, and it updates every time a new image is published or image processing is finished by one task. Figure 10 shows an example of the update of `delay_map` over time. The number in the table represents the image sequence number for publishing or processing. Since the execution time of ORB-SLAM2 is less than the publishing delay, it can process all the images.

Publisher	...	60	61	62	63	64	65	...
ORB-SLAM2	...	60	61	62	63	64	65	...
Deeplabv3	...	59		60			61	...
Faster R-CNN	...	58			59			...
LaneNet	...	60		61			62	...

Delay Map

0	0	0	0	0	0
1	2	2	3	3	3
2	3	4	4	4	6
0	1	1	2	2	3

Fig. 10. An example of the update of `delay_map` over time.

From the example in Figure 10, we can observe that the delay value for ORB-SLAM2 is always 0 while the delay for all three DNN tasks is increasing, whereas Faster R-CNN shows the slowest progress with a delay value of 6. When the delay value is larger than the predefined threshold, the task will start to capture the newest frame from the communication bus and reset the delay value as 0. The threshold is set based on each task's deadline and average execution time. For example, if the deadline of Faster R-CNN is 100ms while the average execution time is 30ms, then the delay threshold for Faster R-CNN would be set as 3. In general, the `delay_map` provides a general solution for managing deadline misses. It also allows the coordinator to adjust data/system/model configurations to reduce time variations.

Coordinator. As the last step of *Prophet*, the coordinator is designed to generate management policies to improve the predictability of the perception system. Therefore, there are two primary goals for coordination. The first is to decrease the delay value for each task to keep pace with the publisher. The second is to average the execution of multiple tasks to avoid cases like some tasks running too fast or others running too slow.

We introduce an `early_exit` policy based on the inference time prediction to achieve the first goal. In general, it means skipping the processing of the second stage and post-processing for object detection and clustering and post-processing for lane detection if the response time has already been longer than the deadline. Unlike the traditional approach with static deadlines, it only drops the execution until it misses the deadline. The *Prophet* drops the execution in the middle of the pipeline based on the observation of DNN inference time variations. Figure 11 illustrates the benefit of the early-exit policy over the deadline-based policy. Suppose there are three tasks to process the same image. The execution of a typical publisher-subscriber system is a callback function on coming frames. All three tasks start to process the n^{th} frame, and the deadline is d^n. After finishing the first stage/part, the intermediate results can predict the execution time for the second stage, which means knowing whether continuing the execution would miss the deadline. Here we assume task 1 and task 2 are predicted to miss the deadline while task 3 is not. Then tasks 1 and 2 would early exit the execution to save time for the next frame, while task 3 would continue to execute. Since it is possible to wait for the coming of the $(n+1)^{th}$ frame, we assume task 2 is waiting for a while and then launch the callback on the $(n+1)^{th}$ frame, while task 1 is not waiting. Compared with the deadline-based policy, we can see the `early_exit` policy helps to save computation for the second stage, which makes it save time for each task.

For the second goal, the coordinator addresses the case of "running too fast" and "running too slow" separately. For the "running too fast" case, the delay value for one task would be close to zero while other tasks are much higher than zero. The coordinator would apply a `yield` policy to ask it to yield CPU and GPU resources to other tasks by skipping the processing of frames. For the "running too slow" case, the coordinator would apply a `skip_frame` policy to ask the task to skip the execution of several frames. Finally, if all the tasks are running much slower than the publisher, the coordinator would apply an `fps` policy on the publisher to decrease the publishing frequency.

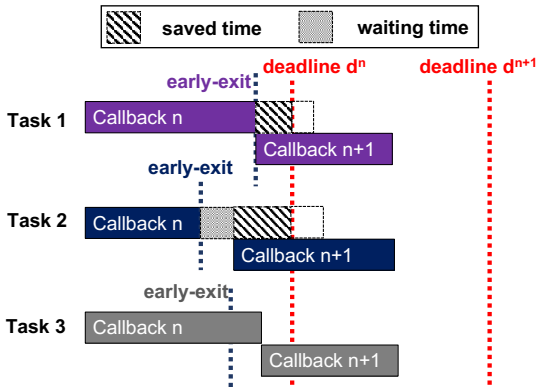

Fig. 11. An illustration of the benefit of early-exit policy over deadline-based policy.

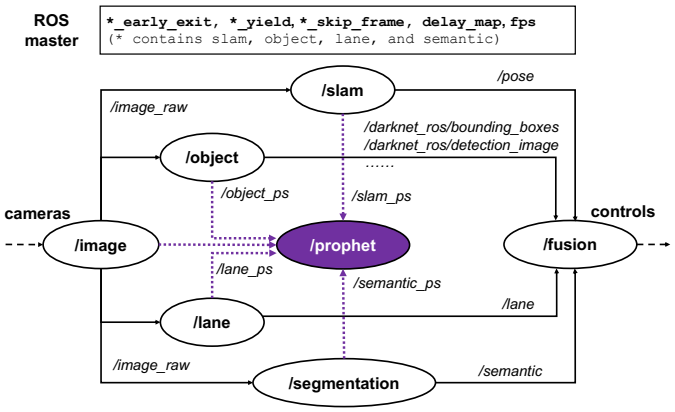

Fig. 12. The ROS implementation of the perception system.

In general, the coordinator is a referee for managing the execution progress of multiple tasks. With the four policies (`early_exit`, `yield`, `skip_frame`, `fps`), the coordinator guarantees the real-time execution and the fairness of multiple task execution.

VI. System Implementation

Autonomous driving vehicles have various applications for sensing, perception, and decision. The system performance is expected to rely on the coordination of several modules. To evaluate the performance of *Prophet*, we integrate its implementation into an AV perception prototype based on ROS in a GPU workstation with Linux RT kernel patched.

A. Overview of the Perception System

Based on the overview autonomous driving system in Figure 1, we develop a ROS framework for the perception system, as shown in Figure 12. Since this paper focuses on the perception latency variations and the planning modules are all rule-based algorithms with stable execution time, the ROS framework does not include that part. The pipeline starts with the */image* node, capturing and publishing images from the cameras. Three ROS nodes subscribe */image_raw* messages and execute the DNN inference on the images, including object detection, lane detection, and semantic segmentation. Another perception node is responsible for Simultaneous Localization and Mapping (SLAM). After perception, four nodes publish their results, covering the position information, objects, lanes, and semantics. The */fusion* node subscribes to these four topics. It synchronizes them to get the sensor fusion results, giving the control module the vehicle's obstacles and open driving space.

Besides, each perception task and the sensor publisher also publishes its execution information based on the process context and intermediate inference results. The *prophet* node subscribes to this information and coordinates task execution by updating the policy parameters accessible by all the tasks.

B. ROS Nodes and Topics

A ROS node (i.e., the nodes in the figure) is a process to perform a particular computation, while ROS topics (i.e., the arrows in the figure) are named buses for ROS nodes to exchange messages [33]. In the perception system, we implement seven ROS nodes to access the sensor data and process the data: */image*, */object*, */slam*, */lane* */segmentation*, */fusion*, and */prophet*. Publish-subscriber-based message sharing is used to transmit messages between these nodes. In our implementation, the algorithm used for SLAM is ORB-SLAM2, which is a pure camera-based approach to capture key points in pixels, localize the vehicles and generate maps simultaneously [31]. Faster R-CNN and LaneNet are deployed for object and lane detection, while Deeplabv3 is deployed for semantic segmentation [6], [8].

The ROS topics are defined as exchanging messages between ROS nodes. Ten ROS topics are implemented to exchange messages, including images, positions, objects, and customized messages. The summarized descriptions of some ROS topics are reported in Table I. There are two messages based on *Image* type: header, height, width, encoding, data, etc. The ROS topic's header contains the sequence ID, timestamp, and frame ID to represent a specific message. Since timestamp and sequence ID are needed for the synchronization, we implement */pose_timestamp* based on */pose*, which contains the position and orientation data. For object detection, bounding boxes are used to present the detected objects, determined by min and max values of the x and y-axis and the probability and the object class. */bounding_boxes* include all the bounding boxes for one image and contain a header inside. The results are shown as different colors inside the image to represent different segments for semantic segmentation. An *Image*-based topic called */semantic* is used to represent the results with the message header. The */lane* topic contains the header and curve, composed of pixels described by three float values. The */*_ps* contains four topics: */slam_ps*, */object_ps*, */lane_ps*, and */semantic_ps*, a customized topic containing a header, process ID, scheduling policy, priority, image_seq, runtime, proposals, and probability.

TABLE I
ROS TOPICS IMPLEMENTED IN THE PERCEPTION SYSTEM.

ROS Topics	Library	Type	Fields
/image_raw	sensor_msgs	Image	header, height, width, encoding, data, etc.
/pose	geometry_msgs	PoseStamped	header, position (x, y, z float64) orientation (x, y, z, w, float64)
/bounding_boxes	darknet_ros_msgs	BoundingBoxes	header, image_header, bounding_boxes
/bounding_box	darknet_ros_msgs	BoundingBox	probability, xmin, ymin, xmax, ymax, id, class
/semantics	sensor_msgs	Image	header, height, width, encoding, data, etc.
/lane	geometry_msgs	Points	header, curve
*/*_ps*	prophet_ros_msgs	ProcessStatus	header, pid, priority, image_seq, runtime, proposals, etc.

C. Message Synchronization

Message synchronization becomes one of the biggest challenges for ROS nodes that need to subscribe to multiple ROS topics and process them together. Generally, message synchronization is based on the timestamp and sequence ID.

The */fusion* node's objective is to combine all the perception results of the same image frame. The first thing is to make a unique ID for each image frame. In the beginning, the */image* node attaches timestamp information and frame ID to each message it publishes. After DNN inference on the coming image frame for four perception nodes, the timestamp and sequence ID of the coming images will be used as the header's timestamp and sequence ID of the new message like */pose*, */semantic*, etc. With unique IDs on each image frame and detection results, the remaining question is how to synchronize them. Our design uses a *message_filter* with Approximate Time Synchronizer to manage the fusion process [34]. The approximate synchronizer sets queue size as 100 and 100ms as the slop, which means the message with a time difference less than 100ms is considered synchronized.

D. Scheduling Policies

Typically, Linux supports four scheduling policies: SCHED-OTHER, SCHED-FIFO, SCHED-RR, and SCHED-DEADLINE [35]. SCHED-OTHER is Linux's default scheduling policy for supporting user applications and maximizing processors' utilization. SCHED-FIFO schedules in a first-come-first-serve method, while SCHED-RR schedules in a round-robin way. SCHED-DEADLINE is a CPU scheduler based on the Earliest Deadline First (EDF) [36]. Two DNN models are used to compare the scheduling performance: Faster R-CNN and PINet. All the tasks' nice values are default values 0. From the profiling results in Figure 2, we can find that Faster R-CNN has a shorter period than PINet. Therefore, under SCHED-FIFO and SCHED-RR, we set the priorities for Faster R-CNN and PINet as 99 and 90, respectively. Faster R-CNN has a higher priority since it has a shorter period. The round-robin interval is the default value of 100ms. The SCHED-DEADLINE policy's runtime, deadline, and period are equally set based on the image publisher's FPS. We configure the image publisher at three types of FPS: 10, 20, and 30 to compare the performance of different scheduling policies. Therefore, the runtime/deadline/period for both Faster R-CNN and PINet are set as 100ms (10FPS), 50ms (20FPS), and 33ms (30FPS), respectively.

Figure 13 shows the CDF of inference latency for Faster R-CNN and PINet under different scheduling policies and FPS. For Faster R-CNN and PINet, we can find that the long tail latency exists for all scheduling policies. From the comparison, we can observe that all scheduling policies show similar inference latency among all the testing cases. Besides, we can observe that a large percentile of image frames are missing deadlines but are still processed until the end, reflecting the fact that the Linux scheduler does not fully terminate the tasks when it misses the deadline. Moreover, we have conducted additional experiments to show the performance comparisons of multi-tenant DNNs, which are included in the Appendix of a longer version [37].

In this paper, we implement an application deadline scheduling with an exit policy and use SCHED-OTHER as the Linux scheduling policy. Each perception task is assigned a 100ms deadline [12]. Before entering the callback function, if the delay has already exceeded 100ms, the frame will be skipped. The delay map and delay threshold are used to check the deadline miss. Given the average inference time of Faster R-CNN, LaneNet, and PINet, the delay thresholds are set as 3, 3, and 2, respectively. The scheduling happens when the callback function returns with a new image frame. Since the image is published periodically, the task model is periodically dispatching jobs.

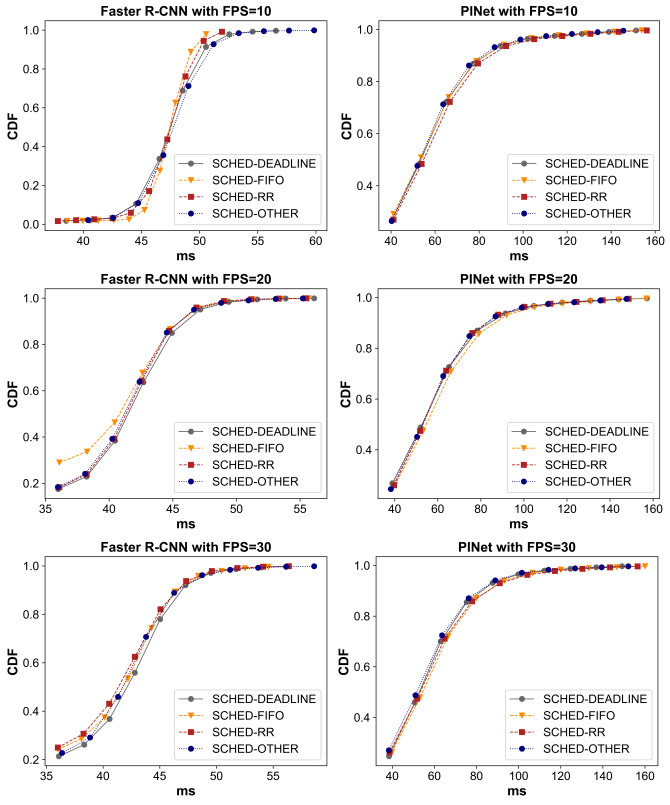

Fig. 13. The CDF of Faster R-CNN and PINet execution latency under different scheduling policies and FPS.

VII. Performance Evaluation

A. Experiment Setup

To begin, we present the experimental setup for the implementation of *Prophet*. We choose a GPU workstation as the computing platform. Besides, we use the KITTI Odometry dataset as the input to the perception pipeline [38]. The image dataset is composed of 4660 image frames. We use a ROS node to publish images at 30 FPS.

Hardware and software setup. The *Prophet* is implemented and evaluated on a GPU workstation. It has 28 Intel® Core™ i9-9940X CPUs with the highest frequency at 3.3GHz. The platform has 4 NVIDIA GeForce RTX 2080 Ti/PCIe/SSE2 GPU cards, providing 304 TOPS in total. Each GPU card has 4352 CUDA cores, supporting 10 Giga Rays/s and 14 Gbps memory speed. The GPU shared memory has 11GB GDDR6 with 352 memory interface width. In addition, the platform has 64 GB DDR4 memory. There are four DDR4-based memory devices, each 16GB with a speed at 2666 MT/s.

The libraries installed for machine learning-related applications include: `CUDA Driver 510.47.03`, `CUDA runtime 11.6`, `TensorFlow 1.15.2`, `torch v1.10.1`, `torchvision v0.11.2`, `cuDNN 8.3.2`, `OpenCV 4.2`, etc. `ROS Melodic` is deployed as the communication middleware. Since accuracy is essential for the autonomous driving scenario, all the DNN models are trained and tested with single precision (FP32) [39].

Metrics. We measure several metrics, including the perception latency (from image publishing to sensor fusion), processor utilization, memory utilization, etc. We calculate latency's statistic metrics for specific time analysis, including minimum, maximum, range, average, variance, processed frames, deadline misses, and coefficient of variation. The range R is defined as the difference between maximum and minimum values. The processed frame is the number of frames from sensor fusion. The higher the value is, the safer the vehicle's planning and control. In addition to the deadline of each task, the sensor fusion also has a deadline. The fusion message that missed the deadline is dropped. The coefficient of variation (c_v) is used to evaluate the relative variability, and it is calculated using the standard deviation σ divided by the mean value μ. c_v is a positive value. The higher the value is, the higher variations the data has.

Coefficient of Variation:

$$c_v = \frac{\sigma}{\mu} \tag{8}$$

B. Single Model Inference Time Prediction

To evaluate the performance of the proposed DNN inference time model, we collect logs when running models on the KITTI image dataset and train the weights for Faster R-CNN, LaneNet, and PINet. Figure 14 shows the results for ground truth and predicted latency. We can observe that the prediction line aligns well with the ground truth line over time.

We collect the prediction results and calculate each image's mean absolute error (MAE) and prediction accuracy.

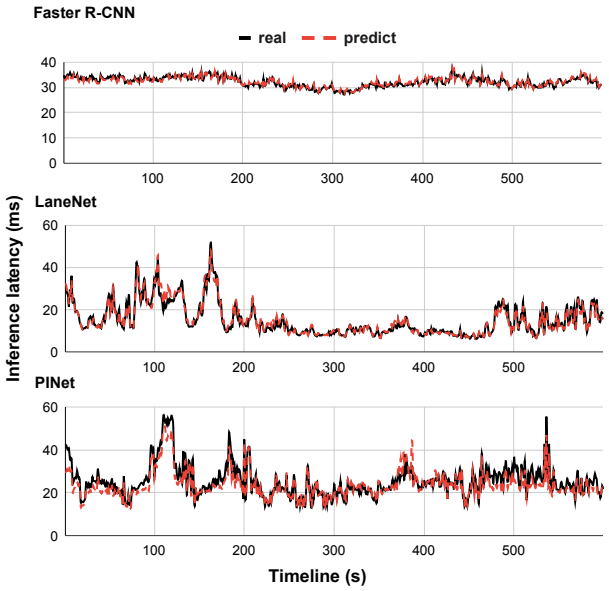

Fig. 14. The ground truth (real) results and predicted latency.

TABLE II
THE AVERAGE REAL/PREDICTED LATENCY, MEAN ABSOLUTE ERROR, AND PREDICTION ACCURACY FOR THREE MODELS.

Model	Real (ms)	Predicted (ms)	MAE (ms)	Accuracy (%)
Faster R-CNN	32.18	32.17	0.33	**98.99**
LaneNet	15.27	15.24	0.99	**94.03**
PINet	25.32	23.72	2.31	**91.68**

The results are shown in Table II. We can observe that the prediction for Faster R-CNN performs the best, with 98.99 percent accuracy and less than 0.35ms average error. LaneNet and PINet have accuracy higher than 90 percent, while the average errors are less than 2.4ms. However, the prediction error is still acceptable compared with the DNN time variation range, which is more than 26ms for LaneNet and PINet.

The high accuracy for time prediction helps the coordinator to exit some tasks early if they are predicted to miss the deadline. The main reason why we have this high prediction accuracy is the observation that the model's multi-stage/multi-branch design generates a random number of intermediate proposals/raw points, while the following stages are designed to process the proposals/raw points one by one. This strong relationship makes it possible for us to leverage the intermediate results to predict whether the deadline is missed or not.

Takeaway: *The intermediate results-based time prediction model in Prophet shows high accuracy, making it possible for the system to know if one particular frame will* miss the deadline before finishing the execution.

C. Multi-Tenant DNNs Coordination

To show the performance of *Prophet* in reducing perception time variations for multi-tenant DNN scenario, we present the evaluation in two cases: the one-GPU case where three perception tasks share the same GPU card and; the multi-GPU case where three perception tasks are distributed on three GPU

TABLE III
LATENCY RESULTS FROM A COMPARISON BETWEEN PROPHET AND THE BASELINE UNDER ONE GPU CASE.

Metrics	Baseline	*Prophet*
MIN (ms)	**19.80**	22.93
MAX (ms)	728.94	**583.47**
Range (ms)	709.14	**560.54**
Average (ms)	196.88	**153.47**
Variance	12227.34	**6466.80**
c_v	0.56	**0.52**
Processed frames	1008	**1271**
Dropped frames	289	**185**
Drop ratio (%)	28.67	**14.56**

cards. Four models/algorithms are executed for both cases: Faster R-CNN, PINet, Deeplabv3, and ORB-SLAM2.

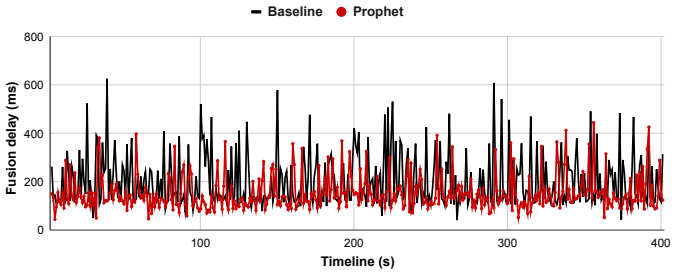

Fig. 15. The fusion delay of *Prophet* and baseline with one GPU card.

One-GPU case. When multiple DNN models share the same GPU, the time variation issue becomes more challenging. Not only does the design of two-stage/two-part detection models cause time variations, but the contention of multiple tasks for GPU resources also causes time variation. The profiling of time variations caused by resource contention would be our future work. To simplify the problem, all tasks are given the same priority on CPU scheduling, and their GPU streams are executed without preemption. In the design of *Prophet*, we leverage the delay map to coordinate multi-tenant task execution to decrease the perception time variations. Figure 15 shows the fusion delay of *Prophet* and baseline in one GPU card case, and Table III shows the statistical analysis on MIN, MAX, range, average, variance, c_v, processed frames, dropped frames, and drop ratio. For three DNN tasks running on one GPU card, the fusion drop delay is set as 200ms. We can find that in one GPU case, *Prophet* helps to reduce the fusion delay compared to the baseline. *Prophet*'s range, variance, and c_v are smaller than the baseline. In addition, the drop ratio is reduced from 28.67% to 14.56%. However, the performance is bounded by the GPU resources. There are still many dropped frames; only 1270 frames are processed out of 4660.

Multi-GPU case. In the multi-GPU case, we distribute the execution of multiple DNNs tasks to separate GPU cards. Faster R-CNN, LaneNet, and Deeplabv3 are mapped to three CUDA devices in our experiments. By isolating the execution of these perception tasks, the inference time of each task is more predictable since there is no GPU resource contention from other tasks, which could make the early exit more

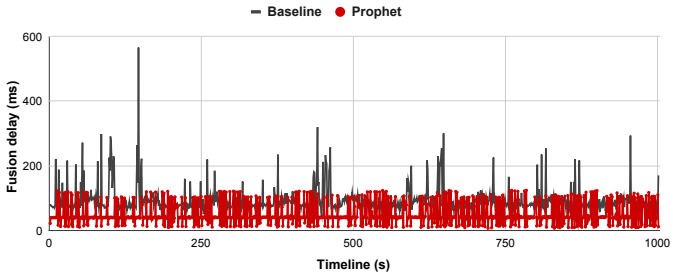

Fig. 16. The fusion delay of *Prophet* and baseline with all four GPU cards.

TABLE IV
LATENCY RESULTS FROM A COMPARISON BETWEEN PROPHET AND THE BASELINE UNDER MULTIPLE GPU CASE.

Metrics	Baseline	*Prophet*
MIN (ms)	40.65	**6.87**
MAX (ms)	1089.94	**399.58**
Range (ms)	1049.29	**392.71**
Average (ms)	93.28	**54.09**
Variance	2212.25	**1243.58**
c_v	**0.50**	0.65
Processed frames	2042	**3496**
Dropped frames	111	**3**
Drop rate (%)	5.43	**0.085**

effective. Figure 16 shows the fusion delay of *Prophet* and baseline in one GPU card case, while Table IV shows the statistical analysis of the fusion delay of the *Prophet* and the baseline. From the timeline with 1000s experiments, we can find that the fusion delay of the *Prophet* is perfectly bounded under 150ms. At the same time, the baseline shows huge time variations, with the largest fusion delay getting close to 600ms. The table shows that the *Prophet*'s MIN, MAX, range, average, and variance are much lower than the baseline, except for the c_v. This is because of the early-exit policy. Since the *Prophet* early exits some frames to save time for the other frames, the fusion delay of some frames would be much lower than the average, which makes the c_v larger. Besides, the *Prophet* has more frames through sensor fusion, which is almost doubled compared with the baseline. We set the fusion deadline for the multi-GPU case as 150ms, and we can find that the dropped frames are reduced from 111 to 3 by *Prophet*, which corresponds to a 0.085 percent drop ratio.

Takeaway: *Under the multi-GPU case, the Prophet makes the fusion delay time bounded by 150ms, reducing the fusion drop ratio to less than 1 percent. Isolate DNN tasks on multiple GPU cards also helps reduce the time variation. The AV perception pipeline becomes more predictable with Prophet.*

VIII. DISCUSSIONS

A. Applicability of the Prophet

There are several conditions for the applicability of the *Prophet*. First, the computing hardware should be a heterogeneous platform with CPU and GPU architecture. The DNN time variations pattern on a pure CPU platform differs from the CPU-GPU platform.

Secondly, in this work, all the DNN models are trained and tested with single precision (FP32), and there is no model compression (pruning, quantization, etc.) [39]–[41]. Generally, DNN models are trained with single precision weights and compressed for faster inference performance. However, DNN prediction accuracy is essential for autonomous vehicles. Although model compression could help to decrease the latency, the impact on the accuracy in real experiments is unpredictable [42].

Thirdly, the proposed system works for all the models profiled in this paper, including five object detection models (Faster R-CNN [6], Mask R-CNN [17], SSD [5], RetinaNet [21], and YOLOv3 [4]), four lane detection models (LaneNet [9], PINet [10], RESA [22], and SCNN [23]), and four semantic segmentation models (UNet [24], Deeplabv3 [8], [25], FCN [26], and LRASPP [27]). Since the DNN structure greatly impacts inference time variations, other DNN models might not show similar time variations. However, if the other DNN model's structure also fits into one-stage and two-stage designs, *Prophet* will still work.

B. Safety-critical Level

The primary goal of *Prophet* is to reduce the impact of DNN inference time variations on the perception pipeline since DNNs are mainly used in AV perception. However, our system is not for a high safety-critical level since we do not guarantee end-to-end predictability from sensing to vehicle control [2]. By proactively dropping frames that are supposed to miss deadlines, the proposed system will help to guarantee the detection of objects/lanes/segmentation. Our system will get rid of cases of missing perception because of abnormal long latency. The understanding of DNN time variations to the AV's planning and control modules would be our future work.

IX. Related Work

Autonomous vehicles are proposed to understand the environment and drive without human intervention [18]. DNNs play an essential role in the sensing, perception, decision, and control tasks in autonomous driving. Generally, the research on DNNs for autonomous driving can be divided into two categories: training DNN models with higher accuracy and improving the runtime performance of trained DNN models.

Many DNN-based algorithms are deployed in autonomous vehicles for object detection, lane detection, semantic segmentation, localization, etc. [11]. The object detection algorithms can be divided into two types: one-stage based algorithms like YOLO and SSD [4], [5]; two-stage based algorithms like Fast R-CNN, Mask R-CNN, etc. [6], [17]. The critical difference is whether there is a proposal bounding box stage. Semantic segmentation is used to detect driving segments. The fully convolutional neural network has been applied and achieves good performance [26]. LaneNet is a lane detection algorithm that uses an instance segmentation problem and applies image semantic segmentation algorithms [9]. Another approach called PINet adds key points estimation with the instance segmentation [10].

After the DNN models are trained, optimizing the model inference in latency, energy consumption, and memory utilization becomes a big challenge. In 2015, Han *et al.* proposed pruning redundant connections and retraining the deep learning models to fine-tune the weights, effectively reducing computing complexity [43]. Reducing the precision of operations and operands is another direction for the runtime optimization of DNN inference. Reducing precision is usually achieved by reducing the number of bits/levels representing the data and reducing computation requirements and storage costs [44]. Prior works [13] observed the time variations for DNN inference in mobile devices and found that inference time follows an approximately Gaussian distribution. However, the statistic-based approach performs poorly when time variations are enormous. Another work [14] on the anytime DNN system also observed the time variations issue and provided a Kalman Filter-based estimation for latency distribution. However, it cannot handle huge time variations since understanding roots causing time variations in DNN inference is missing. A scheduling-based approach is leveraged to provide predictable DNN inference [45]. However, the system is based on a cloud server-level GPU platform which is not practical for an onboard embedded system on the autonomous vehicle. D^3 is work that considers the time variations in AV systems, and they proposed to use dynamic deadlines to address time variations [16]. However, since the roots for DNN inference time variations are not studied, the proposed approach still wastes lots of time processing frames that are supposed to miss the deadline.

X. Conclusion

DNN inference time variations are non-negligible for most models in an AV system, which brings a significant challenge to the predictability of the system. We derive two important insights from a comprehensive empirical study and introduce *Prophet*, which addresses the DNN inference time variation in two-step: the first is to predict the time variations of a single DNN model; the second is to coordinate multiple DNN models inference to reduce the fusion time variations. From the evaluation results on the KITTI dataset, the time prediction of a single model achieves higher than 91% for Faster R-CNN, LaneNet, and PINet. Besides, the perception fusion delay is bounded to a 150ms, and the fusion drop ratio is reduced from 5.4% to 0.085%.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive and helpful feedback. This work was partly supported by the U.S. National Science Foundation under Grants CNS-2103604, CNS-2140346, and IIS-1724227.

XI. Appendix

A. Comparison of Scheduling Policies with Multi-task

To show the impact of different operating system schedulers in the Multi-Tenant DNNs scenario, we conduct experiments on both the baseline and the *Prophet* under four scheduling policies.

Four models/algorithms are executed: Faster R-CNN, PINet, Deeplabv3, and ORB-SLAM2. One thousand images from the KITTI dataset are used as the input. Three DNN models share one GPU card. All the tasks' nice values are default values 0. The priorities of SCHED-FIFO and SCHED-RR are set based on rate monotonic scheduling. From the profiling results in Figure 2, we can find the period increases in the order of ORB-SLAM2, Faster R-CNN, Deeplabv3, and PINet. Therefore, under SCHED-FIFO and SCHED-RR, we set the priorities for ORB-SLAM2, Faster R-CNN, Deeplabv3, and PINet as 99, 90, 80, 70, respectively. Under SCHED-OTHER and SCHED-DEADLINE, all tasks' priority values are 0. The round-robin interval is the default value of 100ms. The SCHED-DEADLINE policy's runtime, deadline, and period are equally set based on the image publisher's FPS. A thousand images from the KITTI dataset is published at 10 FPS as the input. Therefore, the runtime/deadline/period for all the tasks are set as 100ms.

Baseline Performance. Figure 17 shows the CDF of execution latency of four tasks under different OS scheduling policies. We can find that the distribution of execution latency is similar among all scheduling policies. Table V shows the statistical analysis of the fusion delay of the baseline. 200ms is used as the threshold for dropping fusion frames. *SCHED-RR* performs better than the other policies in average, processed frames, drop ratio, etc.

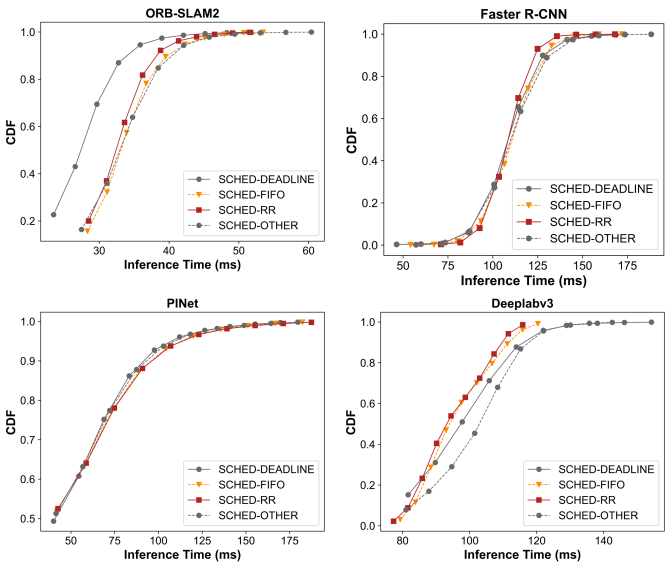

Fig. 17. The CDF of multi-task under different scheduling policies.

Prophet Performance. Figure 18 shows the CDF of execution latency of four tasks under different OS scheduling policies.

TABLE V
Latency results from a comparison between scheduling policies with the baseline.

Metrics	DEADLINE	FIFO	RR	OTHER
MIN (ms)	86.23	**42.01**	43.79	45.91
MAX (ms)	249.20	332.66	**231.24**	292.63
Range (ms)	**162.97**	290.64	187.45	246.71
Average	116.28	117.49	**113.33**	120.98
Variance	445.80	449.02	**206.83**	815.43
c_v	0.18	0.18	**0.13**	0.24
Processed frames	918	936	**971**	883
Dropped frames	29	27	**11**	48
Drop ratio (%)	3.16	2.88	**1.13**	5.44

TABLE VI
Latency results from a comparison between scheduling policies with the *Prophet*.

Metrics	DEADLINE	FIFO	RR	OTHER
MIN (ms)	36.67	35.06	**2.21**	35.64
MAX (ms)	203.89	**195.05**	220.67	210.91
Range (ms)	167.23	**159.99**	218.46	175.27
Average	108.64	106.58	106.56	**105.94**
Variance	1216.54	**605.89**	739.25	1223.64
c_v	0.32	**0.23**	0.26	0.33
Processed frames	998	998	**999**	998
Dropped frames	3	**0**	2	3
Drop ratio (%)	0.3	**0**	0.2	0.3

Table VI shows the statistical analysis of the fusion delay of the *Prophet*. 200ms is used as the threshold for dropping fusion frames. It can be found that both *SCHED-FIFO* has a zero drop ratio. *SCHED-FIFO* processed the most frames.

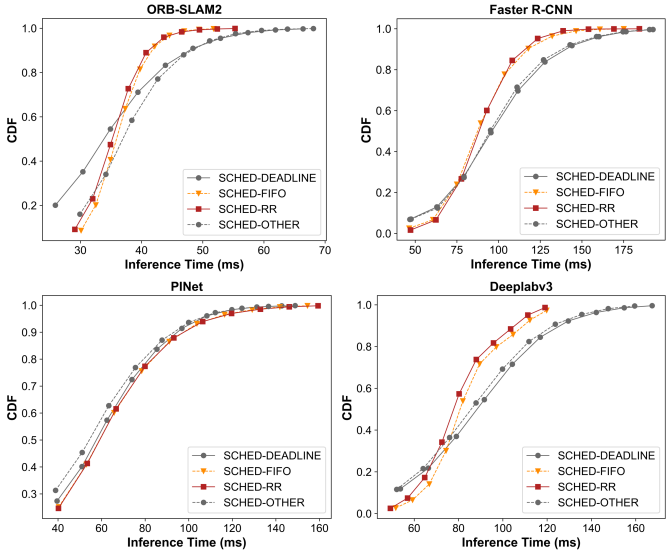

Fig. 18. The CDF of multi-task under different scheduling policies with Prophet.

Comparing the drop ratio from Table V and Table VI, we can find that *Prophet* performs better in reducing dropping frames than the OS's real-time schedulers.

REFERENCES

[1] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies," *IEEE access*, vol. 8, pp. 58 443–58 469, 2020.

[2] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 751–766.

[3] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[4] J. Redmon and A. Farhadi, "YOLOv3: an incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[5] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[8] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *arXiv preprint arXiv:1706.05587*, 2017.

[9] D. Neven, B. De Brabandere, S. Georgoulis, M. Proesmans, and L. Van Gool, "Towards end-to-end lane detection: an instance segmentation approach," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 286–291.

[10] Y. Ko, J. Jun, D. Ko, and M. Jeon, "Key points estimation and point instance segmentation approach for lane detection," *arXiv preprint arXiv:2002.06604*, 2020.

[11] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi, "Computing systems for autonomous driving: State-of-the-art and challenges," *arXiv preprint arXiv:2009.14349*, 2020.

[12] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[13] 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)*. IEEE, 2019, pp. 331–344.

[14] C. Wan, M. Santriaji, E. Rogers, H. Hoffmann, M. Maire, and S. Lu, "ALERT: Accurate learning for energy and timeliness," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 353–369.

[15] P. H. Becker, J. M. Arnau, and A. González, "Demystifying power and performance bottlenecks in autonomous driving systems," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 205–215.

[16] I. Gog, S. Kalra, P. Schafhalter, J. E. Gonzalez, and I. Stoica, "D3: a dynamic deadline-driven approach for building autonomous vehicles," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 453–471.

[17] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2961–2969.

[18] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, 2019.

[19] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, 2008.

[20] W. Shi and L. Liu, "Systems runtime optimization," in *Computing Systems for Autonomous Driving*. Springer, 2021, pp. 81–107.

[21] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[22] T. Zheng, H. Fang, Y. Zhang, W. Tang, Z. Yang, H. Liu, and D. Cai, "Resa: Recurrent feature-shift aggregator for lane detection," *arXiv preprint arXiv:2008.13719*, vol. 5, no. 7, 2020.

[23] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Spatial as deep: Spatial cnn for traffic scene understanding," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[24] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[25] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.

[26] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3431–3440.

[27] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1314–1324.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[29] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[30] "Profiler User's Guide," https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[31] R. Mur-Artal *et al.*, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and rgb-d cameras," *arXiv preprint arXiv:1610.06475*, 2016.

[32] "CUDA Toolkit Documentation v11.6.0," `https://docs.nvidia.com/cuda/archive/11.6.0/`.

[33] "Robot Operating System(ROS), Powering the World's Robots," 2019. [Online]. Available: https://www.ros.org/

[34] "message_filters," `http://wiki.ros.org/message\_filters`.

[35] "sched(7) — Linux manual page," `https://man7.org/linux/man-pages/man7/sched.7.html`.

[36] M. Spuri and G. C. Buttazzo, "Efficient aperiodic service under earliest deadline scheduling." in *RTSS*, 1994, pp. 2–11.

[37] "Prophet: Realizing a Predictable Real-time Perception Pipeline for Autonomous Vehicles," `https://www.weisongshi.org/papers/liu22-prophet.pdf`.

[38] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the KITTI vision benchmark suite," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 3354–3361.

[39] "Floating Point and IEEE 754 Compliance for NVIDIA GPUs," `https://docs.nvidia.com/cuda/floating-point/index.html`.

[40] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[41] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.

[42] R. Yazdani, M. Riera, J.-M. Arnau, and A. González, "The dark side of dnn pruning," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 790–801.

[43] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *arXiv preprint arXiv:1506.02626*, 2015.

[44] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[45] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.