VESPER: A Real-time Processing Framework for Vehicle Perception Augmentation

Kwame-Lante Wright
University of Southern California
Los Angeles, CA
kwamelaw@usc.edu

Pranav Sakulkar
University of Southern California
Los Angeles, CA
sakulkar@usc.edu

Bhaskar Krishnamachari
University of Southern California
Los Angeles, CA
bkrishna@usc.edu

Fan Bai
General Motors
Warren, MI
fan.bai@gm.com

***Abstract*—With today's intelligent vehicles, there are a variety of information-rich sensors, both on and off-board, that can stream data to assist drivers. In the future, we imagine physical infrastructure capable of sensing and communicating data to vehicles to improve a driver's awareness on the road. To process this data and present information to the driver in real-time, we introduce VESPER, a real-time processing framework and online scheduling algorithm designed to exploit distributed devices that are connected via wireless links. A significant feature of the VESPER algorithm is its ability to navigate the trade-off between accuracy and computational complexity of modern machine learning tools by adapting the workload, while still satisfying latency and throughput requirements. We refer to this capability as *polymorphic computing*. VESPER also scales opportunistically to leverage the computational resources of external devices. We evaluate VESPER on an image-processing pipeline and demonstrate that it outperforms offloading schemes based on static workloads.**

I. Introduction

The U.S. Department of Transportation is in the process of developing rules to advance vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication technology [1]. It is believed that this technology will help to improve safety and reduce congestion on the road. As these technologies become a reality, we believe there will be a wealth of information available from various static and mobile sensing sources including video cameras and other sensor feeds, that could be used to assist drivers. However, presenting such streams of sensor data directly to a driver in a raw format would be overwhelming and impractical.

An Advanced Driving Assistance System (ADAS) is needed to extract useful details from these sensors, for instance to give a driver information about traffic or hazards beyond visual range. However, a single vehicle may not be able to handle processing of all the incoming sensor data on its own. In order to meet time constraints, such systems often employ specialized hardware, such as GPUs, or offload to the cloud. However, static offloading schemes are not well suited for vehicular applications. Such schemes are susceptible to changes in resource availability, due to variations in wireless link quality as a vehicle drives around, and thus they will often fail to perform as desired, for instance incurring too high of a latency. Moreover, in this context, unlike traditional mobile applications (where typically only two computation points are considered: on-mobile or in-cloud), there are many points

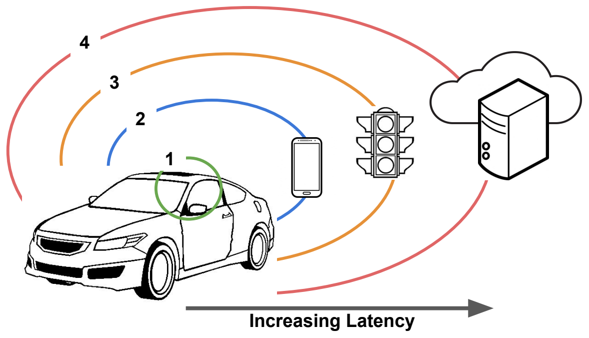


Fig. 1. Potential processing resources available to a car

where computation could take place, as depicted by the loops in Figure 1. These points generally provide a trade-off between latency and computational capability and availability, with the more computationally capable nodes incurring a higher latency and relatively lower availability. A robust system for driving assistance needs to constantly adapt the offloading decisions to effectively utilize all the available processing and communication resources that may be available.

In this work, we recognize that in some cases a working real-time processing system can be more important than an accurate one and we introduce the concept of *polymorphic computing* for this domain. Here polymorphic (coined from poly- "many" and morphic - "forms") refers to different computational pipelines that could be used for a processing task that offer different accuracy-performance tradeoffs. The fundamental idea here is that at a given point in time, an application's workload (resulting from a particular processing pipeline that may be optimized for accuracy) may not be achievable within a given time frame to meet real-time requirements. This could be due to a number of reasons outside the control of the application. However, rather than simply failing, it would be more useful if the application could adapt its workload (in other words, change the actual processing pipeline itself) in a way that it could be completed within the time constraints, albeit at the cost of some reduction in accuracy.

We propose a real-time vehicle sensing and perception augmentation system (VESPER) that incorporates dynamic offloading and polymorphic computing. VESPER provides the following features:

- **Workload Adaptability**: VESPER changes its processing pipeline based on the available computing resources and their link qualities.
- **Support for Intermittent Connectivity**: External computing devices such as the cloud may not always be available to a car, and can come and go frequently. VESPER is adaptive to the changes in the device connectivity.
- **Scalability**: VESPER is capable of handling multiple computing devices (including on the sensor platform, on car, roadside, and in the cloud) seamlessly.

II. Related Works

A variety of computational offloading problems have been studied previously. However, the ability to handle the intermittent connectivity of devices, multiple processing pipelines and real-time constraints, sets the VESPER algorithm apart. We highlight some prior applications of computational offloading below.

Odessa [2] is a system designed to improve the latency and throughput of a streaming application through offloading and parallelism decisions using the Sprout [3] framework. It accomplishes this by making greedy incremental scheduling decisions. While it has demonstrated good performance, Odessa is only designed only for a two-node network, consisting of a mobile device and a server.

Glimpse [4] is a real-time object recognition and tracking system built to run on mobile devices. It utilizes an *active cache* to mitigate the impact of latency variations between a mobile device and the cloud in order to provide continuous performance. While similar to our work in that it addresses real-time constraints, Glimpse's dependence on the cloud is a limitation we overcome with VESPER.

III. Background

In this section we define the problem we are trying to address, introduce an application to help motivate this discussion, and identify the metrics by which the system is evaluated.

A. Driver Perception Augmentation Application

Currently, drivers make most of their driving decisions based on what can been seen directly through their windshields. These decisions may not be optimal and could be potentially improved if the driver is provided with a look-ahead into the future road and traffic conditions. In this work, we consider an application where a drone is streaming images of the road ahead to a car to provide the driver with traffic information. The raw images are not particularly useful, and in fact can be harmful, if the driver needs to divert his/her attention to interpreting them. The system, therefore, requires the implementation of an image processing pipeline for intelligent detection of the vehicles on the road. In order for the system to be helpful, the results of this pipeline need to be computed and delivered to the driver in real-time. This real-time constraint necessitates that the system be capable of adapting to the dynamic availability of computational resources and wireless link quality. The system should accomplish this while providing the highest level of performance possible. We discuss the constraints and performance metrics in detail in the following subsection.

Our image processing pipelines are based on YOLO [6], a real-time object detection system that uses a CNN to detect and localize objects in images. We use two variations of YOLO, namely TinyYOLO and YOLOv2, as our image processing pipelines to detect cars on a highway. As shown in Figure 2, TinyYOLO and YOLOv2 exemplify the performance trade-off at the core of polymorphic computing.

B. Performance Metrics

When the vehicle receives external sensor data (images in our case), VESPER determines which processing pipeline to use and where in order to extract the best information from the image within the time constraints. The following metrics are used to evaluate the performance of the controller algorithm:

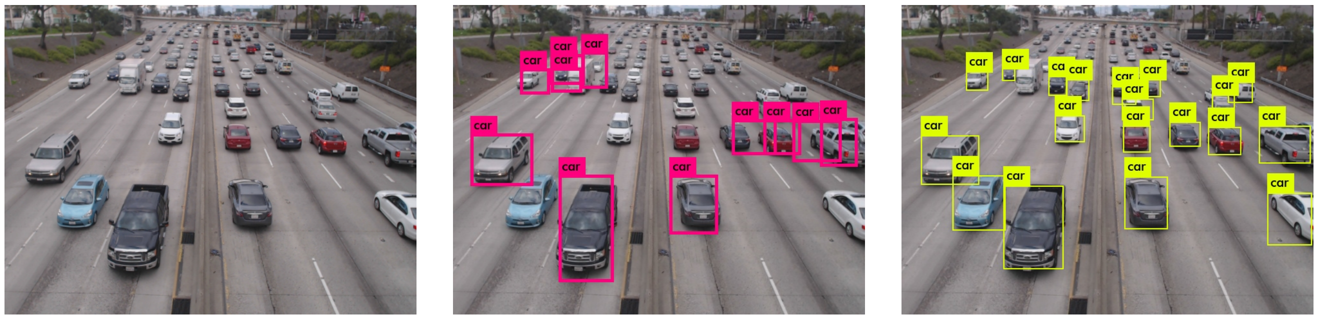
- **Latency**: Latency, or makespan, represents the time it takes an image to make its way through the processing pipeline. It is a function of the chosen pipeline, the resource availability, and the wireless link quality at the time of execution. In our application example, the latency constraint would be dictated by how far the drone is traveling in front of the car, so that the driver has enough time to react to any information provided.
- **Throughput**: To maintain the driver's awareness of traffic conditions ahead, the system should deliver updates to the driver at a reasonable rate, or throughput, measured in frames processed per second. This constraint determines the minimum rate at which the system needs to process images and is a measure of the scheduler's ability to parallelize the workload based on available resources.
- **Accuracy**: The accuracy metric represents how well the output of the processing pipeline fits the ground truth in the real world. The scheduler will attempt to use the most accurate pipeline for the longest period of time while the system is running. Therefore, we believe that the *time-averaged expected accuracy* is a useful measure for algorithm performance. In this work we use mAP as our accuracy value.

The primary goal of the VESPER algorithm is to maximize the system's accuracy while ensuring that the throughput and latency constraints are satisfied. The throughput constraint is a lower bound while the latency constraint is an upper bound. In our envisioned application, these constraints would be dependent on the speed of the car and the distance to the drone as shown in Figure 3.

IV. System Design

A. Framework

The VESPER framework consists of several components, namely the image source, scheduler, dispatcher, token manager, performance monitor, pipeline database, and one or more devices. The framework components are connected as shown in Figure 4.



(a) Drones-eye view of highway (b) TinyYOLO (57.1% mAP @ 207 fps*) (c) YOLOv2 (76.8% mAP @ 67 fps*)

Fig. 2. Illustration of polymorphic computing: two different vision processing pipelines (variations of YOLO), for vehicle detection in images, that offer different trade-offs between accuracy and performance. *on an NVIDIA Titan X GPU [5]

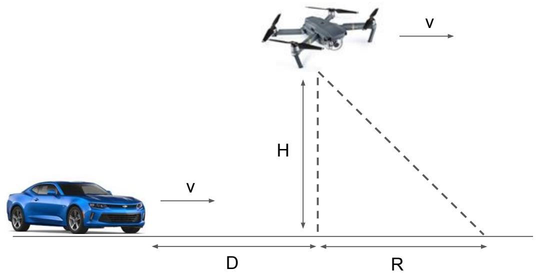


Fig. 3. Parameters affecting the real-time constraints

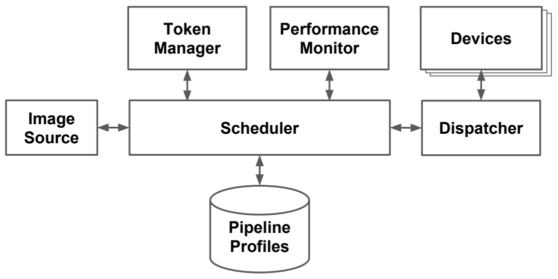


Fig. 4. Framework Architecture

a) Image Source: The nature of our target application requires a constant stream of images. These images are provided by a drone over a wireless link. Due to the transient nature of wireless links and other unpredictable circumstances, we have implemented the ability to control the frame rate used by the drone when supplying the images. This provides some flexibility for the system to adapt its throughput. We discuss this capability in more detail in Section IV-B.

b) Scheduler: The scheduler is the key component of our framework and is where the VESPER scheduling algorithm executes. The scheduling algorithm is responsible for selecting an image processing pipeline so as to maximize the accuracy of the system while satisfying the makespan and throughput constraints. Due to its critical nature, the scheduler is run on the car to make the system robust against intermittent device connectivity, since external devices such as an RSU may not always be connected. Section IV-C describes the scheduler in more detail.

c) Dispatcher: Communication between the scheduler and any devices connected to the system take place through the dispatcher. The dispatcher relies on TCP to ensure that all messages are received.

d) Token Manager: To ensure that we do not assign too much work to a device, we have implemented a token manager. Tokens are associated with a particular device and the number created depends on the amount of threads of work that a device can support. A token is required to assign work to a device.

e) Performance Monitor: This component tracks the various metrics used to assess system performance, including makespan ($\hat{M}$) and throughput ($\hat{T}$). An exponential weighted moving average (EWMA) is used to average some of these values over time. The monitor also tracks per-device performance to allow the scheduler to make more informed decisions. This includes the device's processing rate and round-trip time (RTT).

f) Pipeline Database: We collect the execution times of our image-processing pipelines offline for each type of hardware and store this information in a database. The scheduling algorithm uses this information along with live performance measurements to predict if a pipeline is feasible or not under the current operating conditions.

g) Devices: At any given point, there could be multiple computing devices connected to the car, such as nearby RSUs. Each device receives computing jobs from the dispatcher. For each incoming job, the device timestamps the start and the finish time of execution, which are then used by the performance monitor to profile the execution times of these devices and their corresponding link qualities.

B. Frame Rate Adaptation

To ensure that images are arriving sufficiently fast, the VESPER algorithm applies frame rate adaptation to make sure that the system delivers the desired throughput performance to satisfy the application constraints. VESPER uses a proportional controller which attempts to approach the throughput constraint by maintaining unity of the ratio $T_0/\hat{T}$, where T_0 and $\hat{T}$ represent the system's throughput constraint and measured system throughput, respectively.

We have modified the proportional controller slightly to make it more robust in practice. Firstly, we boost the throughput constraint by 1% to make the system more likely to satisfy this requirement. We've observed empirically that aiming exactly for the constraint would often cause the system to fall short. Secondly, we set a lower and upper bound on the requested frame rate. The lower bound is simply a logical restriction whereas the upper bound is included to prevent any excessive rates from being requested.

C. Scheduling Algorithm

The scheduling component of VESPER determines which image processing pipeline and which devices to use in an attempt to maximize accuracy while ensuring the real-time constraints are satisfied. The scheduler runs on the car and makes its decisions based on the most recent performance data available. As images arrive, they are distributed to scheduled devices using a token-based queuing system.

1) Devices and Tokens: Devices are the workhorses of the VESPER framework. When a device establishes connectivity to the car, the device generates tokens based on the number of GPUs available to be used on that device. These tokens are placed in a FIFO queue. A token is needed by the scheduler in order to assign an image to a device for processing. These tokens essentially limit the number of images that can be in-process at a device at any point in time. A token is consumed from the token queue when a job is sent out to a device and is recreated when the device completes the work. If a device completes its work quickly, its token gets added back to the queue very frequently. VESPER rewards efficient work with more work.

It is possible for a connected device to not be used at all if it fails to meet the system's makespan constraint. To ensure that performance data for such devices do not get stale, VESPER will periodically probe the device with fake work to get fresh measurements. If the link to a particular device is lost, the system may lose some frames that were assigned to that device, but the frames on other devices are still processed as normal. Loss of a few frames is acceptable as VESPER reacts quickly to ensure that the throughput and makespan constraints are still satisfied. Once a device is disconnected, the controller removes the token for that device and it is not considered for further scheduling until it reconnects. Note that there will always be at least one device available to the system, namely the car itself.

2) Performance Measurements: The VESPER scheduler will periodically review the past performance of all the devices and determine the best pipeline to use for processing subsequent images. It accomplishes this by tracking the processing rates and link times for each device. Jobs are timestamped at the car when they leave and return to the Dispatcher. This makes it possible for the car to calculate a job's entire makespan. In addition, jobs are timestamped at the devices when the devices start and finish working on them, allowing VESPER to determine the execution time for a job on each device, without requiring time synchronization with the car. When combined with the pipeline profiles, the execution time can be used to approximate an effective processing rate for each device. By subtracting the execution time from a job's makespan, the latency, or RTT, due to the link can also be determined.

3) Scheduler Logic: VESPER uses the processing rate and link estimates to determine if a pipeline is feasible given the throughput and makespan constraints of the system. If a higher-accuracy pipeline is feasible given the available devices then VESPER will switch to that pipeline. If a device leaves or its performance worsens, VESPER may drop back to a faster, albeit less accurate, pipeline to ensure the real-time constraints are still satisfied.

The design of the algorithm is such that it will terminate faster when the system is performing poorly. The scheduler checks the feasibility of pipelines in increasing order of computational complexity. When the algorithm reaches a pipeline that cannot be accommodated, it will terminate early. The algorithm is $O(PD)$, where P is the number of pipelines available and D is the number of devices connected at the time of execution.

The images sent by the drone are buffered by the car until they expire or are scheduled to be processed by a device. The scheduler loop runs at a fixed frequency and uses device performance data to make its decisions, as described. $\hat{M}$ and $\hat{T}$ are measured over fixed time intervals and are continuously updated by the performance monitor. The monitoring frequency is three times as fast as the scheduler to ensure that the scheduler uses up-to-date measurements.

V. Experimental Setup

We devise a set of experiments to evaluate the performance of VESPER under various conditions. We assume a star topology where the drone (camera) and RSUs wirelessly connect directly to the car. We believe this topology accurately represents a real-world scenario for a driver perception augmentation application.

For our experiment, we imagine that the car communicates to the drone and roadside unit(s) using Wi-Fi. Due to the difficultly of getting permission to fly a drone around moving vehicles to test our system in a live environment, we instead use a network emulation tool called Mahimahi [7] to emulate the links in our experimental scenarios. While the links are emulated, we believe the use of traces obtained from real environments supports our confidence that the results we obtain in-lab will be indicative of what we would expect to see in the real-world and the traces also allow for reproducibility.

A. Image Processing Pipelines

VESPER utilizes the reference implementations of the YOLO and TinyYOLO objection detection pipelines, which are publicly available. Both networks are loaded into memory at runtime and await images for processing. We profiled both pipelines on all of our devices, namely the car and the RSU.

TABLE I
HARDWARE SPECS

Device	CPU	RAM	GPU
Drone (Raspberry Pi 3)	ARM Cortex-A53 (1.2 GHz)	1 GB	-
Car (Desktop)	Intel Core i7-4770 (3.4 GHz)	16 GB	NVIDIA 1050 Ti
RSU (Jetson TX2)	ARM Cortex-A57, Denver 2 (2.0GHz)	8 GB	NVIDIA Pascal

B. Benchmarks

To better assess the performance of VESPER, we implemented a static algorithm within our framework to use as a benchmark. The static algorithm has no capability for frame-rate or pipeline adaptation. This algorithm uses all connected devices regardless of their performance.

C. Hardware Specifications

Table I describes the hardware we used. In all of our experiments, we assign the number of tokens based on the number of GPUs present in each device. For the car and RSU we issued a single token. We did not consider using the drone for neural network computation due to its power limitations, as it is running a Raspberry Pi 3.

Images were captured by a Logitech HD Pro C920 Webcam connected to the drone. The camera captures images at 1080p resolution. At the level of JPEG compression we use for transmission, images are about 20-30 KB in size.

VI. RESULTS

Through a series of experimental scenarios, we evaluate the performance of VESPER and demonstrate its capabilities. The scenarios are run for 30 minutes. Unless otherwise stated, the makespan and throughput constraints for each experiment are $M_0 = 0.8$ seconds and $T_0 = 8.0$ frames/second (fps), respectively. By default, the frame rate adaptation and scheduling algorithms are run every 6 seconds. We refer to this as the control loop time. The control loop time is a tunable parameter that determines how quickly the algorithm takes actions. We've determined experimentally that 6 seconds works adequately.

Through our experiments, we aim to answer the following questions:

- How well does VESPER perform and what overhead is incurred by using this framework?
- How well does VESPER scale to leverage external devices and how responsive is it to changes in computational resource availability and link quality?

A. Scenario 1: Overhead

In the first scenario, we aim to assess the overhead of VESPER by comparing it to a static algorithm in an environment where the car is the only device available for computation. Figure 5 shows the average accuracy, as defined in Section III-B, versus average throughput. The vertical red line represents the throughput constraint for the system. We know if the system is satisfying the throughput constraint if it is located on or to the right of this line. Here, we can see that both the Static and VESPER algorithms met this constraint.

The two horizontal lines in the plot represent the accuracy of the two pipelines available for selection by the scheduler, namely TinyYOLO and YOLOv2. The average accuracy of our system will be located somewhere between those two accuracies, based on how long each pipeline is used during operation. The Static algorithm will always be on one of these lines, as it does not switch pipelines. In this case it is set

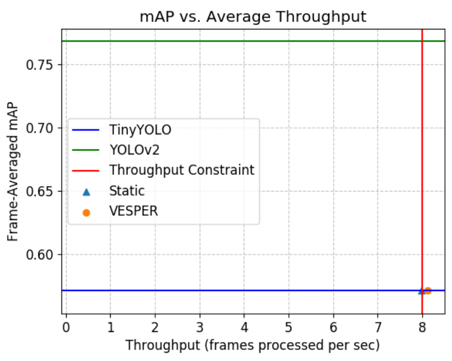


Fig. 5. Scenario 1: Accuracy/Throughput Performance

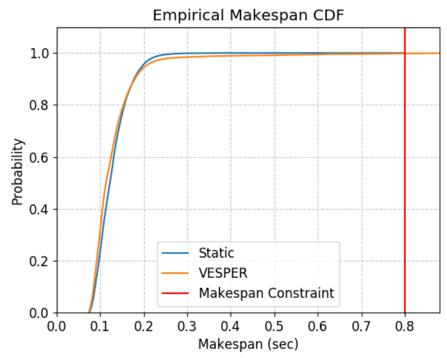


Fig. 6. Scenario 1: Makespan Distribution

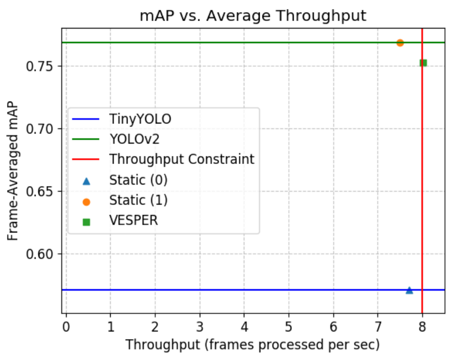


Fig. 7. Scenario 2: Accuracy/Throughput Performance

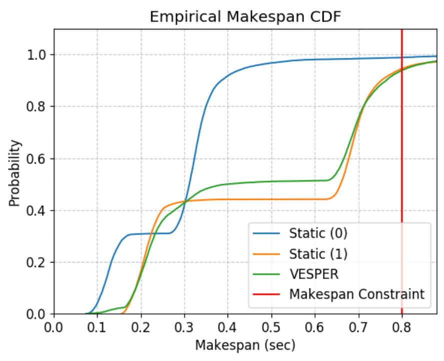


Fig. 8. Scenario 2: Makespan Distribution

to only use TinyYOLO. We can conclude from this plot that VESPER did not find it possible to schedule YOLOv2 while satisfying the throughput constraint since it is also on the blue line. This is actually by design, as we wanted to focus on assessing VESPER's overhead for this scenario. From data we've gathered, the car takes about 0.188 seconds on average to process an image using YOLOv2 and therefore can only support about about 5 fps using that pipeline. On the other hand, TinyYOLO takes about 0.098 seconds per image on the car. With a throughput constraint of $T_0 = 8.0$, VESPER determined that only TinyYOLO could be used.

Through the empirical makespan CDF in Figure 6 we can see that there is little overhead with running the VESPER algorithm over the static case. This plot shows the latency of image processing, from image reception to result, with the vertical red line representing the makespan constraint. We know there were no late images since the CDF hits 1.0 before reaching this line. Comparing the two CDFs, there is no significant penalty incurred when running the VESPER algorithm with no external devices present. The CDFs are fairly sharp since only local processing on the car is performed, meaning link quality does not play an important role here.

B. Scenario 2: Scalability

In the second scenario, we introduce the RSUs as external computing resources and observe VESPER's ability to leverage them to improve its accuracy. Four RSUs are active for the entire experiment (meaning they are operational but still subject to varying link quality). In Figure 7, we observe that VESPER, while maintaining the throughput constraint, is able to achieve an average accuracy between TinyYOLO and YOLOv2.

Whenever a new device connects to the system, the VESPER controller probes the device initially to estimate the link quality and the device execution rate. During this phase, the car is the only device executing the jobs. Since the car can only support the TinyYOLO pipeline while meeting the constraints, the accuracy during the initial phase is low. After the probing phase, which may end after just a single frame for a device, VESPER is occasionally able to change the pipeline to YOLOv2 when conditions allow, giving the system better accuracy. The time-averaged accuracy of VESPER is, therefore, better than TinyYOLO's accuracy. By leveraging the RSUs, VESPER is able improve the average accuracy of the system. Through the makespan CDF, show in Figure 8, we see that while VESPER achieves higher accuracy, this accuracy comes at the cost of makespan due to the heavier processing for YOLOv2. This plot helps to visualize the trade-off that VESPER is making. As long as it continues to satisfy the makespan constraint, VESPER may sacrifice makespan in order to improve accuracy.

VII. Conclusion

In this paper, we have presented VESPER, a real-time *polymorphic computing* framework for driver perception augmentation. VESPER exploits the computational resources of devices connected wirelessly with the car to perform complex processing tasks. It handles intermittently connected devices and allows for workload adaptation, wherein the processing pipeline can be changed based on the available resources of the devices and their link qualities. We have developed the framework and demonstrated its performance using a computer vision task for identifying vehicles in drone images. Through our experiments we have shown that VESPER maximizes the accuracy of the system while satisfying the real-time constraints of latency and throughput for the application.

Acknowledgment

This material is based in part upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053. Any views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] "U.S. DOT Advances Deployment of Connected Vehicle Technology to Prevent Hundreds of Thousands of Crashes," Dec 2016. [Online]. Available: https://www.nhtsa.gov/press-releases/
[2] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling Interactive Perception Applications on Mobile Devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. ACM, 2011, pp. 43–56.
[3] P. S. Pillai, L. B. Mummert, S. W. Schlosser, R. Sukthankar, and C. J. Helfrich, "Slipstream: Scalable Low-Latency Interactive Perception on Streaming Data," in *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 2009, pp. 43–48.
[4] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, Real-time Object Recognition on Mobile Devices," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2015, pp. 155–168.
[5] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.
[6] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-time Object Detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.
[7] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, and H. Balakrishnan, "Mahimahi: A Lightweight Toolkit for Reproducible Web Measurement," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 129–130, 2015.