IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 14, 2020, accepted January 25, 2020, date of publication January 31, 2020, date of current version February 19, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2970728

LoPECS: A Low-Power Edge Computing System for Real-Time Autonomous Driving Services

JIE TANG[1], (Member, IEEE), SHAOSHAN LIU[2], (Senior Member, IEEE), LIANGKAI LIU[3], (Student Member, IEEE), BO YU[2], (Senior Member, IEEE), AND WEISONG SHI[3], (Fellow, IEEE)

[1]Department of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China
[2]PerceptIn, Fremont, CA 94539, USA
[3]Electrical Engineering Computer Science Department, Wayne State University, Detroit, MI 48202, USA

Corresponding author: Jie Tang (cstangjie@scut.edu.cn)

This work was supported in part by the Guangzhou Technology Fund under Grant 201707010148, in part by the Guangdong NSF under Grant 2018A030310408, and in part by the Guangdong Research and Development Key Research of China under Grant 2018B0107003.

ABSTRACT To simultaneously enable multiple autonomous driving services on affordable embedded systems, we designed and implemented LoPECS, a **Lo**w-**P**ower **E**dge **C**omputing **S**ystem for real-time autonomous robots and vehicles services. The contributions of this paper are three-fold: first, we developed a Heterogeneity-Aware Runtime Layer to fully utilize vehicle's heterogeneous computing resources to fulfill the real-time requirement of autonomous driving applications; second, we developed a vehicle-edge Coordinator to dynamically offload vehicle tasks to edge cloudlet to further optimize user experience in the way of prolonged battery life; third, we successfully integrated these components into LoPECS system and implemented it on Nvidia Jetson TX1. To the best of our knowledge, this is the first complete edge computing system in a production autonomous vehicle. Our implementation on Nvidia Jetson demonstrated that it could successfully support multiple autonomous driving services with only 11 W of power consumption, and hence proves the effectiveness of the proposed LoPECS system.

INDEX TERMS Edge computing, QoE (quality of experience), low power, autonomous driving.

I. INTRODUCTION

Many major autonomous driving companies, such as Waymo and Baidu, are engaged in a competition of designing autonomous vehicle which can operate reliably meanwhile in an affordable cost, even in the most extreme environments. Yet, the cost to build such an autonomous vehicle is extremely high, sensors part could easily take over $100,000, the computing system adds another $30,000, resulting in a demo autonomous vehicle easily over $300,000 [1]. Even with the most advanced hardware, having autonomous vehicles co-exist with human-driven vehicles in complex traffic conditions remains a dicey proposition.

To make autonomous driving universally adopted, the major challenge is to simultaneously enable kinds of computation intensive task on a low-power edge computing system with an affordable price. Those autonomous driving services like real-time localization through Simultaneous Localization And Mapping (SLAM), real-time obstacle detection for perception and decision making, should consumes large amount of sensor data and poses huge demands for computing power as well as battery capacity.

However, the design of such a low-power edge computing system is extremely challenging. First, these computation-intensive services are made of complex pipelines and always have tight real-time requirement. For example, inertial measure unit (IMU) data can rush in at a rate as high as 1 KHz in SLAM, requiring the pipeline to consume sensor data at a speed be able to produce 1,000 position updates in a second. Thus, the longest stage in the pipeline cannot take more than 1 millisecond to process. Second, sensor data forms a time series and are independent to each other. That limits the parallelism can be mined for higher efficiency. For movement detection, pictures captured at a rate of 60 frames per second (FPS) flows into CNN pipeline. The objects recognition should be finished within 16 ms. Last, such vehicle computing system has extremely limited energy budget as it runs on the mounted battery. Therefore,

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

it is imperative to optimize power consumption in these scenarios.

To this end, we proposed LoPECS, a **Lo**w-**Po**wer **E**dge **C**omputing **S**ystem to fully exploit the heterogeneous computing resource and explore the offloading potential of edge cloudlet when it passes by. As a demo, a autonomous vehicle the DragonFly Pod, has been developed for a total cost under $10,000 when mass-produced. It is dedicated for low-speed scenarios such as university campuses, industrial parks, and areas with limited traffic.

As far as we know, this is the first paper on a complete edge computing system of a production autonomous vehicle. The contributions of this paper are as follows:

First, to fully utilize the heterogeneous computing resources of low-power edge computing systems, we developed a Heterogeneity-Aware Runtime Layer to schedule autonomous driving computing tasks to heterogeneous computing units for optimal real-time performance. More details can be found in Section 4.

Second, at times computing offloading from vehicle to cloudlet leads to energy efficiency, but whether to offload and how to offload remains an unsolved problem. To address this problem, we developed a vehicle-edge coordinator to dynamically offload tasks to edge cloudlet to optimize user experience in autonomous driving, in terms of lower power and extended battery life. More details of the offloading algorithms can be found in Section 5.

Last but not least, we successfully integrated these components into our proposed LoPECS system and implemented it on Nvidia Jetson TX1. We demonstrated that we could successfully support multiple autonomous driving services with only 11 W of power consumption, and thus proving the effectiveness of the proposed LoPECS system. More details on the system integration can be found in Section 6.

II. AUTONOMOUS DRIVING SERVICES

Before going into the details of the LoPECS system design, let us briefly examine the services needed in autonomous vehicle systems. As shown in Figure 1, a fully functioning autonomous vehicle tightly integrates many technologies, including sensing, localization, perception, decision making, as well as the auxiliary service like smooth interaction with Edge-Cloud platforms for high-definition (HD) map generation, data storage and etc.

Each vehicle uses sensors to perceive environment and safely navigate [1], [2]. Sensor data inputs in localization—the process of understanding its environment—and in making real-time decisions about how to navigate within that perceived environment. These key tasks involve processing a high volume of sensor data (as high as 2GB/S) in real-time and require a complex computational pipeline. In existing designs, an autonomous car must typically be equipped with multiple computing servers, each with several high-end CPUs and GPUs for such high load computing. Consequently, it leaves autonomous vehicles computing systems another big problem in high power consumption — often thousands of watts [20]. Thus, it's crucial to offer a low-power autonomous driving solution with limited vehicle-mounted battery.

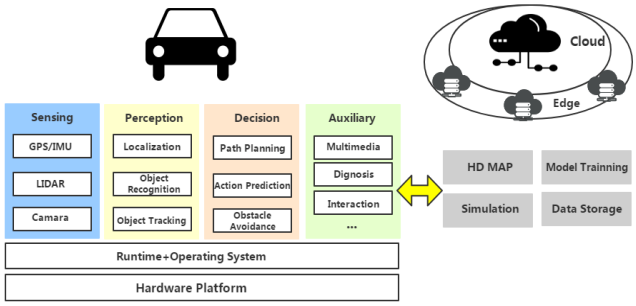


FIGURE 1. Autonomous driving technology stack.

A. SENSING

Normally, an autonomous vehicle consists of several major sensors. Since each type of sensor presents advantages and drawbacks, sensor fusion is used for full sense coverage. These sensors typically include laser imaging detection and ranging (LiDAR), a global positioning system (GPS), an inertial measurement unit (IMU), various cameras, or any combination of these sensors. They are featured in different accuracy, data volume and update frequency, when combined, it generates around multiple gigabytes of raw data per second.

B. PERCEPTION

Perception subsystem percepts the environment by understanding the extracted meaningful information from raw sensor data. Its main tasks include localization, object detection, and object tracking.

1) LOCALIZATION

SLAM refers to the process of constructing or updating the map of an unknown environment while simultaneously keeping track of location of agent. We can use GPS/IMU + camera, which is a visual SLAM, or GPS/IMU + LIDAR, which is a laser-based SLAM. The visual SLAM undergoes the three-step simplified pipeline: 1) triangulating stereo image pairs, a disparity map is used to derive depth information for each point; 2) estimating the motion between the past two frames by establishing correlations between feature points in different frames 3) deriving the current position of the vehicle by comparing the salient features against those in the known map [5].

In DragonFly pod, we use our proprietary SLAM system [7], [8] that utilizes a stereo camera for image generation at 60 FPS, with each frame having the size of 640 X 480 pixels. Meanwhile, the IMU device generates 200 Hz of IMU updates (three axes of angular velocity and three axes of acceleration).

2) RECOGNITION AND TRACKING

The rapid development of deep learning technology guarantees significant object detection and tracking accuracy.

Convolutional Neural Network (CNN) is a type of Deep Neural Network that is widely used in object recognition tasks [4]. Note that in a normal network we would have multiple copies of the convolution, activation, and pooling layers. The network thus can first extract low-level features, and from the low-level it derives high-level features, and at the end it reaches the Fully Connected Layer to generate the labels associated with the input image.

Once an object is identified using a CNN, next comes the automatic estimation of the trajectory of that object as it moves. It is to track nearby moving vehicles and pedestrian and then ensure the current vehicle does not collide with those moving objects

In DragonFly pod, we use the Single Shot Multi-Box Detector [9], which discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location.

C. DECISION

In the decision stage, action prediction, path planning, and obstacle avoidance mechanisms are combined to generate an effective action plan in real time.

The decision unit generates predictions of nearby objects then decides on an action plan based on these predictions. The stochastic model of the reachable position sets of the other traffic participants is built to prediction. Each reachable set are associated with probability distributions. After prediction, path planning comes for best roads choose. The deterministic approach searches all possible paths and utilize a cost function to identify the path. However, its full search may be unable to deliver real-time plans. The probabilistic planners have been utilized to grantees real-time planning decision.

D. MULTIMEDIA

A generic pipeline of speech recognition can be divided into the following stages: First, the speech signal goes through the Feature Extraction stage, which extracts feature vector [28]. Here, we utilize a GMM-based feature extractor; Then the extracted feature vector is fed to the Decoder, using an acoustic model, a pronunciation dictionary, and a language model as input. The decoder then decodes the feature vector into a list of words. Note, we utilize the speech model presented in [10], which uses GMM for classification and HMM for decoding.

E. SERVICES CHARACTERIZATION

1) NO ONE WINS

The characterization study is focused on computer vision with the evaluated workloads listed in Table I. To evaluate CV algorithms on target accelerator platforms, OpenCV CUDA image processing and feature extraction modules, vendor-supplied FPGA benchmarks, and FastCV library are used on GPU-based SoC, FPGA-based Soc, and DSP-based SoC, re-spectively. Based on AlexNet [6], the inference of CNN is evaluated on GPU-based SoC with CuDNNv5 library, and on FPGA-based SoC using hand-crafted OpenCL kernels of convolution layers and fully-connected layers.

TABLE 1. Evaluated workload.

Workload	Description	Ops
Gaussian Blur	640*480 image 5*5 Kernel	15.4M
Convolution	640*480 image 7*7 Kernel	30.1M
Sobel Filter	1920*1080 Image	74.7M
Undistort	640*480 Image	
Feature Detect	640*480 Image	
Optical Flow	52*52 Window	

TABLE 2. Execution time per frame of cv workloads (in ms).

	CPU	DSP	mGPU	GPU	FPGA
Gaussian Blur	2.43	7.17	2.5	4.92	18.3
Convolution	13.1	3.75	6.82	**0.26**	36.8
Sobel Filter	38.5	3.79	3.06	/	13.95
Undistort	23.82	8.01	15.01	**0.35**	/
Feature Detect	18.2	6.89	23.1	9.01	/
Optical Flow	16.91	11.85	/	29.03	9.45

From Table II., it's interesting to find that No single accelerator wins all. Workloads such as Convolution with regular parallel patterns are best suited for GPU execution over other computing platforms, that is because of their favorite in Single Instruction Multiple Thread (SIMT) execution. Similar observation can be made in DL workloads where GPU shows better performance across all layers than FPGA. However, Feature Detect, with more control divergences degrading performance on GPU, is suitable on Single Instruction Multiple Data (SIMD) execution on DSP. Among various Optical Flow algorithms on different platforms, only the two implementing the same Lucas Kanade algorithm are considered, where FPGA shows better performance than GPU. Notice that for Gaussian Blur CPU shows slightly better performance than mGPU, because multicore is boosted for acceleration and there is no overhead for launching kernels.

The shown execution time difference indeed demonstrates the heterogeneity of underlying computing units when they get the chance to perform some autonomous driving operations. If the system can be aware of such heterogeneity, task can be performed in its preferable unit with tailor-made hardware acceleration. Multiple heterogeneous units can work concurrently in a parallel manner for the exploration of parallelism in different grain, task level or fine-grained thread level. Thus, within the same budget, heterogeneity aware execution allows for much great efficiency and longer battery life. In this paper we will explore this heterogeneity by LoPECS, a Low-Power Edge Computing System and talk about its implementation on DragonFly pod.

III. LoPECS ARCHITECTURE

As discussed in the introduction, to enable the affordable and reliable DragonFly Pod, we need to integrate multiple

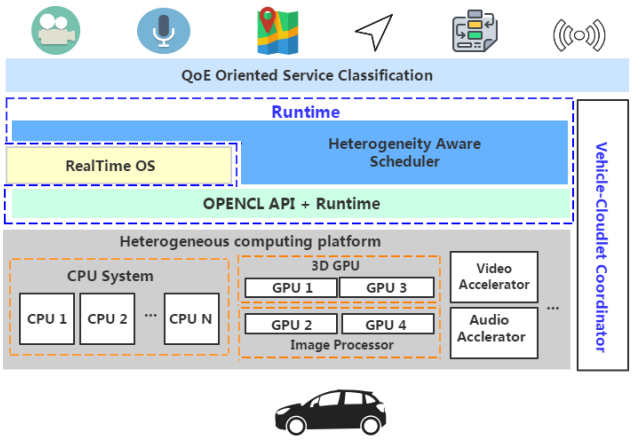


FIGURE 2. LoPECS architecture.

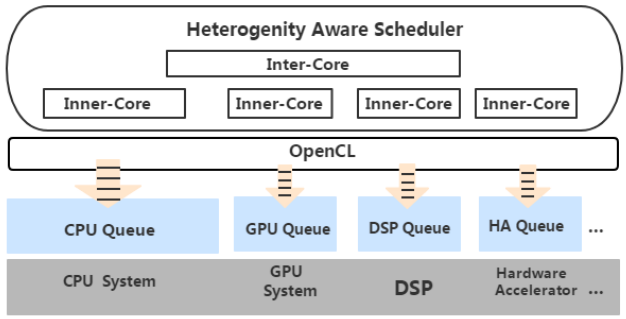


FIGURE 3. The LoPECS runtime design.

autonomous driving services onto a low-power edge computing device.

This poses several challenges: first, as the edge devices usually consist of heterogeneous computing units, computing and energy efficiency can only be achieved if the heterogeneous computing resources can be fully utilized. However, mapping different tasks dynamically to different computing units is complex and challenging, and we do not want to expose this complexity to the service developer. Hence, we need a runtime to dynamically manage the underlying heterogeneous computing resources as well as schedule different tasks onto these units to achieve optimal performance or energy efficiency.

Second, with multiple services running on a resource-constrained device, we need an extremely lightweight operating system to manage these services and facilitate the communications between them. Existing operating systems, such as ROS (Robot Operating System), impose very high computing and memory overheads and thus not suitable for our design.

Third, one way to optimize energy efficiency and to improve the computing capability of edge vehicles is to offload computing workloads to the cloud when possible. However, dynamically deciding whether to offload, and how to offload is another complex and challenging task. To achieve, we need to develop algorithms to handle offloading.

With all these purpose, in this section we propose LoPECS, a **Lo**w-**P**ower **E**dge **C**omputing **S**ystem for autonomous driving service. As shown in Figure 2, At the application layer, currently LoPECS supports localization, obstacle detection, speech recognition and etc. These services support the safe, efficient and realtime driving behaviors. Here, we have introduced a layer named QoE (Qaulity of Experience) Oriented Service Classification. It has the ability to classify different autonomous driving service into *QoE-Time, QoE-Insensitive* and *QoE-Energy*. Such grouping is based on service's features in realtime requirement and hungers for energy cost. With such views, we can get a clear eye on their desires for edge offloading.

To integrate these services, we developed a Realtime OS, an extremely lightweight operating system that manages various services and facilitates their communications with almost zero overheads. Realtime OS serves as the basic communication backbone. Comparing to ROS, Realtime OS is extremely lightweight and optimized for both inter-process communications on the same device, as well as inter-device communications. In this paper, we mainly talk about the task parallelism and cooperation, thus we will leave Realtime OS in our future work.

Below Realtime OS is the LoPECS Runtime layer, which implements two functions: first, it provides an abstraction of the underlying heterogeneous computing resources through and provides acceleration operations; second, it implements a Heterogeneity Aware Scheduling algorithm to manage the mapping of tasks on heterogeneous hardware systems.

In addition, in order to effectively control the energy consumption of autonomous vehicles, LoPECS contains an Vehicle-Cloudlet Coordinator to dynamically offload some tasks to the cloud to achieve optimal energy efficiency. Specifically, taking into account the mobility of vehicles and the cloud availability, we developed an algorithm to dynamically determine the weight of task offload as well as cloud service node selection. We delve into each of these components in the next few sections.

IV. LoPECS RUNTIME

The first major contribution of this paper is the design and implementation of the runtime layer to dynamically map various tasks onto the underlying heterogeneous computing units. This runtime layer is crucial to simultaneously enable multiple autonomous driving tasks on computing and energy resource constrained edge computing systems.

Figure 3 shows the design of the LoPECS Runtime. To manage heterogeneous computing resources, we utilized OpenCL, an open standard for cross-platform, parallel programming of diverse computing units [31]. OpenCL provides the interface for LoPECS to dispatch various applications to the underlying computing resources.

On top of OpenCL, we designed and implemented a *Heterogeneity Aware Scheduler* to manage and dynamically dispatch incoming tasks. Our scheduler is a two-layer design: the inter-core scheduler dispatches incoming tasks, such that they

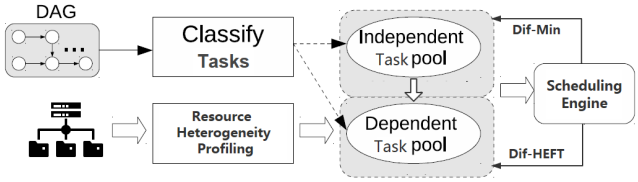


FIGURE 4. The heterogeneity aware scheduler design.

are added to the queue of different heterogeneous computing units based on resource availability and task characteristics. Then within each computing resource queue, the inner-core scheduler determines the order of execution based on the priority and dependency of each task.

A. THE HETEROGENEITY AWARE SCHEDULER

As in Figure 4., the *Heterogeneity Aware Scheduler* classifies tasks into two different categories by identifying data dependencies based on the Directed Acyclic Graph (DAG) analysis. It differentiates underlying kinds of computing resource by the foreknowledge of their heterogeneity. It is the offline profiling results of various autonomous driving tasks on kinds of configured units.

1) CLASSIFYING TASKS BY DEPENDENCIES

Dependencies can appear between tasks, where the next task relies on the output of one or multiple previous tasks and it has to wait until all the previous tasks are finished. So, we can identify the data dependency between jobs by profiling the DAG graph of applications while accepting task submission. With dependencies, we can group task into two pools: one for dependent task and the other for independent.

2) RESOURCE ALLOCATION BETWEEN TASK POOLS

We trigger different scheduler to control the resource allocations between two types of job pools. If there are only independent jobs, the Dif-Min policy in Inter-core Scheduler is developed to dynamically control the resource among these tasks. As Dif-HEFT policy in Inner-core Scheduler is applied to the independent job pool instead, all of the resources allocated to this pool will be allocated to a single task.

B. INTER-CORE SCHEDULER

In detail, the Inter-core Scheduler is triggered for dispatch interdependent tasks to the selected processor. Note that we constrain that each task is only scheduled to one type of computing unit, thus there is no dependencies between tasks. To this end, the inter-core scheduling can be defined to be scheduling independent autonomous driving tasks on heterogeneous multi-core platform [36].

Task scheduling in heterogeneous environments has been proven to be a NP-complete problem and no absolute optimum exists. In [34], authors made a comparison of 11 independent task scheduling heuristics, including Min-Min [29], Max-Min [39], Genetic Algorithm (GA) [40]. The results show that Min-Min has the best comprehensive performance. However, it is still incapable in the cases where cores have big performance difference or application is consists of a large number of short-time tasks. Viewing these shortcomings, to construct a efficient Inter-Core Scheduler for autonomous driving runtime, we propose an improved heuristic named Dif-Min.

In order to better describe the scheduling heuristic, let us first give some definitions as follows [37].

Metatask: a collection of independent tasks to be assigned,

$$Metatask = \{t_i | 0 < i \leq \gamma\}$$

U: the task queue holding unmapped tasks. When scheduling starts, U = Metatask.

Q: the set of heterogeneous cores for scheduling, $Q = \{c_j | 0 < j \leq \chi\}$

$EET(t_i,c_j)$: We use the ETC-Table to store the *EET* value. Each $EET(t_i,c_j)$ in table shows the *Expected Execution Time* of task $t_i | t_i \in Metatask$, when it is running on the unit $c_j | c_j \in Q$. If unit c_k *is the* duplicate core of c_j, we will have $EET(t_i,c_j)=EET(t_i,c_k)$. The *EET* value includes the computation time as well as the time to move the executable and data associated with task t_i from its known source to the destination core. For cases when it is impossible to execute task ti on core c_j, the value of $EET(t_i, c_j)$ is set to infinity.

$mat(c_j)$: Machine availability time for core $c_j, 0 < j \leq \chi$. It is the earliest time core c_j can be available for next round dispatch.

$ct(t_i,c_j)$: The completion time for task ti on core c_j. $ct(t_i,c_j) = mat(c_j) + EET(t_i,c_j)$,

Makespan: The execution time of Metatask. $Makespan = \max ct(t_i,c_j)$, $t_i \in Metatask$, $c_j \in Q$.

Therefore, the purpose of *Inter-core scheduler* is to make wise task dispatching to minimize makespan in autonomous driving on top of heterogeneous hardware. For *Dif-Min,* as its name saying, it will use performance difference and minimal completion time as the basis for heuristics.

To do so, *eq.1* and *eq.2* are proposed to describe the performance difference of task t_i on different cores, one indicates the absolute performance difference and the other shows the relative performance difference. In another word, these two difference indicators demonstrate how much maximal benefits we can get if the task is properly scheduled.

(1) The ratio of the best execution time over the worst one of one computing unit:

$$Div(t_i) = \frac{\max_{j=1}^{|x|}((EET(t_i, c_j)))}{\min_{j=1}^{|x|}(EET(t_i, c_j))} \quad (1)$$

(2) The difference between the best and the worst execution time of each task on each computing unit:

$$Sub(t_i) = \max_{j=1}^{|x|}(EET(t_i, c_j)) - \min_{j=1}^{|x|}(EET(t_i, c_j)) \quad (2)$$

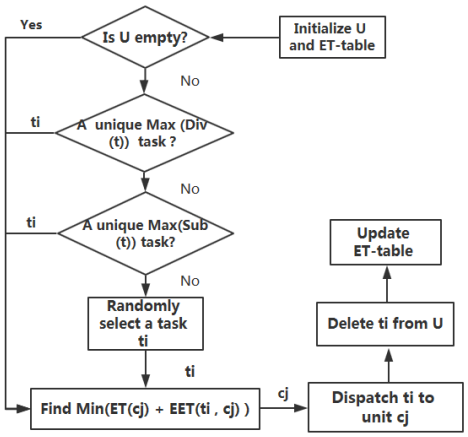


FIGURE 5. Execution flow of inter-core scheduling.

In *Dif-Min*, we also proposed *ET-table,* $1*\chi$ dimension, where χ is the number of computing units. *ET-table is used* to record the sum of execution time of tasks had been allocated to each computing unit. In another word, *ET-table* records the working load each computing unit has taken.

As shown in Figure.5, beginning with $U = Metatask$, *Dif-Min* first find out the task t_i with $Div(t_i)$ =Min($Div(t)$,$t \in U$). After that, it maps the task t_i to the computing unit c_j that can complete all the assignment including t_i in the shortest time, i.e. Min($ET(c_j)$+ $EET(t_i,c_j)$). Once task t_i is dispatched, we update the *ET-table* by adding the execution time of t_i in $M(c_j)$. The computing unit with minimal sum result is the first unit target for next round scheduling, since it can provide the most available time slots for future execution. When task t_i finishes, we remove it from task queue U and we repeat this process until all tasks have been scheduled, i.e U is empty. The time complexity of *Dif-Min* is $O(\chi*\gamma)$, where χ is the number of computing unit and γ is the number of tasks in *Metatask*. Compared with the complexity of $O(\chi*\gamma^2)$ in *Min-Min*, *Dif-Min* can effectively reduce the scheduling complexity.

If the task with largest $Div(t_i)$ is not unique, we further give priority to the task with largest $Sub\ (t_i)$ and dispatch that task first. If the task with largest $Sub(t_i)$ is not unique too, we then randomly select a task from the candidates. The performance difference, $Div(t_i)$ and $Sub(t_i)$, can be viewed as the room for performance improvement when the task t_i is accelerated to the most extend. Thus, task t_i with maximum difference deserves acquirement of the fastest unit since it can contribute the most to the execution time reduction.

C. INNER-CORE SCHEDULER

After Inter-Core Scheduling, all computing units have been assigned with tasks, which are queued to be processed. Now the *Inner-core scheduler* takes over. Different from the Inter-Core scheduling, the inner-core scheduling should mine the dependency between tasks. A task turns to be ready only after all its preceding tasks have been executed. Thus, *Inner-core scheduler* can be described as scheduling multiple dependent tasks involved in the applications to a certain number of heterogeneous processors, expecting to finish the application in the shortest time [40], [42]. In [38], author concludes in a comprehensive way, *HEFT* (Heterogeneous Earliest Finish Time) is the optimal choice.

TABLE 3. Data setup.

Task #	$\lvert\chi\rvert \in \{10, 20, 30, 40, 50\}$
Processor #	$\lvert\gamma\rvert \in \{3. 4, 5, 6\}$
EET Table	$EET(t_i, c_j)\ \lvert i \leq \chi, j \leq \gamma$
Execution Trace	$Task\ i \lvert i \leq \chi$, *start time, duration, dependency*

In eq.3, in *HEFT* the dependency weight of task t_i includes the weight of its succeeding tasks, the average communication cost of data between task t_i and t_j when they are executed in different units, and the average execution cost of t_i. We incorporate the feature of computing unit into HEFT and propose a heterogeneity aware version named *Dif-HEFT*. In *Dif-HEFT,* the priority of t_i has been augmented with its absolute performance difference $Div(t_i)$ *value*. Knowing how much improvement gap between units, the decision on target unit will try to accelerate the execution of t_i to the most extend meanwhile follow up the preexisting dependency topology.

$$Wt(t_i) = \overset{|x|}{\underset{k=1}{Avg}}(EET(t_i, c_i)) + \max_{t_i \in Succ(t_i)} \left\{ Wt(t_j) + \overset{|x|}{\underset{k=1}{Avg}}(Comm(t_i, c_i))_{t_i onc_k} \right\} \quad (3)$$

$$Pri(t_i) = Wt(t_i) + Div(t_i) \quad (4)$$

With *Dif-HEFT*, we can take a simple two-step solution: ***Task selection.*** According to the DAG, $Pri(t_i)$ value of each task $t_i \lvert\ t_i \in U$ is calculated. The task t_i with $Max(Pri(t_i))$ is selected as the candidate to be allocated in this round. If more than one tasks have the same priority, we will select the t_i with $Max(Sub(t_i))$. If there still has competition, a random selection will be used. ***Unit Selection***. According to the *Matching* principle in [36]. *Dif-HEFT* will map the selected t_i to the unit c_j with $Min(\ ct(t_i,c_j))\lvert\ c_j \in Q$. When t_i is finished , we remove it from U and update the remaining dependency to find next candidate task.

D. SIMULATION EVALUATION AND ANALYSIS

In this section, we use simulation to evaluate the performance of proposed *LoPECS Runtime*.

1) EXPERIMENT SETUP

To make simulation, we need to set up four kinds of data in Table III. The number of tasks and processors directly determines the complexity of scheduling. *EET* table stores the expected execution time and $EET(t_i,\ c_j)$ is the expected execution time of task t_i on processor c_j. We generate the *EET* table and the execution trace according to our profiling for autonomous driving workloads in [46].

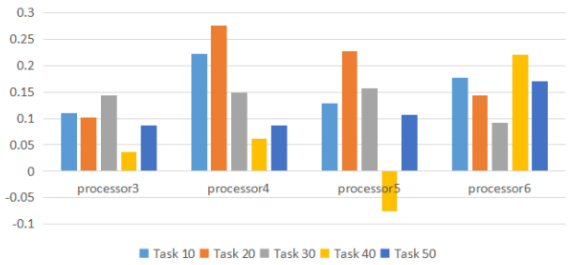


FIGURE 6. Comparison of heterogeneity aware scheduler and user-specific policy.

2) EXPERIMENT RESULT

In the experiment, we generate 100 sets of task execution time data for each group with different number of tasks and processors. In Fig 6, we give the makespan comparison results of *Heterogeneity Aware Scheduler* vs. *User-Specific Policy*. That is because for now all the autonomous driving system leaves the complexity of task coordination to user and no mature work can be found. Here, all makespan data has been normalized. We can clearly see that in most cases, with *Heterogeneity Aware Scheduler* their makespan can averagely be reduced by 10%. In the best case, 20 tasks running on 4 processors, its acceleration can be up to 27.5%. Through further analysis, we can find that this part of efficiency mainly comes from the improvement of *Heterogeneity Aware Scheduler* in two folds: dispatch of task to its hardware preferable units and avoidance of overusing the most powerful core. We can also find there exist a performance downgrade in the case of 40 tasks running on 5 processors. That is because in some case *Heterogeneity Aware Scheduler* may suffer resource equalization and some node will always be invalid for dispatch. However, it happens in very low frequency.

V. QoE ORIENTED SERVICE CLASSIFICATION

Quality of Experience is the intuitive feeling of user when it's served by some service. It can be described as abstract comfort level, but very difficult to be quantified. To the end user of autonomous car, its intuitive feeling is safety and usage time. For a mounted battery, battery life is the core concern for their experience.

To this end, in LoPECS we introduced a *QoE Oriented Service Classification* layer. It can classify tasks distributed at the edges into QoE-*Time, QoE-Energy* and *QoE-Insensitive* three categories. Such classification is based on how their execution state can be sensed by end user. *QoE-Time* and *QoE-Energy* are tagged for those tasks, whose execution has a significant impact on *QoE* in terms of a user noticeable latency or energy cost. Therefore, any efforts on execution time reduction and energy saving should be applied on them to further improve the user experience. Offloading is one of optional way for explore a better user experience.

In the scenario of autonomous driving, each cloudlet node has a fixed communication range. Within the range, the data communication cost can be squeezed to least extend. If out of range, the clients have to spend a large amount of time and energy for unworthy data transfer. For the three services introduced before, *SLAM* is very time-sensitive and requires a position update every 5 ms. If it's offloaded, even in communication range the offloading delay is larger than 5 ms and the resulted user experience is unacceptable even fatal. Thus, Offloading is not the options for *SLAM-like QoE-Time* tasks. For *Objection Recognition* and *Speech Recognition*, system can tolerate a latency as large as 100 ms. It indicates the length of task execution doesn't matter in the way of *QoE* as long as the task can be completed within 100ms. Thus, it leaves great room for energy reduction by offloading task to cloudlet.

In some areas, there may deploy multiple cloudlet nodes and an autonomous vehicle will passes through a number of effective communication ranges. As a result, we should comprehensively consider the state of devices as well availability and distribution of cloudlets in area. And then we make selection by searching for a target cloudlet i in the area that can maximize user experience in expectation.

With such observation, to make offloading the *vehicle-edge Coordinator* should firstly draw a distinction between *QoE-Time* tasks and *QoE-Energy* tasks in the scenario of autonomous driving on edge. Then, by identifying the user experience features and the state of tasks and cloudlet, the *Coordinator* determines whether the task should be offloaded and which cloudlet it should move to. There lists five rules the *Coordinator* must follow below:

1) ***Tasks are created and initialized locally.*** The device is powerful enough to initialize task locally for the sake of a short start-up time.
2) ***Tasks run on local.*** T_{up} is set as the indicator for *QoE-Time*. If the task must be completed in a short time $T|T < T_{up}$, due to some real-time execution requirement, it is *QoE-Time* task and any execution delay will lead to a poor user experience, for example, *SLAM* must be finished in 5ms. Thus, such task should be kept in device to make sure high frequency data manipulation. For some short-time tasks, if they can be finished in time $T|T < T_{up}$, we can leave them on local device as well. That is because offloading tasks to the cloudlets can speed up execution, but it is no more helpful to improve the user experience since their *QoE* requirement has been already met in a local manner.
3) ***Tasks offloaded to the cloudlet.*** T_{down} is set as the indicator for *QoE-Energy*. If the task can tolerant a execution time $T|T > T_{down}$, its insensitivity in execution time leaves great chance for energy reduction. Therefore, tasks can be offloaded to a proper cloudlet for a better user experience in terms of energy can be saved.
4) ***Offloading for Efficiency.*** For the task with execution time $T|\ T_{up} < T < T_{down}$, our decision on task offlodding is decided by the cost function defined in eq. 9. It is a user specific function and demonstrates user's expectation for efficiency of offloading at that time. To guarantee each time we can have efficient offloading

in node i, there must be $C_i^{off} < C_L$. That is to say the cost for offloading must be lower than the one consumed locally.

5) ***Cloudlet Selection for large communication range.*** When offloading is meant to happen, we should select the destination node from a group of available cloudlets. The overhead of offloading comes from data communication. To downgrade the cost to the most we thus should limit offloading just be performed in the largest communication range. The longer time the client vehicle stays within the communication range, the bigger time slice can be reserved for offloading computation, and the more performance and energy consumption benefits we can get.

VI. VEHICLE-EDGE OFFLOADING

The third major contributions of this paper is the design and implementation of an offloading engine that dynamically decides whether and how to offload a computing task for a better User Experience in less power consumption and prolonged battery life. With a lower power and longer life-oriented *vehicle-edge offloading* strategy, all the questions of yes-or-no and how-to in dynamic task offloading should find its answers in the view of *QoE*.

A. VEHICLE-EDGE COORDINATOR

Thus, the core question of the *vehicle-edge Coordinator* is: how to find out an offload-able task and its corresponding destination cloulet node by following the listed five rules. To formalize the question, we start by defining our parameter space as follows:

m: the number of available edge cloudlet nodes;

R_i: communication range of edge cloudlet node, $i = 1, 2, \ldots$,m;

w workload to be offloaded (in terms of number of instructions);

w_i: remaining processing capacity in each edge cloudlet load, $i = 1, 2, \ldots, m$;

f_L: edge client computing speed;

f_{off}^i: computing speed of edge cloudlet node i, $i = 1,2, \ldots,$ m;

v: current moving speed of the client vehicle,

BL: Battery life remains in client vehicle, shown in percentage

(X, Y): current location of the client vehicle;

θ: current heading/orientation of movement of the client vehicle;

D_{in}, D_{out}: transmission data volume as input and output;

r_{in}^i : upstreaming communication bandwidth between client vehicle and edge cloudlet node i;

r_{out}^i : downstreaming communication bandwidth between client vehicle and edge cloudlet node i;

P_{in} : Power of client vehicle when upstream data transferred

P_{out} : Power of client vehicle when downstream data transferred

(X_s^i, Y_s^i): edge cloudlet node physical location, $i = 1, 2, \ldots, m$;

t_a^i(i = 12…, m): the time of the vehicle staying within the communication range of edge cloudlet node i;

t_a^i(i = 12…, m): the time edge cloudlet node i becomes available;

With the parameter above, we can conclude the local execution state in terms of execution time and energy consumption in Eq. 5 and Eq. 6. Here, α and β is the static power factor and dynamic power coefficient respectively [43]. Eq. 7 and Eq. 8 give the computing time and energy cost when execution offloaded into selected cloudlet i. Note that when we talk about cost of time and energy, we only consider the device side. To identify the efficiency of computation in device versus computation offloaded to remote cloudlet node i, we build a cost function in Eq. 9. Here, η is the weight to balance the need for low-power or low-latency. It' a user predefined value and can be dynamically adjusted by a function of B_L. If η set as 1, it's purely a latency sensitive offloading strategy, otherwise it is more apt to energy saving gradually. It shows users' expectation for execution.

Local computing time:

$$t_L = {}^{w}/_{f_L} \tag{5}$$

Local energy consumption: $e_L =$

$$\left(a + \beta * f_L^3\right) * {}^{w}/_{f_L} \tag{6}$$

Offloading computing time:

$$t_{off}^i = {}^{w}/_{f_{off}^i} + D_{in}/r_{in}^i + D^{out}/_{r_{out}^i} \tag{7}$$

Offloading energy consumption:

$$e_{off} = p_{in} * D_{in}/r_{in}^i + p_{out} * D^{out}/_{r_{out}^i} \tag{8}$$

Cost function:

$$\begin{cases} c_L = \eta t_L + (1-\eta)e_L \quad 0 \le \eta \le 1 \\ c_{off}^i = \eta t_{off}^i + (1-\eta)e_{off}^i \end{cases} \tag{9}$$

$$Range_i = \sqrt{\begin{matrix}\left(X + t_c^i * v * \sin\theta - X_c^i\right)^2 \\ +\left(Y + t_c^i * v * \cos\theta - Y_c^i\right)\end{matrix}} \tag{10}$$

Knowing the communication range of edge cloudlet node $Range_i i = 12 \ldots m$ and its coordinate (X_c^i, Y_c^i), by *eq.10* the *Coordinator* can get $t_c^i (i = 12 \ldots m)$, which is the time of client vehicle staying in the communication range of cloudlet i [44]. By follow Rule NO.5, the *Coordinator* considers the cloudlet distribution and the cloudlet states , a maximum $(t_c^i - t_a^i)$ will be found out. Here, we use the available time t_a^i to demonstrate when all the required resource can be standby and when the cloudlet i can finish the last round offloading. Thus, execution should be offloaded to cloudlet node i where $(t_c^i - t_a^i)$ is the maximal in all available cloudlet nodes.

```
IF (t_L < T_up)
    Local Execution;
for( i = 1; i ≤ M-1; i++)   //Offloading
    T = t_c^i - t_a^i; K = 0;
    for(j = 1; j ≤ M-1; j++)
        if(T<t_c^{j+1} - t_a^{j+1})
            T = t_c^{j+1} - t_a^{j+1};
            K = j + 1;
    if(t_L>T_down & e_off^k < e_L)
        return K;
    if(t_L< T_down && c_off^k < c_L)
        return K;
```

FIGURE 7. Offloading algorithm pseudo-code.

B. DYNAMIC TUNING OF T_{down}

In our design, we have $T_{up} = 5$ms, $T_{down} = 100$ms. Such value setting is the results of our profiling on the time tolerance of each edge task. However, a practical task offloading system may have noisy behavior due to continuous variations in the state of vehicles and the cloudlets. It is necessary to develop an adaptive approach to dynamically adjust entry range of task offloading for stable user experience. Here, we do not discuss T_{up}, since it set the baseline for real time of autonomous tasks. T_{down} is the target for improvement and can be further brought down to digest more offloading aiming for energy cost reduction, especially when the battery is low. The relationship between the T_{down} and saved energy can be complex and hard to model. Therefore, we design an adaptive T_{down} tuning approach based on the model-independent *expert fuzzy control (EFC)* technique to integrate with the proposed coordinator [45].

$$C\%(t) = \{e_L(t) - e^i_{off}(t)\}/e_L(t) \tag{11}$$

EFC has two inputs: *C%(t) and BL(t). Input* C%(t) in eq 11. reflects the energy saving in percentage by offloading and *BL(t)* is the remaining battery of client vehicle at time slot t. The output of *EFC* is the T_{down} adjustment. It has five possible values, i.e., f(+2), f(+1), f(0), f(−1) and f(−2) which denote aggressively increasing, increasing, keeping, decreasing and aggressively decreasing the boundary of T_{down} respectively. Note that no matter how T_{down} will fluctuate, the resulted new T_{down} will not exceed 100ms otherwise it will harm the user experience.

Table 4 gives the rule base of *EFC*. The control rules are defined using linguistic variables corresponding to the two inputs,*C%(t)* and *BL(t)*. The fuzzification process converts the numeric inputs into linguistic values such as NL(negative large), NM(negative medium), NS(negative small), Z(zero), PS(positive small), PM(positive medium) and PL(positive large). The rules are in the form of If-Then statements. For example, If *C%(t)* is PL and *BL(t)* is PL, the adjustment in T_{down} is f(-2). The rationale behind this rule is: when the vehicle is almost out of batteries meanwhile in last offloading the gained energy efficiency is very limited, we should lower down T_{down} to permit through more tasks be offloaded for the purpose of saving energy. With such fuzzy control, we can make online decisions of T_{down} to maximize user experiences in energy cost based on vehicle states especially when battery is low.

TABLE 4. The control rule base table.

		BL(t)				
		NL	NS	Z	PS	PL
C%(t)	NL	f(2)	f(1)	f(0)	f(-1)	f(-2)
	N	f(1)	f(1)	f(0)	f(-1)	f(-1)
	Z	f(0)	f(0)	f(0)	f(0)	f(0)
	PS	f(1)	f(1)	f(0)	f(-1)	f(-1)
	PL	f(2)	f(1)	f(0)	f(-1)	f(-2)

VII. LoPECS IMPLEMENTATION ON JETSON

In this section we implement the aforementioned LoPECS architecture with the runtime layer, the Realtime OS, and the offloading engine, onto a Nvidia Jetson TX1 [24]. We examine the detailed performance and power consumption of such implementation and demonstrate that we could successfully support multiple autonomous driving services with only 11 W of power consumption, and thus proving the effectiveness of the proposed LoPECS system.

A. HARDWARE SETUP

The system consists of four parts: the sensing unit, the perception unit, and the decision unit, which are implemented on Jetson TX1, and the execution unit, which is the vehicle chassis. The vehicle chassis receives commands from the Jetson TX1, and executes the commands accordingly. A 2200 mAh battery is used to power the Jetson TX1 board.

The Jetson TX1 SoC consists of a 1024-GFLOP Maxwell GPU, a 64-bit quad-core ARM Cortex-A57, and hardware H.265 encoder/decoder. In addition, onboard components include 4GB LPDDR4, 16GB eMMC flash, 802.11ac WiFi, Bluetooth 4.0, Gigabit Ethernet, and accepts 5.5V-19.6VDC input. Peripheral interfaces consist of up to six MIPI CSI-2 cameras (on a dual ISP), 2x USB 3.0, 3x USB 2.0, PCIe gen2 x4 + x1, independent HDMI 2.0/DP 1.2 and DSI/eDP 1.4, 3x SPI, 4x I2C, 3x UART, SATA, GPIO, and others. Jetson TX1 draws as little as 1 watt of power or lower while idle, around 8-10 watts under typical CUDA load, and up to 15 watts TDP when the module is fully utilized. The four ARM A57 cores automatically scale between 102 MHz and 1.9 GHz, the memory controller between 40MHz and 1.6GHz, and the Maxwell GPU between 76 MHz and 998 MHz.

Regarding the hardware setup, a visual inertial camera module [25] is connected to the TX1 board. This module generates high-resolution stereo images at 60 FPS along with IMU updates at 200 Hz. This raw data is fed to the SLAM pipeline to produce accurate location updates and fed to the

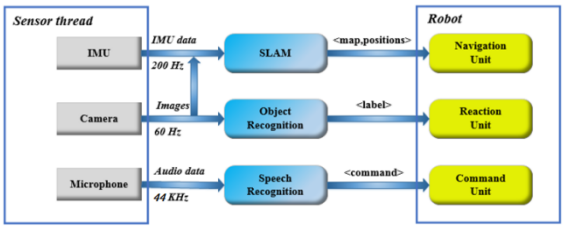


FIGURE 8. System integration.

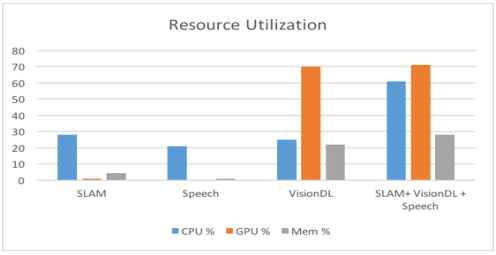


FIGURE 9. Resource utilization on TX1.

CNN pipeline to perform object recognition. In addition, the TX1 board is connected to the underlying chassis through a serial connection. This way, after going through the sensing, perception, and decision stages, TX1 sends commands to the underlying chassis for navigation purpose. For instance, after the SLAM pipeline produces a map of the environment, the decision pipeline can instruct the vehicle to move from location A to location B, and the commands are sent through the serial interface. For speech recognition, to emulate commands, we initiate a thread to constantly perform audio playback to the speech recognition pipeline.

B. SYSTEM ARCHITECTURE

Once we have made a decision on the hardware setup, the next challenge is to design a system architecture to tightly integrates these services. Figure 8 presents the architecture of the system we implement on the Jetson TX1. At the front end, we have three sensor threads to generate raw data: the camera thread generates images at a rate as high as 60 Hz, the IMU thread generates inertial updates at a rate of 200 Hz, and the microphone thread generates audio signal at a rate of 44 KHz. The image and IMU data then get into the SLAM pipeline to produce a position update at a rate of 200 Hz. Meanwhile, as the vehicle moves, the SLAM pipeline also extends the environment map. The position updates, along with the updated map, then get passed to the navigation thread to decide how the vehicle makes its next move. The image data also gets into the object recognition pipeline to extract the labels of the objects that the vehicle encounters. The labels of the objects then get fed into the reaction unit, which contains a set of rules defining the actions to take when a specific label is detected.

For instance, a rule can be that whenever a passenger gets into the vehicle, the vehicle should greet the passenger. The audio data gets through the speech recognition pipeline to extract commands, and then commands are fed to the command unit. The command unit stores a set of predefined commands, and if the incoming command matches one in the predefined command interface, the corresponding action is triggered. For instance, we implement a command ''stop'', whenever the word ''stop'' is heard and interpreted, the vehicle stops all its ongoing actions.

This architecture provides very good separation of different tasks, with each task hosted in its own process. The key to high performance and energy efficiency is to fully utilize the underlying heterogeneous computing resources for different tasks. For instance, feature extraction operations used in the frontend of SLAM as well as CNN computations exhibit very good data parallelism, thus it would be beneficial to offload these tasks to GPU, which frees up CPU resources for other computation, or for energy efficiency. Therefore, in our implementation, the SLAM frontend is offloaded to GPU, while the SLAM backend is executed on CPU; the major part of object recognition is offloaded to GPU; the speech recognition task is executed on CPU. We will explore how this setup behaves on the Jetson TX1 SoC in the next subsections.

C. INTEGRATED PERFORMANCE EVALUATION

In this subsection we study the performance of this system. When running all the services on the system, the SLAM pipeline can process images at 10 FPS on TX1 if we use CPU only. However, once we accelerate the feature extraction stage on GPU, the SLAM pipeline can process images at 18 FPS. In our practical experience, once the SLAM pipeline is able to process images at more than 15 FPS, we have a stable localization service. As a reference, we also measured the SLAM performance on an Intel Core i5 CPU, where at its peak the SLAM pipeline processes images at 15 FPS. Therefore, with the help of GPU, the TX1 SoC can outperform a general-purpose CPU for SLAM workloads.

For the vision deep learning task using Jetson Inference engine, we can achieve 10 FPS in image recognition. This task is mostly GPU-bound. For our low-speed autonomous driving application, the vehicle travels at a fairly slow speed (at 3 m/s), where 10 FPS should satisfy our needs. For the speech recognition, we use Kaldi [10] and it is CPU-bound. We can convert an audio stream into words with 100 ms latency. In our requirement, we can tolerate 500 ms latency for such tasks. In summary, to our surprise, after we enable all these services, TX1 can still satisfy the real-time performance requirement. The main reason is that GPU performs most of the heavy lifting, especially for SLAM and vision tasks.

Next we present the system resource utilization when running these tasks. As shown in Figure 9, when running the SLAM task, it consumes about 28% CPU, 2% GPU, and 4% of system memory. The GPU is mainly used to accelerate feature extraction in this task. When running speech recognition, it consumes about 22% CPU, no GPU, and 2% system memory. For vision-based deep learning task, it consumes 24% CPU, 70% GPU, and 22% of system memory. When combining all three tasks together, the system consumes 60%

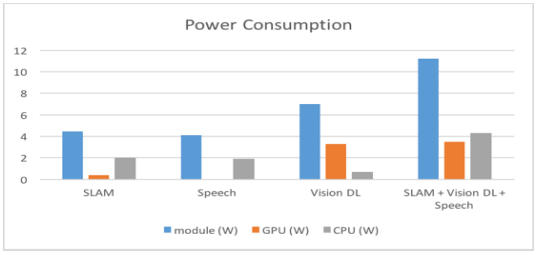


FIGURE 10. Power consumption on TX1.

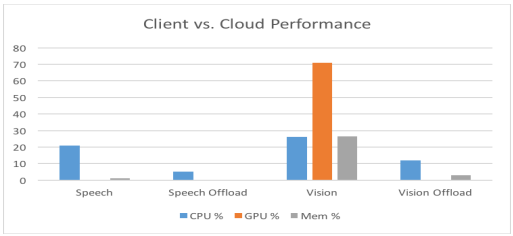


FIGURE 11. Client VS Cloud performance.

CPU, 72% GPU, and 28% of system memory, still leaving enough headroom for other tasks.

Next we present the power consumption behavior. As shown in Figure 10, even when running all these tasks simultaneously, the TX1 module only consumes 11 W, where the GPU consumes 3.5 W, and the CPU consumes 4.2 W. In other words, with an 11 W power envelope, we can enable real-time robot localization, object recognition, and speech recognition on a TX1 SoC module.

D. EDGE AND CLOUD COOPERATION

We deploy the cloud within the local area network. After making this configuration, we have a local cloud that can perform object recognition within 100 ms and speech recognition within 200 ms, meeting the real-time requirement for the robot deployment.

Regarding resource utilization and power consumption, Figure 11 shows the resource utilization of offloading the services compared to executing locally. When offloading the tasks, we send the image or the audio file to the cloud and then wait for the results. For speech recognition, offloading consumes 5% of CPU vs. 20% CPU when executing locally. For object recognition, offloading consumes 12% CPU vs. 25% CPU and 70% GPU. When offloading object and speech recognition tasks and executing the SLAM task locally, the module consumes 5 W. Under this configuration a 2200 mAh battery can power the device for about five hours, which represents a 2.5X boost in running time.

Based on these results, we conclude that in order to optimize energy efficiency, we can deploy edge clouds to host object recognition and speech recognition tasks. Especially in a multiple-vehicle deployment, an edge cloud can be shared by the vehicles.

VIII. RELATED WORK

Several works focus on the functionality of the autonomous driving system. In [11], Franke et al. addresses the challenges that applying autonomous driving system in complex urban traffic. They also propose an approach called Intelligent Stop & Go. Junior is the first work to introduce a full system of self-driving vehicles, which includes sensor models and deployment and software architecture design [12], [13]. Junior presents dedicated and comprehensive information about applications and software flow diagram for autonomous driving. Urmson et al develop an autonomous vehicle called Boss by using sensors including GPS, radar, camera etc [14]. Boss consists of three layers: mission planning layer, behavioral layer, and motion planning layer. Kato et al. present algorithms, libraries and datasets that are required for recognition, decision making and control [15].

There are also several works on evaluating the autonomous driving system and optimizing the performance. KITTI [16], [17] is the first benchmark suite for autonomous driving system. It comprised rich stereo image data and 2D/3D object annotated data. According to different data type, it also provided the dedicated method to generate the ground truth and calculate the evaluation metrics. CAVBench is an edge computing benchmark for Connected and autonomous vehicles, mainly focuses on the performance and power consumption of edge computing systems for autonomous vehicles [35].

Jo et al. apply the distributed system architecture into the design of autonomous driving system [18]. And a system platform is proposed to manage the heterogeneous computing system of the distributed system. The implementation of the proposed autonomous driving system is presented in [19].

In [20], [21], an autonomous driving system based on current award-winning algorithms is implemented and they find three computational bottlenecks for CPU based systems. They compare the performance when heterogeneous computing platforms including GPUs, FPGAs, and ASICs is used to accelerate the computation. With the acceleration approach, their system can meet the performance constraints for autonomous driving system. However, more works can be done on the design of the system to promote the performance except for using hardware to accelerate the algorithms. In [22], Gao propose a safe SOC system architecture. And the security level of autonomous driving application is also discussed. However, the performance is not considered in the design of the system.

Recently, some work begins to enable edge computing in autonomous driving system. Zhang et al. propose an Open Vehicle Data Analysis Platform (OpenVDAP) for connected and autonomous vehicles [23]. OpenVDAP is a full-stack edge-based platform including vehicle computing unit, an isolation-supported and security & privacy-preserved vehicle operation system, an edge-aware application library, as well as task offloading and scheduling strategy. OpenVDAP allows connected and autonomous vehicles to dynamically examine each task's status, computation cost and the optimal scheduling method so that each service could be finished in near real time with low overhead. Meanwhile,

safety and security can also be a vital factor in the design of autonomous driving system.

IX. CONCLUSION

Affordability is the main barrier blocking the ubiquitous adoption of autonomous driving. One of the major contributors to the high cost is the edge computing system, which can easily cost over $20,000 each. To address this problem, we built an affordable and reliable autonomous vehicle, the DragonFly pod, and we target low-speed scenarios, such as university campuses, industrial parks, and areas with limited traffic.

Within this cost structure, we had to simultaneously enable localization, perception, and speech recognition workloads on an affordable and low-power edge computing system. This was extremely challenging as we had to manage these autonomous driving services and their communications with minimal overheads, fully utilize the heterogeneous computing resources on the edge device, and offload some of the tasks to the cloud for energy efficiency.

To meet these challenges, we developed *LoPECS*, an edge computing framework consists of an extremely lightweight operating system: a runtime layer to fully utilize the heterogeneous computing resources of low-power edge computing systems; and an edge-cloud coordinator to dynamically offload tasks to the cloud to optimize edge computing system energy consumption. As far as we know, this is the first complete edge computing system of a production autonomous vehicle.

The results were encouraging we implemented *LoPECS* on a Nvidia Jetson TX1 and we demonstrated that we could successfully support vehicle localization, obstacle detection, and speech recognition services simultaneously, with only 11 W of power consumption, and hence proving the effectiveness of the proposed *LoPECS* system.

In the next step, we plan to extend *LoPECS* to support more heterogeneous edge computing architectures with more diverse computing hardware, including DSP, FPGA, and ASIC accelerators [32]–[34]. Besides low-speed autonomous driving, we believe *LoPECS* has much broader applications: by porting *LoPECS* to more powerful heterogeneous edge computing systems, we can deliver the computing power to L3/L4 autonomous driving; and with more affordable edge computing systems, *LoPECS* can be applied for delivery robots, industrial robots, etc.

REFERENCES

[1] S. Liu, L. Li, J. Tang, S. Wu, and J.-L. Gaudiot, "Creating autonomous vehicle systems," *Synth. Lectures Comput. Sci.*, vol. 6, no. 1, p. 186, Oct. 2017.

[2] S. Liu, J. Tang, C. Wang, Q. Wang, and J.-L. Gaudiot, "A unified cloud platform for autonomous driving," *Computer*, vol. 50, no. 12, pp. 42–49, Dec. 2017.

[3] S. Thrun and J. J. Leonard, *Simultaneous Localization and Mapping, Springer Handbook of Robotics*. Springer, 2008, pp. 871–889.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[5] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn, "Speaker verification using adapted Gaussian mixture models," *Digit. Signal Process.*, vol. 10, nos. 1–3, pp. 19–41, Jan. 2000.

[6] A. Krizhevsky, "Imagenet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012, pp. 1097–1105.

[7] Z. Zhang, S. Liu, G. Tsai, H. Hu, C. C. Chu, and F. Zheng, "PIRVS: An advanced visual-inertial SLAM system with flexible sensor fusion and hardware co-design," Oct. 2017, *arXiv:1710.00893*. [Online]. Available: https://arxiv.org/abs/1710.00893

[8] F. Zheng, G. Tsai, Z. Zhang, S. Liu, C.-C. Chu, and H. Hu, "Trifo-VIO: Robust and efficient stereo visual inertial odometry using points and lines," Mar. 2018, *arXiv:1803.02403*. [Online]. Available: https://arxiv.org/abs/1803.02403

[9] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 21–37.

[10] D. Povey, A. Ghoshal, and G. Boulianne, "The Kaldi speech recognition toolkit," in *Proc. IEEE Workshop Autom. Speech Recognit. Understand. (EPFL)*, Dec. 2011, pp. 1–2.

[11] U. Franke, D. Gavrila, S. Gorzig, F. Lindner, F. Puetzold, and C. Wohler, "Autonomous driving Goes downtown," *IEEE Intell. Syst.*, vol. 13, no. 6, pp. 40–48, Nov. 1998.

[12] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, and D. Johnston, "Junior: The stanford entry in the urban challenge," *J. Field Robot.*, vol. 25, no. 9, pp. 569–597, Sep. 2008.

[13] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun, "Towards fully autonomous driving: Systems and algorithms," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Baden-Baden, Germany, Jun. 2011, pp. 163–168.

[14] C. Urmson *et al.*, "Autonomous driving in urban environments: Boss and the urban challenge," *J. Field Robot.*, vol. 25, no. 8, pp. 425–466, 2008.

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

[16] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2012, pp. 3354–3361.

[17] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The KITTI dataset," *Int. J. Robot. Res.*, vol. 32, no. 11, pp. 1231–1237, Sep. 2013.

[18] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of autonomous Car—Part I: Distributed system architecture and development process," *IEEE Trans. Ind. Electron.*, vol. 61, no. 12, pp. 7131–7140, Dec. 2014.

[19] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of Autonomous Car—Part II: A case study on the implementation of an autonomous driving system based on distributed architecture," *IEEE Trans. Ind. Electron.*, vol. 62, no. 8, pp. 5119–5132, Aug. 2015.

[20] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[21] 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," *SIGPLAN Not.*, vol. 53, no. 2, pp. 751–766, Mar. 2018.

[22] L. Gao, "Safe and secure SOC architecture for autonomous driving," in *Proc. Int. Symp. VLSI Design, Automat. Test (VLSI-DAT)*, Apr. 2018, p. 1.

[23] Q. Zhang, Y. Wang, X. Zhang, L. Liu, X. Wu, W. Shi, and H. Zhong, "OpenVDAP: An open vehicular data analytics platform for CAVs," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 1310–1320.

[24] *Nvidia Jetson TX1*, Accessed: Jun. 20, 2018. [Online]. Available: https://developer.nvidia.com/embedded/buy/jetson-tx1

[25] *PerceptIn DragonFly*. Accessed: Jun. 20, 2018. [Online]. Available: https://www.perceptin.io/dragonfly

[26] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS*. Newton, MA, USA: O'Reilly Media, 2015.

[27] *Nanomsg*. Accessed: Apr. 12, 2018. [Online]. Available: http://nanomsg.org/

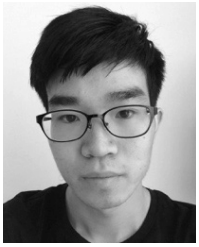
[28] *Percept In's Under-$10K Self-Driving Vehicle*. Accessed: Jun. 20, 2018. [Online]. Available: https://spectrum.ieee.org/cars-that-think/transportation/self-driving/perceptins-sub10k-selfdriving-vehicle

[29] H. Jin, H. Chen, J. Chen, P. Kuang, L. Qi, and D. Zou, "Real-time strategy and practice in service grid," in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf.*, Nov. 2004, pp. 161–166.

[30] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *Proc. HotCloud*, vol. 10, 2010, p. 4.
[31] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66–73, May 2010.
[32] W. Fang, Y. Zhang, B. Yu, S. Liu, and D. ecember. 2., "FPGA-based ORB feature extraction for real-time visual SLAM," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, Dec. 2017, pp. 275–278,
[33] W. Fang, Y. Zhang, B. Yu, and S. Liu, "DragonFly+: FPGA-based quad-camera visual SLAM system for autonomous vehicles," in *Proc. HotChips*, 2018, p. 1.
[34] J. Tang, B. Yu, S. Liu, Z. Zhang, W. Fang, and Y. Zhang, "π-SoC: Heterogeneous SoC architecture for visual inertial SLAM applications," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2018, pp. 8302–8307.
[35] Y. Wang, S. Liu, X. Wu, and W. Shi, "CAVBench: A benchmark suite for connected and autonomous vehicles," in *Proc. 3nd ACM/IEEE Symp. Edge Comput. (SEC)*, Oct. 2018, pp. 30–42.
[36] M.-Y. Wu and W. Shu, "A high-performance mapping algorithm for heterogeneous computing systems," in *Proc. 15th Int. Parallel Distrib. Process. Symposium. IPDPS 2001*, Nov. 2002.pp. 528–532.
[37] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.
[38] V. Boudet. *Heterogeneous Task Scheduling: A Survey*. Accessed: Dec. 25, 2008. [Online]. Available: http://lara.inist.fr/handle/2332/752
[39] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithmsis independent of sizable variances in run-time predictions," in *Proc. 7th IEEE Heterogeneous ComputingWorkshop (HCW)*, 1998, p. 7987.
[40] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *J. Parallel Distrib. Comput.*, vol. 47, no. 1, pp. 8–22, Nov. 1997.
[41] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.
[42] I. Ahmad, M. Dhodhi, and R. Ul–Mustafa, "DPS: Dynamic priority scheduling heuristic for heterogeneous computing systems," *IEE Proc., Comput. Digit. Tech.*, vol. 145, no. 6, pp. 411–418, Nov. 1998.
[43] R. Xu, D. Zhu, C. Rusu, R. Melhem, and D. Mossé, "Energy-efficient policies for embedded clusters," *SIGPLAN Not.*, vol. 40, no. 7, p. 1, Jul. 2005.
[44] H. Zhang, Q. Zhang, and X. Du, "Toward vehicle-assisted cloud computing for smartphones," *IEEE Trans. Veh. Technol.*, vol. 64, no. 12, pp. 5610–5618, Dec. 2015.
[45] Y. Chen, B. Yang, A. Abraham, and L. Peng, "Automatic design of hierarchical takagi–sugeno type fuzzy systems using evolutionary algorithms," *IEEE Trans. Fuzzy Syst.*, vol. 15, no. 3, pp. 385–397, Jun. 2007.
[46] L. Liu, S. Liu, Z. Zhang, B. Yu, and J. Tang, "PIRT: A runtime framework to enable energy-efficient real-time robotic applications on heterogeneous architectures," Feb. 2018, *arXiv:1802.08359*. [Online]. Available: https://arxiv.org/abs/1802.08359

JIE TANG (Member, IEEE) received the B.E. degree in computer science from the University of Defense Technology, in 2006, and the Ph.D. degree in computer science from the Beijing Institute of Technology, in 2012. She was previously a Visiting Researcher with the Embedded Systems Center, University of California Irvine, USA, and a Research Scientist with Intel China Runtime Technology Lab. She is currently an Associate Professor with the School of Computer Science and Engineering, South China University of Technology, Guangzhou, China. She is mainly doing research on computer architecture, autonomous driving, and cloud and run-time system. Dr. Tang is a Founding Member and a Secretary of the IEEE Computer Society Special Technical Community on Autonomous Driving Technologies.

SHAOSHAN LIU (Senior Member, IEEE) received the Ph.D. degree in computer engineering from the University of California Irvine. He is currently a Founder and the CEO of PerceptIn, a company focusing on providing visual perception solutions for autonomous robots and vehicles. Before founding PerceptIn, he was a Founding Member of Baidu USA, and the Baidu Autonomous Driving Unit, in charge of system integration of autonomous driving systems. His research focuses on computer architecture, deep learning infrastructure, robotics, and autonomous driving. He has published over 40 high-quality research articles and holds over 150 U.S. international patents on robotics and autonomous driving, he is also the Lead Author of the best-selling textbook *Creating Autonomous Vehicle Systems*, which is the first technical overview of autonomous vehicles written for a general computing and engineering audience. In addition, to bridge communications between global autonomous driving researchers and practitioners. He has a co-founded the IEEE Special Technical Community on Autonomous Driving Technologies and serves as the Founding Vice President. Dr. Liu is an ACM Distinguished Speak and an IEEE Computer Society Distinguished Speaker.



LIANGKAI LIU (Student Member, IEEE) received the B.S. degree in telecommunication engineering from Xidian University, Xi'an, China, in 2017. He is currently pursuing the Ph.D. degree with Wayne State University, Detroit, MI, USA. His current research interests include edge computing, distributed systems, and autonomous driving.

BO YU (Senior Member, IEEE) received the B.S. degree in electronic technology and science from Tianjin University, Tianjin, China, in 2006, and the Ph.D. degree from the Institute of Microelectronics, Tsinghua University, Beijing, China, in 2012. He is currently the CTO of PerceptIn, Fremont, CA, USA, a company focusing on providing visual perception solutions for robotics and autonomous driving. His current research interests include algorithm and systems for robotics and autonomous vehicles. Dr. Yu is also a Founding Member of the IEEE Special Technical Community on Autonomous Driving.

WEISONG SHI (Fellow, IEEE) received the B.S. degree in computer engineering from Xidian University, in 1995, and the Ph.D. degree in computer engineering from the Chinese Academy of Sciences, in 2000. He is currently a Charles H. Gershenson Distinguished Faculty Fellow and a Professor of computer science with Wayne State University. His research interests include edge computing, computer systems, energy-efficiency, and wireless health. He was a recipient of the National Outstanding Ph.D. Dissertation Award of China and the NSF CAREER Award.