IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 28, 2020, accepted February 9, 2020, date of publication February 12, 2020, date of current version February 26, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2973457

A Container Based Edge Offloading Framework for Autonomous Driving

JIE TANG[1], (Member, IEEE), RAO YU[2], SHAOSHAN LIU[3], (Senior Member, IEEE), AND JEAN-LUC GAUDIOT[4], (Life Fellow, IEEE)

[1]Department of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China
[2]Pony AI, Guangzhou 511453, China
[3]PerceptIn, Fremont, CA 94539, USA
[4]Electrical Engineering Computer Science Department, University of California at Irvine, Irvine, CA 92697, 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 Autonomous driving is one of the most innovative applications nowadays. However, autonomous driving is still suffering from heavy calculation, high energy consumption and strict real-time execution constraints. Different from cloud computing, edge computing deploys calculation, storage and service on the edge of network. It is a better platform to serve efficiency and privacy oriented autonomous driving service offloading. To this end, we proposed a container-based edge offloading framework for autonomous driving. This framework builds an Offloading Decision Module, an Offloading Scheduler Module and an Edge Offloading Middleware on top of the lightweight virtualization. It provides the abstraction and management of the execution environment in the granularity of containers on edge. Therefore, it enables the privacy preserve and resource isolation for autonomous driving execution constraints. Its utility preferable offloading schedule strategy formalized the multi-application multi-edge nodes mapping problem into a multiple multidimensional knapsack problem (MMKP) and gave a utility oriented greedy algorithm (GA) for real-time solving. The experimental results show that the proposed framework has high feasibility and isolation meanwhile can guarantee millisecond-level autonomous driving offloading on edge.

INDEX TERMS Edge computing, offloading, energy efficient, container, autonomous driving.

I. INTRODUCTION

As a new way of transportation, autonomous driving technology develops rapidly. Based on the huge volume of sensor data, continuous operations of sensing, perception and decision making all demand a large amount of calculation. Such intensive computing brings serious energy consumption and heat dissipation problem and leads to short battery life. Meanwhile, most of the autonomous driving applications are delay sensitive, which require results returned in a very short time. Therefore, Autonomous Vehicles (AVs) are urgent for some solutions for both energy and execution efficiency. In traditional vehicle-cloud network, the tremendous network delay makes it difficult for autonomous tasks to be completed in time. As an extension of cloud, Edge Computing extends computing, storage and sharing capabilities from cloud to network edges. Such a more compact end-to-end access mode is very suitable for deploying computing migration for those autonomous driving services with high update frequency, low service delay and wide service coverage. Edge computing now turns to be the hope of low-latency while energy-efficient autonomous driving service[1].

Edge computing can provide service offloading and data provision for autonomous vehicles. Through effective service scheduling, it can reduce the computation cost of autonomous vehicles and improve the user experience in driving. But we must recognize that it is necessary to deal with the offloading requests in terms of performance independence, privacy independence and experience independence when multiple offloading services bursts simultaneously. At the same time, it must meet the requirements that these independences will not hurt the effectiveness of computing offloading. Therefore, it needs a more lightweight solutions rather than the current virtual machine (VM)-based virtualization

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

technology for service offloading. Furthermore, as another challenge, the orchestration of lightweight virtualized runtimes is needed.

To this end, in this paper, we study the demand of autonomous driving driven edge computing and propose a container-based offloading framework on edge. Here, we apply containers, which are a lightweight virtualization concept, i.e., less resource and time consuming and look like a better solution for more interoperable application scheduling in edge computing. The proposed container-based framework aims at providing privacy and safety precaution meanwhile takes application finish time as a constraint condition. The main contributions of this article are as follows:

- We analyzed the execution pipeline of autonomous driving and checked its features for computation offloading architectures. With the observation, we proposed a container-based offloading framework for autonomous driving on edge. It includes a offloading decision module, a offloading scheduler module and a offloading middleware to manage the offloading pipeline and resource allocation for all containers in their entire life cycle. The use of such framework can construct a dynamic isolated operating environment for utility maximized but secured autonomous driving application offloading on edge.
- We model the problem of offloading multiple applications into edge nodes as a Multiple Multidimensional Knapsack Problem (MMKP) and gave a resource-constrained offloading utility preferable strategy. It was implemented into a greedy algorithm for MMKP to meet the strict requirement of response delay for autonomous driving applications.
- We evaluate the performance of proposed framework. The simulation results demonstrate that: the container is really a lightweight and secured offloading carrier on edge for autonomous driving. And the container-based framework is with less overhead and can realize autonomous driving computing offloading at millisecond level.

The rest of this paper is organized as follows. In Section II, the scene of application offloading from AVs to MEC (Mobile Edge Computing) nodes is presented. In Section III we stated the container-based offloading framework. In Section IV and V, we review the fundamental reason of choosing Docker for offloading and give the architecture of container-based offloading. In section VI, we describe the multiple edge node offloading scheduling. In Sections VII, we share the detailed experimental methodologies and results to demonstrate the effectiveness of container-based offloading. Finally, we summarize the related work and concluded in Section VIII.

II. VISION OF APPLICATION OFFLOADING FOR AUTONOMOUS DRIVING

The working process of AVs is mainly composed of three stages: Sensing, Perception and Decision.

(1). In the Sensing stage, AVs collect data through GPS, IMU, camera and other vehicular sensors, and perform sensor data pre-processing and sensor fusion.

(2). In the Perception stage, three parts of work need to be accomplished: localization, object recognition and object tracking. AVs complete meter-level localization by integrating GPS and IMU data, and further complete centimeter-level localization by comparing laser point cloud, camera frame data and high precision map (HD Map). In object recognition and tracking, deep learning is used to quickly acquire the semantics and moving direction of objects in the perceptual range.

(3). In the Decision stage, AVs combine prediction, path planning and obstacle avoidance to determine the next action. Prediction mainly uses a stochastic model to predict the correlation probability of other vehicles' accessible location set. In this process, AVs detect obstacles and predict obstacles' behavior to plan trajectory and complete control operations like acceleration, deceleration and orientation.

According to the completion time limit, [2] further divides autonomous driving applications into three categories: Real-time application, interactive application and auxiliary application. For real-time applications, there has a strict limitation on completion time. Taking SLAM [3] as an example, in low-speed AVs position updates are usually carried out every 5 ms. Serious accidents may occur if SLAM fails to return results within 5 ms. For interactive applications, its time tolerance is relatively loose. For example, the tolerable delay time of speech recognition and feature recognition in low-speed AVs is 100 ms. If its offloading can be completed in a relatively short time less than 100ms, the interactive application can be offloaded to edge nodes. For auxiliary applications, it generally does not have real-time response requirements and system diagnostics for error prediction is one case of them. However, such applications are usually computationally expensive and greatly threaten battery life. For the purpose of energy saving and battery life prolonging, it is better to offload this kind of applications to edge nodes.

With the development of 5G technology, the data transmission rate between edge nodes and AVs will be much higher. Lower network latency gives more tolerance to the completion of offloaded applications. More and more autonomous driving applications can achieve both execution and energy efficiency through vehicle-edge collaborative computing. When multiple applications from multiple autonomous driving vehicles are offloaded in the same edge node, security and privacy must be carefully considered. If the offloaded application is disturbed by other applications or hackers on the edge node, the wrong results may affect vehicle decision-making and cause serious accidents. In order to ensure the safety of AVs, on the one hand, it is necessary to isolate the operating environment of offloaded applications and improve its ability to suppress malicious acts. On the other hand, data isolation should be guaranteed to prevent a large amount of users' privacy information from being illegally acquired.

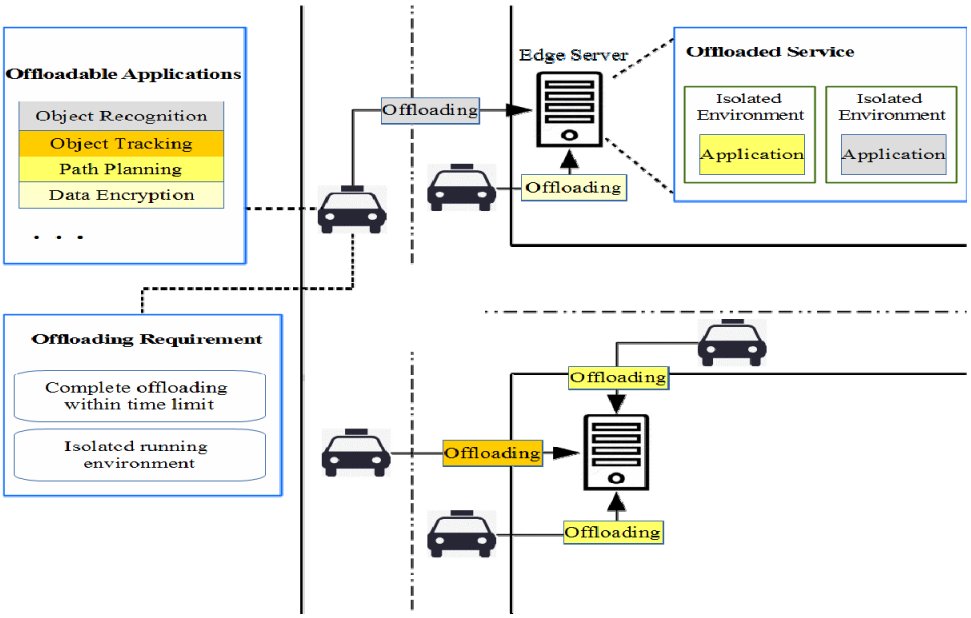

FIGURE 1. Edge-AVs coworking scene.

Thus, we can get an envision in Figure 1 that multiple edge nodes are located on the RAN (Radio Access Network) side and there are a lot of applications from nearby AVs that can be offloaded to them. An edge node can serve multiple vehicles at one time and a vehicle can offload multiple applications to different edge nodes. Such application offloading can effectively alleviate the energy consumption and heat dissipation problems of AVs. To this end, on one hand, the resource utilization efficiency of multiple edge nodes should be maximized to meet the requirements of low latency and high security of autonomous driving. On the other hand, during offloading, multiple offloaded applications should be operating environment isolated and service data isolated to meet the high security and privacy requirements of autonomous driving.

III. CONTAINER BASED EDGE OFFLOADING FRAMEWORK

Edge computing is pushing computing applications, data, and services away from centralized cloud data center architectures to the edges of the underlying network. It is very suitable to deploy computation and data provision related services close to users so as to improve the user experience in dedicated scenarios.

The challenge of computing offloading for autonomous driving in Edge is to meet its requirements in security, privacy and efficiency at the same time. Virtualization is an answer to the need for scheduling offloading services as manageable but independent units. The evolution of virtualization has improved multi-tenancy capabilities and resulted container techniques leads to more lightweight solutions. To this end, we proposed a container-based edge offloading framework for autonomous driving. Through container-based light-weight virtualization in edge nodes, the application running environment can be isolated to provide safer offloading service for AVs. An offloading scheduling strategy for multiple edge nodes under resource constraints is proposed to meet the strict requirement of response delay for autonomous driving applications.

As shown in Figure 2, there are three agents in the proposed framework: AVs, Edge Servers and a Node Coordinator.

Offloading Decision Module locates in AVs, which are used to decide whether to offload services or not. Autonomous driving applications can be divided into three types: real-time applications, interactive applications and auxiliary applications. Offloading decision is made based on the application type and other concerns like data transmission, calculation load, network speed, etc. Detail of the offloading decision algorithm is not discussed in this paper. The existing decision theory [4]–[6] is mature enough that it can be directly applied in the decision of offloadable autonomous driving applications. We just need to guarantee that the offloading decision module will follow three conditions below to do the computing offloading:

(1). The residual computing power of edge servers can support the offloaded application return the results within the limited time.

(2). The energy consumption for computing offloading is less than that for vehicular execution.

(3). The remaining memory capacity of edge servers can meet the memory requirement of the offloaded application.

Node Coordinator manages multiple edge servers within a valid scope. It monitors the status of multiple edge servers through a service management module, updates the health status and residual resources of edge servers in real-time, schedules multiple target offload edge nodes under resource

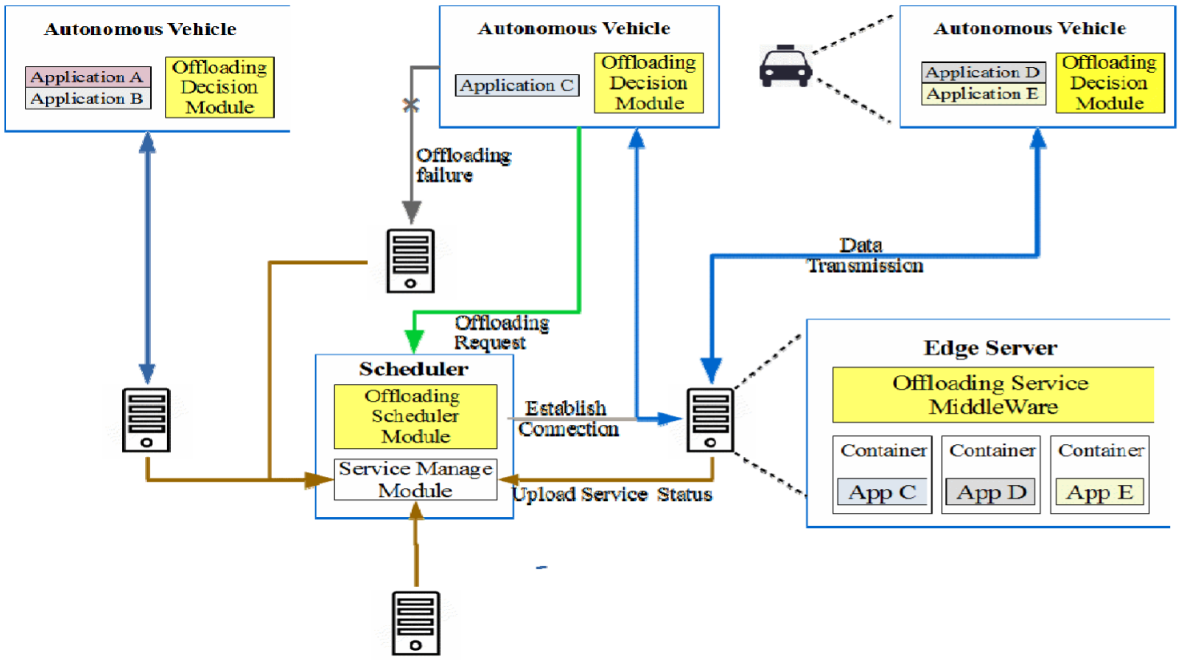

FIGURE 2. Container-based autonomous driving computing offloading framework.

constraints. Through the offloading scheduler module by using the utility maximized scheduling strategy, the optimal offloading targets are selected out and the connections between Edge servers and AVs are established.

Offloading Service Middleware is deployed in edge servers, which can quickly respond to service offloading dispatch. The middleware uses containers to isolate the running environment, application data and hardware resources to realize the security of offloaded applications. The middleware also can flexibly change containers' resources provision according to the requirements of offloaded applications to ensure the execution efficiency.

In order to speed up response time and lower transmission cost, it is preferable to offload applications to the nearest edge server. AVs first obtain the status of such the nearest edge server to check whether its offloading requirements can be satisfied. If so, the nearest edge server is directly selected as an offloading target. There is no need to initiate a request to Node Coordinator to schedule multiple candidate edge nodes. If not, the AVs send an offloading schedule request to Node Coordinator and wait to be dispatched to another appropriate edge server. The selected edge server uses a Docker container to visualize the environment for each offloaded application and returns the result to AVs after offloading completes. So, it can realize the isolation between applications.

IV. USING DOCKER FOR OFFLOADING

Existing computing offloading frameworks can be divided into two categories. One kind framework provides fine-grained computing offloading services based on remote code execution, which can achieve method-level offloading, such as MAUI, a system based on.NET execution environment [7]. Another kind of offloading framework builds VM (Virtual Machine) on the server-side. Each VM has its own virtualized hardware and OS. It is applicable for most programming languages and execution environments and can support application-level coarse-grained offloading. The cloudlet system [8] on edge is its typical implementation.

In order to meet both the safety and response time requirements of autonomous driving applications at the same time, the container-based solution turns to be the optimal choice for offloading. As shown in Figure 3., Containers are much lighterweight than VMs. Container images can be built simply by packaging application code, task dependencies, and environment profiles in an image copy. The startup of a container is also faster than VM. Recent Linux distributions–part of the Linux container project LXC–provide kernel mechanisms named namespaces and cgroups to isolate processes on a shared operating system [10]:

Namespace isolation enables the isolations of groups of processes. This ensures that they cannot sense the resources in other groups. Different isolating kernel and version identifiers own their dedicated namespaces for process isolation, network interfaces, access to inter-process communication, mount-points.

cgroups(control groups) use limit enforcement, accounting and isolation manage to limit the resource access for process groups, e.g., limiting the memory available to a specific container. This allows better isolation between isolated applications on a host. This also restricts the containers in multi-tenant host environments.

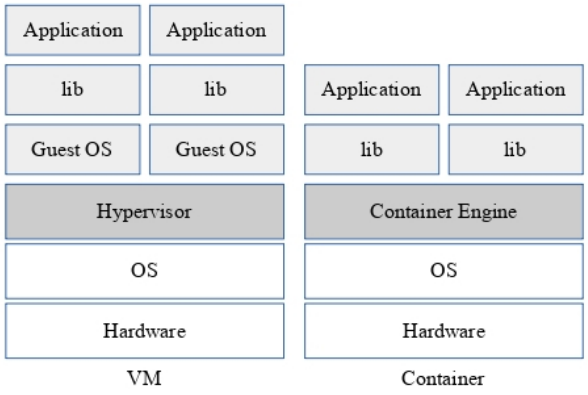

FIGURE 3. Comparisons between containers and VMs.

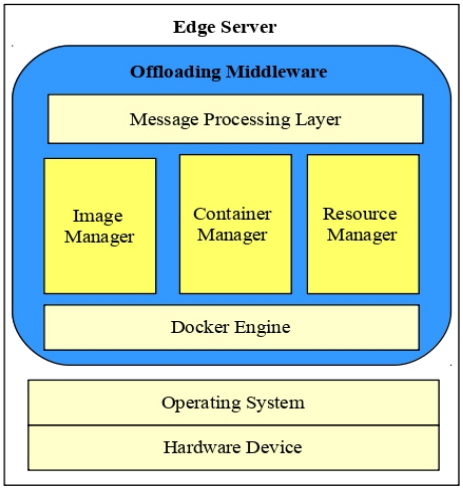

FIGURE 4. Total energy saving made by three scheduling.

With this observation, in this paper, we use Docker [9] as the container engine for offloading isolation. Docker builds on top of Linux LXC techniques and now is the most popular container solution for illustrating containerization. Therefore, Docker is no doubt the optimal one to build a light-weight offloading service and it can better meet the requirements in autonomous driving offloading scenarios.

V. CONTAINER-BASED EDGE OFFLOADING MIDDLEWARE

As shown in Figure 4, the container-based Edge Offloading Middleware includes a Message Processing Layer, an Image Manager, a Container Manager and a Resource Manager. These modules are built on top of the Docker Engine, which provides interfaces for the inter-operations between modules. Message Processing Layer takes care of offloading requests receive, signaling interaction, data transmission and so on. Image Manager focuses on management and update of local Docker images. Container Manager is used to create and destroy Docker containers, meanwhile manages the running status of containers. Resource Manager is used to manage the available resources in the server and configure resource limitation parameters for each container.

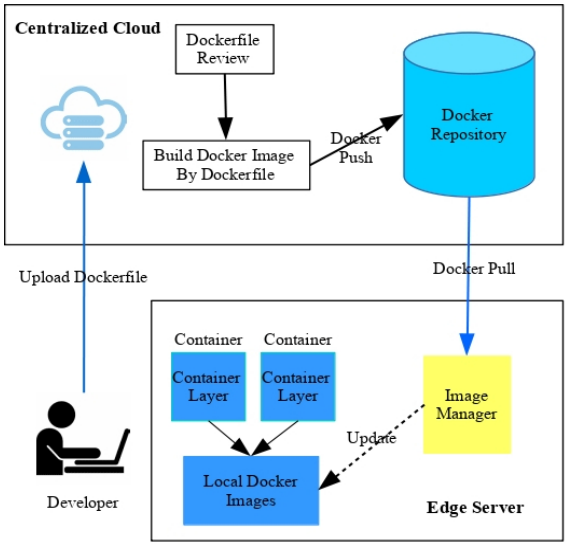

FIGURE 5. Image manager module.

After receiving an offloading request, Message Processing Layer goes through identification and authentication for vehicles requesting offloading by signaling interactions between AVs and target edge nodes. After that, the category of to-be-offloaded applications and their execution data will be sent out from AVs. Image Manager and Container Manager set up corresponding containers by container initialization and image loading. Meanwhile, Resource Manager will allocate some amount of hardware resources to the created containers according to corresponding resource limits. After these operations, the container-based edge node starts to serve computing offloading. The result will be sent back to AVs after execution on edge is completed.

A. IMAGE MANAGER

As shown in Figure 5, the central cloud maintains a Docker image repository to store audited images written by developers. The layered nature of Docker images allows developers to create new images based on existing ones. When multiple containers are created based on the same image, each container only needs to add a writable container layer on top of the basis image, and multiple containers can share all layers within the image. Thus, newly created containers do not have to copy the contents in the image. this feature reduces the cost of container creation and saves the image storage space on the local edge node.

Image Manager manages local images and periodically pulling images update from the central cloud image repository. Image Manager selects the appropriate image from the local list according to the category of to-be-offloaded application, and then moves to container initialization. Container Manager initializes the container and loads the program and all required dependencies. AVs do not need to upload its executable files and related dependencies, but only needs to provide application category and execution data, thus the amount of data transmission can be greatly cut down.

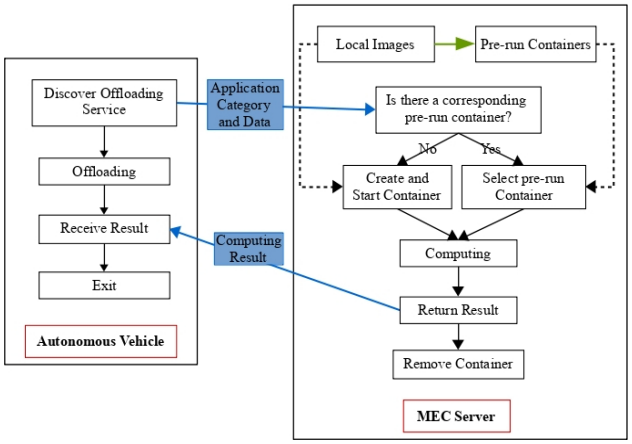

FIGURE 6. Pre-run containers strategy.

B. CONTAINER MANAGER

The main work of the Container Manager is to create and destroy containers, meanwhile manage the status of containers in operation. The Docker Engine already provides corresponding interfaces for these operations. It is known that container creation speed and boot speed is faster than VM and can be as fast as second or even millisecond level. However, in the scenario of computing offloading for autonomous driving, some interactive applications require much shorter response time. The delay effect of container creation is really a concern. In order to meet the real-time offloading requirements of interactive applications, we proposed a Pre-Run container strategy.

As shown in Figure 6, according to the proposed Pre-Run container strategy, Container Manager Pre-Runs some containers for selected to-be-offloaded applications and maintain them just with a very small amount of computing and memory share. When an offloading request arrives, Container Manager selects a Pre-Run container according to the application category, and calls Resource Manager to adjust its resource allocation to start offloading calculation. Containers implements light-weight isolation therefore there wastes little computing and memory capacity to maintain such dormant containers. Besides, multiple containers built with the same image share all read-only layers that their creation will not take up additional storage space on edge servers. It can be concluded that this Pre-Run strategy can create containers in a quite short time just with a very small cost and it's suitable for time-limited application offloading in autonomous driving scenarios.

C. RESOURCE MANAGER

In autonomous driving scenario, it is very important to isolate environment and resources for offloaded services. Resource Manager based on Docker manages kinds of hardware resources including computing units, memory and disk. If an application in a container requests memory exceptionally or maliciously, it may occupy a large amount of memory, leading to memory exhaustion in the edge server. In the worst case, it may cause OOM (Out of Memory Exception). Resource Manager records the amount of memory remaining in the edge server and then sets up the memory use limits for each container. Based on such settings, when creating containers, deleting and adjusting containers, the amount of remaining memory can be adjusted in time. It can limit the malicious memory consumption into an isolated container environment, that misbehavior in using the memory resource will not impact other offloaded services.

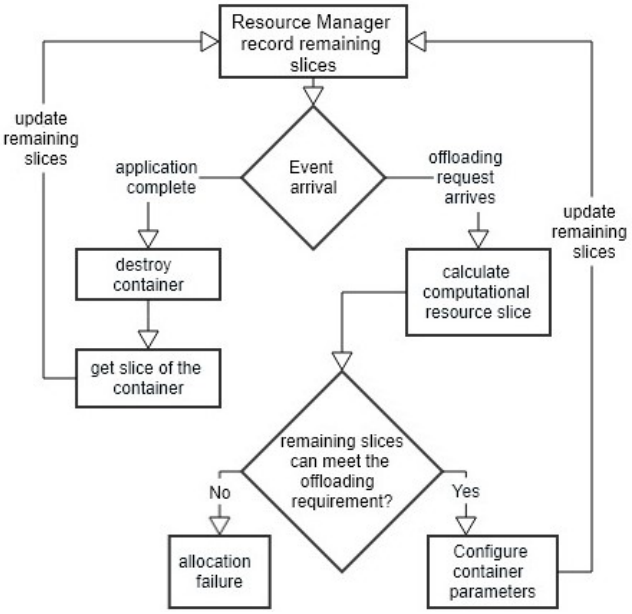

FIGURE 7. Maintenance of computing resource slices.

Offloaded applications of AVs need to be completed within a limited time. When multiple applications compete for computing resources, the finish time of applications running in the container turns to be unpredictable. It is possible that applications cannot be completed in a to-be-guaranteed limited time. In order to realize computing resource management by the granularity of container, Resource Manager divides all available computing resources into N_{total} slices and configures quotes for each container. A container's computational resource slice N_{share} is derived from the following formula:

$$N_{share} = \lceil \frac{V_{container}}{V_{total}} \rceil * N_{total} \quad (1)$$

$V_{container}$ is the computing power allocated to the container and V_{total} is the total computing power of edge server that can provide for offloading service, which can be obtained by benchmark profiling. As long as the sum of N_{share} values of all containers is less than N_{total}, the computing power each container is guaranteed will be no less than $V_{container}$, even when there is server computing resources contention. As shown in Figure 7, the Resource Manager maintains the computing power slices during the entire process and ensures that each container can have sufficient computing resources to undertake offloaded service.

VI. MULTIPLE EDGE NODE OFFLOADING SCHEDULING

The proposed framework also introduces a Node Coordinator to deal with the offloading schedule in multiple edge node scenario. Node Coordinator deploys an offloading scheduler and a service management module. The service management module manages the service status of multiple edge nodes, including their health status and amount of remaining resources. When an AVs fails to offload to its nearest edge node, it can request Node Coordinator for larger-wide offloading target selection. The offloading scheduler will pick up an appropriate edge node from candidates within coordination scope and service offloading redirects.

The discussion in this chapter is based on the following assumptions:

1) It is assumed that the energy consumption generated by the operations on the on-board computing platform is known, the amount of memory required for the normal operation of the application is known, and the computing resources on the MEC server we are talking about only include the CPU.

2) It is assumed that the amount of data to be transmitted of offloading is known, and the data uploading speed and downloading speed between the vehicle and the MEC server are known. The network transmission between the vehicle and the MEC server is relatively stable, and serious network fluctuation is not considered.

3) It is assumed that the speed of application completion is directly proportional to the computing power provided by the MEC server. The overhead of the MEC server system itself is ignored when switching between multiple cores is ignored. The container is assumed to have been pre-created, so the cost of starting the container is not considered.

4) The calculation unloading mode of unmanned vehicles is coarse-grained unloading, which is divided according to the application, and the application cannot be divided again.

Symbols used in this section and their meanings are stated in Table I.

TABLE 1. Symbol for offloading.

Symbol	Definition for Offloading
$T_{execution}(i)$	Execution time of the i^{th} application in local way
I	App collection to be uninstalled
J	Available MEC server collections
$E_{local}(i)$	$i \in I$: the energy consumption of the i^{th} application in the local execution of the vehicle
$E_{offload}(i, j)$	Unloading energy consumption of the i^{th} application to the j^{th} MEC server, $j \in J$
$T_{limit}(i)$	Completion time limit of the ith application
$T_{transmit}(i, j)$	Data transmission time from the i^{th} application to the j^{th} MEC server
$W_{compute}(i)$	Calculation amount of the i^{th} application
x_{ij}	Decide whether to offload the i^{th} application to the j^{th} MEC server, 0-1 variable
V_{ij}	The computing power required to offload the i^{th} application to the j^{th} MEC server
N_{ij}	CPU share needed to offload the i^{th} application to the j^{th} MEC server
RN_j	The remaining CPU share of the j^{th} MEC server
M_i	Memory requirements of the i^{th} application
RM_j	The remaining memory of the j^{th} MEC server
u_{ij}	The utility of uninstalling the i^{th} application to the j^{th} MEC server
U	Total utility

A. SINGLE APPLICATION SCHEDULING

When the nearest edge node fails to take care of AV offloading requirement due to insufficient resources, it can ask help from Node Coordinator. Node Coordinator will choose another capable edge node by scheduling policy and build the direct connection for avs and the selected edge node.

Node Coordinator uses six-tuple to organize the information necessary for offloading. Such six-tuple is $<T_{limit}, W_{transmit}, W_{compute}, P_{transform}, E_{local}, M>$. T_{limit} refers to the application completion time limit. $W_{transmit}$ refers to the execution data that needs to be transmitted. $W_{compute}$ refers to the calculation workload in the application. $P_{transform}$ refers to energy consumption in data transmission. E_{local} refers to energy consumption when the application runs in local mode. M refers to the memory requirements of the application. A successful computing offloading need to guarantee the following rules and they are also the criterions for offloadable service selection:

1). The remaining computing resources of the edge node can make sure application results returned within a time limit.

$$T_{Limit} \leq \frac{W_{transmit}}{B} + \frac{W_{transmit}}{B} \tag{2}$$

2). The energy consumed in computing offloading is less than that for local execution.

$$E_{local} \leq P_{transform} * \frac{W_{transmit}}{B} \tag{3}$$

3). The remaining memory of the edge node can meet the execution requirements in terms of memory.

$$M \leq RM \tag{4}$$

Here, RV refers to the remaining computing resources in the edge server, RM refers to the remaining memory resource in the edge server, B refers to the bandwidth.

The faster data transmission between vehicle and server, the less energy vehicle consumes in computing offloading data in-and-out. Therefore, AVs tend to choose an edge node with the shortest data transmission time to save more energy. When the Node Coordinator receives an offloading request, it filters out q available edge nodes nearby and the set of these selected servers is J. The selected servers must satisfy the requirement that energy consumed in offloading is less than that in local execution and data transmission time is less than the finish time limit of offloaded application.

$$\forall j \in J, \quad E_{offload}(j) < E_{local} \text{ and } T_{transmit}(j) \leq T_{limit} \tag{5}$$

Here, $E_{offload}(j)$ refers to the offloading energy cost when edge node j targeted. $T_{transmit}(j)$ refers to the time required to transmit data to edge node j. Let U_j be the AVs' utility of offloading application to edge node j. U_j can be expressed as:

$$U_j = E_{local} - E_{offload}(j) \tag{6}$$

The coordinator needs to select an edge node with the largest U_j from set J to provide offloading service. The process is as follows:

a). Order set J according to the ascending order of offloading energy consumption and generate a q-length list: $E_{offload}(1) \leq E_{offload}(2) \leq \ldots \leq E_{offload}(q)$.
b). Traverse the list and check whether the remaining resources of traversed server j can meet the offloading requirement $T_{limit} \leq \frac{W_{transmit}}{B_j} + \frac{W_{transmit}}{RV_j}$ && $M_{min} \leq RM_j$, M_{min} is the minimal memory share. If there exists, jump to (c. Otherwise, continue traversing. If traversal completes, jump to (d.
c). Edge node j is selected for service offloading.
d). No edge node was selected. The application will run in local mode.

B. MULTIPLE APPLICATIONS SCHEDULING

In the previous section, the strategy is made for a single application schedule. In several cases, there may exist multiple offloading requests in a very short period of time. For example, more than one offloading request might be sent out from several AVs simultaneously, especially in morning and evening rush hour. In these scenarios, the single application schedule strategy works quite bad since it does not take the resources competition between different applicants into consideration. Therefore, it leans to give preference to a specific application and the efficiency of the other offloading will be severely hurt. In this section, we will discuss how to schedule fairly in terms of utility when multiple applications coexist.

Here, we assume the Node Coordinator receives p offloading requests in a very short time. The set of applications to be offloaded is I and there are q available edge nodes and J available node servers. Due to the different network transmission states between AVs and edge nodes, the energy consumption for AVs offload application into edge server will be different too.U_{ij} is utility can be obtained if application i offloaded is to server j, which can be expressed by formula (7). When $E_{local}(i) \leq E_{offload}(i, j)$, we will get a negative u_{ij}, $u_{ij} \leq 0$, which indicates offloading application i to server j is not a profitable solution.

$$u_{ij} = E_{local}(i) - E_{offload}(i, j) \tag{7}$$

x_{ij} is the indication of whether to offload the application i to the server j. The value of x_{ij} can only be 0 or 1 since here we already set all offloaded applications are inseparable. As shown in Formula (8), the total utility U of offloading is equal to the sum of offloading utility of all applications.

$$U = \sum_{i \in I} \sum_{j \in J} x_{ij} u_{ij} \tag{8}$$

Each edge node has a limited amount of resources for computing and memory data caching. It is necessary to ensure that the amount of resources allocated to all offloaded applications will not greater than the amount of remaining resources of the server these offloading residents. The memory requirement of the application i is M_i. The remaining resource of the service j is RM_j. The slice of computing resources required for the application i to be offloaded on the j server is N_{ij}. Server's remaining slice of computing resources is RN_j. To fulfill the limitations above, it must ensure that $\sum$i∈I, $x_{ij}M_i \leq RM_j$ and $\sum$i∈I, $x_{ij}N_{ij} \leq RN_j$. And N_{ij} can be derived from equation (9).

$$N_{ij} = \lceil \frac{V_{ij}}{V_{total(j)}} \rceil * N_{total}(j) \tag{9}$$

V_{ij} represents the computing power required to offload application i into server j. $T_{limit}(i)$ represents the finish time limit of application i. $T_{transmit}(i, j)$ represents the data transmission time between application i and target offloading server j. $W_{compute}(i)$ represents the calculation workload of application i. $V_{total}(j)$ represents the total amount of computing resources of server j. $N_{total}(j)$ represents the total slices of computing resources of server j.

$$\max \ U = \sum_{i \in I} \sum_{j \in J} x_{ij} u_{ij} \tag{10}$$

$$\text{s.t. } x_{ij} \in \{0, 1\}, \quad i \in I, \ j \in J \tag{11}$$

$$\sum_{j \in J} x_{ij} \leq 1, \quad i \in I \tag{12}$$

$$\sum_{i \in I} x_{ij} N_{ij} \leq RN_j, \quad j \in J \tag{13}$$

$$\sum_{i \in I} x_{ij} M_i \leq RM_j, \quad j \in J \tag{14}$$

In summary, the model of multiple applications offloading by multiple edge servers can be described above. Formula (10) represents the optimization objective, i.e. Maximizing the entire offloading utility of all to-be-offloaded applications. Formula (11) represents that to-be-offloaded application is indivisible, either be offloaded completely or run locally. Formula (12) represents the limits that the to-be-offloaded application can only be offloaded to one specific edge node. Formula (13) is a computational resource constraint, which means the total slices of computing resource allocated by the server for offloading cannot be greater than the remaining computing slices. Formula (14) is memory constraint, which means the total memory allocated by the server for offloading should not be greater than the remaining memory volume.

Each edge server can be considered as a knapsack with distinct computing resources and memory; each application can be considered as an item with two weights. We can model this problem as an MMKP (multiple dimensional knapsack problem). In this problem, if all edge servers have sufficient memory resources, the problem will be transformed into MKP (multiple knapsack problem). MKP is the reduction from

MMKP and MKP is NPC(NP-Complete) problem, so we can conclude the problem of multiple application offloading on multiple edge servers is also an NPC problem, which cannot get an optimal solution in polynomial time.

C. MMKP SOLUTION FOR AUTONOMOUS DRIVING

In the above section, we construct a model for multiple applications multiple edge servers scheduling, which has been proved as MMKP. MMKP is essentially a combination of MDKP (multi-dimensional knapsack problem) and MKP (multiple knapsack problem), and it is also an NPC problem that no optimal solution in polynomial time [11].

In the scenario of autonomous driving, the finish time of application offloading is relatively strict. Even though the branch-and-bound method can find the global optimal solution, but its problem solving is too slow [12]. When the number of offloaded tasks reaches 30, the branch-and-bound method needs more than 100 seconds to solver. Existing heuristic algorithms cannot guarantee a feasible solution time for this problem as well. If the solution of MMKP takes too much time, offloaded applications will complete overtime and it is unacceptable especially for autonomous driving applications. Here, we propose a Greedy Algorithm (GA). Utility in formula (8) is used as the greedy selection target, and the global maximum utility value is selected at each step to try to allocate. GA cannot ensure the global optimal solution, but it ensures the solution search will finish in time and the quality of offloading service will not be affected. GA is designed as follows:

Input: p*q matrix u, p*q matrix N, array M, array RN, array RM
Output: matrix x
create p*q matrix x
create list *u_list*
for i ← 1 to p **do**
    for j ← 1 to q **do**
    if u[i, j] > 0 **do**
        u_list.push([*u*[i, j], i, j])
sort *u_list* by descending order
for elem in u_list **do**
i ← elem[1]
if app i is solved **do**
        continue
j ← elem[2]
// server j lack resources
if N[i, j] > RN[j] or M[i] > RM[j] **do**
        continue
x[i][j] ← 1
RN[j] ← RN[j] - N[i, j]
RM[j] ← RM[j] - M[i]
if all the apps are allocated **do**
        break
return x

TABLE 2. Resources configure method.

Memory	--memory parameter
CPU share	--cpu-shares parameter
CPU number	--cpus parameter
GPU	Use nvidia-docker configuration

VII. IMPLEMENTATION ON DOCKER

We have implemented a simple version of the proposed offloading framework by Python. In our design, we use Docker as a container engine. It uses a predefined Docker file to construct a Docker image. The way to make different resource isolation listed in Table 1. We use **–memory** parameter to limit the memory usage of containers. For CPU shares, one way is to use **–cpus** parameters to limit the maximum number of CPU that can be used by the container. It represents the CPU usage per unit time. The other is using **–cpu-shares** parameter to limit the CPU time ratio that container processes can access. For GPU shares, we use **nvidia-docker** to specify how many GPU block will be dispatched for a container.

In our implementation, the server builds kinds of Docker images and creates a certain number of Pre-Run containers. Each Pre-Run container is mounted in different directories for the purpose of data isolation. With such implementation, a more isolated envieroment can be created and will be more adopted for the offloading services dedicated for autonomous driving applications.

The server provides an HTTP interface to receive service offloading requests and return corresponding results. The server chooses a Pre-Run container by application category then reconfigures its CPU and memory quantum with **–docker-update**. All required execution data are stored in container mount directory, and **–docker-exec** is called to perform the actual calculation, and the result is sent back when the calculation is completed.

VIII. EXPERIMENTS

A. FEASIBILITY OF OFFLOADING FRAMEWORK

In this subsection, we will discuss the feasibility of the offloading framework in terms of data transmission, container initialization and workload computing. Their sum-up must be less than the finish time limit of offloaded application. The details of edge servers we used are listed in Table 3.

1) DATA TRANSMISSION

With the development of 5G technology, the data transmission rate will reach more than 10Gbps in the future. Thus, network delay can be limited in 1ms and data transmission of MB data will only take a few milliseconds or less [13]. Since we are using containers as offloading runtime, there already packed a large volume of code/setup data in image and data transmitted will be further reduced. Network delay is no longer the bottleneck for computing offloading.

TABLE 3. Edge server information.

Edge server	
CPU type	Intel(R) i7-6700
Physical cores number	4
Threads number	8
Memory	16G
Storage	1TB
Operating system	Ubuntu18.04
Docker version	18.03.0-ce

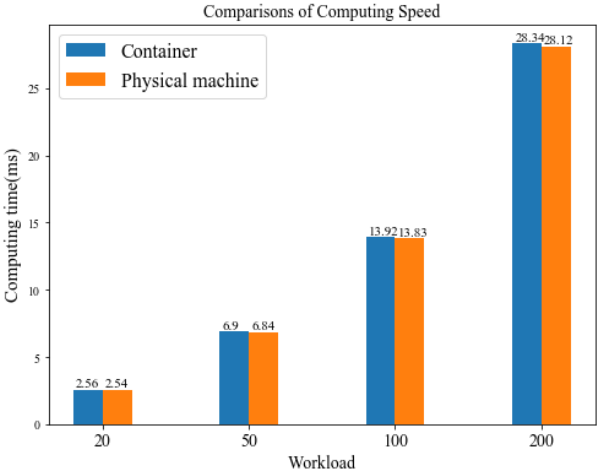

FIGURE 8. Comparisons of computing speed.

TABLE 4. Longest running time by CPU slices.

Workload	50	100	200
cpus	50.6ms	53.2ms	58.2ms
cpu-shares	45.5ms	46.1ms	50.4ms

2) COMPUTING

To prove the feasibility of using Docker container, we compare the performance of tool ***Sysbench*** executed in Docker container mode and physical machine mode. Sysbench is a cross-platform multi-threaded benchmark testing tool. It can test the computing power of servers under different workloads and different thread numbers. As shown in Figure 8, comparisons are made when 8 threads are used, and the results data is an average of 500 times run. It can be seen that containers can provide computing power very close to that of physical machines. The overhead container introduced is tolerable for offloading.

Here, we also give the execution time data, one by using the cpu-shares parameter and the other using cpus parameter. Three offloaded applications are with the workload of 50, 100 and 200. We recorded the longest running time of each application in 20 repeated experiments for both restriction modes, as shown in Table 4. It can be seen that all the applications can return within 60ms and both cpu slices methods can effectively guarantee the execution speed of offloaded application. However, using the cpu-shares parameter makes a shorter finish time due to its flexibility.

TABLE 5. Resource usage in pre-run.

	Number of Containers				
	200	400	600	800	1000
CPU (%)	0.16%	0.23%	0.35%	0.48%	0.57%
Memory(GB)	0.31	0.58	0.83	1.18	1.4

TABLE 6. Offloading time comparison.

	Data Encryption	Path Planning	Object Recognition
Data Volume (KB)	100-200	<10	500-800
Transmission (ms)	34	9	167
Calculation	29	58	238
Offloading	63	67	1077

TABLE 7. Rules of data generation.

Description	Range	Unit
E_{local}	5∼15	Unit energy consumption
$W_{transmit}$	200∼2000	KB
B	1000∼5000	Mbps
$W_{compute}$	100∼1000	Unit of calculation
T_{limit}	80∼150	ms
$P_{transform}$	0.1	Unit energy consumption/ms
M	1∼4	GB
V_{total}	20∼40	Unit of calculation/ms
N_{total}	10000	-
RN	2∼Ntotal	-
RM	2∼20	GB

3) CONTAINER INITIALIZATION

In order to reduce the delay of container initialization, the Container Manager Pre-Runs some containers with very few resources and simply adjust its resources quote when requests arrive.

Here, we compared Pre-Run strategy with Full-load strategies. Full load is to create and start a new Docker container from images each time requests arrive. In our test, the Full-load strategy takes an average of 1.2 seconds to initialize a container. It is far more exceeds the finish time limit of most autonomous driving applications. Pre-Run strategy only takes about 10 ms to initialize the container.

Table 5 shows the CPU and memory usage as the number of Pre-Run containers increases. It can be seen that a single Pre-Run container occupies very little resources. When the number of Pre-Run containers reaches 1000, the CPU occupancy is still less than 1%, the memory occupancy exceeds 1.4G. Maintaining hundreds of Pre-Run containers is not too stressful for the server.

B. EFFICIENCY OF OFFLOADING FRAMEWORK

We have already implemented a simple version Offloading framework by Python. In this section, we test its

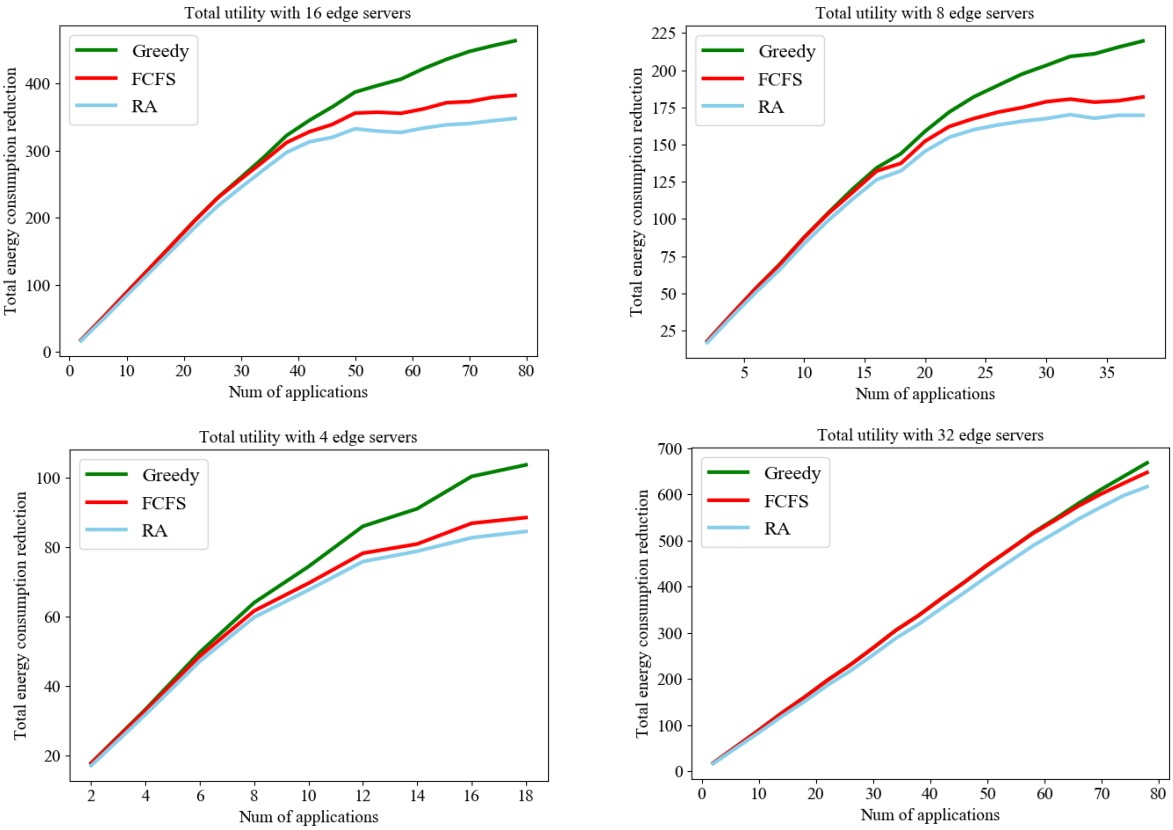

FIGURE 9. Total energy saving made by three scheduling.

offloading efficiency by using three common applications in autonomous drivings: data encryption, path planning and object recognition. Data encryption is implemented by AES, which is an encryption standard widely used in the world [14]. Path planning is implemented by A-star algorithm, which is a heuristic path planning algorithm with fast solving speed [15]. Object Recognition is implemented by YOLOv3 [16] and it uses COCO, an image data set provided by Microsoft team [17].

In our implementation, the server builds corresponding Docker images and creates Pre-Run containers for the above three applications. Each Pre-Run container is mounted in different directories for the purpose of data isolation. The server provides an HTTP interface to receive service offloading requests and return corresponding results. Another machine in LAN is used as a client to send offloading requests to the server. The transmission rate between client and server is about 100 Mbps. Through 50 groups of repeated experiments, the average offloading time is shown in table 5. The data transmission time includes delay for data upload and download, and the calculation time includes the container initialization time and actual calculation time.

For data encryption and path planning, its offloading efficiency is very high. It can complete calculation and return results in 100ms. This response time is short enough to meet the efficiency requirements of computing offloading in autonomous driving. But in object recognition, the offloading is inefficient. That because the computing unit in the server is not powerful enough to run matrix and convolution operations. If we can well tune the hardware setup in the server, Yolov3 can be completed in about 20 ms if GPU introduced [16].

C. OFFLOADING SCHEDULE FOR MULTIPLE EDGE NODE

Here, we use First Come First Server (FCFS) and Random Algorithms (RA) to compare with the Greedy Algorithms (GA). We also use branch and bound (B&B) method [12] to find out the optimal solution in some combination. Some missing of B&B data is due to the long time for solution search in some large-scale scenarios and not suitable for performance comparison.

At present, there is no data set for autonomous driving computing offloading. Therefore, under the background of 5G transmission, we manually generate data for comparison and the rules for data generation are shown in Table 6. Utility and computing resource requirements of applications allocated to multiple edge servers are calculated based on randomly generated data. Figure 9 shows the total energy saving made by three scheduling strategies when there deployed 4, 8, 16 and 32 edge servers. In order to ensure the stability

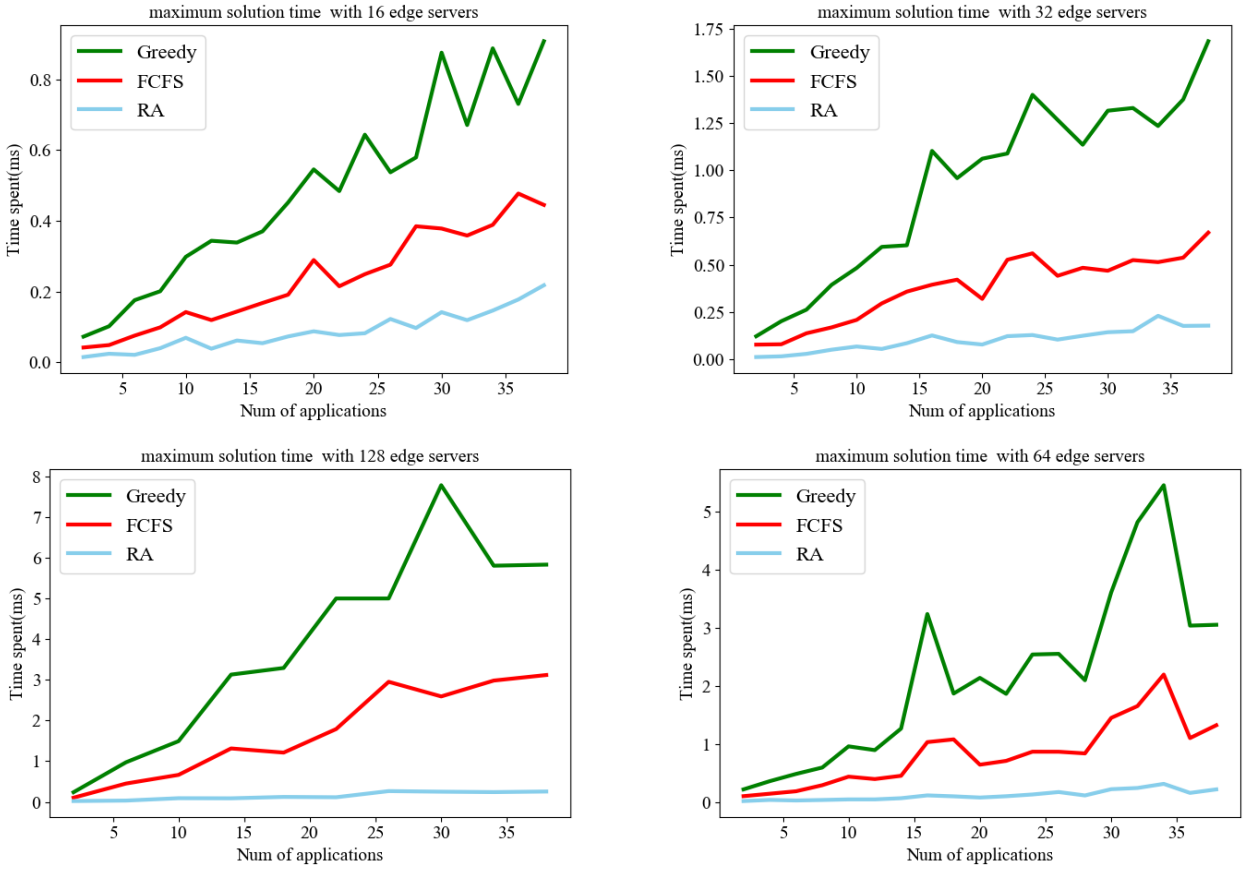

FIGURE 10. Solution time of three scheduling.

of those results, 200 experiments were repeated and average data for each combination are obtained. Shown in Figure 8, it can be seen that when the number of offloading requests is relatively small, GA is similar to the FCFS in performance, very close to the optimal results B&B gives. This is because when there is ample resource each request can be offloaded to its most ideal edge server with a great chance. However, as the number of offloading requests increases and the resources of edge servers reduces, GA still can make better scheduling decisions due to its consideration of greediness. FCFS and RA underperform for its blindness.

Figure 10 shows the curve of solution time as the number of offloading requests increases. 200 experiments were carried out on different combinations of the number of applications and edge servers. The maximum finish time of 200 experiments was selected for display. When the scale of the problem is medium, such as 32 applications and 20 edge servers, GA can return results within 1ms or millisecond level. But when the scale of the problem is too large, such as 40 applications and 128 edge servers, the solution time is close to 10 ms. 10 ms scheduling delay indeed can give a neglect-able impact on offloading. When the number of edge servers and offloading requests are extremely large, GA is no more an option for offloading.

IX. RELATED WORK

Autonomous driving research is still in its infancy. There exist only a few works on discussing how AVs cooperate with Edge nodes.

OpenVDAP[18]. is a real-world edge computing system that 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. Reference [19] proposed an infrastructure-based vehicle control system that shares internal states between edge and cloud servers, dynamically allocates computational resources and switches necessary computation on collected sensors according to network conditions in order to achieve safe driving. Reference [20] proposed a two-level edge computing architecture for automated driving services in order to make full use of the intelligence at the wireless edge for coordinated content delivery.

Edge Computing moves the focus from heavy-weight data center clouds to more lightweight virtualized resources, distributed to bring services to the users. Researchers have identified lightweight virtualization and deploy container technology to meet the needs. A container is essentially a packaged self-contained, ready-to-deploy set of parts of

applications, that might be in the form of binaries and libraries to run the applications [21]. Container tools like Docker are frameworks built around container engines [22]. Docker has started to develop its own orchestration solution and Kubernetes [23] is another relevant project, but a more comprehensive solution that would address the orchestration of complex application stacks.

Code and computation offloading techniques formulated for cloud technologies are now commonly introduced in edge computing paradigms [24]. CloneCloud [25] and ThinkAir [26] are two examples application and system virtualization-based code offloading frameworks. Researchers [27] also proposed an architecture-aware compiled code offloading framework by offloading native code. However, such framework is only used for smartphone applications. It made use of LLVM compiler to generate intermediate code to support kinds of mobile applications running on heterogeneous mobile and sever ends. All offloading are compiled by LLVM back-end compilers into intermediate code binaries for different native hardware at runtime. Researchers [28] present another framework for pre-compiled vector instruction translation and offloading in heterogeneous computing architectures. It can get the chance of avoid the execution overhead of compiled code offloading. It maps and translates ARM vector intrinsic to x86 vector intrinsic such that an application programmed for ARM architecture can be executed on the x86 architecture without any modification. Such mapping leads to considerate execution efficiency.

We can witness the progress of autonomous driving, edge computing and computing offloading in their own fields. We also can validate the truth that applications on autonomous driving vehicles need a isolated meanwhile secured environment for real-time workload offloading. Edge computing shows up as the best solutions. That is the reason why we look at the insides of container-based offloading by edge for autonomous driving. That is also the big difference we can tell from those previous works.

X. CONCLUSION

In order to meet the requirements of efficiency, security and privacy of autonomous driving., A container-based edge offloading framework is proposed in this paper. Using this framework, AVs can offload applications to edge serves with high isolation and without performance harms. The Edge Offloading Middleware uses a container Pre-Run strategy for fast container bootup and dynamic resource management in category of memory and computing source. Offloading Scheduler describes the multiple applications multiple edge nodes schedule as an MMKP problem and proposed a greedy algorithm for polynomial-time solution search. Experiments show that the proposed offloading framework can support millisecond-level computing offloading while guaranteeing the safety and privacy of AVs. The greedy-based strategy has less scheduling overhead and can effectively increase the total utility of edge servers.

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[2] Y. Wang, S. Liu, X. Wu, and W. Shi, "CAVBench: A benchmark suite for connected and autonomous vehicles," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Oct. 2018, pp. 30–42.

[3] J. Van Brummelen, M. O'Brien, D. Gruyer, and H. Najjaran, "Autonomous vehicle perception: The technology of today and tomorrow," *Transp. Res. C, Emerg. Technol.*, vol. 89, pp. 384–406, Apr. 2018.

[4] M. Kamoun, W. Labidi, and M. Sarkiss, "Joint resource allocation and offloading strategies in cloud enabled cellular networks," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2015, pp. 5529–5534.

[5] W. Labidi, M. Sarkiss, and M. Kamoun, "Energy-optimal resource scheduling and computation offloading in small cell networks," in *Proc. 22nd Int. Conf. Telecommun. (ICT)*, Apr. 2015, pp. 313–318.

[6] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang, "Energy-efficient offloading for mobile edge computing in 5G heterogeneous networks," *IEEE Access*, vol. 4, pp. 5896–5907, 2016.

[7] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, 2010, pp. 49–62.

[8] Z. Pang, L. Sun, Z. Wang, E. Tian, and S. Yang, "A survey of cloudlet based mobile computing," in *Proc. Int. Conf. Cloud Comput. Big Data (CCBD)*, Nov. 2015, pp. 268–275.

[9] C. Anderson, "Docker [software engineering]," *IEEE Softw.*, vol. 32, no. 3, p. 102-c3, May 2015.

[10] R. Rosen. *Resource Management: Linux Kernel Namespaces and Cgroups*. Accessed: May 2013. [Online]. Available: http://www.haifux.org/lectures/299/netLec7.pdf

[11] H. Kellerer, U. Pferschy, and D. Pisinger, "Multidimensional knapsack problems," in *Knapsack Problems*. Berlin, Germany: Springer, 2004, pp. 235–283.

[12] J. Wang, T. Liu, K. Liu, B. Kim, J. Xie, and Z. Han, "Computation offloading over fog and cloud using multi-dimensional multiple knapsack problem," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2018, pp. 1–7.

[13] G. A. Akpakwu, B. J. Silva, G. P. Hancke, and A. M. Abu-Mahfouz, "A survey on 5G networks for the Internet of Things: Communication technologies and challenges," *IEEE Access*, vol. 6, pp. 3619–3647, 2018.

[14] G. Singh and S. Supriya, "A study of encryption algorithms (RSA, DES, 3DES and AES) for information security," *Int. J. Control Automat.*, vol. 67, no. 19, pp. 33–38, Apr. 2013.

[15] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, and L. Jurišica, "Path planning with modified a star algorithm for a mobile robot," *Procedia Eng.*, vol. 96, pp. 59–69, Jan. 2014.

[16] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767

[17] T. Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, Sep. 2014, pp. 740–755.

[18] 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.

[19] K. Sasaki, N. Suzuki, S. Makido, and A. Nakao, "Vehicle control system coordinated between cloud and mobile edge computing," in *Proc. 55th Annu. Conf. Soc. Instrum. Control Eng. Jpn. (SICE)*, Sep. 2016, pp. 1122–1127.

[20] Q. Yuan, H. Zhou, J. Li, Z. Liu, F. Yang, and X. S. Shen, "Toward efficient content delivery for automated driving services: An edge computing solution," *IEEE Netw.*, vol. 32, no. 1, pp. 80–86, Jan. 2018.

[21] S. Soltesz, H. Po'tzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, 2007.

[22] J. Turnbull. (2014). *The Docker Book*. [Online]. Available: http://www.dockerbook.com/

[23] *Production-Grade Container Orchestration—Kubernetes*. Accessed: Jul. 4, 2019. [Online]. Available: https://kubernetes.io/

[24] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 12, pp. 3590–3605, Dec. 2016.
[25] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.
[26] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 945–953.
[27] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim, "Architecture-aware automatic computation offload for native applications," in *Proc. 48th Int. Symp. Microarchitecture (MICRO)*, 2015, pp. 521–532.
[28] J. Shuja, S. Mustafa, R. W. Ahmad, S. A. Madani, A. Gani, and M. K. Khan, "Analysis of vector code offloading framework in heterogeneous cloud and edge architectures," *IEEE Access*, vol. 5, pp. 24542–24554, 2017.

JIE TANG (Member, IEEE) received the B.E. degree from the University of Defense Technology, in 2006, and the Ph.D. degree from the Beijing Institute of Technology, in 2012, both in computer science. She was a Visiting Researcher with the Embedded Systems Center, University of California at Irvine, Irvine, CA, USA, and a Research Scientist with the Intel China Runtime Technology Laboratory. 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, cloud, and run-time systems. She is also a Founding Member and the Secretary of the IEEE Computer Society Special Technical Community on Autonomous Driving Technologies.

RAO YU received the master's degree in computer science from the South China University of Technology, Guangzhou, China. He also received two Chinese patents in autonomous driving and edge computing. He is currently a System Engineer with Pony AI., Ltd. His research interests include big data, cloud computing, and system software stack.

SHAOSHAN LIU (Senior Member, IEEE) received the Ph.D. degree in computer engineering from the University of California at Irvine, Irvine, CA, USA. 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, as well as the Baidu Autonomous Driving Unit, in charge of system integration of autonomous driving systems. His researches focus on computer architecture, deep learning infrastructure, robotics, and autonomous driving. He has published more than 40 high-quality research articles. He holds more than 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 co-founded the IEEE Special Technical Community on Autonomous Driving Technologies and serves as the Founding Vice President. He is a Senior Member of an ACM Distinguished Speak and the IEEE Computer Society Distinguished Speaker.

JEAN-LUC GAUDIOT (Life Fellow, IEEE) received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electronique et Electrotechnique, Paris, France, in 1976, and the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles, Los Angeles, CA, USA, in 1977 and 1982, respectively. He was the Chair of the Department, from 2003 to 2009. During his tenure, the department underwent significant changes. These include the hiring of 12 new faculty members (three senior professors) and the remarkable rise in the U.S. News and World Report®rankings of the Computer Engineering program from 42 to 28 (46 to 36 for the Electrical Engineering program). He is currently a Professor of the Electrical Engineering and Computer Science Department, University of California at Irvine, Irvine, CA, USA. In 1999, he became a Fellow of the IEEE, For Contributions to the Programmability and Reliability of Dataflow Architectures. He was elevated to the rank of AAAS Fellow in 2007, For Distinguished Contributions to the Design and Analysis of Highly Efficient Multiprocessor and Memory System Architectures.