LAVEA: Latency-aware Video Analytics on Edge Computing Platform

Shanhe Yi
College of William and Mary
syi@cs.wm.com

Zijiang Hao
College of William and Mary
hebo@cs.wm.com

Qingyang Zhang
Wayne State University
Anhui University, China
qyzhang@wayne.com

Quan Zhang
Wayne State University
quan.zhang@wayne.com

Weisong Shi
Wayne State University
weisong@wayne.com

Qun Li
College of William and Mary
liqun@cs.wm.com

ABSTRACT

Along the trend pushing computation from the network core to the edge where the most of data are generated, edge computing has shown its potential in reducing response time, lowering bandwidth usage, improving energy efficiency and so on. At the same time, low-latency video analytics is becoming more and more important for applications in public safety, counter-terrorism, self-driving cars, VR/AR, etc. As those tasks are either computation intensive or bandwidth hungry, edge computing fits in well here with its ability to flexibly utilize computation and bandwidth from and between each layer. In this paper, we present LAVEA, a system built on top of an edge computing platform, which offloads computation between clients and edge nodes, collaborates nearby edge nodes, to provide low-latency video analytics at places closer to the users. We have utilized an edge-first design and formulated an optimization problem for offloading task selection and prioritized offloading requests received at the edge node to minimize the response time. In case of a saturating workload on the front edge node, we have designed and compared various task placement schemes that are tailed for inter-edge collaboration. We have implemented and evaluated our system. Our results reveal that the client-edge configuration has a speedup ranging from 1.3x to 4x (1.2x to 1.7x) against running in local (client-cloud configuration). The proposed shortest scheduling latency first scheme outputs the best overall task placement performance for inter-edge collaboration.

CCS CONCEPTS

•Networks →Cloud computing; •Computing methodologies →Object recognition; •Software and its engineering →Publish-subscribe / event-based architectures;

KEYWORDS

computation offloading, edge computing

1 INTRODUCTION

Edge computing (also termed fog computing [4], cloudlets [28], MEC [24], etc.) has brought us better opportunities to achieve the ultimate goal of a world with pervasive computation [28]. This new computing paradigm is proposed to overcome the inherent problems of cloud computing and provide supports to the emerging Internet of Things (IoT) [14, 33, 37]. Typically, when using the cloud, all the data generated shall be uploaded to the cloud data center before processing. However, considering nowadays a huge amount of data is being intensively generated at the edge of the network, transferring the data at such scale to the distant cloud for processing will add burdens to the network and lead to unacceptable response time, especially for latency-sensitive applications. More specifically, as for edge computing, we aim to provide *edge analytics*, which focuses on data analytics at or near the places (the network edge) where data is generated [30]. Data analytics done at the edge of the network has many benefits such as gathering more client side information, cutting short the response time, saving network bandwidth, lowering the peak workload to the cloud, and so on.

Among many edge analytic applications, in this paper, we focus on delivering video analytics at the edge. The ability to provide low latency video analytics is critical for applications in the fields of public safety, counter-terrorism, self-driving cars, VR/AR, etc [32]. In video edge analytic applications, we consider typical client devices such as mobile phones, body-worn cameras or dash cameras mounted on vehicles, web cameras at toll stations or highway checkpoints, security cameras in public places, or even video captured by UAVs [35]. For example, in "Amber Alert", our system can automate and speedup the searching of objects of interest by vehicle recognition, vehicle license plate recognition and face recognition utilizing various web cameras deployed at highway entrances, or dash cameras or cameras of smartphones mounted on cars.

Simply uploading all the captured video or redirecting video feeds to the cloud cannot meet the requirement of latency-sensitive applications, because the computer vision algorithms involved in

object tracking, object detection, object recognition, face and optical character recognition (OCR) are either computation intensive or bandwidth hungry. In addressing these problems, mobile cloud computing (MCC) is proposed to run heavy tasks on resource rich cloud node to improve response time or energy cost. This technique utilizes both the mobile and cloud for computation. An appropriate partition of tasks that makes trade-off between local and remote execution can speed up the computation and preserve mobile energy at the same time [7, 13, 15, 21, 31]. However, there are still concerns of cloud about the limited bandwidth, the unpredictable latency, and the abrupt service outage. Existing work has explored adding intermediate servers (cloudlets) between mobile client and the cloud. Cloudlet is an early implementation of the cloud-like edge computing platform with virtual machine (VM) techniques. The edge computing platform in our work has a different design on top of lightweight OS-level virtualization which is modular – easy to deploy, manage, and scale. Compared to VM, the OS-level virtualization provides resource isolation in a much lower cost. The adoption of container technique leads to a server-less platform where the end user can deploy and enable edge computing platform on heterogeneous devices with minimal efforts. The user programs (scripts or executable binaries) will be encapsulated in containers, which provide resource isolation, self-contained packaging, any-where deploy, and easy-to-configure clustering. The end user only needs to register events of interest and provide corresponding handler functions to our system, which automatically handle the events behind the scene.

In this paper, we are considering a 3-tier mobile-edge-cloud deployment and we put most of our efforts into the mobile-edge side and inter-edge side design. To demonstrate the effectiveness of our edge computing platform, we have built the **L**atency-**A**ware **V**ideo **E**dge **A**nalytics (LAVEA) system. We divide the response time minimization problem into three sub-problems. First, we select client tasks that benefit from being offloaded to edge node in reducing time cost. We formulated this problem as a mathematical optimization problem to choose offloading tasks and allocate bandwidth among clients. Unlike existing work in mobile cloud computing, we cannot make the assumption that edge node is as powerful as cloud node which can process all the tasks instantly. Therefore, we consider the increasing resource contention and response time when more and more tasks are running on edge node by adding latency constraints to the optimization problem. Second, upon receiving offloading task requests at each epoch, the edge node runs these tasks in an order to minimize the makespan. However, the offloaded tasks cannot be started when the corresponding inputs are not ready. To address this problem, we employed a classic two-stage job shop model and adapted Johnson's rule with topological ordering constraint in a heuristic to prioritize the tasks. Last, we enable inter-edge collaboration leveraging nearby edge nodes to reduce the overall task completion time. We have investigated several task placement schemes that are tailored for inter-edge collaboration. The findings provided us insights that lead to an efficient prediction-based task placement scheme.

In summary, we make the following contributions:

- We have designed an edge computing platform based on a server-less architecture, which is able to provide flexible computation offloading to nearby clients to speed up computation-intensive and delay-sensitive applications. Our implementation is lightweight-virtualized, event-based, modular, and easy to deploy and manage on either edge or cloud nodes.
- We have formulated an optimization problem for offloading task selection and prioritized offloading requests to minimize the response time. The task selection problem co-optimizes the offloading decision and bandwidth allocation , and is constrained by the latency requirement, which can be tuned to adapt to the workload on edge node for offloading. The task prioritizing is modeled as a two-stage job shop problem and a heuristic is proposed with the topological ordering constraint.
- We have evaluated several task placement schemes for inter-edge collaboration and proposed a predication-based method which efficiently estimates the response time.

2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce the background of edge computing and relevant techniques, present our observations from preliminary measurements, and discuss the scenarios that motivate us.
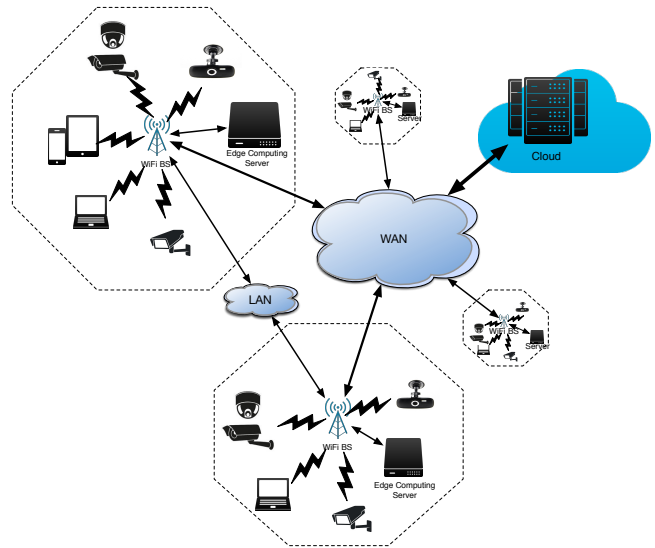

Figure 1: An overview of edge computing environment

2.1 Edge Computing Network

In the paper, we consider an edge computing network as shown in Figure 1, in which we focus on two types of nodes, the client node (in this paper, we call it client for short) and the edge server node (in this paper, we call it edge, edge node, or edge server for short). We assume that clients are one-hop away from edge server via wire or wireless links. When a client connects to the edge node, we implicitly indicate that the client will first connect to the

correspond access points (APs) using cable or wireless and then utilize the services provided by the co-located edge node. In a sparse edge node deployment, a client will only connect to one of the available edge nodes nearby at certain location. While in a dense deployment, a client may have multiple choices on selecting the multiple edge servers for services. Implicitly, we assume that there is a remote cloud node which can be reached via the wide area network (WAN).

To understand the factors that impact the feasibility of realizing practical edge computing systems, we have performed several preliminary measurements on existing network and shown the results in Fig. 2 and Fig. 3. In these experiments, we measured the latency and bandwidth of combinations between clients nodes with different network interfaces connecting to edge (or cloud) nodes. Based on the measurements of bandwidth, all clients have benefits in utilizing a wire-connected or advanced-wireless (802.11ac 5Ghz) edge computing node. In terms of latency, wire-connected edge nodes is the best while the 5Ghz wireless edge computing nodes have larger means and variances in latency compared to the cloud node in the closest region due to the intrinsic nature of wireless channels. Therefore, in this paper, we pragmatically assume that edge nodes are connected to APs via cables to deliver services with better latency and bandwidth than the cloud. Therefore, in such a setup, the cloud node can be considered as a backup computing node, which will be utilized only when the edge node is saturated and experiences a long response time.
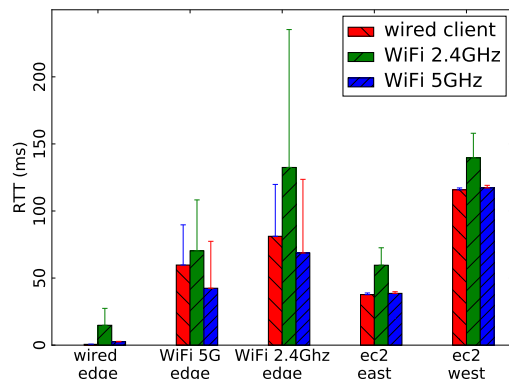

Figure 2: Round trip time between client and edge/cloud.

2.2 Serverless Architecture

Serverless architecture or Function as a Service (FaaS), such as AWS Lambda, Google Cloud Function, Azure Functions, is an agile solution for developer to build cloud computing services without the heave lifting of managing cloud instances. To use AWS Lambda as an example, AWS Lambda is a event-based, micro-service framework, in which a user-supplied Lambda function as the application logic will be executed in response to corresponding event. The AWS cloud will take care of the provisioning and resource management for running Lambda functions. At the first time a Lambda function is created, a container will be built and launched based on the configurations provided. Each container will also be provided a small
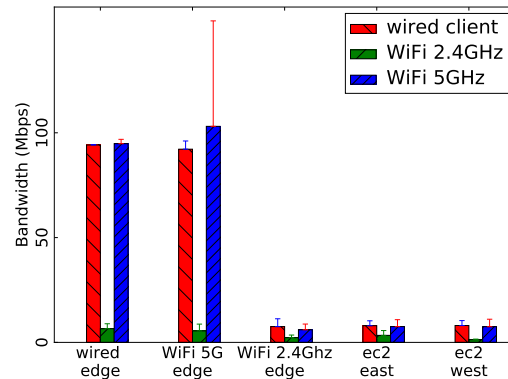

Figure 3: Bandwidth between client and edge/cloud.

disk space as transient cache during multiple invocations. AWS has its own way to run Lambda functions with either reusing an existing container or creating a new one. Recently, there is AWS Lambda@Edge [1], that allows using serverless functions at the AWS edge location in response to CDN event to apply moderate computations. We strongly advocate the adoption of serverless architecture at the edge computing layers, as serverless architecture naturally solves two important problems for edge computing: 1) serverless programming model greatly reduces the burden on users or developers in developing, deploying and managing edge applications, as there is no need to understand the complex underlying procedures to run the applications or heavy lifting of distributed system management; 2) the functions are flexible to run on either edge or cloud, which lowers the barrier of edge-cloud inter-operatability and federation. Recent works have shown the potentials of such architecture in low latency video processing tasks [11] and distributed computing tasks [20], and there have been research efforts of incorporating serverless architecture in edge computing [8].

2.3 Video Edge Analytics for Public Safety

Video surveillance is of great importance for public safety. Besides the "Amber Alert" example, there are many other applications in this field. For example, secure cameras deployed at public places (e.g. the airport) can quickly spot unattended bags [42], police with body-worn cameras can identify suspects and suspicious vehicles during approaching, and so on. Because those scenarios are urgent and critical, the applications need to provide the quickest responses with best efforts. However, most tasks in video analytics are undoubtedly computationally intensive [26]. While running on resource constrained mobile clients or IoT devices directly, the latency in computation, battery drain (if battery-powered), or even heat dissipation will eventually ruin the user experience, failing to achieve the performance goals of the applications. If running on cloud nodes, transferring large volume of multimedia data will incur unacceptable transmission latency and additional bandwidth cost. Being proposed as a dedicated solution, the deployment of edge computing platform enables the quickest responses to these video analytics tasks which require both low latency and high bandwidth.

In this paper, we mainly focus on building video edge analytics platform and we demonstrate our platform using the application

of Automated License Plate Recognition (ALPR). Even though we integrate specific application, our edge platform is a general design and can be extended for other application with little modifications. An ALPR system usually has four stages: 1) image acquisition, 2) license plate extraction, 3) license plate analysis, and 4) character recognition [2, 10]. Each of the stages involves various computer vision, pattern recognition, and machine learning algorithms. Migrating the execution of some algorithms to powerful edge/cloud node can significantly reduce the response time [34]. However, offloaded tasks require intermediate data, application state variables, and corresponding configurations to be uploaded. Some of the algorithms produce large amount of intermediate data will add delay to the whole processing time if offloaded to remote cloud. We believe that a carefully designed edge computing platform will assist ALPR system to expand on more resource-constrained devices at more locations and provide better response time at the same time.

3 LAVEA SYSTEM DESIGN

In this section, we present our system design. First, we will discuss our design goals. Then, we will overview our system design and introduce several important edge computing services.

3.1 Design Goals

- **Latency**. The ability to provide low latency services is recognized as one of the essential requirements of edge computing system design.
- **Flexibility**. Edge computing system should be able to flexibility utilize the hierarchical resources from client nodes, nearby edge nodes and remote cloud nodes.
- **Edge-first**. By edge-first, we mean that the edge computing platform is the first choice of our computation offloading target.

3.2 System Overview

LAVEA is intrinsically an edge computing platform, which supports low-latency video processing. The main components are edge computing node and edge client. Whenever a client is running tasks and the nearby edge computing node is available, a task can be decided to run either locally or remotely. We present the architecture of our edge computing platform in Figure 4.

3.2.1 Edge Computing Node. In LAVEA, the edge computing node provides edge computing services to the mobile devices nearby. The edge computing node attached to the same access point or base station as clients is called the edge-front. By deploying edge computing node with access point or base station, we ensure that edge computing service can be as ubiquitous as Internet access. Multiple edge computing nodes can collaborate and the edge-front will always serve as the master and be in charge of the coordination with other edge nodes and cloud nodes. As shown in Figure 4, we use the light-weight virtualization technique to provide resource allocation and isolation to different clients. Any client can submit tasks to the platform via client APIs. The platform will be responsible for shaping workload, managing queue priorities, and scheduling tasks. Those functions are implemented via internal APIs provided by multiple micro-services such as queueing service, scheduling

service, data store service, etc. We will introduce several important services later in this section.

3.2.2 Edge Client. Since most edge clients are either resource constrained devices or need to accommodate requests from a large number of clients, an edge client usually runs lightweight data processing tasks locally and offloads heavy tasks to edge computing node nearby. In LAVEA, the edge client has a thin client design, to make sure all the clients can run it without introducing too much overhead. For low-end devices, there is only one worker to make progress on the assigned job. The most important part of client node design is the profiler and the offloading controller, acting as participants in the corresponding profiler service and offloading service. With profiler and offloading controller, a client can provide offloading information to the edge-front node and fulfill offloading decision received.

3.3 Edge Computing Services

3.3.1 Profiler Service. Similar to [7, 21, 31], our system uses a profiler to collect task performance information on various devices, since it is difficult to derive an analytic model to accurately capture the behavior of the whole system. However, we have found that the execution of video process tasks is relatively stable (when input and algorithmic configurations are given) and a profiler can be used to collect relevant metrics. Therefore, we add a profiling phase to the deployment of every new type of client devices and edge devices. The profiler will execute instrumented tasks multiple times with different inputs and configurations on the device and measure metrics including but not limited to execution time, input/output data size, etc. The time-stamped logs will be gathered to build the task execution graph for specific tasks, inputs, configurations, and devices. The profiler service will collect those information, on which LAVEA relies for offloading decisions.

3.3.2 Monitoring Service. Unlike profiler service which gathers pre-run-time execution information on pre-defined inputs and configurations, the monitoring service is used to continuously monitoring and collect run-time information such as the network, system load, from not only the clients but also nearby edge nodes. Monitoring the network between client and edge-front is necessary since most edge clients are connected to edge-front server via wireless link. The condition of wireless link is changing from time to time. Therefore, we need to constantly monitor the wireless link, to estimate the bandwidth and the latency. Monitoring system load on the edge client provides flexible workload shaping and task offloading from client to the edge. This information is also broadcasted among nearby edge nodes. When an edge-front node is saturated or unstable, some tasks will be assigned to nearby edge nodes according to the system load, the network bandwidth, and network delay between edge nodes as long as there is still benefit instead of assigning tasks to cloud node.

3.3.3 Offloading Service. The offloading controller will track tasks running locally at the client, exchange information with the offloading service running on the edge-front server. The variables gather in profiler and monitoring services will be used to as inputs to the offloading decision problem which is formulated as an optimization problem to minimize the response time. Every time
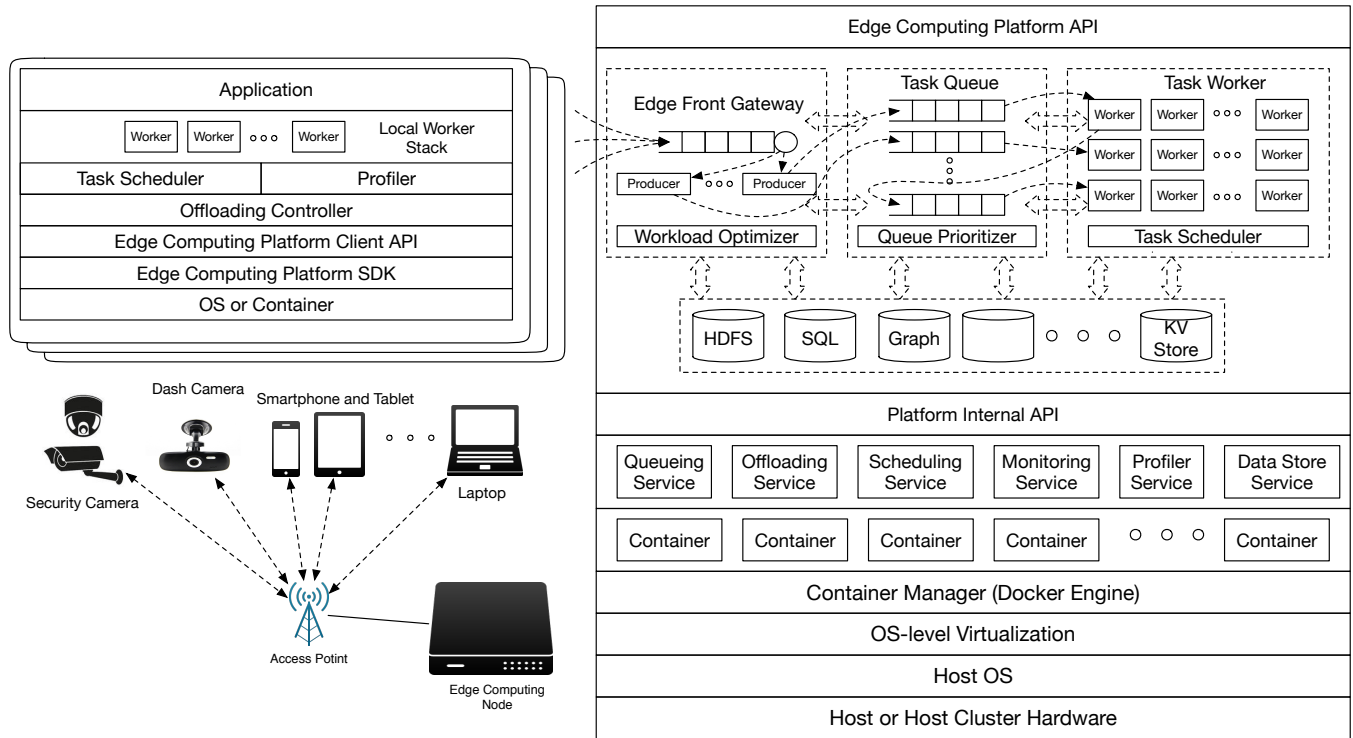
Figure 4: The architecture of edge computing platform

when a new client registers itself to the offloading services, after the edge-front node collects enough prerequisite information and statistics, the optimization problem is solved again and the updated offloading decisions will be sent to all the clients. Periodically, the offloading service also solves the optimization problem, and update offloading decisions with its clients.

4 EDGE-FRONT OFFLOADING

In this section, we describe how we select tasks of a job to run remotely on the edge server in order to minimize the response time.

We consider selecting tasks to run on the edge as a computation offloading problem. Traditional offloading problems are offloading schemes between clients and remote powerful cloud servers. In literature [7, 21, 31], those system models usually assume the task will be instantly finished remotely once the task is offloaded to the server. However, we argue that this assumption will not hold in edge computing environment as we need to consider the various delays at the server side especially when lots of clients are sending offloading requests. We call it *edge-front computation offloading* from the perspective of client:

- Tasks will be only by offloaded from client to the nearest edge node, which we call the edge front.
- The underlying scheduling and processing is agnostic to clients.
- When a mobile node is disconnected from any edge node or even cloud node, it will resort to local execution of all the tasks.

We assume that edge node is wire-connected to the access point, which indicates that the out-going traffic can go through edge node with no additional cost. The only difference between offloading task to edge node and cloud node, is that the task running on edge node may experience resource contention and scheduling delay while we assume task offloaded to cloud node will get enough resource and be scheduled to run immediately. In light work load case, if there is any response time reduction when this task is offloaded to cloud, then we know that there is definitely benefit when this task is offloaded to the edge. The reasons are 1) an edge server is as responsive as the server in the cloud data center, 2) running a task on edge server experiences shorter data transmission delay as client-edge link has much larger bandwidth than edge-cloud link which is usually limited and imbalanced by the Internet service providers (ISPs). Therefore, in this section, we focus on the task offloading only between client and edge server, and we will discuss integrating nearby edge nodes for the heavy work load scenario in the next section.

4.1 Task Offloading System Model and Problem Formulation

In the paper, we call a running instance of the application as a job, which is usually a set of tasks. The job is the unit of work that user submits to our system while the task is the unit of work for our system to make scheduling and optimization decisions. These tasks generated from each application will be queued and processed either locally or remotely. By remotely, we mean run the task on an edge node. For simplicity, we consider that all clients are running

instances of applications processing same kind of jobs, which is typically the case in our edge application scenario. However, our system can easily extend to heterogeneous applications running on each client device.

We choose to work on the granularity of task since those tasks are modularized and can be flexibly combined to either achieve a speed processing or form a workflow with high accuracy. In our ALPR application, each task is usually a common algorithm or library of computer vision. For example, We have analyzed an open source ALPR project called OpenALPR [22] and illustrate its task graph in Fig. 5.
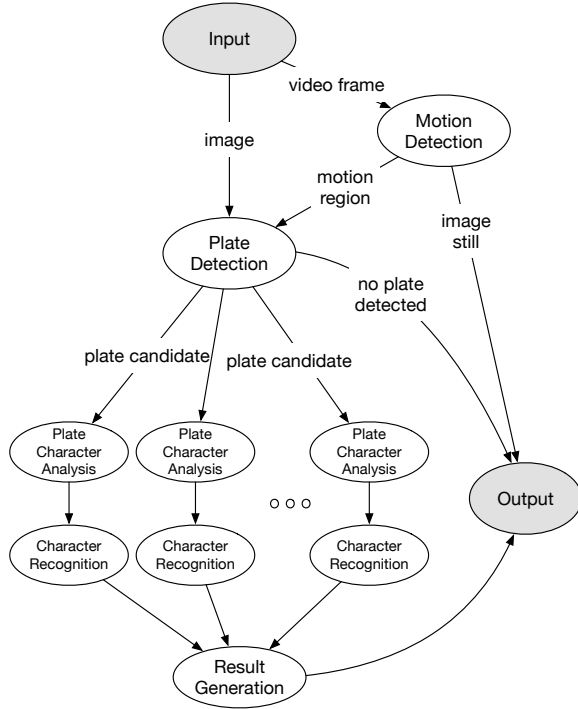

Figure 5: The task graph of OpenALPR.

Then we consider there are N clients and only one edge server connected as shown in Fig. 1. This edge server could be a single server or a cluster of servers. Each client $i, i \in [1, N]$ will process the upcoming job upon request, e.g. recognizing the license plates in video streams. Usually, those jobs will generate heavy computation task and could benefit from offloading some of them to the edge server. Without loss of generality, we use a graph of task to represent the complex task relations inside a job, which is essentially similar to the method call graph in [7], but in a more coarse granularity. For a certain kind of job, we start with its directed acyclic graph (DAG) $G = (V, E)$, which gives the task execution sequence. Each vertex $v \in V$ weight is the computation or memory cost of a task (c_v), while each edge $e = (u, v), u, v \in V, e \in E$ weight represents the data size of intermediate results (d_{uv}). Thus, our offloading problem can be taken as a graph partition problem, in which we need to assign a directed graph of tasks to different computing nodes (local, edge, or cloud), with the purpose to minimize

certain cost. In this paper, we primarily try to minimize the job finish time.

The remote response time includes the communication delay, the network transmission delay of sending data to the edge server, and the execution time on that server. We use an indicator $I_{v,i} \in \{0, 1\}$ for all v in V and for all $i \in [1, N]$. If $I_{v,i} = 1$, then the task v at client i will run locally, otherwise, it will run on the remote edge server. For those tasks running locally, the total execution time for client i is a sum as

$$T_i^{local} = \sum_{v \in V} I_{v,i} c_v / p_i \tag{1}$$

where p_i is the processor speed of client i.

Similarly, we use

$$\overline{T}_i^{local} = \sum_{v \in V} (1 - I_{v,i}) c_v / p_i \tag{2}$$

to represent the execution time of running the offloaded tasks locally. For network, when there is an offloading decision, the client need to upload the intermediate data (outputs of previous task, application status variables, configurations, etc) to the edge server in order to continue the computation. The network delay is modeled as

$$T_i^{net} = \sum_{(u,v) \in E} (I_{u,i} - I_{v,i}) d_{uv} / r_i + \beta_i \tag{3}$$

where r_i is the bandwidth assigned for this client and β_i is the communication latency which can be estimated using round trip time between the client i and the edge server.

For each client, the remote execution time is

$$T_i^{remote} = \sum_{v \in V} (1 - I_{v,i})(c_v / p_0) \tag{4}$$

where p_0 is the processor speed of the edge server.

Then our offloading task selection problem can be formulated as

$$\min_{I_i, r_i} \sum_{i=1}^{N} (T_i^{local} + T_i^{net} + T_i^{remote}) \tag{5}$$

The offloading task selection is represented by the indicator matrix **I**. This optimization problem is subject to the following constraints:

- The total bandwidth

$$\text{s.t.} \quad \sum_{i=1}^{N} r_i \leq R \tag{6}$$

- Like existing work, we restrict the data flow to avoid ping-pong effect in which intermediate data is transmitted back and forth between client and edge server.

$$\text{s.t.} \quad I_{v,i} \leq I_{u,i}, \quad \forall e(u, v) \in E, \forall i \in [1, N] \tag{7}$$

- Unlike exiting offloading frameworks for mobile cloud computing, we take the resource contention or schedule delay at the edge side into consideration by adding a end-to-end delay constraint.

$$\text{s.t.} \quad \overline{T}_i^{local} - (T_i^{net} + T_i^{remote}) > \tau, \quad \forall i \in [1, N] \tag{8}$$

where τ can be tuned to avoid selecting borderline tasks that if offloaded will get no gain due to the resource contention or schedule delay at the edge.

4.2 Optimization Solver

The proposed optimization is a mixed integer non-linear programming problem (MINLP), where the integer variable stands for the offloading decision and the continuous variable stands for the bandwidth allocation. To solve this optimization problem, we start from relaxing the integer constraints and solve the non-linear programming version of the problem using Sequential Quadratic Programming method, a constrained nonlinear optimization method. This solution is optimal without considering the integer constraints. Starting from this optimal solution, we optionally employ branch and bound (B&B) method to search for the optimal integer solution or simply do an exhaustive search when the number of clients and the number of tasks of each job are small.

4.3 Prioritizing Edge Task Queue

The offloading strategy produced by the task selection optimizes the "flow" time of each type of job. At each time epoch during the run time, the edge-front node receives a large number of offloaded tasks from the clients. Originally, we follow the first come first serve rule to accommodate all the client requests. For each request at the head of the task queue, the edge-front server first checks if the input or intermediate data (e.g. images or videos) available at the edge, otherwise, the server waits. This scheme is easy to implement but substantial computation is wasted if the network IO is busy with a large size file and there is no task that is ready for processing. Therefore, we improve the task scheduling with a task queue prioritizer to maintain a task sequence which minimizes the makespan for the task scheduling of all offloading task requests received at a certain time epoch. Since the edge node can execute the task only when the input data has been fully received or the depended tasks have finished execution. We consider that an offloaded task has to go through two stages: the first stage is the retrieval of input or intermediate data and state variables; the second stage is the execution of the task.

We model our scheduling problem using the flow job shop model and apply the Johnson's rule [19]. This scheme is optimal and the makespan is minimized, when the number of stages is two. Nevertheless, this model only fits in the case that all submitted job requests are independent and no priorities. When considering task dependencies, a successor can only start after its predecessor finishes. By enforcing the topological ordering constraints, the problem can be solved optimally using the B&B method [5]. However, this solution hardly scales against the number of tasks. In this case, we adapt the method in [3] and group tasks with dependencies and execute all tasks in a group sequentially. The basic idea is applying Johnson's rule in two levels. The first level is to decide the sequence of tasks with in each group. The difference in our problem is that we need to decide the best sequence among all valid topological orderings. Then the bottom level is a job shop scheduling problem in terms of grouped jobs (i.e. a group of tasks with dependencies in topological ordering), in which we can utilize Johnson's rule directly.

4.4 Workload Optimizer

If the workload is overwhelming and the edge-front server is saturated, the task queue will be unstable and the response time will be accumulated indefinitely. There are several measures we can take to address this problem. First, we can adjust the image/video resolution in client-side configurations, which makes well trade-off between speed and accuracy. Second, by constraining the task offloading problem, we can restrain more computation tasks at the client side, Third, if there are nearby edge nodes which are favored in terms of latency, bandwidth, and computation, we can further offload tasks to nearby edge nodes. We have investigated this case with performance improvement considerations in the Section 5. Last, we can always redirect tasks to the remote cloud, just like offloading in MCC.

5 INTER-EDGE COLLABORATION

In this section, We improve our edge-first design by taking the case when the incoming workload saturates our edge-front node into consideration. We will first discuss our motivation of providing such option and list the corresponding challenges. Then we will introduce several collaboration schemes we have proposed and investigated.

5.1 Motivation and Challenges

The resources of edge computing node are much richer than client nodes but are relatively limited compared to cloud nodes. While serving an increasing number of client nodes nearby, the edge-front node will be eventually overloaded and become non-responsive to new requests. As a baseline, we can optionally choose to offload further requests to the remote cloud. We assume that the remote cloud has unlimited resources and is capable to handle all the requests. However, running tasks remotely in the cloud, the application need to bear with unpredictable latency and limited bandwidth, which is not the best choice especially when there are other nearby edge nodes that can accommodate those tasks. We assume that under the condition when all available edge nodes nearby are exhausted, the mobile-edge-cloud computing paradigm will simply fall back to the mobile cloud computing paradigm. The fallback design is not in the scope of this paper. In this paper, we mainly investigate the inter-edge collaboration with the prime purpose to alleviate the burden on edge-front node.

When the edge-front node is saturated with requests, it can collaborate with nearby edge nodes by placing some tasks to these not-so-busy edge nodes, such that all the tasks can get scheduled in a reasonable time. This is slightly different from balancing the workload among the edge nodes and the edge-front node, in that the goal of inter-edge collaboration is to better serve the client nodes with submitted requests, rather than simply making the workload balanced. For example, an edge-front node that is not overloaded does not need to place any tasks to the nearby edge nodes, even when they are idle.

The challenges of inter-edge collaboration are two-fold: 1) we need to design a proper inter-edge task placement scheme that fulfills our goal of reducing the workload on the edge-front node while offloading proper amount of workload to the qualified edge nodes; 2) the task placement scheme should be lightweight, scalable, and easy-to-implement.

5.2 Inter-Edge Task Placement Schemes

We have investigated three task placement schemes for inter-edge collaboration.

- Shortest Transmission Time First (STTF)
- Shortest Queue Length First (SQLF)
- Shortest Scheduling Latency First (SSLF)

The STTF task placement scheme tends to place tasks on the edge node that has the shortest estimated latency for the edge-front node to transfer the tasks. The edge-front node maintains a table to record the latency of transmitting data to each available edge node. The periodical re-calibration is necessary because the network condition between the edge-front node and other edge nodes may vary from time to time.

The SQLF task placement scheme, on the other hand, tends to transfer tasks from the edge-front node to the edge node which has the least number of tasks queued upon the time of query. When the edge-front node is saturated with requests, it will first query all the available edge nodes about their current task queue length, and then transfer tasks to the edge node that has the shortest value reported.

The SSLF task placement scheme tends to transmit tasks from the edge-front node to the edge node that is predicted to have the shortest response time. The response time is the time interval between the time when the edge-front node submits a task to an available edge node and the time when it receives the result of the task from that edge node. Unlike the SQLF task placement scheme, the edge-front node keeps querying the edge nodes about the queue length, which may has performance issue when the number of nodes scales up and results in a large volume of queries. We have designed a novel method for the edge-front node to measure the scheduling latency efficiently. During the measurement phase before edge-front node chooses task placement target, edge-front node sends a request message to each available edge node, which appends a special task to the tail of the task queue. When the special task is executed, the edge node simply sends a response message to the edge-front node. The edge-front node receives the response message and records the response time. Periodically, the edge-front node maintains a series of response times for each available edge node. When the edge-front node is saturated, it will start to reassign tasks to the edge node having the shortest response time. Unlike the STTF and SQLF task assignment schemes, which choose the target edge node based on the current or most recent measurements, the SSLF scheme predicts the current response time for each edge node by applying regression analysis to the response time series recorded so far. The reason is that the edge nodes are also receiving task requests from client nodes, and their local workload may vary from time to time, so the most recent response time cannot serve as a good predictor of the current response time for the edge nodes. As the local workload in the real world on each edge node usually follows certain pattern or trend, applying regression analysis to the recorded response times is a good way to estimate the current response time. To this end, we recorded measurements of response times from each edge node, and offloads tasks to the edge node that is predicted to have the least current response time. Once the edge-front node starts to place task to a certain edge node, the

estimation will be updated using piggybacking of the redirected tasks, which lowers the overhead of measuring.

Each of the task placement schemes described above has some advantages and disadvantages. For instance, the STTF scheme can quickly reduce the workload on the edge-front node. But there is a chance that tasks may be placed to an edge node which already has intensive workload, as STTF scheme gathers no information of the workload on the target. The SQLF scheme works well when the network latency and bandwidth are stable among all the available edge nodes. When the network overheads are highly variant, this scheme fails to factor the network condition and always chooses edge node with the lowest workload. When an intensive workload is placed under a high network overhead, this scheme potentially deteriorates the performance as it needs to measure the workload frequently. The SSLF task placement scheme estimates the response time of each edge node by following the task-offloading process, and the response time is a good indicator of which edge node should be chosen as the target of task placement in terms of the workload and network overhead. The SSLF scheme is a well trade-off between previous two schemes. However, the regression analysis may introduce a large error to the predicted response time if inappropriate models are selected. We believe that the decision of which task placement scheme should be employed for achieving good system performance should always give proper considerations on the workload and network conditions. We evaluated those three schemes through a case study in the next section.

6 SYSTEM IMPLEMENTATION AND PERFORMANCE EVALUATION

In this section, we first brief the implementation details in building our system. Next, we introduce our evaluation setup and present results of our evaluations.

6.1 Implementation Details

Our implementation aims at a serverless edge architecture. As shown in system architecture of Fig. 4, our implementation is based on docker container for the benefits of quick deployment and easy management. Every component has been dockerized and its deployment is greatly simplified via distributing pre-built images. The creation and destruction of docker instances is much faster than that of VM instances. Inspired by the IBM OpenWhisk [18], each worker container contains an action proxy, which uses Python to run any scripts or compile and execute any binary executable. The worker container communicates with others using a message queue, as all the inputs/outputs will be jsonified. However, we don't jsonified image/video and use its path reference in shared storage. The task queue is implemented using Redis as it is in memory and has very good performance. The end user only needs to 1) deploy our edge computing platform on heterogeneous devices with just a click, 2) define the event of interests using provided API, 3) provide a function (scripts or binary executable) to process such event. The function we have implemented is using open source project OpenALPR[22] as the task payload for workers.

6.2 Evaluation Setup

6.2.1 Testbed. We have built a testbed consisting of four edge computing nodes. One of the edge nodes is the edge-front node, which is directly connected to a wireless router using a cable. Other three nodes are set as nearby edge computing nodes for the evaluation of inter-edge collaboration. These four machines have the same hardware specifications. They all have a quad-core CPU and 4 GB main memory. The three nearby edge nodes are directly connected to the edge-front node through a network cable. We make use of two types of Raspberry Pi (RPi) nodes as clients: one type is RPi 2 which is wired to the router while the other type is RPi 3 which is connected to router using built-in 2.4 Ghz WiFi.

6.2.2 Datasets. We have employed three datasets for evaluations. One dataset is the Caltech Vision Group 2001 testing database, in which the car rear image resolution (126 images with resolution 896x592) is adequate for license plate recognition [25]. Another dataset is a self-collected 4K video contains rear license plates taken on an Android smartphone and is converted into videos of different resolutions(640x480, 960x720, 1280x960, and 1600x1200.). The other dataset used in inter-edge collaboration evaluation contains 22 car images, with the various resolution ranging from 405x540 pixels to 2514x1210 pixels (file size 316 KB to 2.85 MB). The task requests use the car images as input in a round-robin way, one car image for each task request.

6.3 Task Profiler

Beside the round trip time and bandwidth benchmark we have presented in Fig. 2 and Fig. 3 to characterize the edge computing network, we have done profiling of the OpenALPR application on various client, edge and cloud nodes.
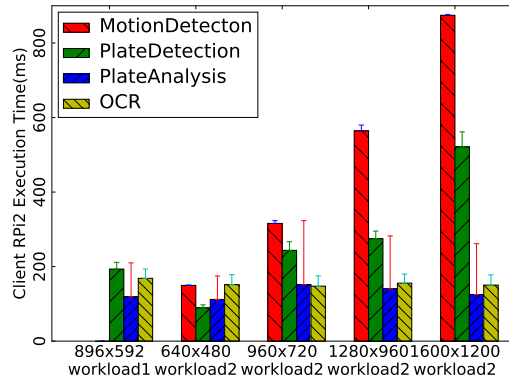

Figure 6: OpenALPR profile result of client type 1 (RPi2 quad-core 0.9GHz)

In this experiment, we are using both dataset 1 (workload 1) and dataset 2 (workload 2) at various resolutions. The execution time for each tasks are shown in Fig. 6, Fig. 7, Fig. 8, and Fig. 9. The results indicate that by utilizing an edge nodes, we can get a comparable amount of computation power close to clients for computation-intensive tasks. Another observations is that, due to the uneven optimizations on heterogeneous CPU architectures, some tasks are better to keep local while some others should be
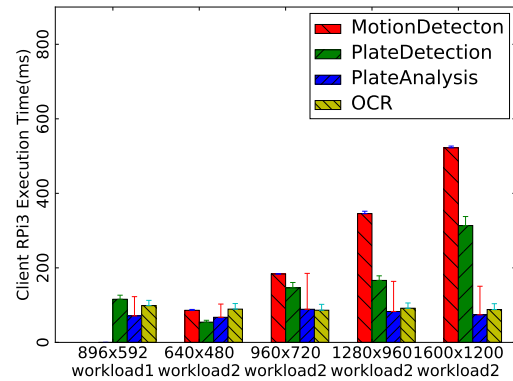

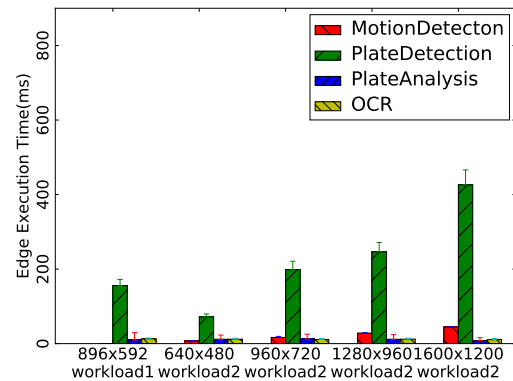Figure 7: OpenALPR profile result of client type 2 (RPi3 quad-core 1.2GHz)

Figure 8: OpenALPR profile result of a type of edge node (i7 quad-core 2.30GHz)
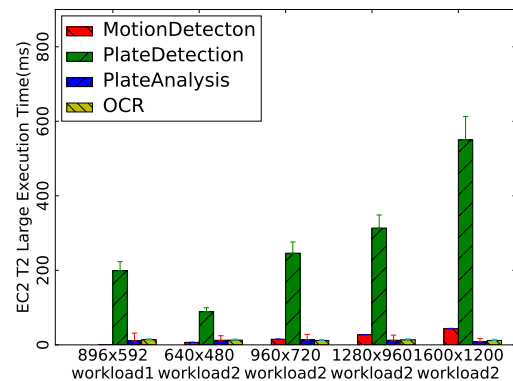

Figure 9: OpenALPR profile of a type of cloud node (AWS EC2 t2.large Xeon dual-core 2.40GHz)

offloaded to edge computing node. This observation justified the need of computation offloading between clients and edge nodes.
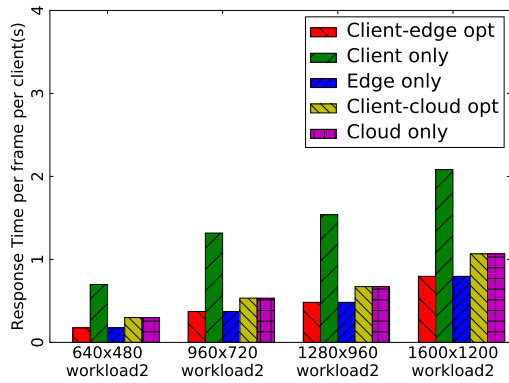
Figure 10: The comparison of task selection impacts on edge offloading and cloud offloading for wired clients (RPi2).
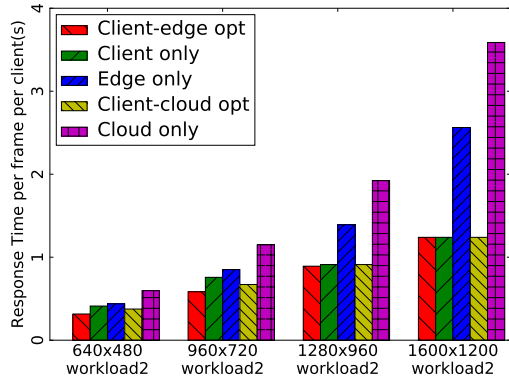

Figure 11: The comparison of task selection impacts on edge offloading and cloud offloading for 2.4 Ghz wireless clients (RPi3).

6.4 Offloading Task Selection

To understand how much the execution time can be reduced by splitting tasks between the client and the edge, or between the client and the cloud, we design an experiment with workloads generated from dataset 2 on two setups of scenarios: 1) one edge node provides service to three wired client nodes that have the best network latency and bandwidth; 2) one edge node provides service to three wireless 2.4 Ghz client nodes that have latency with high variance and relatively low bandwidth. The result of the first case is very straightforward: the clients simply upload all the input data and run all the tasks on the edge node in edge offloading or cloud node in cloud offloading, as shown in Fig. 10. This is mainly because using Ethernet cable can stably provide lowest latency and highest bandwidth, which makes offloading to edge very rewarding. We didn't evaluate 5 Ghz wireless client since this interface is not supported on our client hardware while we anticipate similar results as the wire case. We plot the result of a 2.4 Ghz wireless client node with offloading to an edge node or a remote cloud node in the second case in Fig. 11. Overall, the results showed that by offloading tasks to an edge computing platform,

the application we choose experienced a speedup up to 4.0x on wired client-edge configuration compared to local execution, and up to 1.7x compared to a similar client-cloud configuration. For clients with 2.4 Ghz wireless interface, the speedup is up to 1.3x on client-edge configuration compared to local execution, and is up to 1.2x compared to similar client-cloud configuration.
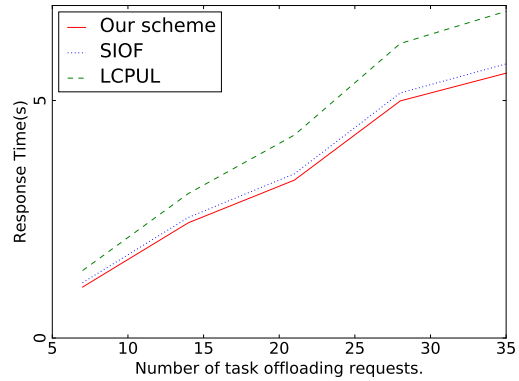

Figure 12: The comparison result of three task prioritizing schemes.

6.5 Edge-front Task Queue Prioritizing

To evaluate the performance of the task queue prioritizing, we collect the statistical results from our profiler service and monitoring service on various workload for simulation. We choose the simulation method because we can freely setup the numbers and types of client and edge nodes to overcome the limitation of our current testbed to evaluate more complex deployments. We add two simple schemes as baselines: 1) shortest IO first (SIOF): sorting all the tasks against the time cost of the network transmission. 2) longest CPU last (LCPUL): sorting all the tasks against the time cost of the processing on the edge node. In the simulation, base on the combination of client device types, workloads and offloading decisions, we have in total seven types of jobs to run on the edge node. We increase the total number of jobs and evenly distributed them among the seven types and report the makespan time in Fig. 12. The result showed that LCPUL is the worst among those three schemes and our scheme outperforms the shortest job first scheme.

6.6 Inter-Edge Collaboration

We also evaluate the performance of the three task placement schemes (i.e., STTF, SQLF and SSLF) discussed in Section 5, through a controlled experiment on our testbed. For evaluation purpose, we configure the network in the edge computing system as follows. The first edge node, denoted as "edge node #1" afterwards, has 10 ms RTT and 40 Mbps bandwidth to the edge-front node. The second edge node, "edge node #2", has 20 ms RTT and 20 Mbps bandwidth to the edge-front node. The third edge node, "edge node #3", has 100 ms RTT and 2 Mbps bandwidth to the edge-front node. Thus, we emulate the situation where three edge nodes are in the distance to the edge-front node from near to far.
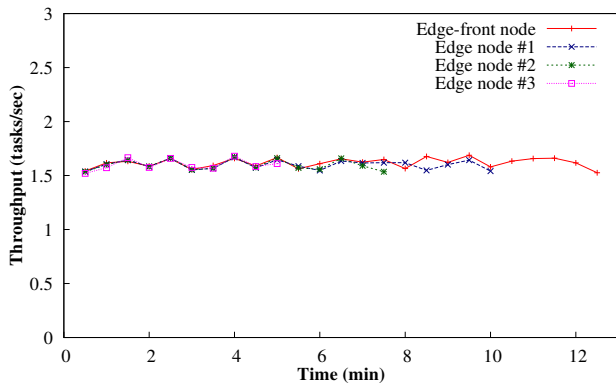
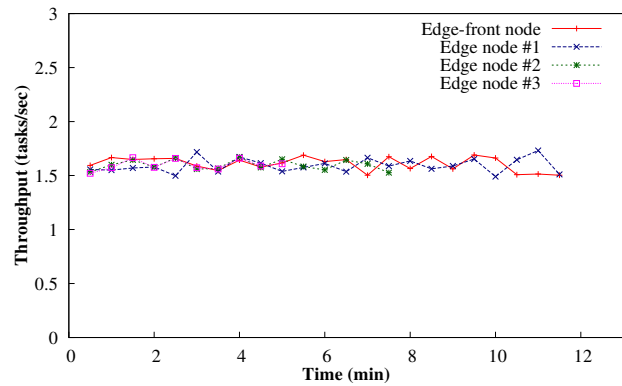Figure 13: Performance with no task placement scheme.
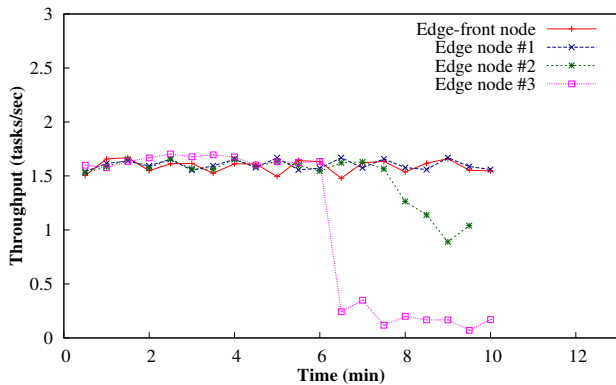

Figure 14: Performance of STTF.
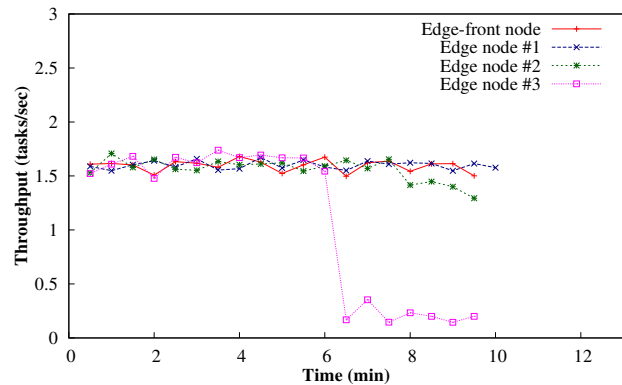

Figure 15: Performance of SQLF.

Figure 16: Performance of SSLF.

We use the third dataset to synthesize a workload as follows. In the first 4 minutes, the edge-front node receives 5 task requests per second, edge node #1 receives 4 task requests per second, edge node #2 receives 3 task requests per second, and edge node #3 receives 2 task requests per second, respectively. No task comes to any of the edge nodes after the first 4 minutes. For the SSLF task placement scheme, we implement a simple linear regression to predict the scheduling latency of the task being transmitted, since the workload we have injected is uniform distributed.

Fig. 13 illustrates the throughput on each edge node, when no task placement scheme is enabled on the edge-front node. The edge-front node has the heaviest workload and it takes about 12.36 minutes to finish all the tasks. We consider this result as our baseline.

Fig. 14 is the throughput result of STTF scheme. In this case, the edge-front node only transmits tasks to edge node #1, because edge node #1 has the highest bandwidth and the shortest RTT to the edge-front node. Fig. 17 reveals that the edge-front node transmits 120 tasks to edge node #1 and no task to other edge nodes. As edge node #1 has heavier workload than edge node #2 and edge node #3, the STTF scheme has limited improvement on the system performance: the edge-front node takes about 11.29 minutes to finish all the tasks. Fig. 15 illustrates the throughput result of SQLF

scheme. This scheme works better than the STTF scheme, because the edge-front node transmits more tasks to less-saturated edge nodes, efficiently reducing the workload on the edge-front node. However, the edge-front node intends to transmit many tasks to edge node #3 at the beginning, which has the lowest bandwidth and the longest RTT to the edge-front node. As such, the task placement may incur more delay then expected. From Fig. 17, the edge-front node transmits 0 task to edge node #1, 132 tasks to edge node #2, and 152 tasks to edge node #3. The edge-front node takes about 9.6 minutes to finish all the tasks.

Fig. 16 demonstrates the throughput result of SSLF scheme. This scheme considers both the transmission time of the task being placed and the waiting time in the queue on the target edge node, and therefore achieves the best performance of the three. As mentioned, edge node #1 has the lowest transmission overhead but the heaviest workload among the three edge nodes, while edge node #3 has the lightest workload but the highest transmission overhead. In contrast, edge node #2 has modest transmission overhead and modest workload. The SSLF scheme takes all these situations into consideration, and places the most number of tasks on edge node #2. As shown in Fig. 17, the edge-front node transmits 4 tasks to edge node #1, 152 tasks to edge node #2, and 148 tasks to edge node #3 when working with the SSLF scheme. The edge-front node

takes about 9.36 minutes to finish all the tasks, which is the best result among the three schemes. We infer that the third scheme will further improve the task complete time if more tough network conditions and workloads are considered.
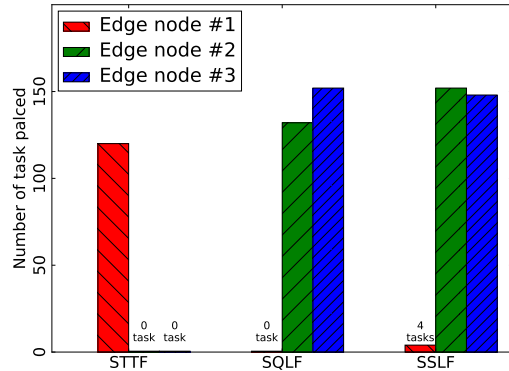

Figure 17: Numbers of tasks placed by the edge-front node.

7 RELATED WORK

The emergence of edge computing has drawn attentions due to its capabilities to reshape the land surface of IoTs, mobile computing, and cloud computing [6, 14, 32, 33, 36–38]. Satyanarayanan [29] has briefed the origin of edge computing, also known as fog computing [4], cloudlet [28], mobile edge computing [24] and so on. Here we will review several relevant research fields towards video edge analytics, including distributed data processing and computation offloading in various computing paradigms.

7.1 Distributed Data Processing

Distributed data processing has close relationship to the edge analytics in the sense that those data processing platforms [9, 39] and underlying techniques [16, 23, 27] can be easily deployed on a cluster of edge nodes. In this paper, we pay specially attentions to distributed image/video data processing systems. VideoStorm[40] made insightful observation on vision-related algorithms and proposed resource-quality trade-off with multi-dimensional configurations (e.g. video resolution, frame rate, sampling rate, sliding window size, etc.). The resource-quality profiles are generated offline and a online scheduler is built to allocate resources to queries to optimize the utility of quality and latency. Their work is complementary to ours, in that we do not consider the trade-off between quality and latency goals via adaptive configurations. Vigil [42] is a wireless video surveillance system that leveraged edge computing nodes with emphasis on the content-aware frame selections in a scenario where multiple web cameras are at the same location to optimize the bandwidth utilization, which is orthogonal to the problems we have addressed here. Firework [41] is a computing paradigm for big data processing in collaborative edge environment, which is complementary to our work in terms of shared data view and programming interface.

While there should be more on-going efforts for investigating the adaptation, improvement, and optimization of existing distributed

data processing techniques on edge computing platform, we focus more on the task/application-level queue management and scheduling, and leave all the underlying resource negotiating, process scheduling to the container cluster engine.

7.2 Computation Offloading

Computation Offloading (a.k.a Cyber foraging [28]) has been proposed to improve resource utilization, response time, and energy consumption in various computing environments [7, 13, 21, 31]. Work [17] has quantified the impact of edge computing on mobile applications and found that edge computing can improve response time and energy consumption significantly for mobile devices through offloading via both WiFi and LTE networks. Mocha [34] has investigated how a two-stage face recognition task from mobile device can be accelerated by cloudlet and cloud, In their design, clients simply capture image and sends to cloudlet. The optimal task partition can be easily achieved as it has only two stages. In LAVEA, our application is more complicated in multiple stages and we leverage client-edge offloading and other techniques to improve the resource utilization and optimize the response time.

8 DISCUSSIONS AND LIMITATIONS

In this section, we will discuss alternative design options, point out current limitations, and future work that can improve the system.

Measurement-based Offloading. In this paper, we are using a measurement-based offloading (static offloading), i.e, the offloading decisions are based on the outcome of periodic measurements. We consider this as one of the limitations of our implementations, as stated in [15] and there are several dynamic computation offloading schemes have been proposed [12]. We are planning to improve the measurement-based offloading in the future work.

Video Streaming. Our current data processing is image-based, which is one of the limitations of our implementation. The input is either in the format of image or in video stream which is read into frames and sent out. We believe by utilizing existing video streaming techniques in between our system components for data sharing will further improve the system performance and open more potential opportunities for optimization.

Discovering Edge Nodes. There are different ways for the edge-front node to discover the available edge nodes nearby. For example, every edge node intending to serve as a collaborator may open a designated port, so that the edge-front node can periodically scan the network and discover the available edge nodes. This is called the "pull-based" method. In contrast, there is also a "push-based" method, in which the edge-front node opens a designated port, and every edge node intending to serve as a collaborator will register to the edge-front node. When the network is in a large scale, the pull-based method usually performs poorly because the edge-front node may not be able to discover an available edge node in a short time. For this reason, the edge node discovery is implemented in a push-based method, which guarantees good performance regardless of the network scale.

9 CONCLUSION

In this paper, we have investigated providing video analytic services to latency-sensitive applications in edge computing environment.

As a result, we have built LAVEA, a low-latency video edge analytic system, which collaborates nearby client, edge and remote cloud nodes, and transfers video feeds into semantic information at places closer to the users in early stages. We have utilized an edge-front design and formulated an optimization problem for offloading task selection and prioritized task queue to minimize the response time. Our result indicated that by offloading tasks to the closest edge node, the client-edge configuration has speedup range from 1.3x to 4x (1.2x to 1.7x) against running in local (client-cloud) under various network conditions and workloads. In case of a saturating workload on the front edge node, we have proposed and compared various task placement schemes that are tailed for inter-edge collaboration. The proposed prediction-based shortest scheduling latency first task placement scheme considers both the transmission time of the task and the waiting time in the queue and outputs overall performance better than the other schemes.

REFERENCES

[1] Amazon Web Service. 2017. AWS Lambda@Edge. http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html. (2017).
[2] Christos-Nikolaos E Anagnostopoulos, Ioannis E Anagnostopoulos, Ioannis D Psoroulas, Vassili Loumos, and Eleftherios Kayafas. 2008. License plate recognition from still images and video sequences: A survey. *IEEE Transactions on intelligent transportation systems* 9, 3 (2008), 377–391.
[3] KR Baker. 1990. Scheduling groups of jobs in the two-machine flow shop. *Mathematical and Computer Modelling* 13, 3 (1990), 29–36.
[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 13–16.
[5] Peter Brucker, Bernd Jurisch, and Bernd Sievers. 1994. A branch and bound algorithm for the job-shop scheduling problem. *Discrete applied mathematics* 49, 1 (1994), 107–127.
[6] Yu Cao, Songqing Chen, Peng Hou, and Donald Brown. 2015. FAST: A fog computing assisted distributed analytics system to monitor fall for stroke mitigation. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*. IEEE, 2–11.
[7] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, NY, USA, 49–62. DOI:https://doi.org/10.1145/1814433.1814441
[8] Eyal de Lara, Carolina S Gomes, Steve Langridge, S Hossein Mortazavi, and Meysam Roodi. 2016. Hierarchical Serverless Computing for the Mobile Edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 109–110.
[9] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[10] Shan Du, Mahmoud Ibrahim, Mohamed Shehata, and Wael Badawy. 2013. Automatic license plate recognition (ALPR): A state-of-the-art review. *IEEE Transactions on circuits and systems for video technology* 23, 2 (2013), 311–325.
[11] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi
[12] Wei Gao, Yong Li, Haoyang Lu, Ting Wang, and Cong Liu. 2014. On exploiting dynamic execution patterns for workload offloading in mobile cloud applications. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. IEEE, 1–12.
[13] Mark S Gordon, Davoud Anoushe Jamshidi, Scott A Mahlke, Zhuoqing Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently.. In *OSDI*, Vol. 12. 93–106.
[14] Zijiang Hao, Ed Novak, Shanhe Yi, and Qun Li. 2017. Challenges and Software Architecture for Fog Computing. *Internet Computing* (2017).
[15] Mohammed A Hassan, Kshitiz Bhattarai, Qi Wei, and Songqing Chen. 2014. POMAC: Properly Offloading Mobile Applications to Clouds. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*.
[16] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.
[17] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2016. Quantifying the Impact of Edge Computing on Mobile Applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 5.
[18] IBM. 2017. Apache OpenWhisk. http://openwhisk.org/. (April 2017).
[19] Selmer Martin Johnson. 1954. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly* 1, 1 (1954), 61–68.
[20] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed computing for the 99%. *arXiv preprint arXiv:1702.04024* (2017).
[21] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. 2009. Wishbone: Profile-based Partitioning for Sensornet Applications.. In *NSDI*, Vol. 9. 395–408.
[22] OpenALPR. 2017. OpenALPR – Automatic License Plate Recognition. http://www.openalpr.com/. (April 2017).
[23] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.
[24] M Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, and others. 2014. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative* (2014).
[25] Brad Philip and Paul Updike. 2001. Caltech Vision Group 2001 testing database. http://www.vision.caltech.edu/html-files/archive.html. (2001).
[26] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. 2012. Real-time computer vision with OpenCV. *Commun. ACM* 55, 6 (2012), 61–69.
[27] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 36.
[28] Mahadev Satyanarayanan. 2001. Pervasive computing: Vision and challenges. *IEEE Personal communications* 8, 4 (2001), 10–17.
[29] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (2017), 30–39.
[30] Mahadev Satyanarayanan, Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, Wenlu Hu, and Brandon Amos. 2015. Edge analytics in the internet of things. *IEEE Pervasive Computing* 14, 2 (2015), 24–31.
[31] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, and Ellen Zegura. 2014. Cosmos: computation offloading as a service for mobile devices. In *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*. ACM, 287–296.
[32] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct 2016), 637–646. DOI: https://doi.org/10.1109/JIOT.2016.2579198
[33] Weisong Shi and Schahram Dustdar. 2016. The Promise of Edge Computing. *Computer* 49, 5 (2016), 78–81.
[34] Tolga Soyata, Rajani Muraleedharan, Colin Funai, Minseok Kwon, and Wendi Heinzelman. 2012. Cloud-Vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*. IEEE, 000059–000066.
[35] Xiaoli Wang, Aakanksha Chowdhery, and Mung Chiang. 2016. SkyEyes: adaptive video streaming from UAVs. In *Proceedings of the 3rd Workshop on Hot Topics in Wireless*. ACM, 2–6.
[36] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. 2015. Fog Computing: Platform and Applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*. IEEE, 73–78.
[37] Shanhe Yi, Cheng Li, and Qun Li. 2015. A Survey of Fog Computing: Concepts, Applications and Issues. In *Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata '15*. ACM, 37–42.
[38] Shanhe Yi, Zhengrui Qin, and Qun Li. 2015. Security and privacy issues of fog computing: A survey. In *Wireless Algorithms, Systems, and Applications*. Springer, 685–695.
[39] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. *HotCloud* 10 (2010), 10–10.
[40] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 377–392. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang
[41] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong. 2016. Firework: Big Data Sharing and Processing in Collaborative Edge Environment. In *2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. 20–25. DOI: https://doi.org/10.1109/HotWeb.2016.12
[42] Tan Zhang, Aakanksha Chowdhery, Paramvir Victor Bahl, Kyle Jamieson, and Suman Banerjee. 2015. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 426–438.