NetVision: On-demand Video Processing in Wireless Networks

Zongqing Lu, *Member, IEEE*, Kevin Chan, *Member, IEEE*, Rahul Urgaonkar, Shiliang Pu, and Thomas La Porta, *Fellow, IEEE*

***Abstract*—The vast adoption of mobile devices with cameras has greatly contributed to the proliferation of the creation and distribution of videos. For a variety of purposes, valuable information may be extracted from these videos. While the computational capability of mobile devices has greatly improved recently, video processing is still a demanding task for mobile devices. We design an on-demand video processing system, *NetVision*, that performs distributed video processing using deep learning across a wireless network of mobile and edge devices to answer queries while minimizing the query response time. However, the problem of minimal query response time for processing videos stored across a network is a strongly NP-hard problem. To deal with this, we design a greedy algorithm with bounded performance. To further deal with the dynamics of the transmission rate between mobile and edge devices, we design an adaptive algorithm. We built NetVision and deployed it on a small testbed. Based on the measurements of the testbed and by extensive simulations, we show that the greedy algorithm is close to the optimum and the adaptive algorithm performs better with more dynamic transmission rates. We then perform experiments on the small testbed to examine the realized system performance in both stationary networks and mobile networks.**

***Index Terms*—Video processing, edge computing, wireless networks.**

I. Introduction

The proliferation of handheld mobile devices and wireless networks has facilitated the generation and rapid dissemination of vast numbers of videos. Videos taken for various purposes may contain valuable information about past events that can be exploited for on-demand information retrieval. For example, a distributed video processing problem may involve a query of a set of mobile devices to find a specific vehicle in a region of a city. Various stored videos within mobile devices, not necessarily for the intention of capturing the object of interest, may provide valuable information for such queries. However, the processing requirements for such applications approach the computational limits of the mobile devices. Although the computational capacity of mobile devices has greatly improved in the past few years, processing (multiple) videos is still overwhelming for mobile devices.

In this paper, we consider the detection of objects in videos on a wireless network consisting of mobile and edge devices. Instead of storing and processing videos locally, mobile devices can choose to upload videos to much more capable devices (*i.e.*, computers with a much powerful GPU), which can significantly accelerate video processing. Although these devices are standalone platforms, they are used to process videos on behalf of mobile devices and hence we call them edge devices. However, due to the availability gap (the time between when the video is taken and when it is uploaded) [1] and when a query is issued, edge devices will not likely have the pertinent video pre-stored, especially when the query is about recent events. Therefore, to reduce the delay of the on-demand information retrieval from videos related to a query, the related videos can be processed either locally on the mobile devices or transmitted and processed on the edge devices.

Based on this use of wireless networks for video processing, there are clear scenarios to which this can be applied. Example scenarios are: (*i*) *emergency response and video forensics*, in which authorities attempt to identify objects or people of interest in videos captured by surveillance systems or other mobile devices that may have been either present or deployed in the time and area of interest; (*ii*) *wireless surveillance systems*, which support queries about the content of videos captured by wireless cameras via on-demand information retrieval from the videos. In these scenarios, edge devices can be deployed to support the storage and processing of videos to address on-demand information queries about past events. Without edge devices, this process is significantly delayed, resulting in serious consequences in the event that the query is not addressed satisfactorily.

As an example, an information query may be the following "*did a red truck drive through downtown today?*" Then, all related videos stored on either mobile devices or edge devices taken in proximity of the "downtown" area need to be processed to detect the presence of a "red truck". The query will reach all devices in the network and finds all of the potentially related videos based on video metadata (*e.g.*, GPS, timestamp). The network needs to determine where to process each video (locally or offloaded to edge devices), and to which edge device to upload each video. This approach should minimize the time required to process all of the related videos, which is referred to as the *query response time*.

To enable such information queries, we design a system, *NetVision*, that performs distributed video processing using deep learning across a wireless network to answer queries while minimizing the query response time. However, the problem of processing pertinent videos distributed throughout a wireless network with minimal query response time, which is referred to as the *processing scheduling* problem, turns out to be a strongly NP-hard problem. To deal with this,

Z. Lu is with the Department of Computer Science, Peking University. Email: zongqing.lu@pku.edu.cn. K. Chan is with Army Research Laboratory. E-mail: kevin.s.chan.civ@mail.mil. R. Urgaonkar is with Amazon. E-mail: urgaonka@amazon.com. S. Pu is with Hikvision. Email: pushiliang@hikvision.com. T. La Porta is with the Department of Computer Science and Engineering, Pennsylvania State University. E-mail: tlp@cse.psu.edu.

we design a greedy algorithm with bounded performance, which determines whether or not to offload each video, and schedules a transmission sequence to offload videos from a set of mobile devices before processing the videos. To cope with the dynamics of the transmission rate between mobile and edge devices during this process, we further design an adaptive algorithm, which makes such decisions in runtime. We built and deployed NetVision on a small testbed. Based on the measurements of the testbed, we perform simulations to extensively evaluate the proposed algorithms. We also perform experiments on the small testbed to examine the realized system performance in different network scenarios. The major contributions of this paper are summarized as follows.

- We formulate the processing scheduling problem for on-demand video processing to determine the optimal video offloading and transmission sequence in terms of minimizing the query response time.
- We design a greedy algorithm with bounded performance, which exploits average completion time of nodes as a criterion to consecutively determine each video offloading. The performance of the greedy algorithm is close to the optimum and much better than other approaches.
- We propose an adaptive algorithm with very low message overhead to collect information from nodes and then determine video offloading during runtime. The adaptive algorithm performs better when the transmission rate between mobile and edge devices is more dynamic.
- We built NetVision with these two algorithms and deployed it on a network of mobile devices and an edge device for information queries. Experimental results on the testbed verify the realized system performance in stationary networks.
- We further built an emulation environment for mobile networks featured with dynamic transmission rates. By plugging the small testbed into this environment, we further confirm the benefit of NetVision in mobile networks.

The rest of this paper is organized as follows. Section II reviews related work. Section III gives the overview. The greedy algorithm is presented in Section IV, followed by the adaptive algorithm in Section V. Section VI gives the implementation of NetVision. Section VII evaluates the performance, Section VIII discusses NetVision, and Section IX concludes the paper.

II. Related Work

The proliferation of mobile devices with cameras, such as smartphones and tablets, has substantially increased the prevalence of images and videos. Images and videos taken by mobile devices create opportunities for many applications and have attracted considerable attention from research communities. Much of the research focuses on images. Yan *et al.* [2] studied real-time image search on smartphones. Qin *et al.* [3] investigated tagging images, integrating information of people, activity and context in a picture. Wang *et al.* [4] optimized the selection of crowdsourced photos based on the metadata of images including GPS location, phone orientation, *etc*. Hua *et al.* [5] designed a real-time image sharing system for disaster environments. Likamwa *et al.* [6] investigated the energy optimization of image sensing on smartphones. Some works focus on videos [7]–[12]. However, none of these works consider on-demand information retrieval from videos stored on networked mobile devices.

Mobile cloud computing bridges the gap between the limitations of mobile devices and increasing mobile multimedia applications. Mobile devices can potentially perform offloading of computational workloads to either improve resource usage or augment performance. MAUI [13] and ThinkAir [14] are the system frameworks to support method-level computation offloading by code migration. Dynamic execution patterns and context migration is investigated for code offloading in [15]. Virtual Machine synthesis is exploited for offloading in [16]. A few works focus on the latency of mobile offloading. Wang *et al.* [17] considered reducing task completion by adaptive local restart. Kao *et al.* [18] optimized the latency with energy constraints by task assignment of mobile offloading. Unlike existing work that focuses on workload offloading from individual mobile device and concerns about when and how much to offload, the major focus of this paper is to optimize the latency of video processing across many mobile and edge devices by determining to which edge device to offload and the video offloading sequence.

Optimizing the computing of deep learning applications on mobile devices are recently investigated by compressing parameters of Convolutional Neural Networks (CNNs) [19] [20], by distributing computation to heterogeneous processors on-board [21] [22], and by jointly maximizing the performance of all the concurrently running deep learning applications under resource constraints [23]. DeepCache [24] is a principled cached design to improve the execution efficiency of deep learning for continuous mobile vision. However, the computing capability of mobile devices (even equipped with mobile GPUs) is still far behind powerful GPU-accelerated workstations.

Glimpse [25] is a continuous, real-time object recognition system for camera-equipped mobile devices, where the algorithm for object recognition runs on the server. Vigil [26] is a real-time distributed wireless surveillance system, where edge computing nodes co-located with cameras are leveraged to perform simple vision analytic functions and analytic information together with significant associated video frames are uploaded to the cloud. MCDNN [27] is also a continuous video processing system on cloud-backed mobile devices, systematically trading off accuracy for resource usage. VideoEdge [28] is a system that trades off between the accuracy of video analytics and resources in a hierarchy of cameras, private clusters, and public clouds. Unlike these works, we focus on on-demand information retrieval and optimally processing videos stored across a wireless network of many mobile and edge devices.

This work is also related to machine scheduling, *e.g.*, multiprocessor scheduling. However, unlike existing work, such as [29] [30], that considers thermal or energy as the cost of scheduling a job, we consider the cost as a part of the completion time at the scheduled machine. This changes the problem dramatically and hence existing solutions cannot be applied to our problem.

III. Overview

A. The Big Picture

We consider a wireless network that consists of mobile and edge devices, where mobile devices can directly communicate with edge devices via wireless links. When an information retrieval query is initiated, videos on the nodes related to the query need to be processed to answer the query. Note that when we say node or network node, it refers to either a mobile device or an edge device. In such networks, queries can be easily disseminated in the network and then parsed at each node to find the related videos, *e.g.*, based on metadata of videos, such as GPS locations and timestamps, or using VideoMec [12]. The dissemination and parsing of the queries is important to this process but is not the focus of this paper.

Since mobile devices have limited computational capability, processing videos on mobile nodes may result in long processing times, especially when there are many videos to be processed. Therefore, besides processing videos locally, mobile devices can also offload videos to edge devices and process videos remotely. However, the offload process incurs other delays, *e.g.*, the processing delay at the edge device and communication delay. Moreover, we consider deep learning for video processing. Although deep learning (*e.g.*, CNNs) can be greatly accelerated by a GPU using parallel computing, processing even a single video will fully occupy the GPU and thus videos have to be processed sequentially. Therefore, when an edge device is busy processing a video, it has to put other videos into a queue. Moreover, we do not consider mobile to mobile offloading since mobile devices have similar computational capacity and such offloading rarely benefits when considering these delays together.

Moreover, due to the constraints of video processing techniques (*e.g.*, the feature extraction for action recognition requires all frames be available beforehand [31]), nodes can process videos only when the video has been fully received[1]. Considering this together with the limitation of wireless link capacity, when more than one mobile device needs to offload videos to the same edge device, it is desirable to transmit the videos sequentially rather than in parallel such that the edge device can process videos early. Similarly, each mobile device should offload videos sequentially as well. For example, assuming a node needs to transmit two videos with the same size to another node and transmitting one video costs time t, if the two videos are transmitted one by one, the receiver can start processing the first video at t and the second video at $2t$. However, if the two videos are sent out simultaneously, the receiver can only start processing at time $2t$.

In addition, it is possible that different edge devices are receiving videos from different mobile devices simultaneously. This can be accomplished by assigning different wireless channels at edge devices so as to avoid potential interference. These constraints on video processing and communications complicate the problem of processing videos throughout a wireless network, specifically, when we aim to take advantage of edge devices to optimize the query response time. In summary, each node can only send or receive one video at a time, which is referred to as the communication constraint in this paper.

[1]Very large videos can be easily segmented into smaller videos by preprocessing based on the change of scene or context for storing and transmission. We assume that the videos of mobile devices have already been preprocessed. More sophisticated optimization incorporating with pre-processing is to be considered in future work.

B. The Processing Scheduling Problem

To minimize the query response time, which is the time required to process all the videos related to the query, we need jointly consider several factors: which nodes should process which videos, and what transmission sequence to perform the video offloading, as each node can only transmit (or receive) one video at a time. The *processing scheduling* problem is to find such a video offloading and transmission sequence that minimizes the query response time.

The processing scheduling problem is NP-hard, which can be proved by reduction from *machine scheduling* [32]. Considering the special case where the communication delay of videos is zero, processing scheduling can be seen as a generalization of machine scheduling with the constraint that certain jobs can be only scheduled on some machines (*i.e.*, videos stored at a mobile device can only be processed at this mobile device or remotely at edge devices), which is also NP-hard. Thus, processing scheduling is NP-hard in the strong sense. We do note that there is past work on machine scheduling, considering different constraints. However, to the best of our knowledge, they do not consider the cost of scheduling a job (*i.e.*, the communication delay of an offloaded video) as a part of the completion time at the scheduled machine. This change seems minor, however it makes the problem completely different and hence existing solutions cannot be applied to our problem. We will show the performance of the scheme that does not consider the communication delay in Section VII. Moreover, the processing scheduling problem is also generic, because in many applications job/task scheduling indeed incurs costs at the completion time, especially when across networked devices. Therefore, our solution can be generalized to other applications.

Let V represent the set of videos stored in the network and related to the query, and let U denote the set of nodes in the network. U_c denotes the set of edge devices and U_d denotes the set of mobile devices, where $U = U_c \cup U_d$. The query response time $\mathbf{T}_{\max}$ is the maximum time to complete processing of the assigned videos among all of the nodes. *For edge devices, the assigned videos are the videos stored locally and the videos scheduled to be offloaded from mobile devices. For mobile devices, the assigned videos are the locally stored videos excluding offloaded videos.* Let T_k, $k \in U$ denote the completion time of node k and then $\mathbf{T}_{\max} = \max_{k \in U} T_k$. The processing scheduling problem is to minimize $\mathbf{T}_{\max}$.

C. Completion Time

First, we show how to calculate the completion time of nodes with the assignment of videos. Each video i assigned at node k, has processing delay $p_{i,k}$ and communication delay $c_{i,k}$. Note that $p_{i,k}$ may vary across different types of queries

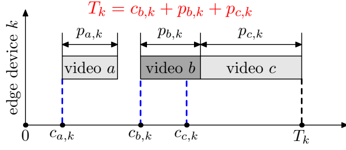


Fig. 1. An example of calculating the completion time of edge devices

that require different video processing solutions; and $c_{i,k}$ is the time from the initiation of the query to when node k fully receives video i. Note $c_{i,k} = 0$ represents video i is locally stored at node k.

Since videos may be scheduled to be processed by edge devices instead of locally by mobile devices, we need to account for the communication delay incurred by the offload. As a result, the completion time of edge devices is not simply equal to the sum of processing delay of videos assigned to each node. Further, an edge device, say k, may also spend time waiting for assigned videos. Therefore, for each video assigned to k, we check if the video offload to k is completed before k finishes processing existing videos or previously received videos. If the offload is complete, the edge device does not incur any waiting time, otherwise, the waiting time of k for the video needs to be included in T_k. Therefore, T_k is calculated as the sum of the processing delay of videos assigned at node k and the waiting time for each video to be offloaded.

Fig. 1 is an example of calculating the completion time of edge device k involving the offloading of videos with various cases of processing and communication delays. As discussed above, the completion time T_k is equal to the sum of the processing delay and waiting time, and hence $T_k = p_{a,k} + p_{b,k} + p_{c,k} + c_{a,k} + (c_{b,k} - p_{a,k} - c_{a,k})$. It is simplified as $T_k = c_{b,k} + p_{b,k} + p_{c,k}$, which can be interpreted that if node k spends time waiting for a video, then the time before processing the video can be denoted by the communication delay of the video. Thus, the calculation of the completion time of nodes can be generalized as

$$T_k = \max_{i \in V_k}(c_{i,k} + \sum_{j \in V_k} \alpha_{i,j,k} p_{j,k}), \tag{1}$$

where V_k denotes the set of videos assigned to node k, and $\alpha_{i,j,k} = 1$, if $c_{j,k} \geq c_{i,k}$, otherwise 0. Note that (1) can also be used to calculate the completion time of mobile devices. Since, for mobile device k, $c_{i,k} = 0$ and $\alpha_{i,j,k} = 1$ in (1), $T_k = \sum_{i \in V_k} p_{i,k}$.

D. Communication Delay

The communication delay not only depends on the size of videos and the transmission rate, but also the transmission sequence of the mobile devices and the receiving sequence for each of the edge devices. For example, mobile device k is scheduled to first offload video a to an edge device m and then transmit video b to another edge device n. Then, to calculate the communication delay of $c_{b,n}$, we need to determine when node k can start to transmit video b to n, which is actually the time when node k finishes offloading video a to m or the time when the video scheduled before b, say c, in the receiving sequence at n is received. Thus, we have

$$c_{b,n} = D_b / r_{k,n} + \max\{c_{a,m}, c_{c,n}\}, \tag{2}$$

where D_b is the size of video b, and $r_{k,n}$ denotes the transmission rate from k to n. From (2), we can see the calculation of communication delay is nonlinear.

E. Mathematical Formulation

Suppose $\mathbf{x}$ is a solution from the solution space χ for processing scheduling, where $\mathbf{x}$ decides which videos each mobile device should offload, to which edge devices these videos should be sent, and the transmission sequence of all the offloaded videos. The problem then can be formulated as

$$\begin{aligned} \min_{\mathbf{x} \in \chi} \quad & \max_{k \in U} \max_{i \in V_k(\mathbf{x})} (c_{i,k}(\mathbf{x}) + \sum_{j \in V_k(\mathbf{x})} \alpha_{i,j,k} p_{j,k}) \\ s.t. \quad & \alpha_{i,j,k} = 1, \text{ if } c_{j,k}(\mathbf{x}) \geq c_{i,k}(\mathbf{x}), \text{ otherwise } 0, \\ & \forall k \in U, \ \forall i, j \in V_k(\mathbf{x}), \end{aligned} \tag{3}$$

where $V_k(\mathbf{x})$ denotes the set of videos to be processed at node k of solution $\mathbf{x}$, $c_{i,k}(\mathbf{x})$ denotes the communication delay of video i under solution $\mathbf{x}$ and $c_{i,k}(\mathbf{x})$ is subject to the constraint that each node can only send or receive one video at any time.

The processing delay of each video can be easily obtained based on the size of video, the node profile, and the execution profile of the processing method, as in [13] and [33]. Therefore, for a specific node and processing method, the processing delay is proportional to the size of the videos (discussed in Section VII). As the calculation of communication delay is nonlinear, thus (3) cannot be further formulated by Integer Linear Programming (ILP), which can be solved by the ILP solver by integer relaxation.

Therefore, the processing scheduling problem is intrinsically the major challenge to build the on-demand video processing system in wireless networks. To deal with this, we propose a greedy algorithm with bounded performance to solve the processing scheduling problem in Section IV. Then, we propose an adaptive algorithm in Section V.

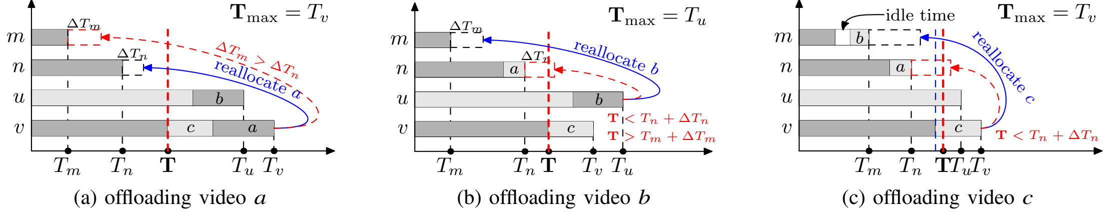
IV. Greedy Algorithm

In this section, we describe the design of the greedy algorithm, give the performance analysis and discuss how the greedy algorithm can be easily and efficiently implemented.

A. The Algorithm

The processing scheduling problem addresses how to offload videos from mobile devices to edge devices to minimize the maximum completion time for the entire process, which equivalently can be seen as averaging the completion time of all the nodes.

Intuitively, it is desirable for edge devices not to be idle since they are able to process the videos faster than the mobile devices. Even more preferable is that they are processing and receiving videos simultaneously. We consider two situations in which this may occur. Initially, the edge device may have locally stored videos to process; therefore, it is desirable to have the mobile devices upload larger videos first. This is also true when the disparity of the completion time among the nodes is the greatest. After several offloading steps, there may be some convergence in terms of the average completion



(a) offloading video a (b) offloading video b (c) offloading video c

Fig. 2. Illustration of the greedy algorithm, where m and n are edge devices, and u and v are mobile devices.

time. When the completion time among the nodes has less variability, it is better to offload videos with small size. Based on these intuitions, we design the greedy algorithm, which offloads a video from the mobile device with the maximum completion time to an edge device each step and improves $\mathbf{T}_{\max}$ step by step. The algorithm works as follows.

1. Calculate the completion time for each node according to (1), and then calculate the average completion time of nodes, denoted as

$$\mathbf{T} = \frac{\sum_{i\in U} T_i s_i}{\sum_{i\in U} s_i}, \tag{4}$$

   where s_i denotes the processing rate of node i. Note that the completion time of edge devices may include idle time for waiting for an incoming video.
2. For the mobile device that has maximum completion time, say i, find the videos that have sizes less than or equal to $(\mathbf{T}_{\max} - \mathbf{T})s_i$. Note that $\mathbf{T}_{\max} = T_i$.
3. Then, the set of videos is assessed from largest to smallest to find the first pairing of video and edge device such that, if the video is offloaded to the edge device, it has the minimal increase in completion time among all edge devices and its completion time is still less than or equal to $\mathbf{T}$.
4. If there is no valid pair, select the smallest video on mobile device i and offload it to the edge device, say m. edge device m is chosen such that the completion time of m is minimal among all edge devices and $T_m < \mathbf{T}_{\max}$ after the offloading of the video.
5. If $T_m \geq \mathbf{T}_{\max}$ (*i.e.*, $\mathbf{T}_{\max}$ cannot be reduced by offloading videos from mobile devices to edge devices), the process stops; otherwise iterate the process from step 1.

Let us use Fig. 2 as an example to illustrate the algorithm. There are four nodes in the network, where m and n are edge devices and u and v are mobile devices. First, each node calculates its own completion time. In Fig. 2a, since no videos have been offloaded, the completion time is simply the sum of the processing delay of videos. Then, we calculate $\mathbf{T}$ according to (4). In (4), $\sum_{i\in U} T_i s_i$ can be seen as the sum of workload at each node and $\sum_{i\in U} s_i$ is the processing power of all nodes. Thus, $\mathbf{T}$ is the weighted average completion time, assuming that the future offloading of videos does not incur any idle time on any edge devices and videos can be fragmented to any sizes. Therefore, $\mathbf{T}$ can be seen as a criterion to determine video offloading at each step, which avoids overloading the edge device.

As previously mentioned, videos that are offloaded from mobile device v should be smaller than $(T_v - \mathbf{T})s_v$. In Fig. 2a, these videos are a and c. For video offloading, we first consider the increase of the completion time of edge devices (*i.e.*, the joint consideration of the workload at the edge device and the communication delay of the video). Moreover, we consider the completion time itself. If it is longer than $\mathbf{T}$ after the assignment of the video, we should choose a smaller video. In Fig. 2a, since $D_a > D_c$ (recall D denotes the size of the video) video a is considered for offloading first. Although edge device n has more workload than m, the offloading of video a results in less increase in the completion time for n than for m (*i.e.*, $\Delta T_n < \Delta T_m$ and $T_n + \Delta T_n < \mathbf{T}$), so video a is offloaded to edge device n. Note that ΔT_n (ΔT_m) is the increase in completion time when a is allocated to n (m), which can be easily calculated using (1).

After that, we recalculate $\mathbf{T}$. Since the offloading of video a does not incur any idle time at edge device n, $\mathbf{T}$ is the same as before. As in Fig. 2b, currently, $\mathbf{T}_{\max} = T_u$ and thus the video offloading will be from mobile device u. Although n has more workload than m (which means n may not have to be idle in waiting for b), the offloading of video b to n incurs more communication delay than m; *i.e.*, $c_{b,n} = c_{a,n} + D_b/r_{u,n}$, where $r_{u,n}$ denotes the transmission rate between u and n, while $c_{b,m} = D_b/r_{u,m}$, assuming $r_{u,m} = r_{u,n}$. Since $\mathbf{T} < T_n + \Delta T_n$ and $\mathbf{T} > T_m + \Delta T_m$, video b will be offloaded to edge device m.

Due to the idle time of m incurred by the offloading of video b, $\mathbf{T}$ increases as shown in Fig. 2c. To determine the assignment of videos, the processing delay at edge devices can be easily calculated, but the communication delay is more complicated to compute as discussed before. For example, in Fig. 2c, $c_{c,m} = \max\{c_{a,n}, c_{b,m}\} + D_c/r_{v,m}$. So, video c is assigned at m rather than n since $T_n + \Delta T_n > \mathbf{T}$. The algorithm terminates with this offloading as the remaining two videos on mobile devices are large and the offloading of these videos can no longer reduce $\mathbf{T}_{\max}$.

The processing scheduling problem can be seen as balancing the completion time at each node. Thus, $\mathbf{T}$ is employed as a criterion for video offloading at each iteration, since $\mathbf{T}$ can be treated as the optimal average completion time. Moreover, at each step, we consider the increase of the completion time at edge devices, which is a joint consideration of the incurred communication delay and idle time at edge devices. Therefore, by regulating video offloading by $\mathbf{T}$ and minimizing the increase of completion time at edge devices, the greedy algorithm can reduce $\mathbf{T}_{\max}$ step-by-step towards the optimum.

B. Performance Analysis

For each offloading step, the greedy algorithm attempts to minimally increase the completion time for the edge device.

However, when the completion time with the minimal increase is more than $\mathbf{T}$, the greedy algorithm chooses to balance the completion time among edge devices to avoid overload. Moreover, due to the heterogeneity of both processing rates and transmission rates, it is hard to give a tight bound on the performance of the greedy algorithm. However, in the following, we try to give some insights on the algorithm performance with the variability of these rates.

Let t be the last time when all edge devices are busy (idle time does count as busy), $x = \frac{\sum_{i \in V} D_i}{\sum_{j \in U} s_j}$ (x can be seen as the average completion time of nodes without considering idle time), and let y denote the processing delay of the video with the largest size at the edge device, which is scheduled to process the last offloaded video (assuming that $\mathbf{T}_{\max}$ is determined by an edge device). Since $\mathbf{T}$ is explored as a criterion to determine video offloading, together with (1), we have

$$t \leq x + \sum_{i \in U_c} \frac{x s_i}{r_i}, \tag{5}$$

where, to simplify the analysis, we assume that videos offloaded at an edge device i are transmitted at a constant rate r_i from mobile devices. In (5), $\sum_{i \in U_c} \frac{x s_i}{r_i}$ gives the worse case of communication delay, assuming all videos are offloaded to edge devices and incur idle time at edge devices. Therefore, (5) gives the worse case of t.

Moreover, the last video offloading of the greedy algorithm minimizes the completion time at the assigned edge device among all edge devices. Therefore, by including the processing delay and communication delay of the last video, we have

$$\mathbf{T}_{\max} \leq t + y + \sum_{j \in U_c} \frac{y s_j}{r_j}. \tag{6}$$

By plugging (5) into (6), we have

$$\mathbf{T}_{\max} \leq x + \sum_{i \in U_c} \frac{x s_i}{r_i} + y + \sum_{j \in U_c} \frac{y s_j}{r_j}. \tag{7}$$

Let $\mathbf{T}^*$ denote the optimal maximum completion time and clearly we have

$$x \leq \mathbf{T}^*$$
$$y \leq \mathbf{T}^*.$$

Together with (7), we have

$$\mathbf{T}_{\max} \leq 2\mathbf{T}^*(1 + \sum_{i \in U_c} \frac{s_i}{r_i}). \tag{8}$$

The approximation ratio of the greedy algorithm is given in (8). We can see, from (8), when the processing rate of edge devices is high, the communication delay has a great impact on the completion time of edge devices. Thus, the approximation ratio goes up. When the transmission rate is high, the processing delay dominates the completion time and then the approximation ratio approaches 2. Although the bound on the performance is not tight, as will be shown in Section VII, the greedy algorithm performs much better than this bound.

Different from machine scheduling that has simple algorithms with an approximation ratio only related to the number of machine, our problem is much more difficult. The completion time is tightly coupled with not only processing rate but also transmission rate. Moreover, the completion time has a nonlinear relation with transmission rate. These make it almost impossible to derive an approximation ratio independent of processing rate and transmission rate.

For the computational complexity, as one video is offloaded during each iteration, there are at most $|V|$ iterations for the greedy algorithm. For each iteration, the videos stored at the mobile device with the maximum completion time are iterated over all the edge devices to minimize the increase in completion time. Therefore, the computational complexity of the greedy algorithm is $O(|U||V|^2)$.

C. *Discussion*

The greedy algorithm is a centralized approach and requires the information of all the videos *a priori*. When a query is initiated, the information (*e.g.*, data size) about videos stored in the network and related to the query needs to be collected at one node, *e.g.* an edge device, to run the greedy algorithm. The solution is then sent to the other nodes. Alternatively, the information can be collected at each node and each node may run the greedy algorithm. This is feasible, since the information collected is small and the computational complexity of the algorithm is low.

The solution of the processing scheduling problem determines which videos are offloaded from mobiles and edge devices. It also determines the transmission sequence, but this sequence is shown not to be trivial. For example, in Fig. 2, for mobile device v, the sending sequence is a and then c. However, v may not transmit c immediately after a; it must be transmitted after m receives video b from u. Therefore, when there is a video for which the mobile device cannot locally determine the transmission start time, the receiving edge device will inform the mobile device when it is ready to receive. Although such coordination incurs additional communication overhead (and idle time), the overhead is low since there is at most one message for each offloaded video.

In NetVision, the greedy algorithm is run on edge devices and hence edge devices control the mobile's transmission of the videos to the edge devices, as will be discussed in Section VI. The greedy algorithm is designed for the scenario where mobile and edge devices are stationary (*e.g.*, wireless surveillance systems) and the transmission rate between them is steady (or varies slightly). To cope with the scenario with high dynamics of transmission rate, we propose an adaptive algorithm.

V. Adaptive Algorithm

In this section, we consider the case where the transmission rate between mobile and edge devices dynamically changes during the on-demand querying of videos process (but assume that all nodes stay connected to the network during the process). So, we can capture the impact of the factors that can influence the transmission rate, such as channel quality and mobility, and hence the video processing performance.

Due to the dynamics of the transmission rate, the communication delay of offloaded videos also varies. This makes the processing scheduling problem more difficult, because we do not know how the transmission rate changes *a priori*. Since the communication delay of an offloaded video is only known after the transmission of the video is completed, it is better to determine video offloading in realtime for such scenarios. Therefore, we propose an adaptive algorithm that makes video offloading decisions during runtime, through consideration of the transmission rate, the communication delay and the completion time.

A. The Algorithm

We assume the same query is issued to the network of mobile and edge devices. Unlike the greedy algorithm which determines video offloading beforehand, the adaptive algorithm decides video offloading in runtime. Therefore, it is more resilient to dynamic transmission rates, which is the main advantage over the greedy algorithm.

Intuitively, to offload videos in runtime, the designed algorithm should gradually reallocate videos from mobile devices, balance the workload among edge devices, and prevent edge devices from being overloaded. Moreover, the adaptive algorithm should not incur too much communication overhead, which would delay the video transmission. Based on these considerations, the adaptive algorithm is designed to adapt to the dynamics of transmission rate and reduce $\mathbf{T}_{\max}$ dynamically as videos are transmitted and others are being processed.

To describe the adaptive algorithm, we first give the overall workflow and then detail how the edge device decides whether to accept offload requests from mobile devices and how the mobile device decides to which edge device to offload the video based on the replies from edge devices.

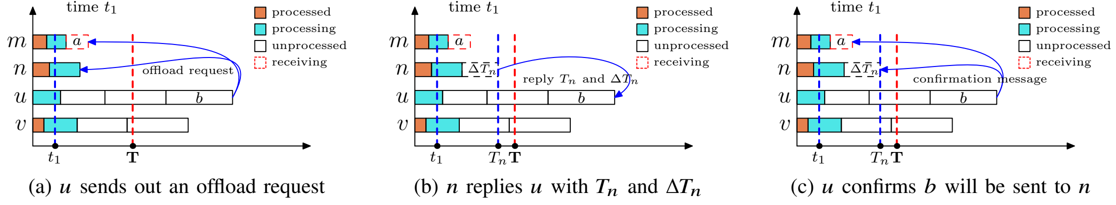
Upon receiving the query, each node identifies locally stored videos related to the query and broadcasts the information of the data size of each such video and its processing rate to other nodes. Then, each node will have the information to calculate the completion time of all other nodes in the network, which includes the data size, the locality, and the communication delay of each video, and the processing rate of each node. Later, when video offload occurs, the locality and communication delay of the video is updated. These information is maintained and updated at each node. For each mobile device, it also needs to probe each edge devices to obtain current transmission rate (the probing will be discussed in Section VI). After that, all nodes start to process videos.

For processing, each mobile device continuously processes videos from small to large in size, while each edge device can process any video it currently has in any order as the order will not impact the completion time on the edge device. For video offloading, each time a mobile device offloads the largest video, for which it has not completed processing (*i.e.*, it is possible to offload the video that is being processed). When a mobile device is ready to offload videos (*i.e.*, it is not transmitting any video), it will broadcast an offload request to inform all the edge devices. When edge devices receive an offload request, they will add the request into a set of unhandled requests.

When an edge device is ready to receive videos, (*i.e.*, it is not receiving any video), it will determine whether to accept the received requests and reply to the accepted request. Based on the replies from edge devices, the mobile device will eventually determine to which edge device the video should be offloaded. After making the decision, the mobile device will broadcast a confirmation message to edge devices to inform them of the selected edge device and the estimated communication delay of the video, and then start transmitting the video. When other edge devices receive the message, they will mark the offload request from the mobile device as handled and then update the locally maintained information, *i.e.*, changing the location of the video from the mobile device to the edge device and add the estimated communication delay for the video. Note that the estimated communication will be replaced by the actual communication delay when the mobile device finishes the offloading and sends out other request. This process continues until all videos are processed.

An edge device needs to decide whether to accept received requests when it is ready to receive videos. An edge device, say m, which is not currently receiving a video, first calculates the completion time of each node based on its maintained information, and then calculates $\mathbf{T}$ according to (4). From the set of unhandled requests, it selects the request from the mobile device, u, that has the maximum completion time among the set. Then, edge device m calculates the completion time T_m and the increase ΔT_m if the video is offloaded to m. If $T_u = \mathbf{T}_{\max}$, edge device m will accept the offload request when $T_m < \mathbf{T}_{\max}$ and then send T_m and ΔT_m to u. If $T_u < \mathbf{T}_{\max}$, edge device m will accept the request of u only if $T_m \leq \mathbf{T}$, otherwise, m will skip the request.

A mobile device needs to decide to which edge device to offload the video based on the replies from edge devices. Mobile device u may receive multiple replies. It will choose the edge device that has the minimal completion time if the received completion times are more than $\mathbf{T}$. Otherwise, it will select the edge device whose completion time is less than $\mathbf{T}$ such that the increase in the completion time of the chosen edge device is minimal. As edge devices finish receiving an offloaded video at different timestamps, a mobile device usually receives only one reply during a short period time. Therefore, a timeout (very short, *e.g.*, a second) is set up after a mobile device receives the first reply. The timeout is used only when there are more edge devices than mobile devices or when to determine the first video to offload, where a mobile device may receives multiple replies. For all other cases, a mobile device will make decision based on the first received reply, in order to not block the offloading. Moreover, it is possible that a mobile device will not receive a response from edge devices. This is because the completion time of the mobile device is always shorter than $\mathbf{T}_{\max}$. Note that while waiting for the response from edge devices, mobile devices continuously process videos. After mobile device u selects the edge device, it will broadcast a confirmation message and then start offloading the video. When edge devices receive the message, they will mark the request as handled and update locally maintained information accordingly as discussed above. The unselected edge devices that are ready to receive videos will continuously process the



(a) u sends out an offload request (b) n replies u with T_n and ΔT_n (c) u confirms b will be sent to n

Fig. 3. Illustration of the adaptive algorithm, where m and n are edge devices, and u and v are mobile devices.

unhandled requests if the request set is not empty.

See Fig. 3 as a simple example to illustrate the adaptive algorithm. As in Fig. 3a, at time t_1, mobile device u is ready to offload videos and thus it sends out an offload request of video b to edge devices m and n. Since m is currently receiving video a, it will add the request into the set of unhandled requests. Since n is not currently receiving a video, it will calculate T_n and ΔT_n if video b is offloaded to itself based on the current transmission rate between n and u, and then send them to u as shown in Fig. 3b. When u receives the reply, it will decide to offload b to n, because it only gets one reply. Before offloading b to n, it will first send out a confirmation message as in Fig. 3c. When n receives the confirmation message, it will setup the connection to receive b. Meanwhile when m receives the message, it will mark the offload request from u as handled.

B. Discussion

Since, typically, there are more mobile devices than edge devices in the network, an edge device is most likely to decide whether to accept a request when it finishes receiving a video rather than when it receives an offload request. As the edge device selects the request of the mobile device that has the maximum completion time among the set of unhandled request, the adaptive algorithm will gradually decrease $\mathbf{T}_{\max}$ by handling each offload request until it cannot be reduced.

The confirmation message from a mobile device is designed to inform edge devices that the offload request has been handled and the estimated communication delay of the video to be offloaded, which will be used to calculate $\mathbf{T}$ at each edge device when it handles other offload requests. The communication delay is estimated based on the transmission rate before offloading each video. Since the transmission rate may vary during offloading, the actual communication delay will be different than what is estimated. However, each edge device will be notified of the completion of each video offload (by the offloading mobile device) and then the other edge devices can update their previously received estimation by the actual communication delay. Therefore, the difference between the actual completion time at each edge device and the estimated will only vary by the actual communication delay of one video. Thus, it only slightly impacts the criterion $\mathbf{T}$ and the performance of the adaptive algorithm.

As message overhead can delay video offloading, the adaptive algorithm is designed to produce messages with as little overhead as necessary. At the beginning of video processing, each node will broadcast a message including the information of locally stored videos and thus there will be $|U|$ messages. As discussed before, the edge device will most likely handle the request after receiving a video, and thus there is most likely one reply for each request. Therefore, for each offloaded video, there will be three messages, *i.e.*, request, reply and confirmation. In the worst case that all videos are offloaded to edge devices, the overall message overhead of the adaptive algorithm is $3|V| + |U|$. The small number of messages is sufficient to obtain all the information to determine video offloading. Moreover, a node needs to compute the completion time of all nodes when it (for edge devices) decides to accept the offload request or when it (for mobile devices) selects the edge device. However, the computational overhead is low, *i.e.*, $|V|$. For the worst case that all videos are offloaded, the sum of computation overhead of all nodes is only $2|V|^2$.

As edge devices can also communicate with each other, we could consider transfer of videos among edge devices to balance the workload. However, we decided against this, because video offloading among edge devices incurs additional communication delay. That means a video might be transferred multiple times before being processed and thus increase the communication delay. As a result, it might also increase the communication delay of other videos due to the constraint that each node can only send or receive one video at a time. However, the adaptive algorithm requires only one transfer for each offloaded video, and instead of balancing workload by transferring videos among edge devices, it balances the workload when offloading videos from mobile devices to edge devices.

The adaptive algorithm estimates the communication delay of each offloaded video based on the transmission rate just before offloading and makes video offloading decision in realtime. Therefore, it is more suitable for the scenarios where the transmission rate dynamically changes during video processing. Moreover, the adaptive algorithm is executed based on the interactions between the mobile device and edge device, and thus NetVision needs the corresponding component for each as will be discussed in Section VI.

VI. NetVision

We have built NetVision, a system of on-demand information query and video processing. NetVision consists of two components: NetVision edge and NetVision mobile as illustrated in Fig. 4.

NetVision edge is running on edge devices and implemented in C++ based on Linux. Inside NetVision edge, we built a simple *query system* with a GUI using Qt to issue queries and collect responses. When a query is initiated, it first sends the query to all other network nodes using a *messaging service*, which is built using Google Protocol Buffers. The information sent back from other nodes is gathered at a

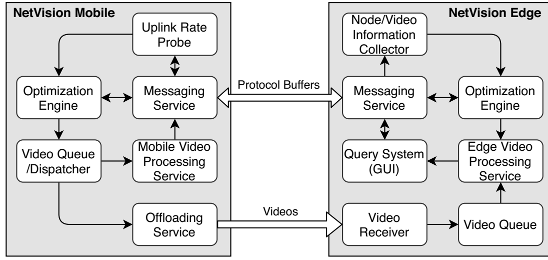
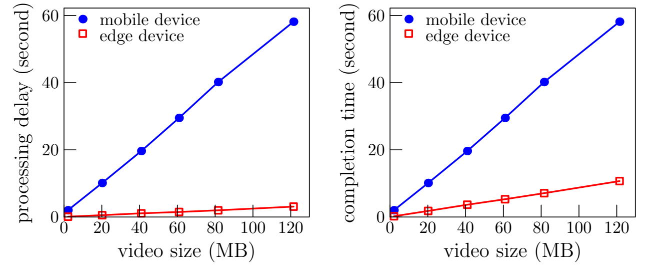


Fig. 4. Architecture of NetVision.



(a) processing delay *vs* video size (b) completion time *vs* video size

Fig. 5. Processing delay and completion time of videos with different sizes for mobile device and edge device, where videos have the resolution 1920×1080, bit rate 16Mbps, frame rate 30fps, and the transmission rate between mobile device and edge device is 16MBps.

node/video information collector. The information includes the node's processing rate, transmission rates, sizes of videos related to the query, *etc*. This information is the input for the *optimization engine*. The optimization engine adopts the greedy algorithm or adaptive algorithm for different network scenarios to determine how to process videos across the network, and then it behaves accordingly for signaling and message passing, as discussed in Section IV and V.

The *video queue* holds local videos and videos received from mobile devices. We currently implemented our video processing approach for object detection and recognition based on AlexNet [34] (a CNN) using Caffe [35] [36] (a deep learning framework) as the *video processing service* on both NetVision edge and mobile. The difference between them is that video processing on the NetVision edge can be greatly accelerated by powerful GPUs. The results of video processing across all the nodes are collected at the query system to respond to the query.

NetVision mobile is currently implemented in Java based on Android. In addition to the similar components with NetVision edge, NetVision mobile has a unique component, the *uplink rate probe*. Similar to [13], each time NetVision mobile offloads a video, it takes the opportunity to obtain an estimate of the uplink rate. When the estimate is not fresh enough, it conducts a measurement by sending 64KB data over TCP to an edge device to obtain a fresh estimate.

In our current implementation, the greedy algorithm is run only on the NetVision edge. The *optimization engine* on NetVision mobile only performs interactions with the NetVision edge for the adaptive algorithm. Moreover, the *video dispatcher* controls local video processing and video offloading according to the determined processing schedule and signaling from the NetVision edge. Since none of off-the-shelf mobile devices support GPU acceleration for deep learning, video processing on the NetVision mobile is run only on CPU.

VII. Performance Evaluation

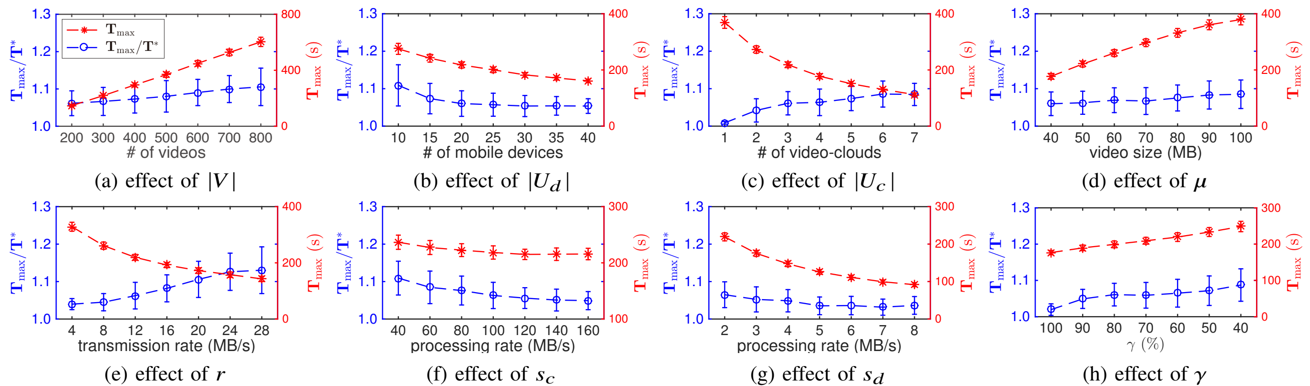
In this section, we first evaluate the proposed algorithms by extensive simulations based on the empirically gathered measurements. Then we investigate the system performance on a small testbed in stationary networks and also in mobile networks by plugging the small testbed into an emulated mobile environment.

A. Algorithm Performance

1) Processing Delay: First, we evaluate the processing delay of videos in terms of data size on mobile and edge devices. We ran our video processing approach on both tablets (Nexus 9) and an edge device implementation (Dell Precision T7500 with GeForce GTX TITAN X 12 GB GPU). We took several videos with different sizes using the tablet and processed them on both the tablet and edge device. Fig. 5a gives the comparison of the processing delay between the tablet and edge device. From Fig. 5a, we can see that GPU can greatly accelerate video processing. For a video of 60MB, the tablet takes about 30 seconds, whereas the edge device takes only about 0.5 second. Both linearly increase with the data size of videos. When taking the communication delay of videos into consideration, as shown in Fig. 5b, the completion time of processing each video (offloaded from the tablet) on the edge device is still much less than that of the tablet. Note that the specifications of videos, such as resolution, frame rate and bit rate, may affect the processing delay.

2) Greedy Algorithm vs Optimum: In order to evaluate the performance of the proposed algorithms, we setup a simulation environment. The videos are generated with different data sizes following normal distributions with different μ and σ. To capture the heterogeneity of the processing rate, the processing rates of mobile and edge devices are set uniformly and randomly to between $[\gamma s_d, s_d]$ and between $[\gamma s_c, s_c]$, respectively, where s_d denotes the maximum processing ratio for mobile devices and s_c denotes the maximum processing rate of edge devices. Also, the transmission rate between a mobile device and an edge device is set uniformly and randomly to $[\gamma r, r]$. The number of videos $|V|$, the number of mobile devices $|U_d|$, the number of edge devices $|U_c|$, r, μ, σ, γ, s_d and s_c are system parameters for simulations. We used the following default settings for the parameters: $|V| = 300$, $|U_d| = 20$, $|U_c| = 3$, $r = 12$MB/s, $\mu = 50$MB, $\sigma = 20$MB, $\gamma = 0.6$, $s_d = 2$MB/s and $s_c = 100$MB/s, where the settings of s_d and s_c correspond to the measurements in the previous section.

We evaluate the greedy algorithm and compare it with the optimum achieved by an exhaustive search in various settings. For each setting, we generate one hundred instances according to the randomness of simulation setup, including video size, processing rate, and transmission rate. The two solutions run on these instances. The performance is compared in terms of $\mathbf{T}_{\max}/\mathbf{T}^*$ to demonstrate how the greedy algorithm approximates the optimum, and the value of $\mathbf{T}_{\max}$ is also illustrated.



(a) effect of $|V|$ (b) effect of $|U_d|$ (c) effect of $|U_c|$ (d) effect of μ

(e) effect of r (f) effect of s_c (g) effect of s_d (h) effect of γ

Fig. 6. Comparison between the greedy algorithm and the optimum in terms of $\mathbf{T}_{\max}/\mathbf{T}^*$ and the value of $\mathbf{T}_{\max}$, where the default setting is $|V| = 300$, $|U_d| = 20$, $|U_c| = 3$, $\mu = 50$MB, $\sigma = 20$MB, $r = 12$MB/s, $s_d = 2$MB/s, $s_c = 100$MB/s and $\gamma = 0.6$.

Fig. 6 demonstrates the effects of system parameters on the performance of the greedy algorithm. For each evaluated parameter, all other parameters use the default settings. From Fig. 6a, we can see $\mathbf{T}_{\max}/\mathbf{T}^*$ slightly increases with the increased number of videos. When using 200 videos, the greedy algorithm is less than 10% worse than the optimum, and it is less than 20% when using 800 videos. The increase is caused by increased video offloading when there are more videos to be processed. Correspondingly, when there are more mobile devices in the network, each mobile device has fewer videos to process and thus less video offloading. Therefore, the greedy algorithm performs better as the number of mobile device increases in Fig. 6b. When there is only one edge device in the network, the greedy algorithm achieves the optimum shown in Fig. 6c. The difference rises when the number of edge devices goes up, but it tends to flatten out when the number of edge device increases further.

In Fig. 6d, the greedy algorithm performs close to the optimum in the settings with different average video sizes. Fig. 6e demonstrates the effect of transmission rates. When the transmission rate increases, mobile devices tend to offload more videos to edge devices as offloading videos costs less than before. This leads to an increased deviation between the greedy algorithm and the optimum, though both $\mathbf{T}_{\max}$ and $\mathbf{T}^*$ decrease.

The completion time of edge devices is determined by the processing delay and communication delay of videos. When the processing rate of edge devices increases, the processing delay decreases and thus the greedy algorithm performs better as in Fig. 6f. Moreover, $\mathbf{T}_{\max}/\mathbf{T}^*$ also declines when mobile devices are more computationally powerful as indicated in Fig. 6g, because fewer videos are offloaded when mobile devices have higher processing rates. The effect of the diversity of processing rates and transmission rates is captured in Fig. 6h; *i.e.*, such diversity leads to slightly increased $\mathbf{T}_{\max}$ and deviation from the optimum.

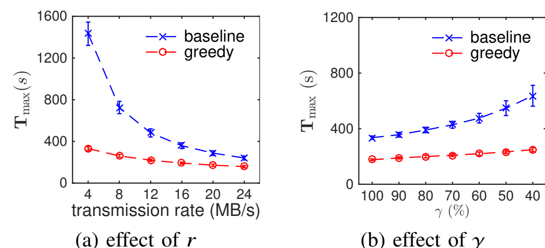
In summary, through extensive simulations, we can see that the performance of the greedy algorithm is close to the optimum in various settings (no more than 20% worse than the optimum) and it is much less than the theoretical upper bound as in (8).

3) Greedy Algorithm vs Baseline: We also compare the greedy algorithm with a *baseline* scheme that does not consider communication delay and iteratively offloads a video from the mobile device that has the maximum completion time to the edge device that has the minimum. As illustrated in Fig. 7, the greedy algorithm performs much better than the baseline. When the transmission rate increases, the impact of the communication delay on the completion time decreases and thus the difference between these two algorithms narrows, as in Fig. 7a. Moreover, the baseline is more sensitive to the increased diversity of processing rates and transmission rates as indicated in Fig. 7b. Therefore, we can conclude that the greedy algorithm that considers both processing delay and communication delay is much better than the baseline that considers only processing delay.

4) Adaptive Algorithm vs Greedy Algorithm: The adaptive algorithm is designed for the scenarios where the transmission rate varies during video processing. To model the dynamics of the transmission rate, we adopt a Markov chain [37]. Let R denote a vector of transmission rates $R = [r_0, r_1, \ldots, r_l]$, where $r_i < r_{i+1}$. The Markov chain moves at each time unit. If the chain is currently in rate r_i, then it can change to adjacent rate r_{i-1} or r_{i+1}, or remain in the current rate with the same probability. Therefore, for a given vector, *e.g.*, of four rates, the transition matrix can be defined as

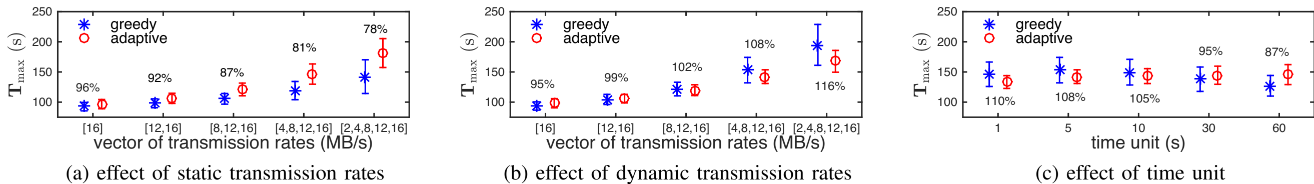
$$P = \begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \end{array} \begin{array}{c} \begin{array}{cccc} r_0 & r_1 & r_2 & r_3 \end{array} \\ \begin{bmatrix} 1/2 & 1/2 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 \\ 0 & 1/3 & 1/3 & 1/3 \\ 0 & 0 & 1/2 & 1/2 \end{bmatrix} \end{array}.$$

In the simulations, the transmission rate between a mobile device and edge device is initially set to a randomly selected



(a) effect of r (b) effect of γ

Fig. 7. Comparison between greedy algorithm and baseline in terms of $\mathbf{T}_{\max}$, where the default setting is $|V| = 300$, $|U_d| = 20$, $|U_c| = 3$, $\mu = 50$MB, $\sigma = 20$MB, $r = 12$MB/s, $s_d = 2$MB/s, $s_c = 100$MB/s and $\gamma = 0.6$.



(a) effect of static transmission rates (b) effect of dynamic transmission rates (c) effect of time unit

Fig. 8. Comparison between adaptive algorithm and greedy algorithm in terms of $\mathbf{T}_{\max}$, where the default setting is $|V| = 100$, $|U_d| = 10$, $|U_c| = 2$, $\mu = 50$MB, $\sigma = 20$MB, $s_d = 2$MB/s, $s_c = 100$MB/s, $\gamma = 0.6$, $t = 5$s, and $R = [4, 8, 12, 16]$. The percentage gives the ratio of greedy algorithm over adaptive algorithm in terms of $\mathbf{T}_{\max}$.

rate from R and it dynamically changes according to the transition matrix each time unit t. The greedy algorithm determines video offloading and transmission sequence based on the initially assigned transmission rates before processing videos. Then, the simulation runs and produces the runtime $\mathbf{T}_{\max}$ for the greedy algorithm. The adaptive algorithm runs during video processing and determines video offloading during runtime of simulations.

First, we compare the adaptive algorithm with the greedy algorithm under static transmission rates. As shown in Fig. 8a, the greedy algorithm outperforms the adaptive algorithm in various vectors of transmission rates. Note that the percentage in Fig. 8 gives the ratio of $\mathbf{T}_{\max}$ of the greedy algorithm over the adaptive algorithm. Moreover, the difference between the greedy algorithm and adaptive algorithm expands with the increased diversity of transmission rates, *i.e.*, the ratio changes from 96% to 78%. In the adaptive algorithm, edge devices can only accept the offload request after receiving previously offloaded video to adapt to the variation of transmission rate. Therefore, when a mobile device selects an edge device for offloading, the edge devices that are currently receiving videos are not considered. However, the greedy algorithm makes offloading decisions beforehand and considers every edge device at each step. Therefore, the greedy algorithm performs better under static transmission rates.

When transmission rates change dynamically, the performance of the greedy algorithm and the adaptive algorithm is shown in Fig. 8b, where the time unit $t = 5$s. When transmission rates are more stable, *e.g.*, $R = [16]$ or $[12, 16]$, the greedy algorithm performs better than the adaptive algorithm. When transmission rates are more dynamic, *e.g.*, $R = [8, 12, 16]$, $[4, 8, 12, 16]$ or $[2, 4, 8, 12, 16]$, the adaptive algorithm outperforms the greedy algorithm. Fig. 8c gives the performance comparison in terms of time unit of the Markov chain. As short time intervals produce a dynamic transmission rate during video processing, the adaptive algorithm performs better when time interval is short, and vice versa.

In summary, as expected, the greedy algorithm is preferred for the scenarios where the transmission rate is steady, while the adaptive algorithm is more suitable for the scenarios where the transmission rate is dynamic.

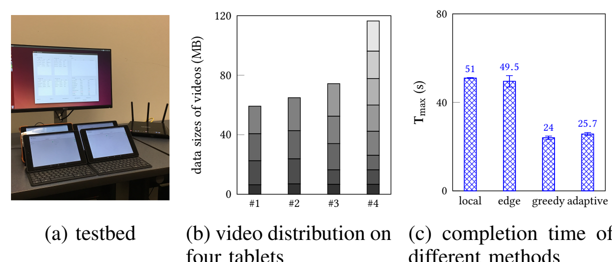
B. System Performance

1) Stationary Network: We deployed NetVision on a small testbed that includes four Nexus 9 tablets and the edge device implementation which are connected through a WiFi router which supports 802.11b/g/n, as shown in Fig. 9a. In order to obtain different uplink data rates for each tablet, the four tablets are spread with different distances to the WiFi router and are kept stationary. Queries are issued from the edge device to the tablets, targeting different objects. To respond to a query, all videos stored on the testbed will be processed. For the video processing implementation, frames are extracted from a video and then object detection is performed on the frames.

In the experiments, we configured NetVision to run different methods, which include the greedy algorithm, the adaptive algorithm, *local* (videos are processed locally), and *edge* (all videos are offloaded to the edge device for processing). Experiments are performed on a small set of videos (21 clips with a bit rate of about 16Mbps and frame rate 30fps, average size 16 MB). The sizes and the distribution of the videos are illustrated in Fig. 9b. The distribution of the 21 video clips on the tablets are generated following a normal distribution.

We ran the experiment 10 times (issued 10 queries) using each method and measured the query response time, which is the time elapsed from the issue of a query to when all processing results are collected at the edge device. We can see, from Fig. 9c, both the greedy algorithm and adaptive algorithm greatly outperform other two. The deviation of *edge* is larger than others, because all videos are offloaded to the edge device. This also implicitly indicates the fluctuation of uplink data rates during video offloading. The greedy algorithm, however, still outperforms the adaptive algorithm. This shows that the greedy algorithm can tolerate a small fluctuation of uplink data rates.

Moreover, the greedy algorithm offloads more videos than the adaptive algorithm in the experiments, as shown in Table I. The difference comes from the different strategies they adopt to determine the offloading. The greedy algorithm determines an offloaded video from a subset of videos a mobile device has, while the adaptive algorithm always chooses to offload the largest video from a mobile device. Therefore, the greedy algorithm usually offloads more videos than the adaptive algorithm. However, offloading more videos does not necessarily



(a) testbed (b) video distribution on four tablets (c) completion time of different methods

Fig. 9. Experiment setup, video distribution, and performance of different methods on the testbed.

TABLE I
VIDEOS PROCESSED AT DIFFERENT LOCALITIES ON TESTBED

	On the Edge	Locally
Greedy Algorithm	7	14
Adaptive Algorithm	4/5	17/16

mean better performance, as it is affected by dynamics of the uplink data rate, which we will discuss in Section VII-B2. In addition, the algorithms may offload a different number of videos during each run (*e.g.*, the adaptive algorithm offloaded 4 or 5 videos to the edge device in the experiments), which mainly depends on the uplink data rate at runtime.

Based on the experimental results on the testbed, we conclude that although the uplink data rate fluctuates in stationary networks, the greedy algorithm obtains better performance than the adaptive algorithm.

2) Mobile Network: In order to evaluate NetVision in mobile networks (featured with more dynamic uplink data rates), we setup an emulation environment using CORE [38] and EMANE [39]. The basic idea is to let real data traffic between the tablets and edge device go through the emulated mobile environment, and thus investigate the performance of NetVision in mobile networks.

Emulation Setup. As depicted in Fig. 10, we first connect each tablet to a WiFi router via the wireless link, and wire each WiFi router to the workstation. On the workstation, we setup a virtual machine (VM), which is bridged with the four physical network interfaces connected with the WiFi routers. The emulator CORE is running on the VM. In CORE, we use the physical interface tool (RJ45) to connect each network interface of the VM. By these configurations, the four tablets are connected into CORE and represented by four wireless nodes in the emulation. Similarly, the edge device (workstation) is connected into CORE via a virtual network interface between the workstation and VM, also represented by a wireless node in the emulation.

Using the configurations above, all network traffic between the tablets and edge device goes through CORE. Further, we setup a wireless network of the five nodes in CORE, and we use EMANE to emulates physical and data link layers using its pluggable PHY and MAC models. Currently, the MAC layer is configured with 802.11b/g, which is the WLAN mode with the highest data rate that EMANE can support, and the unicast rate is set to 54Mbps. The PHY layer is configured with 20MHz bandwidth at the frequency of 2.347GHz, and other parameters are set using default values.

The dimension of the emulation area is $200{\times}200m^2$, the

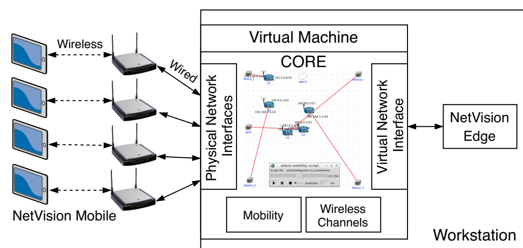


Fig. 10. Emulation Setup. The tested is plugged into the emulation. All network traffic between NetVision edge and NetVision mobile goes through the emulated mobile network.

TABLE II
MOBILITY SETTINGS

	Random Waypoint	Random Walk	Manhattan Grid
Area		$200{\times}200\ m^2$	
Duration		240 seconds	
Min/Max Speed		5/10 m/s	
Max Pause		10 seconds	
Grids			10×10

node of the edge device is statically placed at the center, and all other four nodes are mobile. Therefore, the maximum possible distance between a mobile node and the edge device is about 140m, which matches the approximate outdoor range of 802.11b/g.

To add mobility into the emulation, we use BonnMotion [40] to generate movements for the four mobile nodes. We consider three mobility models, which are *random waypoint*, *random walk*, and *Manhattan grid*. The parameter settings of BonnMotion to generate these movements are shown in Table II. The movement duration is set to 240s, which means the movement is repeated every 240 seconds. However, as we will see, processing all the videos takes much less time than 240 seconds, and thus nodes can finish processing videos during one run. The min/max node speed is set to 5/10 m/s, and the maximum pause is 10 seconds. For Manhattan grid, the area is divided into 10×10 grids.

In the emulation, to eliminate the effect of the real wireless link between a tablet and WiFi router on the path between the tablet and edge device, we put the pair of a tablet and WiFi router together and then place each pair in a different room to avoid interferences. By doing so, we can get a higher throughput for the real wireless link and thus make the data rate between a tablet and the edge device bottlenecked at the emulated wireless link between the mobile node and edge node in CORE. In addition, the WiFi router supports 802.11b/g/n, while the WLAN in CORE only supports 802.11b/g. Therefore, overall, the wireless link in CORE regulates the data rate between a tablet and the edge device, and hence node mobility in CORE can generate a dynamic data rate between the tablet and edge device.

Experiments. Similar to the experiments in Section VII-B1, we configured NetVision to run different methods for the experiments. For each method under each mobility pattern, we issued 10 queries from the edge device and then measured the query response time. The results are shown in Fig. 11. As the data rate between a mobile device and the edge device is low in emulation, processing all the videos at the edge device takes very long time (*i.e.*, several minutes). For clarity of illustration, we do not include it in Fig. 11. Moreover, since the video distribution on the tablets remains the same as in Fig. 9b, processing all videos locally takes the same time, compared

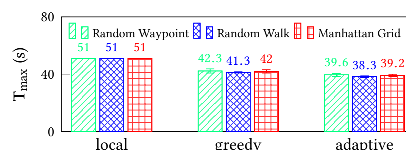


Fig. 11. Comparison among local, greedy algorithm, and adaptive algorithm in terms of $\mathbf{T}_{\max}$ in emulation under different mobility patterns.

to Fig. 9c.

We can see, from Fig. 11, the adaptive algorithm outperforms the greedy algorithm under all the three mobility patterns. Moreover, the mobility patterns only slightly affect the performance. Compared to Fig. 9c, the query response time increases more than 50% for both algorithms, because the uplink data rate between the mobile device and edge device in emulation is lower than in the experiment on the testbed. Due to the lower uplink data rate, fewer videos are offloaded, as depicted in Table III for both the greedy algorithm and adaptive algorithm, compared to Table I. Although the greedy algorithm still offloads more videos than the adaptive algorithm, its performance is worse than the adaptive algorithm. This is because the greedy algorithm makes the offload decisions beforehand. However, due to the dynamics of uplink data rates, these decisions may not be optimal after the change of uplink data rates. The offloaded videos may take much longer than expected to arrive at the edge device. Therefore, offloading more videos does not necessarily result in better performance. On the contrary, the adaptive algorithm makes the offload decision at runtime after each offloading is completed. Therefore, it is less sensitive to the change of data rates.

Based on the experimental results in emulation, we conclude that in mobile networks, where the uplink data rates are more dynamic, the adaptive algorithm obtains better performance than the greedy algorithm.

In summary, the experiments in stationary and mobile networks verify the advantages of the greedy algorithm and adaptive algorithm over other methods. By adopting these two algorithms, NetVision can optimize on-demand video processing in different network scenarios.

VIII. Discussion

NetVision is designed to process videos on-demand across a wireless network that consists of mobile and edge devices for information retrieval using deep learning (*i.e.*, CNNs). NetVision works as a distributed computing platform to optimize the query response time. Currently, NetVision employs AlexNet and Caffe for deep-learning based video processing, but it is compatible with any deep learning model and framework. Moreover, NetVision can be easily adapted to mobile GPUs whenever they are available for acceleration of deep learning on mobile devices. It is worth noting that although mobile GPUs can accelerate the computing of deep learning, the computational capability of mobile GPUs is still far behind powerful workstation GPUs (*e.g.*, Tegra K1 *vs* TITAN Xp). Therefore, the gap of computational capability between the mobile device and edge device still exists and thus NetVision can still provide improved performance in such networks.

Limitations. NetVision currently has several limitations. NetVision only supports one query at a time. Supporting simultaneous queries is challenging (as it changes the problem formulation) and will be our future work. Energy consumption is also important for NetVision, especially when mobile devices are energy-constrained. We will also consider the joint optimization of query response time and energy in future work. In wireless networks, missed messages and connectivity disruptions happen often. How to quickly cover from these circumstances is also very important, which needs careful systematic design and will also be our future work.

TABLE III
Videos Processed at Different Localities in Emulation

	On the Edge	Locally
Greedy Algorithm	3	18
Adaptive Algorithm	2	19

IX. Conclusion

In this paper, we designed NetVision, a system to perform distributed video processing across a wireless network to answer queries and optimize the query response time. We formulated the processing scheduling problem, which is a strongly NP-hard problem. To deal with this, we designed a greedy algorithm with bounded performance. To handle the dynamics of the transmission rate between mobile and edge devices, we further proposed an adaptive algorithm. We built and deployed NetVision on a small testbed. Based on the empirically gathered measurements, we first performed simulations to extensively evaluate the proposed algorithms. Results show that the performance of the greedy algorithm is close to the optimum and much better than other methods, and the adaptive algorithm performs better with more dynamic transmission rates. We also performed experiments on the testbed to examine the realized system performance in stationary networks. We further built an emulation environment for mobile networks. By plugging the testbed into this environment, we further confirmed the benefit of NetVision in mobile networks.

Acknowledgment

This work was supported in part by Network Science CTA under grant W911NF-09-2-0053, NSF China under grant 61872009, and Hikvision. A preliminary version of this work appeared in the Proceedings of IEEE ICNP 2016 [41].

References

[1] Y. Jiang, X. Xu, P. Terlecky, T. Abdelzaher, A. Bar-Noy, and R. Govindan, "Mediascope: selective on-demand media retrieval from mobile devices," in *IPSN*, 2013.

[2] T. Yan, V. Kumar, and D. Ganesan, "Crowdsearch: exploiting crowds for accurate real-time image search on mobile phones," in *MobiSys*, 2010.

[3] C. Qin, X. Bao, R. Roy Choudhury, and S. Nelakuditi, "Tagsense: a smartphone-based approach to automatic image tagging," in *MobiSys*, 2011.

[4] Y. Wang, W. Hu, Y. Wu, and G. Cao, "Smartphoto: a resource-aware crowdsourcing approach for image sensing with smartphones," in *MobiHoc*, 2014.

[5] Y. Hua, H. Jiang, and D. Feng, "Real-time semantic search using approximate methodology for large-scale storage systems," in *INFOCOM*, 2015.

[6] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision," in *MobiSys*, 2013.

[7] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *MobiSys*, 2011.

[8] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan, "Scalable crowd-sourcing of video from mobile devices," in *MobiSys*, 2013.

[9] P. Jain, J. Manweiler, A. Acharya, and K. Beaty, "Focus: clustering crowdsourced videos by line-of-sight," in *SenSys*, 2013.
[10] Z. Chen, W. Hu, K. Ha, J. Harkes, B. Gilbert, J. Hong, A. Smailagic, D. Siewiorek, and M. Satyanarayanan, "Quiltview: a crowd-sourced video response system," in *HotMobile*, 2014.
[11] F. Chen, C. Zhang, F. Wang, and J. Liu, "Crowdsourced live streaming over the cloud," in *INFOCOM*, 2015.
[12] Y. Wu and G. Cao, "Videomec: A metadata-enhanced crowdsourcing system for mobile videos," in *IPSN*, 2017.
[13] 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 *MobiSys*, 2010.
[14] 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 *INFOCOM*, 2012.
[15] W. Gao, Y. Li, H. Lu, T. Wang, and C. Liu, "On exploiting dynamic execution patterns for workload offloading in mobile cloud applications," in *ICNP*, 2014.
[16] F. Hao, M. Kodialam, T. Lakshman, and S. Mukherjee, "Online allocation of virtual machines in a distributed cloud," in *INFOCOM*, 2014.
[17] Q. Wang and K. Wolter, "Reducing task completion time in mobile offloading systems through online adaptive local restart," in *ICPE*, 2015.
[18] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," in *INFOCOM*, 2015.
[19] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *CVPR*, 2016.
[20] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *ICLR*, 2016.
[21] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *IPSN*, 2016.
[22] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *MobiCom*, 2016.
[23] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *MobiCom*, 2018.
[24] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *MobiCom*, 2018.
[25] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *SenSys*, 2015.
[26] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *MobiCom*, 2015.
[27] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *MobiSys*, 2016.
[28] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," in *SEC*, 2018.
[29] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele, "Thermal-aware global real-time scheduling on multicore systems," in *RTAS*, 2009.
[30] A. Das, A. Kumar, and B. Veeravalli, "Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 869–884, 2016.
[31] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *ICCV*, 2015.
[32] J. K. Lenstra, A. H. G. R. Kan, and P. Brucker, "Complexity of machine scheduling problems," *Annals of Discrete Mathematics*, vol. 1, pp. 343–362, 1977.
[33] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *EuroSys*, 2011.
[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.
[35] "Caffe," http://caffe.berkeleyvision.org/.
[36] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *MM*, 2014.
[37] A. Fu, P. Sadeghi, and M. Médard, "Dynamic rate adaptation for improved throughput and delay in wireless network coded broadcast," *IEEE/ACM Transactions on Networking*, vol. 22, no. 6, pp. 1715–1728, 2014.
[38] "CORE," https://www.nrl.navy.mil/itd/ncs/products/core.
[39] "EMANE," https://www.nrl.navy.mil/itd/ncs/products/emane.
[40] "BonnMotion," https://sys.cs.uos.de/bonnmotion/.
[41] Z. Lu, K. S. Chan, R. Urgaonkar, and T. La Porta, "On-demand video processing in wireless networks," in *ICNP*, 2016.

Zongqing Lu is an Assistant Professor in the Department of Computer Science, Peking University. He received the B.S. and M.S. degrees from Southeast University, China, and the Ph.D. degree from Nanyang Technological University, Singapore, 2014. Prior to joining Peking University in 2017, he was a postdoc in the Department of Computer Science and Engineering, Pennsylvania State University. His research interests fall at the intersection of distributed systems and machine learning.

Kevin Chan is research scientist with the Computational and Information Sciences Directorate at the U.S. Army Research Laboratory. Previously, he was an ORAU postdoctoral research fellow at ARL. His research interests are in network science and dynamic distributed computing, with past work in dynamic networks, trust and distributed decision making and quality of information. He has been an active researcher in ARL's collaborative programs, the Network Science Collaborative Technology Alliance and Network and Information Sciences International Technology Alliance. Prior to ARL, he received a PhD in Electrical and Computer Engineering (ECE) and MSECE from Georgia Institute of Technology. He also received a BS in ECE/EPP from Carnegie Mellon University.

Rahul Urgaonkar is an Operations Research Scientist with the Modeling and Optimization group at Amazon. Previously, he was with IBM Research where he was a task leader on the US Army Research Laboratory (ARL) funded Network Science Collaborative Technology Alliance (NS CTA) program. He was also a Primary Researcher in the US/UK International Technology Alliance (ITA) research programs. His research is in the area of stochastic optimization, algorithm design and control with applications to communication networks and cloud-computing systems. Dr. Urgaonkar obtained his Masters and PhD degrees from the University of Southern California and his Bachelors degree (all in Electrical Engineering) from the Indian Institute of Technology Bombay.

Shiliang Pu received the Ph.D. degree in applied optics from the University of Rouen, Mont-Saint-Aignan, France, in 2005. He is currently the Executive Vice Director of the Research Institute with Hikvision, Hangzhou, China. He is also responsible for the company's technology research and development work on video intelligent analysis, image processing, coding, and decoding. His current research interests include image processing and pattern recognition.

Thomas La Porta is the Director of the School of Electrical Engineering and Computer Science at Penn State University. He is an Evan Pugh Professor and the William E. Leonhard Chair Professor in the Computer Science and Engineering Department. He received his B.S.E.E. and M.S.E.E. degrees from The Cooper Union, New York, NY, and his Ph.D. degree in Electrical Engineering from Columbia University, New York, NY. He joined Penn State in 2002. He was the founding Director of the Institute of Networking and Security Research at Penn State. Prior to joining Penn State, Dr. La Porta was with Bell Laboratories where was the Director of the Mobile Networking Research Department. He is an IEEE Fellow and Bell Labs Fellow. He also won two Thomas Alva Edison Patent Awards. Dr. La Porta was the founding Editor-in-Chief of the IEEE Transactions on Mobile Computing. He has published numerous papers and holds 39 patents.