LinkShare: Device-Centric Control for Concurrent and Continuous Mobile-Cloud Interactions

Bo Hu
Yale University
New Haven, USA

Wenjun Hu
Yale University
New Haven, USA

ABSTRACT

Mobile applications have become increasingly sophisticated. Emerging cognitive assistance applications can involve multiple computationally intensive modules working continuously and concurrently, further straining the already limited resources on these mobile devices. While computation offloading to the edge or the cloud is still the de facto solution, existing approaches are limited by intra-application operations only or edge-/cloud-centric scheduling. Instead, we argue that operating system level coordination is needed on the mobile side to adequately support the prospects of multi-application offloading. Specifically, both the local mobile system resource and the network bandwidth to reach the cloud need to be allocated intelligently among concurrent offloading jobs.

In this paper, we build a system-level scheduler service, LinkShare, that wraps over the operating system scheduler to coordinate among multiple offloading requests. We further study the scheduling requirements and suitable metrics, and find that the most intuitive approaches of minimizing the end-to-end processing time or earliest-deadline first scheduling do not work well. Instead, LinkShare adopts earliest-deadline first with limited sharing (EDF-LS), that balances real-time requirements and fairness. Extensive evaluation of an Android implementation of LinkShare shows that adding this additional scheduler is essential, and that EDF-LS reduces the deadline miss events by up to 30% compared to the baseline.

ACM Reference Format:
Bo Hu and Wenjun Hu. 2019. LinkShare: Device-Centric Control for Concurrent and Continuous Mobile-Cloud Interactions. In *The Fourth ACM/IEEE Symposium on Edge Computing (SEC 2019), November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318216.3363303

SEC 2019, November 7–9, 2019, Arlington, VA, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363303

1 INTRODUCTION

Mobile-cloud computing has been a cornerstone of the mobile landscape in the last decade. This ranges from offloading energy-hungry computation to the cloud, speeding up computation using more computing power on the cloud, leveraging more storage space on the cloud, to benefiting from more elaborately trained models for learning based applications.

Existing mobile-cloud interactions tend to be cloud-centric, since the mobile side is less powerful. Consider, for example, typical workloads involving computation offloading to the cloud. While different applications might interact with distinct backend servers, implicitly it is assumed that there is only one active mobile-cloud interaction session at a time.

However, mobile applications are becoming increasingly sophisticated, enhancing our interaction with the environment or providing cognitive assistance (Section 2). Some scenarios might involve multiple concurrent applications or modules working together (Google Tango [3], Gabriel [22] and DeepEye [33]), where these modules are individually computationally intensive and require computation offloading. Worse, these applications embody a vision of continuous operations, further straining the already limited resources on the mobile devices. These represent canonical examples of multiple concurrent and continuous mobile-cloud sessions.

Although there has been a multitude of computation offloading work over the past fifteen years [13, 15, 17–21, 28–30, 32, 35, 36, 41], existing approaches are limited by intra-application operations only or cloud-centric scheduling.

Instead, the emerging multi-application scenarios can exhibit a heterogeneous network and server execution model, involving different server backends. This suggests different perceived network latencies and server processing capabilities across applications. Further, they need to share the wireless interfaces, which can only be controlled at the device. Cloud-centric management is no longer sufficient in this concurrent mobile-cloud interaction paradigm. We need to shift the control to a device-driven paradigm. In other words, some operating system level coordination is needed on the mobile side to adequately support the prospects of multi-application offloading.

An analysis of the current mainstream offloading mechanisms suggests that the main consideration is scheduling the network transfer to enable remote computation. In particular, the bottleneck is often the transfer along the first wireless hop. This is important given the advent of mobile edge computing promising to bring the remote server closer. However, the combination of application workloads, network transfer time, and the server processing time complicates the picture. Most of the canonical applications have soft real-time constraints for user interaction. This requires a balancing act between minimizing the end-to-end processing time and meeting deadlines.

For example, the most intuitive approaches of minimizing the end-to-end processing time (Shortest Job First, or SJF) and earliest deadline first (EDF) scheduling both turn out to be inadequate. The former (SJF) gives no guarantee about meeting deadlines and can cause fairness issues (and even starvation). This is because minimizing processing times can penalize a job as a result of *another job* using a less powerful backend. EDF, on the other hand, cannot handle heavy-tailed network transfer time distribution. Section 3 analyzes this in detail and suggests adding limited sharing to EDF (EDF-LS). Essentially, when we detect a large network transfer, we serve the next two jobs queued in a round-robin fashion.

Following the above study, we build a system-level scheduler service, LinkShare (Section 4) and implement it on Android (Section 4.3). LinkShare wraps over the operating system scheduler to coordinate among multiple offloading requests, incorporating the EDF-LS algorithm.

Using benchmark applications representing face recognition, optical character recognition, speech recognition, and license plate recognition, we find that this additional scheduling decision is essential, and EDF-LS reduces the deadline miss rate by up to 30% compared to the baseline EDF (Section 5). When the workload is light, EDF-LS hardly incurs penalty from sharing, and in fact often outperforms EDF. Even if the network bandwidth increases, the deadline concern does not disappear, especially alongside improved server capability.

Although the specific system is motivated by multiple mobile offloading jobs, the issues discussed and the system architecture are generic to concurrent mobile-cloud interactions. In fact, any concurrent, inter-application network-bound requests could benefit from the LinkShare service. More broadly, LinkShare as a framework is also applicable when the "offloading" destination is an edge server or a nearby device instead. It is also amenable to other scheduling algorithms as warranted by the application requirements. We believe LinkShare represents a first step towards coordinating an IoT ecosystem tethered to the mobile device.

In summary, this work studies concurrent and continuous mobile-cloud interactions involving different server backends and argues for a device-centric control mechanism. Specifically, our contribution is three-fold.

First, we analyze the scheduling complexity arising from concurrent offloading workloads. Our study points to the need to balance between meeting application deadlines and avoiding blocking heavy workloads. We adopt limited sharing in addition to Earliest Deadline First.

Second, we build a general framework as a system-level service, LinkShare, that extends the operating system scheduler for concurrent inter-application network-bound requests and incorporates the above scheduling algorithm.

Third, extensive evaluation of an Android implementation of LinkShare confirms the importance of this additional scheduler and shows that EDF-LS achieves the balancing goal.

2 MOTIVATION

2.1 Emerging Application Scenarios

Mobile Augmented Reality (AR) Games. Mobile AR games have been showing their potential on mobile platforms. One example is the dominoes game included in the Google Tango [3] project. Instead of getting a set of dominoes at hand, users can place virtual dominoes in the real world and arrange them through their mobile phones. It requires the application to understand the user's surroundings, remember the exact locations of the dominoes already placed, and display the virtual dominoes on the screen in the meantime. To enable all these functions, motion tracking, depth perception, area learning and video rendering modules are needed. The first two modules help with understanding the environment, while area learning helps with remembering the previous operations and video rendering concurrently displays the virtual dominoes. All these modules coordinate with one another to make the dominoes game work correctly.

Wearable Cognitive Assistance. Wearable devices for cognitive assistance have been suggested for more than a decade [37]. More recently, Gabriel [22] provides interactive cognitive assistance using Google Glass to help people suffering from cognitive decline, such as those with Alzheimer's disease. The patients are often unable to remember the names of friends or remember to perform daily tasks. When looking at a person that the user might know, the assistant will tell the user the name of the person immediately. When looking at his/her plants, it will remind him/her to water them. These two scenarios require face recognition and object recognition respectively. As we cannot predict when the user may meet with a friend or walk around his/her garden, these two modules must run continuously and simultaneously.

Smart Video Surveillance. Smart home has become a popular concept in recent years, and smart video surveillance is a key technology to ensure the security in smart homes. According to [24], the key to security is situation awareness. The system needs to keep track of "who are the people in a space?"

and "what are the subjects in the space doing?". To answer these questions, learning techniques like face recognition, object recognition, activity tracking are widely employed. To quote one report [1], “one of the biggest factors assisting market growth" is the real-time access and monitoring capability. All the above modules need to run concurrently to generate comprehensive real-time alerts.

Observations. Common across these examples, they are all computation intensive and latency sensitive, and have a continuous flavor. More important, there are multiple modules either within a single application or across multiple applications. Take the face recognition module as an example, the average execution time on Google Glass, a Samsung Galaxy Android smartphone, and a remote server are 2912, 537, and 41 ms respectively [14]. To maintain a high refresh rate for an interactive face recognition application, it is impractical for all these modules to run entirely on the local device.

2.2 Limited Execution Model Currently

The de facto approach to handling computation-intensive mobile applications is to offload computation to the cloud (or, more recently, to the edge). However, the current execution model typically focuses on only one computation-intensive workload running at a time, and most optimizations are applied within a single application. This single-workload model does not suit the emerging applications described above, whether commercialized applications like Google Tango (multiple modules running concurrently within the same application), or research prototypes such as Gabriel (multiple applications running concurrently within a single device), DeepEye [33](multiple deep vision models running concurrently), and MCDNN [25] (multiple deep neural network applications running concurrently, sharing the same library).

Instead, system-level cross-application coordination support is needed to manage simultaneous execution of these applications, but existing approaches are still limited to specific scenarios and do not address concurrently offloading to different backend servers. Tango takes a hardware approach without offloading. MCDNN operates at the DNN library level, aiming to process jobs locally as much as possible. DeepEye limits its use cases to multiple deep vision applications. Gabriel is currently designed for concurrent mobile applications supported by a common backend server and takes a cloud-centric approach.

2.3 Towards device-centric scheduling

The entire offloading job involves local processing and/or data transfer to the remote server followed by remote processing. Therefore, we need to consider sharing and scheduling for each component.

With concurrent offloading, different apps may use distinct backend servers, due to either technical or business-related reasons. For an example of the latter, consider the two mainstream public cloud platforms: Amazon EC2 and Google Compute Engine (GCE). Price-wise, for the same server computation capacity, GCE is cheaper than Amazon EC2. However, the EC2 deployment covers more datacenters around the world (22 total, 7 in the US) compared to GCE (21 total, 6 in the US). Therefore, a cost conscious developer who only needs to provide regional service may opt for a server on GCE, while another developer aiming for a better service coverage worldwide may favor EC2.

We now face a new situation where the control needs to be shifted to the mobile device. Since the concurrent applications and their backend servers are likely heterogeneous, such device-driven control really matters.

Complicating the picture even further, applications have different service requirements, manifested in deadlines for real-time user interactions [40]. On-device scheduling can make a huge difference. Even a simple reordering of the offloading requests could dramatically improve the performance of one module without hurting the others. We will study specific examples next.

3 SCHEDULING OFFLOADING JOBS

3.1 What to schedule on-device

The anatomy of an offloading job. The processing of a job involving offloading includes several components: potentially some local processing on the mobile device, transferring the data to the cloud, and processing on the remote server(s). The second component can be further split into transfer on the wireless access link and on the wired path to the server. The split between local and remote processing is specific to the offloading mechanism adopted in the application.

Within these components, the first component is traditionally managed by the operating system scheduler and access to the wireless link is also initiated by the mobile device. The remaining components, transfer on the wired path and the server processing, are beyond the control of the mobile device. Therefore, the main scheduling decision in our context concerns sharing the wireless link between concurrent offloading workloads.

Offloading mechanism and scheduling. Existing main-stream offloading mechanisms are method-based [15, 17] or, similarly, based on small execution units [36]. In other words, the whole application process can be divided into many execution units, and each unit is run either locally or remotely.

Consider face recognition. The recognition function is treated as a standalone method in MAUI [17]. Invoking this method on a single frame from the video feed comprises an

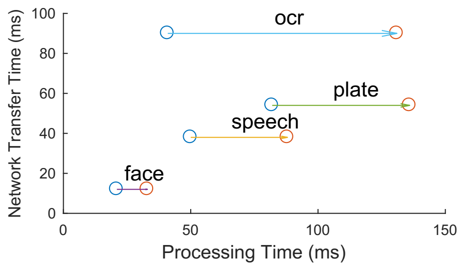


Figure 1: Processing Time Breakdown

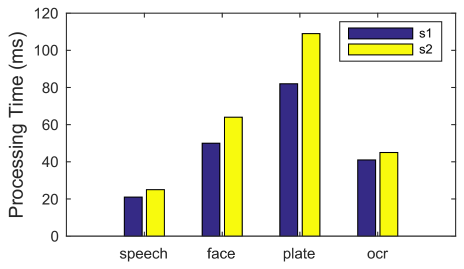


Figure 2: The impact of server capacity

execution unit. When this application operates on a video feed, each frame is processed in its entirety either locally or remotely. Therefore, we can decouple the decision between local vs remote and how to order the remote ones.

To summarize, an offloading scheduler needs to make two decisions: how many jobs to process locally (then leave to the OS to schedule), and how to order the offloading jobs. Since the first problem has been studied extensively [13, 18, 19, 28, 30, 32, 35, 41], we focus on the second problem in this paper.

3.2 The complexity of offloading

The offloading performance is affected by multiple factors. The server capability determines the remote processing time. The server's physical location along with the network conditions affect the data transfer time. In particular, wireless links are susceptible to interference and multipath fading.

Figure 1 shows the processing time breakdown of our benchmark applications, assuming a 10 Mbps network upload speed. For each application, the point on the left shows the remote processing time, while the point on the right shows the end-to-end processing time (i.e., with data transfer time added to the remote processing time). The timing results are averaged over 500 runs per application. The application and experiment setups are detailed in Section 5.1.

We can see that plate recognition is clearly computation bound, whereas optical character recognition (OCR) is network bound. While face recognition, plate recognition and OCR are expected to operate on the same image frame[1], face recognition does not need the entire frame, locally or remotely. The built-in face detection function in the vision library can narrow down the region of interest before handing the recognition application a subset of the frame containing these regions only. Therefore, the input size for face recognition is much smaller than the entire frame.

We then measure the processing times on two different servers, S_1 and S_2. S_1 has an Intel Core i5-4570 processor (Quad Core, 6 MB Cache, 3.2 GHz) and 8 GB 1600 MHz DDR3 Memory. This is the same server used for the previous figure and in our evaluation throughout. S_2 is a workstation with an Intel Core i5-4430 Processor (Quad Core, 6 MB Cache, 3.0 GHz) and 16 GB 1600 MHz DDR3 Memory. We randomly select 500 data samples from the input dataset and Figure 2 shows the processing times of the benchmark applications on the two servers.

Although the server specifications appear similar, the processing time for face recognition and plate recognition differ by almost 25%. For face recognition, the processing time difference is comparable to the data transfer time.

Since the processing times across applications differ anyway, offloading the same application to distinct servers is analogous to offloading different applications to the same server. Therefore, we implicitly capture the offloading complexity mentioned above by studying a suite of applications.

One issue remains, however. The benchmark applications are representative of the continuous applications described in Section 2, for which there is an implicit deadline requirement for processing to ensure smooth user interaction. Meeting deadlines can be more important than optimizing for the end-to-end processing time, though the latter is an essential contributing factor to the former. We next explore suitable scheduling metrics to capture desirable performance goals.

3.3 Scheduling metrics and algorithms

Recall that our main goal is to schedule data transfer and hence order the offloading requests. Scheduling is a well-studied topic. Common metrics and criteria include fairness, minimizing job time (in our case, this means the shortest network transfer time, or minimal network queuing delay), and deadline-awareness for real-time jobs. We consider these metrics and the associated standard algorithms in turn. Note that there are multiple definitions for fairness. Since we are concerned with a wireless link, we use the common notion of slot-based fairness, which is also consistent with round-robin in the OS.

[1]As explained in Section 5.1, we cannot train in real time, so the applications are fed with synthetic video feeds. The input data for plate recognition and OCR are different, hence the transfer times differ.

Fair sharing vs serial execution. Perhaps one of the simplest scheduling approaches is to serve all jobs in parallel, i.e., serving the data transfers from all offloading jobs simultaneously using technique like parallel TCP [23]. Each application can get the same share of the bandwidth, which is referred to as fair sharing. Instead, we argue for sequential offloading (as adopted by First-Come-First-Serve, or FCFS) instead of fair sharing.

Under blocking-based offloading scenarios, a remote execution process cannot make progress until it has received all the data. Consider a scenario where the concurrent offloading tasks are from speech recognition and plate recognition, the former arriving first. Figure 3a compares the end-to-end processing time for each task (the sum of the network transfer time and the computation time) between fair sharing and FCFS, the latter clearly outperforming the former. This is mainly because fair sharing incurs higher queuing delay for each task and hence a much higher average network transfer time of 84 ms, 30% more than that of FCFS. This in fact reconfirms that fairness often does not align with optimal performance [39].

Therefore, *serial execution can achieve better average application processing times by reducing the average network transfer times.*

To pre-empt or not to. The next question is whether to preempt any jobs. Devices are not limited to relatively powerful mobile phones, and could be embedded devices with limited memory and high context-switching overhead. For these situations, the cost of moving state between memory and storage to pre-empt one process and bring in the next could be prohibitive. Therefore, *we focus on non-pre-emptive scheduling techniques that can be applied across a range of devices.*

Processing throughput vs deadline. To optimize for performance, perhaps the most intuitive approaches are minimizing the job completion time and meeting job deadlines (in our case, bounding the *end-to-end processing time*). Correspondingly, the canonical scheduling algorithms are Shortest Job First (SJF) and Earliest Deadline First (EDF).

In particular, it might appear natural to minimize the end-to-end processing time, since this could be directly achieved with SJF and *appear* to lead to meeting deadlines. However, this approach can unintentionally penalize one offloading job as a result of *another job* using a less powerful backend. Therefore, the SJF decision should be based on minimizing the network transfer time alone. For example, if we offload OCR and plate recognition concurrently, according to the processing time breakdown in Figure 1, the scheduler based on the shortest *end-to-end processing time* will schedule plate recognition first, and produce 179 ms average end-to-end processing time, while that of the shortest *network transfer time* will choose the reverse order of offloading, which will produce 160 ms end-to-end processing time.

Consider the scenario of concurrent speech recognition and plate recognition jobs again. Figure 3b shows that, regarding the average end-to-end processing time, SJF outperforms EDF by only 6%. However, if both had a 150 ms processing deadline requirement, both meet the deadlines under EDF, while only plate recognition manages so when using SJF.

Therefore, *the scheduling decision should primarily depend on the network transfer time instead of the end-to-end processing time; further, if the main scheduling concern is deadline-awareness, minimizing the average processing time with SJF does not work well even in the simplest two-app scenarios.* This is despite the fact that a low end-to-end processing time is essential to meet the deadline.

The tale of the tail. So far, EDF appears to be a winner. However, one consequence of its non-pre-emptive nature is tail latency. EDF is not tail-robust [34], i.e., its performance will suffer when the network transfer times of different applications follow a heavy-tail distribution.

Now consider a face recognition job and an OCR job in the queue. Figure 3c shows that both meet their deadlines (150 ms, marked by the vertical dashed line) under both fair sharing and EDF. However, the average end-to-end performance of face recognition suffers under EDF. This is mainly because the network transfer time of OCR is disportionately large compared to that for face recognition. Although OCR should be prioritized given the EDF policy, it appears that fair sharing is the right approach instead. Lack of tail-robustness manifests when the heavy workload essentially blocks the light workload due to non-preemptive scheduling.

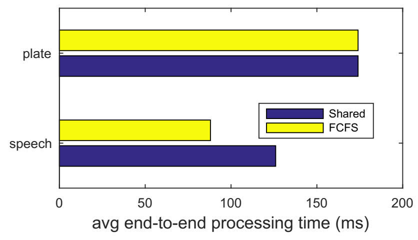
Therefore, *EDF suffers when the network transfer time follows a heavy-tail distribution, but sharing can mitigate this problem.*

Summary. To conclude this study, the winning scheduling objective appears to be prioritizing meeting deadlines but also trying to be tail-robust at the same time.

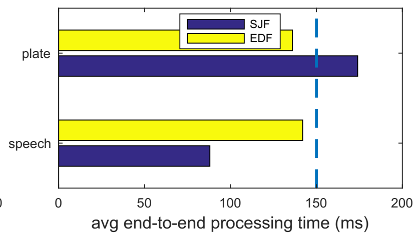
3.4 EDF with Limited Sharing

To balance the requirements of meeting application deadlines and avoiding a long tail-latency for network transfer, we augment Earliest Deadline First with Limited Sharing (EDF-LS). The idea is very simple. EDF-LS starts with EDF as the baseline while dynamically determining whether to multiplex the shared link between several transfers. To simplify further, we only consider two consecutive transfer tasks.

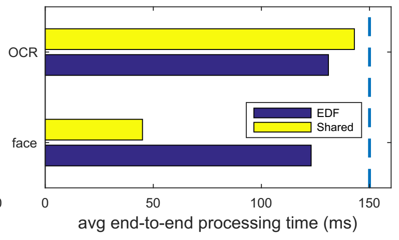
Suppose the two consecutive tasks in the queue, T_1 and T_2, are already ordered based on EDF. Their estimated network transfer times are N_1 and N_2 respectively. The limited sharing will be enabled if and only if the following condition holds: $N_1 > S \cdot N_2$. S captures how much the heavy-transfer task is



(a) Fair sharing vs FCFS



(b) SJF vs EDF



(c) EDF vs Sharing

Figure 3: The impact of scheduling algorithms on the end-to-end processing time

affecting the other, hence whether sharing is waranted. We set $S = 5$ empirically (Section 5.4).

This way, our scheduling algorithm can perform like EDF when the network transfer distribution is short-tailed. When a large transfer task arrives, however, the sharing enabled can allow small tasks to make progress as well. Note that sharing carries a cost, and therefore limiting sharing is essential to ensure the overall performance.

3.5 Discussion

The choice of offloading destination. Although the above scheduling algorithm is set in the specific context of current mobile offloading practice, many issues to consider are generic to multiple concurrent offloading sessions. The destinations can be a nearby (edge) server, the cloud, or even another nearby mobile device. For example, we can envisage offloading from a phone to a pad, such as offloading graphics rendering for multi-party gaming. There can also be more complicated link sharing mechanisms. The above expressions can be adjusted to reflect more general offloading scenarios.

Intra-application dependencies. So far we have assumed that the individual modules within an application are independent from one another. It is conceivable that two modules to be offloaded may instead be logically correlated. For example, one module may need to wait for the result from another module. We believe such correlation should be managed by the application itself by setting appropriate per-module deadlines to reflect the correlation, and therefore we do not consider such dependencies in this paper.

Fending off greedy applications. Another implicit assumption in our discussion is that each application will suggest reasonable deadlines for their offloading requests. If instead, a greedy (or malicious) application wants to game the system, it can intentionally set a tight deadline to gain more network resource. One solution to counter this is to track the variance of the end-to-end processing time distribution of each application and infer whether any application had been favored throughout.

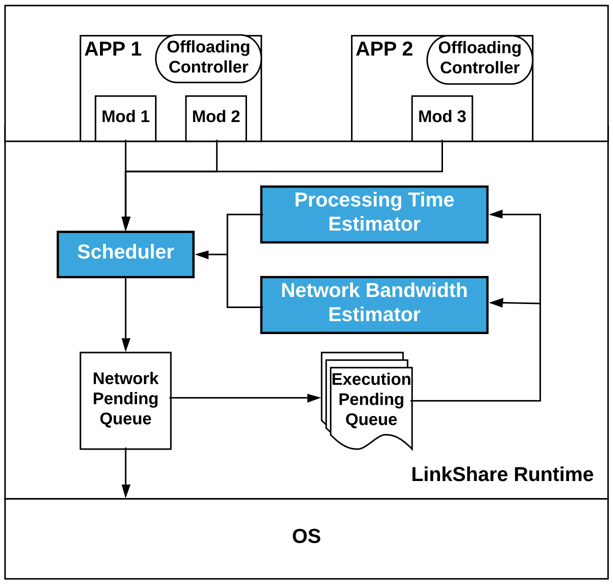


Figure 4: LinkShare architecture.

4 LINKSHARE AS A SERVICE

We next build a system level service, LinkShare, that coordinates multiple offloading jobs. Fundamentally, LinkShare is an extension to the operating system scheduler. It takes into account the wireless link bandwidth, remote processing capability, application processing deadline requirements and their effects on the actual end-to-end processing times. To support the central goal of making scheduling decisions, it collects information of past runs, estimates future execution times, and computes an appropriate schedule.

LinkShare sits between the applications and the operating system. Figure 4 shows the system architecture. There are two control paths: *application driven scheduling* and *background monitoring*. We describe the individual modules next.

Note that our system architecture is independent of the exact scheduling algorithms. Different scheduling algorithms can be substituted if needed.

4.1 Application driven scheduling

The application first decides whether to offload any workload remotely using an existing offloading runtime or its custom decision module, as discussed in Section 3.1. If offloading is needed, it sends the request to LinkShare, along with the deadline for completing the remote processing (e.g., in terms of the longest permitted processing time). LinkShare then determines the order to execute the offloading requests, in terms of the order of data transfer to the offloading destinations, using the Earliest Deadline First with Limited Sharing (EDF-LS) detailed in Section 3.4.

All offloading requests from applications are first enqueued in the *network pending queue*. Given our EDF-LS algorithm, this is essentially a priority queue, where the network transfer completion deadline (computed from the time instant at which LinkShare receives the request from the application, the estimated remote processing time and the processing deadline specified by the application) serves as the priority indicator.

The scheduler continues to empty the *network pending queue* as long as there are pending offloading requests. Once network transfer is completed, an offloading request is moved from the *network pending queue* to the corresponding per-application[2] *execution pending queue* and remains there until a response is received from the remote server.

4.2 Lightweight background monitoring

In the background, LinkShare tracks the states of each scheduled request, including the size of data transfer s_d, and the start and end time of the network transfer t_s and t_e.

Remote processing time estimation. To assess whether an application deadline can be met, it is essential to estimate the remote processing time. The main challenge here is to the intrinsic variability of processing times. Take popular learning-base mobile applications as an example. The variability may arise from (a) caching, (b) using multiple models simultaneously to speed up inference, (c) input variation, and (d) other customized optimizations [16].

Given all the variability, we estimate the average processing time (i.e., per-application) instead of the per-job processing time (i.e., for each data transfer). This is for two reasons.

First, it simplifies the system implementation significantly to be lightweight on a mobile device. In contrast, current mainstream per-data-transfer processing time estimation techniques are all learning based and incur high computation complexity. [15]

Second, since the mobile applications of interest to us usually have soft real-time requirements, an accurate estimate of the *average* processing time is sufficient for our scheduling goal of meeting deadlines.

[2]Strictly speaking, "per-application" refers to "per-module" in this section.

We adopt the exponentially weighted moving average (EWMA) to estimate the average, per-application remote processing time. The estimate T_{est} is initialized to 0 and updated in an event-driven manner. The completion of each offloading request at time t_r triggers an update for T_{est} based on the previous estimate and the observed processing time ($T_{cur} = t_r - t_e$) incurred by this completed request. Empirically, we set the smoothing parameter λ to 0.3.

$$T_{est} = T_{prev-est} \cdot (1 - \lambda) + \lambda \cdot T_{cur}$$

Network bandwidth estimation. The sharing component of EDF-LS is triggered based on the projected network transfer times. Therefore, network bandwidth estimation is also essential to LinkShare. Since wireless link bandwidth is known to fluctuate, we again resort to the classic EWMA. Triggered when an offloading request completes its network transfer, the bandwidth estimate is updated as follows:

$$B(t+T) = B(t) + \alpha \cdot (B(t) - \frac{u(T)}{T})$$

Where $B(t)$ is the estimated network bandwidth at time t, T is the interval between two updates, u is the number of bytes sent over T, and α is the smoothing parameter. This way, we can estimate the network bandwidth without incurring any network overhead.

While these above estimation algorithms work well in our evaluations, we realize that they represent a set of very simple heuristics. More sophistication can be taken into consideration to make these estimation more robust and accurate. For example, when estimating remote processing time, the estimation accuracy will increase if we can coordinate the real-time workload information on remote servers. Similar argument can be applied to network bandwidth estimation. We defer these and other improvements to future work, and currently choose to focus on the potential benefits brought by scheduling of multiple offloading applications.

Security considerations. Tracking the variance of the end-to-end processing time distribution of each application might not be resistant to malicious applications that overclaim deadlines from launch time. Fortunately, proper access control can efficiently protect our system from these applicaitons. This can be achieved by incorporating a reputation system (such as Credence [38]) into LinkShare. Each offloading request can be tagged with its application source. The tracking of end-to-end processing time of each offloading request can help to establish a reputation record for each application. The application whose deadline is always far beyond its end-to-end processing time can be identified and barred from time to time.

4.3 Implementation details

We implement LinkShare as a background application level service in Android Nougat OS with API version 24.

Since our scheduler runs in the user space, we enforce the schedule computed by setting the priority of thread that processes the scheduled request to `Thread.MAX_PRIORITY`, using the Andriod API `Thread.setPriority()`.

LinkShare APIs. LinkShare exposes the following APIs to the applications: `offloadRequest`, `decisionMade`, `notifySent`, and `notifyComplete`.

`offloadRequest` is used by an application to send an offloading request to the scheduler, containing the module name, input data size, process id, timestamp and the end-to-end processing time deadline. Once a schedule has been computed, the scheduler sends a `decisionMade` message to all applications with pending requests, indicating the module to be offloaded first.

`notifySent` is used by an application after completing sending its data to the remote server. This is intended to notify the scheduler to schedule the next offloading request in the queue and trigger the network bandwidth estimator to update the runtime bandwidth `notifyComplete` is used by an application when a computation job is completed. This notifies the scheduler to mark this scheduled request as completely and trigger the remote processing time estimator to update the estimated remote processing time for this specific application.

The following pseudocode shows code snippets for a face recognition module to send an offloading job with and without our scheduler service. The lines in bold indicate the code patch needed to use our service. All changes are on the mobile side, with little overhead.

Application code change. We opt to expose APIs to applications instead of making LinkShare transparent for two reasons. First, the application needs to set the deadline requirement explicitly anyway, since this cannot be inferred accurately. The system can only track the processing times of past runs and estimate the *bounds* (the shorest or the longest) on the deadlines that can be met. Second, the scheduling framework is not restricted to computation offloading. Therefore, we highlight the APIs needed to coordinate multiple concurrent network-bound applications in general.

Offloading framework for experiments. We also implement an offloading framework for experiments. This is independent from our scheduler service.

We define our own communication interface for service-application communication using Android Interface Definition Language (AIDL [6]). AIDL makes it easy to handle multi-threaded asynchronous inter-process communication.

The communication framework between our mobile device and server is built on gRPC [7]. gRPC is portable and provides a stream model, which we use to implement an asynchronous communication channel between a client and the server. The client does not need to wait for server response before sending another request.

```
// Conventional application code
if(offloadMode == true) {
    asyncRemoteRecognition.sent(image);
   while(!asyncRemoteRecognition.complete()) {
     continue;
   }
   result = asyncRemoteRecognition.result;
}

// With the scheduler service
offloadRequest(modName, timeStamp, pid,
      inputDataSize, deadline);

while(true) {
   if(receive decisionMade) {
      offloadMode = decision.offloadMode;
   }
}

if(offloadMode == true) {
   asyncRemoteRecognition.sent(image);
   notifySent(modName, timeStamp);
   while(!asyncRemoteRecognition.complete()) {
     continue;
   }
   result = asyncRemoteRecognition.result;
   notifyComplete(modName, timeStamp);
}
```

5 EVALUATION

5.1 General setup

Application scenario, benchmarks, and test data. We emulate a Gabriel [22] like scenario where multiple recognition applications concurrently operate on the camera and audio feeds from a phone and collectively provide assistance to the phone user to interact with the environment. Each video frame is processed simultaneously by one or more applications. We built four benchmark applications: face recognition, plate recognition, speech recognition, and an optical character recognition (OCR) application. These mimic similar commercial applications but are simplified to contain only the most relevant library functions. They are also extensively instrumented to provide the measurements we need, which we cannot obtain from commercial applications.

The *Face recognition* module recognizes the faces in a given video frame. We build the module using the Local Binary Patterns Histograms (LBPH) face recognizer in OpenCV [8]. The experiment data are taken from the unconstrained facial images database [31], which is a set of real-world photographs selected from the large photobank [10]. We use the cropped images dataset, where faces were automatically extracted from the original photographs. This set contains images of 605 individuals, on average 7.1 images per person.

The *Plate recognition* module recognizes the car license plate in real time. We build this using the OpenALPR library which is a popular open source plate recognition library [9]. We use the plate dataset from [4], which contains 500 images of the rear views of various vehicles, the resolution of each image is 640×480.

The *Speech recognition* module is built on the CMU Sphinx speech recognizer [2]. This application can recognize a set of spoken commands given by the users, including “open application", “call home", “close application", “take a picture", “tell me the location" and “tell me the time". The corresponding input data are the voice recordings from different occupants in our lab, 60 samples in total, 10 for each command. Note that our speech recognition application is completely processed on a remote server, like the main-stream speech driven assistant *Siri*.

The *OCR application* is built on the tesseract open source OCR engine [11]. It extracts text information from images. This is to emulate a scenario where users go abroad and do not understand the local language, so they use their phones to extract the textual information from signposts and translate it to familiar languages. We use the KAIST Scene Text Database [27] as the input, which contains signposts and brands under different light conditions.

Note that these applications all contain specific pre-trained models. During the run time, they only perform *inference* on any input data. This is because we cannot train these applications in real time yet. Therefore, we emulate a video feed using data from the test sets specified above to generate 4 parallel feeds[3]. For each application, we generate a long sequence by randomly drawing data from the corresponding test set and feed this sequence to the application. Across applications, we make the corresponding frames similar in size as much as possible.

Canonical testbed setup. We use a Samsung S9 smartphone as the mobile device, with an octa-core (2.7 GHz quad-core M2 Mongoose + 1.7 GHz quad-core Cortex A53) processor, 4 GB of RAM and 64 GB of internal storage. We use two servers, each with an Intel core i5-4570 processor (Quad-core, 6 MB Cache, 3.2 GHz) and 8 GB 1600 MHz DDR3 Memory.

Unless otherwise stated, the network upload speed is 10 Mbps. We use this as the baseline since the average wireless bandwidth measured was 8.63 Mbps in a recent network report [5]. This is the perceived TCP transfer throughput, not the physical layer rate on the wireless link.

[3]While this approach may not fully mimic the inter-frame correlation in a real video feed, it does not affect our current evaluation. In a real video feed, we could leverage temporal correlation between frames and skip a few runs for some applications as a further optimization, but this is orthogonal to making scheduling decisions.

The network transfer is over the WiFi link from the phone to the nearest access point. Since the WiFi networks in our lab are usually faster than the reported average speed and fluctuate between successive runs, we emulate a stable link at a target upload speed by rate limiting the WiFi transfers. We consider other network conditions later.

We also explored using other phones and server capabilities (both the network latency to the server and the processing capability). We find these do not change the qualitative observations. As long as the server is much more powerful than the phone such that offloading is worthwhile, the performance of our scheduler is indifferent to the phone processing capability. Further, the effects of server conditions are captured in applications showing different combinations of the network transfer and server processing times. Therefore, we only report results based on the canonical setup.

Note that our application benchmarks are not fully optimized, so local processing might take even longer than it could be. To offset that, we also use less powerful servers. This means that the effect of network transfer time overhead in our setting is in fact less pronounced than in scenarios with faster phones and servers. We will discuss this in the context of upload speed scaling in Section 5.3.

5.2 Scheduler Performance

We aim to answer two questions: (a) is the additional scheduling added by LinkShare essential? (b) How well does EDF-LS perform in terms of meeting application deadlines? Both can be addressed together by comparing the performance of several scheduling algorithms.

Scheduling algorithms for comparison. We compare the following algorithms: (a) *Fair sharing*: This algorithm achieves slot-based fairness on the wireless link. (b) *First come first serve (FCFS)*: this is the most naive non-preemptive scheduling algorithm, as well as the default network scheduler; therefore it is a proxy for the scheduling decision in the absence of LinkShare. (c) *Shortest job first (SJF)*: this is the algorithm that minimizes the network transfer time. (d) *Earliest deadline first (EDF)*: this is the optimal deadline-aware algorithm among the non-idling non-preemptive scheduling algorithms. (e) *Earliest deadline first with limited sharing (EDF-LS)*: our algorithm.

Workloads. Given the network transfer time distribution of individual application benchmarks, we assemble two workloads, referred to as *heavy-load* and *light-load*.

The heavy-load includes all four benchmark applications running concurrently. This is a general case, and likely more common in future. The light-load setting involves only face, plate, and speech recognition offloading concurrently, without OCR. This division is motivated by the network transfer time disparity. On average, the network transfer time of OCR is nearly the sum of the transfer times for the other three applications.

Setting the processing deadline. Since the actual deadline requirements of the commerical applications are proprietary, we determine the deadline based on the tail of the processing time for each application. We consider three options, using the 90th, 95th and 99th percentile of all these applications to determine their individual deadlines respectively.

Note that we assume the application deadline and the request sending interval are coupled together. This is reasonable since these applications operate on continuous video feeds, and the frame processing deadline partly arises from the need to keep with the frame capture rate. Unless stated otherwise, therefore, we will set the sending interval between two consecutive data frames in each application to be their individual deadline.

Performance metric. We use the deadline miss rate to assess deadline-awareness. This is defined as the ratio of the number of data frames that miss the deadline to the total number of data frames sent, turned into a percentage.

Light load. Here we set the deadline as 1.2× the 90th, 95th and 99th percentile of the end-to-end processing time. Figure 5 shows the performance of different scheduling algorithms performance given these deadline settings. We first note that each application favors specific scheduling strategies, and one often benefits at the expense of another.

FCFS vs EDF-LS: As we can see from all three figures, FCFS suffers when considering face recognition (the least bandwidth hungry in this case) but outperforms all the other scheduling algorithms when considering plate recognition (the most bandwidth hungry in this case). This is because the offloading request arrival pattern determines the performance of FCFS. As the request interval of face recognition is the smallest among these three applications, the probability of another application blocking the next data frame transfer is the highest for face recognition among these applications.

Recall that FCFS is a proxy for not employing LinkShare at all, i.e., managing offloading requests without adjusting the network transfer order. Figure 5 clearly shows this additional scheduling is essential.

Fair sharing vs EDF-LS: In contrast, fair sharing outperforms all the other scheduling algorithms for face recognition, but performs the worst for the other two applications. This is because, intrinsically, sharing is beneficial to the application that is less bandwidth hungry. However, it is at the cost of performance degradation of other applications that are more bandwidth hungry.

SJF vs EDF-LS: When given tighter deadlines, SJF shares similar performance to that of EDF-LS. When the deadline requirements are relaxed to 1.2× the 99th percentile, EDF-LS outperforms SJF for speech recognition application and delivers similar performance to the other applications. The main reason is SJF is that not aware of different deadline requirements across applications. Speech recognition incurs a slightly shorter network transfer *time* than plate recognition on average. However, in terms of the network transfer *deadline*, plate recognition has a much longer deadline than that of speech recognition. This means transfer the data frame for speech recognition before plate recognition is a better choice in general, but making scheduling decisions based on the network transfer *time* is a bad idea.

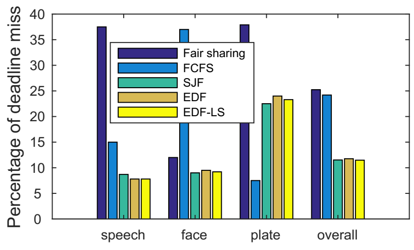
EDF vs EDF-LS: In the light workload scenario, EDF and EDF-LS share similar performance across all applications. This is expected, as light load rarely triggers sharing, and the performance of EDF-LS should converge to that of EDF. Sharing may incur a penalty in this case, as it slows down one job without helping another job much.

Heavy load. We set the individual application deadline to be 1.5× the 90th, 95th and 99th percentile of the end-to-end processing time of the respective application. We experimented with factors ranging from 1.2 to 1.5, and found that 1.5 is a suitable value to allow all these applications to mostly meet their deadlines.

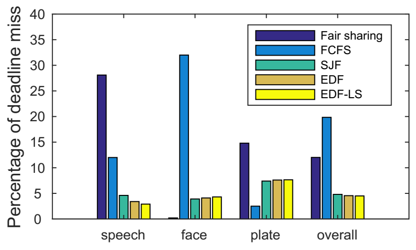
Figure 6 shows the performance of different scheduling algorithms given three different deadline settings. We mainly discuss the comparison between EDF and EDF-LS in this case, as the performance of the other scheduling algorithms is similar to the situation under light load.

The main difference between EDF and EDF-LS can be seen in the performance of face recognition and OCR. EDF-LS largely outperforms EDF for face recognition, at the cost of slight performance degradation of OCR. Specifically, the performance improvement margin is 30% given a tighter deadline (the 90th percentile case) and still 15% given a loose deadline (the 99th percentile case). This highlights the importance of mitigating head-of-queue blocking that would have originally happened to EDF. In fact, this blocking issue is not restricted to EDF. It also plagues other non-preemptive scheduling algorithms like SJF and FCFS. By mitigating this with limited sharing, we can reduce the missing rate of face recognition to less than 3%.

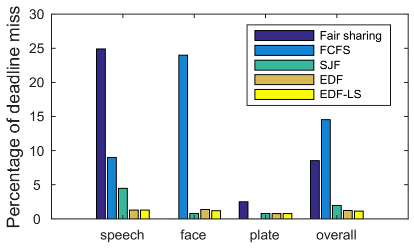
Summary. The additional scheduling added by LinkShare over the default OS scheduler is essential, and EDF-LS can achieve up to 30% reduction in the deadline miss rate compared to the baseline EDF.



(a) 90th

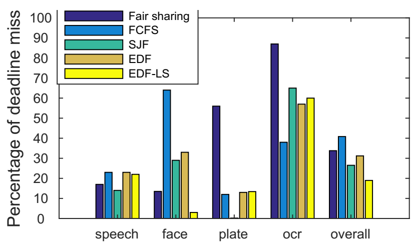


(b) 95th

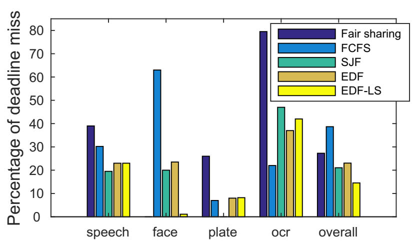


(c) 99th

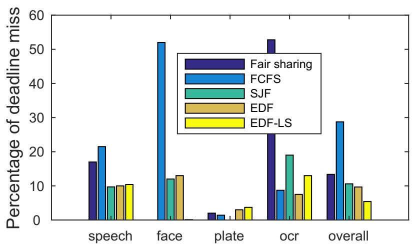
Figure 5: Deadline miss rate under light load at 10 Mbps



(a) 90th



(b) 95th



(c) 99th

Figure 6: Deadline miss rate under heavy load at 10 Mbps

5.3 Impact of Network Conditions

As network transfer is the major bottleneck in our system, intuitively the link quality dictates the performance of EDF-LS. We consider various bandwidths on steady links and two real traces of fluctuating networking conditions.

Steady links. Since wireless networks continue to evolve, we may expect increasingly higher bandwidth. We are therefore interested to see whether the problem remains under better network conditions.

We repeat the previous heavy-workload experiments under two additional bandwidth settings, 15 Mbps and 20 Mbps, as a proxy to show the system performance as the upload bandwidth increases. Note that, as the network bandwidth will affect the estimate of the end-to-end processing time, the deadline under a higher network bandwidth will be tighter than that under 10 Mbps.

Figures 7a and 7b show the performance with 15 Mbps and 20 Mbps upload speeds. These show that EDF-LS consistently incurs the lowest miss rates for face recognition and similar miss rate compared to EDF for other applications.

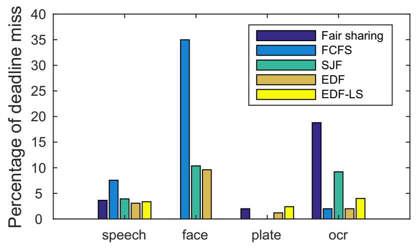
We see that the increasing network bandwidth has the largest impact on the performance on fair sharing and FCFS. This is because higher bandwidths reduce the network queuing time and thus mitigate the effect of bad scheduling decisions.

One interesting observation is that, for face recognition, the performance of EDF and SJF hardly change even as the network bandwidth goes up. The reason is that deadline misses occur due to the blocking nature of these two algorithms (face recognition is blocked from data transfer by a network-bound job), not its own network transfer time. The blocking effect remains even at higher network speeds.

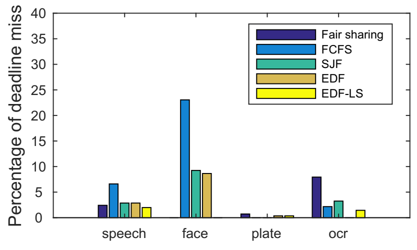
More generally, even as the bandwidth increases further, the scheduling problem is likely to remain, because the server processing capacity is likely to increase in tandem. Fundamentally, the cause of missing deadlines in our context is the relative network queuing delay, i.e., the absolute queuing delay divided by the end-to-end processing time. This depends heavily on the ratio between the network transfer time and the remote processing time.

Fluctuating network conditions. Being aware of the potential influence of mobile network conditions to the evaluation results, we next study the performance as the link condition fluctuates.

We first measure WiFi and LTE bandwidths by running `iperf3`, using our mobile phone as the client sending traffic to one of our servers. Again, these are the TCP throughput numbers, instead of the raw physical layer rates. The WiFi trace is captured in a cafe near campus, while the LTE trace is collected as one walks from the student dormitory to the

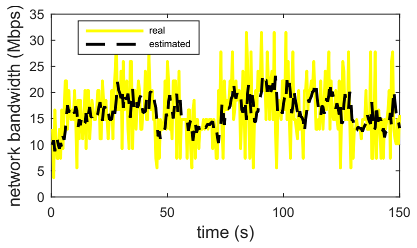


(a) 15 Mbps

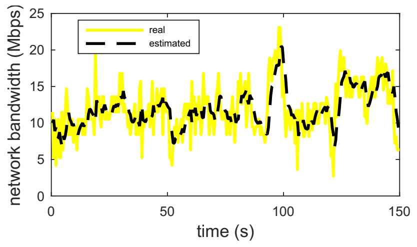


(b) 20 Mbps

Figure 7: Performance under different network upload speeds

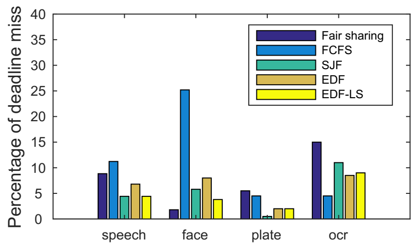


(a) WiFi

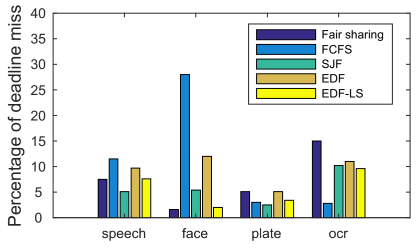


(b) LTE

Figure 8: Real WiFi and LTE traces



(a) WiFi



(b) LTE

Figure 9: Performance under real WiFi and LTE network conditions

department building. We record the bandwidth every 0.5 second, which is the minimal interval supported by `iperf`. Each trace covers a 150-second period.

Figures 8a and 8b plot the bandwidth timeseries for the WiFi and LTE traces captured respectively (the "real" lines). The "estimated" lines in these two figures also show the estimated network bandwidth using the bandwidth estimation module in LinkShare. Our module indeed manages to track the trend of average network bandwidth changes.

We then replay these traces to evaluate the performance under fluctuating network conditions. We assume the same deadline and request sending interval settings (1.5 times the 99th percentile of the end-to-end processing time) as that in the 10 Mbps, heavy-workload for both WiFi and LTE.

Figures 9a and 9b show the performance under WiFi and LTE respectively. EDF-LS can still show similar performance to that under steady links.

An interesting observation is that EDF-LS outperforms EDF for *all* applications, not just face recognition. This is especially pronounced for LTE. Similarly, fair sharing fares much better over fluctuating links than steady links. The reason for both observations is similar to what happened at increasing network upload speeds. Sharing can intrinsically benefit from a larger bandwidth. In a highly fluctuating network, the network bandwidth spikes can be better used when multiplexed between different applications.

5.4 Microbenchmarks

Finally, we performance several microbenchmarks to assess the design and overhead of our scheduling algorithm.

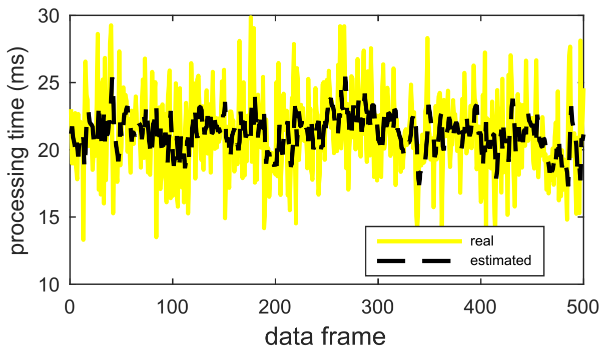
Scheduler overhead. The scheduler overhead is mainly from the scheduling decision time. This takes less than 2 ms.

Processing time estimation. We already assessed the accuracy of the *network bandwidth estimation* module in the previous subsection. Here we use face and speech recognition as two examples to assess the accuracy of the *processing time estimation* module, shown in Figure 10.

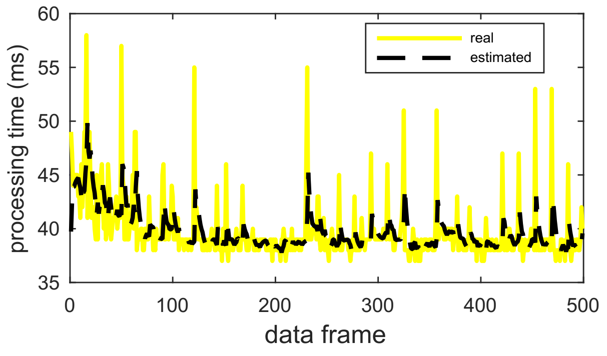
Face recognition is a representative application with relatively stable processing time pattern, while speech recognition represents a version with more fluctuation. OCR and plate recognition share similar processing time patterns as that of speech recognition. We run each application 500 times and randomly selected input data from their respective dataset. As we can see from the figure, our processing time estimation module manages to track the trend of the average processing time changes. Quantitatively, the root mean square errors of the processing time for speech and face recognition are 2.56 ms and 1.54 ms respectively.

Sharing parameters. Recall that the sharing component of EDF-LS is triggered based on a threshold, and the parameter S hints at the disparity between the projected network transfer times between two consecutive offloading requests (Section 3.4). Therefore, we need a suitable value for S.

To avoid the randomness due to the input stream, we first run the heavy-load combination under the 99th percentile deadline constraints with $S = 3$ and 10 Mbps network upload



(a) Face recognition



(b) Speech recognition

Figure 10: Processing time estimation

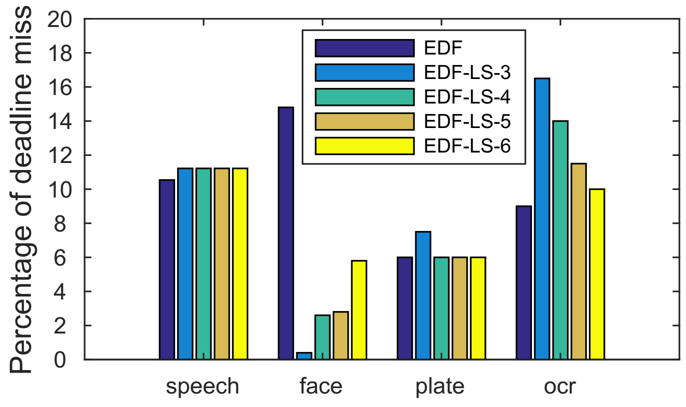


Figure 11: The value of sharing parameter

bandwidth. We record the processing time and file size for each data frame. Then, we replay the trace in simulation and try different S values to assess its impact.

Figure 11 shows the results for $S = 3, 4, 5,$ and 6. The main difference comes from the performance of face recognition and OCR. The higher the S value, the less benefit face recognition gained from the limited sharing. A good trade-off between respecting the deadline and mitigating heading-of-queue blocking is achieved when the S value is around 4 to 5.

As S increases (i.e., requiring higher transfer time disparity before enabling sharing), the performance of EDF-LS approaches that of EDF, which means limited sharing is rarely activated. Based on this analysis, we set the sharing parameter value to be 5.

Discussion. Another dimension of our algorithm is the choice of the maximum number of workloads that can share the link bandwidth. The optimal number is workload dependent. For our workloads, sharing between two consecutive jobs appear to be optimal and the most robust choice.

Another point worth mentioning is that, for now, the choice of sharing parameter S aims at minimizing the average percentage of deadline missing across all workloads. That is to say, implicitly, we treat all the workload applications the same. However, the preference of different user might make the utilities of these workload applications different. We believe it is an interesting future work if introducing utility of each application as a factor when making scheduling decision.

6 RELATED WORK

For most of the last decade, the computation-intensive requirement from mobile applications has been met with computation offloading to a cloud server. New dedicated hardware has also emerged in recent years. However, existing solutions are limited as new applications are on the horizon, calling for more inter-module or inter-application coordination. We will discuss relevant resource management approaches in the context of common offloading practice.

Dedicated Hardware. The Google Tango project [3] uses dedicated hardware to power computation-intensive workloads locally. It does not depend on external resources and simplifies application development. Resource management is simply delegated to the operating system. However, the need for additional hardware and often high power consumption are ill-suited for wearable devices like Google Glass, which generally require remote support.

Single-module Offloading. Computation offload to the cloud has been explored extensively in the last 15 years [13, 18, 19, 28, 30, 32, 35, 41]. Several recent works [15, 17, 20, 21, 29, 36] focus on seamlessly partitioning the workload spanning the mobile device and the remote server.

They assume only one foreground application at a time. All other applications within the same device are treated as background processes, with lower priorities. This is indeed true for previous scenarios. However, emerging applications are more sophisticated and all the modules concerned must be served simultaneously, given the same priority. Single-module offloading techniques are insufficient.

Multi-module Offloading. MCDNN [25] provides a common platform for multiple applications concurrently utilizing deep neural networks (DNN). When reasoning about the workload split between the device and the cloud for each DNN application, MCDNN does coordinate on-device resource usage between applications. However, the coordination is specific to DNN applications run on MCDNN and integrates decisions to trade off classification accuracy for less resource usage.

Cloud-centric offloading. Gabriel [22], mentioned earlier, offloads all its computation intensive workloads to the nearby cloudlet, where both the control and computation units are located. The underlying assumption is all of these applications can be processed on a single backend server. However, even for Google Glass, apart from a handful of built-in applications like Google Now and Google Maps, most applications are from third-party developers, who may maintain their own servers. It may be impractical to make all these applications offload to the same destination. In that case, the cloud-centric offloading scheme can not be applied. Instead, a device-centric approach seems more promising.

Deadline-aware scheduling. Deadline-aware scheduling has been studied extensively, most recently in data center networking [12, 26, 39]. [39] controls the network transfer rate to meet the deadline, but rate control is difficult for mobile scenarios. [26] pre-empts flows to approximate a range of scheduling algorithms. However, we explain earlier that the overhead of link switching in wireless is high.

Summary. Although there is a multitude of offloading schemes to handle computation-intensive workloads on mobile devices, none appears to provide the coordination needed between multiple offloading modules to serve the emerging applications outlined above. LinkShare fills in this gap with a generic on-device scheduling service for all applications interacting with backend servers. LinkShare augments standard deadline-aware scheduling to suit our needs.

7 CONCLUSION

In this paper, we investigated concurrent and continuous mobile-cloud interactions in the form of simultaneous offloading jobs. These share the wireless link from the mobile device but might involve different server backends. Therefore, we need device-centric management, instead of the more common cloud-centric control.

We build a system-level service, LinkShare, that wraps over the operating system scheduler to coordinate among multiple offloading requests. We study the scheduling requirements and suitable metrics, and find that the most intuitive approaches of minimizing the end-to-end processing time and earliest deadline first scheduling do not work well. Instead, LinkShare incorporates limited sharing between consecutive workloads in the case of heavy-tailed distribution. LinkShare is implemented in Android. Extensive evaluation shows that adopting this additional scheduler is essential and that EDF-LS can achieve up to 30% reduction in the deadline miss rate compared to the baseline EDF.

The issues we studied are not specific only to computation offloading to a remote cloud. Even with the advent of edge computing, the same bottleneck remains. LinkShare as a framework is also applicable when the offloading destination is a nearby server or another IoT device instead. We believe this is a first step towards coordinating an IoT ecosystem tethered to the mobile device from the perspective of the mobile frontend.

ACKNOWLEDGMENTS

We would like to thank all the reviewers and our shepherd Dr. Landon Cox for their helpful comments and suggestions. This work is funded by the National Science Foundation under Grant No. CNS-1815115.

REFERENCES

[1] 2016. IP Video Surveillance and VSaaS Market by Type (IP camera, Monitors, Storage, VMS, Video Analytics, Cloud Storage by product software, Cloud storage by deployment, VSAAS, Hosted VSAAS, Managed VSAAS, Hybrid VSAAS, Integration Services) and Application (Banking & Financial, Retail, Healthcare, Government & higher, security, Manufacturing & corporate, Residential, Entertainment & Casino) - Global Opportunity Analysis and Industry Forecast, 2015 - 2022. https://www.alliedmarketresearch.com/IP-video-surveillance-VSaaS-market

[2] 2017. *CMUSphinx Project*. https://cmusphinx.github.io

[3] 2017. *Google Tango Project*. https://get.google.com/tango/

[4] 2017. *License Plate Detection, Recognition and Automated Storage*. http://www.zemris.fer.hr/projects/LicensePlates/english/results.shtml

[5] 2017. *Speedtest Market Report in USA*. http://www.speedtest.net/reports/united-states/

[6] 2019. *Android Interface Definition Language*. https://developer.android.com/guide/components/aidl.html

[7] 2019. *gRPC*. http://www.grpc.io

[8] 2019. *LBPH Face recognizer*. http://docs.opencv.org/trunk/df/d25/classcv_1_1face_1_1LBPHFaceRecognizer.html

[9] 2019. OpenALPR. http://www.openalpr.com/cloud-stream.html

[10] 2019. Photo bank of the Czech News Agency. http://multimedia.ctk.cz/en/foto/

[11] 2019. Tesseract Open Source OCR Engine. https://github.com/tesseract-ocr/tesseract

[12] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 435–446.

[13] Rajesh Balan, Jason Flinn, Mahadev Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. 2002. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 87–92.

[14] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 155–168.

[15] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: elastic execution between mobile

device and cloud. In *Proceedings of the sixth conference on Computer systems*. ACM, 301–314.

[16] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System.. In *NSDI*. 613–627.

[17] 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*. ACM, 49–62.

[18] Jason Flinn, Dushyanth Narayanan, and Mahadev Satyanarayanan. 2001. Self-tuned remote execution for pervasive computing. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*. IEEE, 61–66.

[19] Huber Flores and Satish Srirama. 2013. Mobile code offloading: should it be a local decision or global inference?. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 539–540.

[20] Mark S Gordon, David Ke Hong, Peter M Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 137–150.

[21] 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.

[22] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 68–81.

[23] Thomas J Hacker, Brian D Athey, and Brian Noble. 2001. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE, 10–pp.

[24] Arun Hampapur, Lisa Brown, Jonathan Connell, Ahmet Ekin, Norman Haas, Max Lu, Hans Merkl, and Sharath Pankanti. 2005. Smart video surveillance: exploring the concept of multiscale spatiotemporal tracking. *IEEE Signal Processing Magazine* 22, 2 (2005), 38–51.

[25] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 123–136.

[26] Chi-Yao Hong, Matthew Caesar, and P Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 127–138.

[27] Jehyun Jung, SeongHun Lee, Min Su Cho, and Jin Hyung Kim. 2011. Touch TT: Scene text extractor using touchscreen interface. *ETRI Journal* 33, 1 (2011), 78–88.

[28] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. 2010. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services*. Springer, 59–79.

[29] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*. IEEE, 945–953.

[30] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. 2015. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 521–532.

[31] L. Lenc and P. Král. 2015. Unconstrained Facial Images: Database for Face Recognition under Real-world Conditions. In *14th Mexican International Conference on Artificial Intelligence (MICAI 2015)*. Springer, Cuernavaca, Mexico.

[32] Zhiyuan Li, Cheng Wang, and Rong Xu. 2001. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 238–246.

[33] Akhil Mathur, Nicholas D Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. 2017. DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models using Wearable Commodity Hardware. (2017).

[34] Jayakrishnan Nair, Adam Wierman, and Bert Zwart. 2010. Tail-robust scheduling via limited processor sharing. *Performance Evaluation* 67, 11 (2010), 978–995.

[35] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. 2010. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 347–356.

[36] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. 2011. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 43–56.

[37] M Satyanarayanan. 2004. From the editor in chief: Augmenting cognition. *IEEE Pervasive Computing* 3, 2 (2004), 4–5.

[38] Kevin Walsh and Emin Gün Sirer. 2006. Experience with an Object Reputation System for Peer-to-Peer Filesharing.. In *NSDI*, Vol. 6. 1–1.

[39] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 50–61.

[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 *NSDI*, Vol. 9. 1.

[41] Wenzhang Zhu, Cho-Li Wang, and Francis CM Lau. 2002. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*. IEEE, 381–388.