Adaptive Cloud Offloading for Vehicular Applications

Ashwin Ashok†, Peter Steenkiste‡, Fan Bai¬
†Georgia State University, ‡Carnegie Mellon University, ¬General Motors Research

***Abstract*—The growing number of sensor–based interactive applications and services are pushing the limits of the on–board computing resources in vehicles. With vehicles increasingly being connected to the Internet, offloading the computation to cloud–computing infrastructures is an attractive solution. However, the large sensory data inputs of interactive applications makes offloading challenging across dynamic network conditions, and different application requirements or policies. To address this challenge, we design a system to adaptively offload specific vehicular application components or *modules* to the cloud. We particularly develop heuristic mechanisms for the placement and scheduling of modules on the On–Board Unit (OBU) and a cloud server under dynamic networking conditions during driving. Through an experimental evaluation of the end–end application response time using our prototype vehicular cloud offloading system, we show that our mechanism can help meet application response time constraints.**

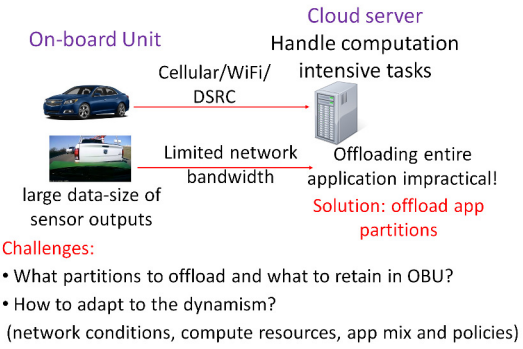


Fig. 1. Motivation for offloading in vehicular context (DSRC = Dedicated Short Range Communication)

I. Introduction

Computing requirements for vehicular applications are increasing tremendously, particularly with the growing interest in embedding new class of interactive applications and services using on–board sensory inputs. For example, autonomous driving and novel driver–safety enhancement applications are becoming increasingly dependent on on–board cameras and motion sensors. The on–board computing resources in vehicles have been primarily allocated to accommodate the driving mechanics of the vehicles. The growing need for embedding interactive applications and services requires reallocation of the limited on–board computing resources. Unlike software, updating the hardware information technology (IT) resources in vehicles to keep up with increasing demands of such applications is challenging as it cannot be done automatically. Moreover, it must be possible to manage such updates over the long lifetimes of vehicles; 10–15 years.

A potential solution to address the computing challenges in vehicles is through cloud–computing, by migrating or *offloading* the interactive applications from vehicle on–board–units (OBU) to a remote computer or *cloud*. Interactive vehicular applications typically have large sensor inputs, for example a video footage for a vision application, and offloading the entire application to the cloud is impractical due to the unreliable wireless connectivity between the vehicle and cloud. Offloading across such unreliable network conditions can result in large round–trip times (RTT). To address the offloading challenge, in this paper, we propose to design a system that selectively offloads only *parts* of applications. These parts may correspond to specific functions, or *tasks*, that are deemed computationally intensive by our system.

Prior approaches to cloud–offloading design have primarily proposed distributing computation by migrating virtual machines [1] or offloading parts of the application code between the mobile client and cloud [2] or low–level memory abstractions [3]. The challenge with such designs is the large communication overhead to transport machine and application state. Distributed computing techniques using remote–procedure call (RPCs) based architectures (CORBA: http://www.corba.org/) have existed for a long time, however, these techniques assume high network reliability which is not guaranteed in a vehicular case.

Challenges in selectively offloading application parts. Designing a system for offloading specific application tasks to the cloud in a vehicular environment brings about some fundamental challenges (Figure 1): (i) *Dynamic Allocation*; The system must allocate OBU and cloud CPU cycles to schedule execution of a collective set of application tasks. The scheduling must take into account the round–trip latency to offload application tasks over a limited network bandwidth. It must also be able plan CPU and network bandwidth allocations for scheduling applications for different policy considerations; for example, minimize network bandwidth expense or minimize OBU cycles usage. Overall, this takes shape of a multi–dimensional (multiple applications and across variable constraints) scheduling problem which are known to be NP–hard. (ii) *Run–time Adaptation*; The system must be able to conduct the allocation and offloading process during run–time

978-1-5090-5197-7/16/$31.00 ©2016 IEEE

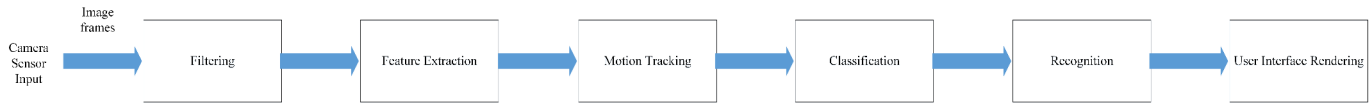


Fig. 2. Application modules of a computer vision based gesture recognition application

as information about what applications will be initiated is not known apriori.

Adaptive Cloud–Offloading for Vehicular Applications. To address the offloading challenges, we design an adaptive cloud–offloading system that incorporates a heuristic mechanism for resource allocation. The mechanism partitions application into tasks and plans placements of those tasks in OBU or cloud based on variations in network bandwidth, OBU cycles availabilities and policies. In this way, the mechanism tries to identify if a schedule is possible even before applications are to be scheduled at run–time and adapt accordingly based on the planning done through resource allocation.

In our design, we particularly focus on ensuring that the end–to–end response time[1] requirements of applications are being met. We define a term *deadline* as the maximum duration within which an application must complete execution to be regarded useful by the system user. *The goal of our offloading system design is to ensure that the response time of embedded vehicular applications are within their stipulated deadlines.* In this regard, we make the following specific contributions in this paper:

1) We design framework for offloading that is flexible and can deal with vehicular applications with large sensory inputs. ;
2) We develop a heuristic mechanism for partitioning and scheduling, which includes partitioning applications into tasks & identifying placement of such tasks, and computing a schedule;
3) We develop a mechanism for adapting applications' scheduling at run–time based on the dynamics of network variations and design policy selections;
4) We prototype an end–end system implementation of a cloud offloading system. We develop four proof–of–concept applications on an Android platform that integrate our offloading system;
5) We experimentally evaluate the effectiveness of our prototype offloading system, using the response time of applications as a metric, across variable network conditions, deadlines and policy.

II. System Overview

A. Application Task Model

Sensor–based applications in vehicles typically are a collection of tasks with different properties such as long/short execution times and large/small data size inputs, etc. We characterize the properties of these tasks through a model.

[1]total execution time of the application measured from initiation to completion (deliver final output)

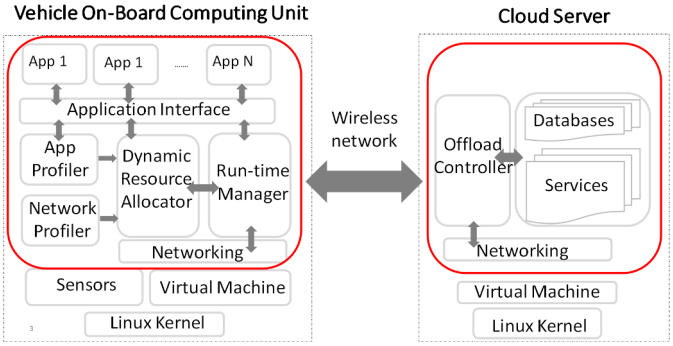


Fig. 3. Service–Oriented Cloud–Offloading System Architecture

We divide an application into individual units called *modules* (Figure 2). A module is a collection of application tasks that can be treated as a monolithic function and which can be reused independently across different compatible systems. In this work, we will consider a sequential execution flow of modules where the data output of a module is the data input to the subsequent module.

B. Cloud–Offloading System Architecture

In our offloading system architecture we consider that applications' module functionalities are available in the cloud in the form of *services*. As shown in Figure 3, the system includes a vehicle OBU (client) running multiple applications and cluster of machines in the cloud (server) that provide application module functionalities as services. Through a *service based approach* we avoid the need to migrate code and/or application state at run–time avoiding huge communication and management overheads.

The process of offloading a module involves the system appropriately identifying the cloud service corresponding to the module, migrating the data input for that module, executing the module in cloud machine and returning back the output to the client. This offloading process is handled and managed through appropriate software controllers in the client and cloud machines. The client–server interactions are conducted over a wireless network connection.

We treat that the information regarding the modules are dynamically shared with the cloud server when the client registers with the cloud service for the first time, and updated on–the–fly when required. The client includes an application profiler and a network profiler that integrate as libraries with the application source–code through a software interface. The application profiler records the execution time, and the input, output data size of each module in each application in the workload. The network profiler periodically records the

network bandwidth (referred to as network speed in the rest of this manuscript), computed based on the round–trip time measurements of probe packets between the client and the cloud server through a cellular connection.

The core components of our cloud–offloading system include: (a) Dynamic resource allocator, that hosts a heuristic mechanism to allocate OBU and cloud CPU cycles, and network bandwidth to execute a workload (collection of applications), (b) Run–time Manager, that makes run–time decisions of workload execution.

C. Design Scope and Policies

The scope of our system design includes:
(i) Applications arriving at the resource allocator within a window of time (specified by designer) must be considered in the workload for resource allocation.
(ii) The allocator will consider only aperiodic or bursty application arrivals.
(iii) No part(s) of applications deemed as safety–critical will be offloaded.
(iv) Preemption of execution of an application module is not allowed.

We aim to support the following policies in our cloud–offloading system:

1) *Minimize OBU CPU cycles expense*: Offload as many modules as possible to the cloud and minimize commitment to OBU CPU.
2) *Minimize network bandwidth expense*: Offload only as many modules as necessary and conserve the available network bandwidth.
3) *Minimize maximum lateness of applications*: Offload as many applications as possible with a maximum limit for apps being late.

III. Dynamic Resource Allocation

The goal of resource allocation is to make decisions on what application modules are partitioned and placed in OBU or cloud, and to find a schedule for their execution.

In general, both, placement and scheduling fall under the class of NP–hard problems. While several heuristics for scheduling problems have been proposed before [4], the vehicular cloud–offloading context brings about the challenge of capturing the variabilities in computing (CPU cycles) and network resources (bandwidth) through a single optimization problem.

We approach this problem by developing a heuristic mechanism based on the key insights we gained about vehicular offloading system:
(1) A sequential execution control flow of modules implies that the data size will shrink across the application modules. So, for each application, once a module is placed in cloud, it implies that the subsequent modules in the application also are placed in the cloud. This means that there is only one placement decision to be taken for each application.
(2) The usage of the network bandwidth and OBU CPU cycles can be traded–off. The placement decisions on what modules will need to be offloaded if the OBU is over–committed (leading to applications missing deadlines) can be planned before scheduling applications.

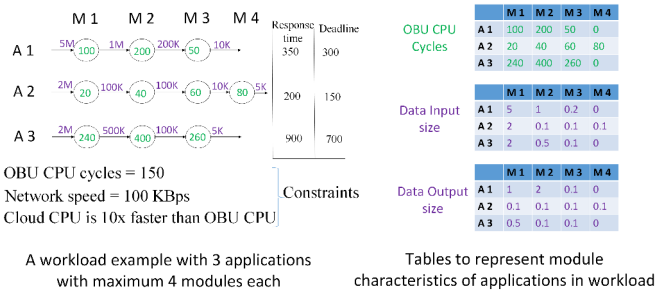


Fig. 4. Illustration of a tabular structure to represent workload

A. Workload Structuring

Traditionally, heuristic approaches for real–time placement and scheduling have used tree data–structures. However, generating the tree to exhaustively represent all possible placement options is time consuming, and requires large storage space. In addition, traversing a tree is a time consuming operation and can lead to long wait times for applications' execution.

In our design, we organize applications by representing their module characteristics in the form of tabular representations. As illustrated in Figure 4 each table represents one module characteristic; (i) OBU CPU cycles (represented in terms of execution time), (ii) size of data input to the module, and (iii) size of data output from the module. Here, each row represents one application, and each column corresponds to a module from the application corresponding to that row. We ensure that the columns in each row are indexed as per their sequential execution order in that application.

We also use a tabular data structure to represent the placement options, where each column corresponds to one module and the value in each cell of a column corresponds to the placement, represented as a binary value; where 1 corresponds to cloud and 0 corresponds to OBU. We generate these columns by regrouping the modules of each application (retaining their sequential execution order) into a single linear array. In this way we represent one placement option, encompassing all the applications in the workload, through a binary representation in each row.

We use the tabular data–structure because it helps in conserving both, storage space and execution time to generate and/or navigate across the structure. It applies well to our case where both dimensions, the array of applications and the list of modules in each application can be concisely represented in one structure.

B. Placement and Scheduling

We develop a heuristic mechanism for placement and scheduling that involves three stages:
(1) **Proactive placement to meet constraints:** Find placements where the OBU CPU cycles and network bandwidth

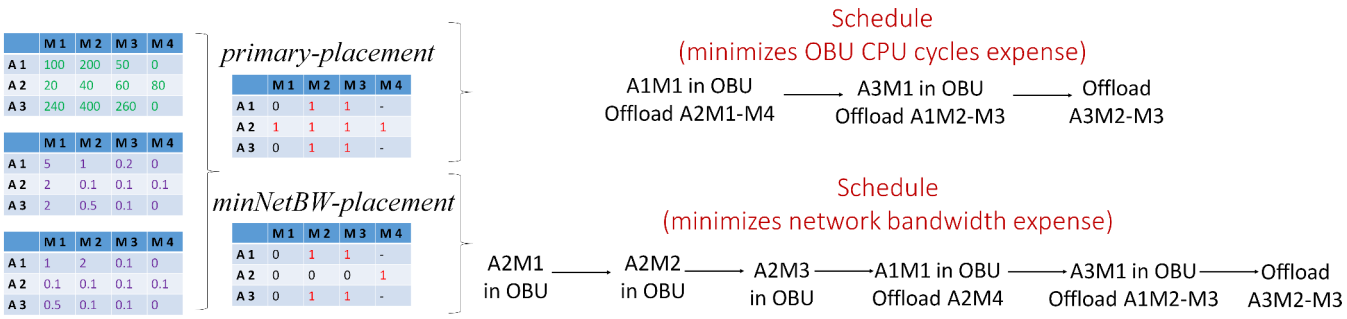


Fig. 5. Example showing the *primary–placement* and *minNetBW–placement* for workload in Figure4, and the schedules computed for each.

constraints are met, and which can help find a schedule to meet deadlines.

(2) **Placement planning for different policies:** Plan placements for each design policy.

(3) **Schedule computation:** Prepare schedule for selected placement. Update placement selection and recompute schedule if applications cannot meet deadline.

In our design, we use a quantity called *slack*, for each application, defined as the difference between the application's deadline and its response time (negative slack means application is late). In our design, for the sake of simplicity, if the placement and schedule is not known, we compute the slack by considering the response time as the execution time of the application assuming it is the only one to be executed in the OBU. The slack is updated with updates in placement and schedule.

1) Proactive placement to meet constraints: We aggressively search for a placement option that offloads as many modules as possible to the cloud within the given network bandwidth constraint. We will refer to this placement as *primary–placement*.

Algorithm for finding *primary–placement*

Step 1: Select application with least–slack

Step 2:

Select least indexed module not placed in cloud

Place the module in cloud.

Accumulate the module data size (input + output) to be migrated

Compute Required network bandwidth = cumulative data size/timeWindow. (where, timeWindow = $\min$ Available networking time across selected applications)

Step 3:

Check if (Required network bandwidth) $\geq$ (Available network bandwidth). If not, go to Step 1

If yes, stop placement of the module to cloud. Modules not placed in cloud are placed in OBU

Update slack for all applications

Compute residual OBU cycles (total OBU cycles available $-$ total OBU cycles used)

Step 4:

If residual OBU cycles > 0 AND $\min$ slack > 0, select placement as *primary placement*

If not, make workload policy changes and go to Step 1. Consider either of the following placement policies:

(i) applications can be late up to a specific limit

(ii) remove application(s) from the workload

2) Placement planning for different policies: In this stage, we define mechanisms for adjusting the placement selection if the policy consideration changes.

POLICY 1: Minimize OBU CPU cycles expense

The *primary–placement* selection process essentially addresses this policy.

POLICY 2: Minimize network bandwidth expense

Considering *primary–placement*, based on our understanding of the network bandwidth and OBU CPU cycles tradeoff (Insight 2), we compute the network bandwidth expense per unit investment of residual OBU cycles, per application. Starting with the least indexed module for each application, for each offloaded module of the application, we compute the residual OBU cycles and slack considering the module is replaced back in OBU. We update the placement if the residual OBU cycles and slack are non–negative after iterating over all applications for replacements. We will refer to this placement as *minNetBW–placement*.

In the process of placement selection, those that do not meet the criterion are pruned. In the tabular representation of the placement this means that if a module M_i is placed in cloud, then any placement option where M_j $(j \geq i)$ is placed in OBU is pruned – based on Insight 1.

POLICY 3: Minimize maximum lateness of applications

This policy is not addressed in the placement stage. It will be considered only in schedule computation stage, if no schedule is found for meeting deadlines within the given network bandwidth and OBU CPU cycles constraints using other two policies.

3) Schedule computation: The placement process identifies the *primary–placement*, and *minNetBW–placement*. Depending on the policy consideration, the scheduling process considers either *primary–placement* or *minNetBW–placement* and tries to find a schedule based where applications meet their

deadlines. We show the placements and schedule computed for the workload example from Figure 4 in Figure 5.

Scheduling algorithm
Step 1:
Update slack for each application based on placement
Step 2:
Select application with least–slack.
Schedule least–indexed module. Update slack.
Repeat Step 2 if no application is late.
Exit if all modules have been scheduled.
Step 3:
If any application is late, update placement*
Recompute schedule for updated placement (go to Step 1).
Step 4:
Exit when all modules have been scheduled (*schedule found*)
Exit when *no schedule found***

**Updating placements.* Placement updates follow this strategy: (a) Pick least indexed module placed in OBU from application that is most late, and place in cloud, and (b) Place offloaded modules from the application that is not late and has highest slack, in OBU. The idea here is to balance the network bandwidth and OBU CPU cycles expenses for the updated placement. The process (b) will not be necessary if the network bandwidth increases or if there is sufficient bandwidth to offload the module as in (a).

***No schedule found.* In this case, the workload policy must be adapted such as relaxing the deadline or removing application from the workload can be initiated. However, if the system allows the applications to be late, the schedule is recomputed considering to POLICY 3 to *minimize the maximum lateness across applications*.

Why least–slack first? The least–slack first approach helps to distribute the slack across the set of applications while computing the schedule. In this way, if an application with least–slack will get late in the next scheduling iteration (as the deadline is close), it gets compensated by an application with large slack.

IV. Run–Time Execution and Adaptation

The run–time manager adapts schedule execution when the network bandwidth changes or when new applications arrive at the workload. Since preemption of a module execution is not allowed, the schedule is updated only upon its completion. The robustness of the adaptation to changes to network bandwidth and new application arrivals depends on the policies used in Section III.

The network bandwidth is represented in terms of the network speed (bytes/sec) and measured as the ratio of the total data size and RTT of probe packets communicated between OBU and cloud. Upon detection of network bandwidth variation of greater than a preset threshold, the system updates the placement and schedule.

The run–time controller adapts to the following cases of network bandwidth reductions:

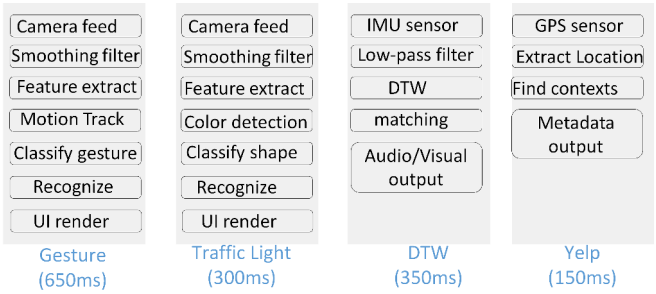


Fig. 6. Modules of our prototype apps (numbers represent the response time of each app when only one app executes in the workload)

(a) Bandwidth drops: Select the placement planned for this specific amount of bandwidth drop.

(b) Network connection is intermittent: Select the average of the minimum value of network bandwidth across past L (designer choice) network profile events. Select the placement planned for this specific amount of bandwidth drop.

(c) Periodic outage: Select the placement planned for the effective bandwidth drop due to the outage.

(d) Unpredictable outage: All modules are placed in OBU.

Upon an increase in the bandwidth, if the policy is to minimize OBU CPU cycles, the placement is updated using the *primary–placement* selection process. If the policy is to minimize network bandwidth usage, no change in placement is made.

When new applications arrive at the workload, the placement and schedule is recomputed. However, the adaptation of the schedule to new application arrivals will depend on the policies used in Section III. If the policy is to minimize OBU CPU cycles expense, the placement and the schedule is recomputed when the application arrives. If the policy is to minimize the network bandwidth expense, depending on the scheduling approach, the application may be scheduled or made to wait.

V. Prototype Implementation

A. Cloud Server Infrastructure

We implemented the cloud services in one of the cluster nodes in a private cloud server that contained 40 CPU cores at 2.3GHz processor speed and ran Ubuntu Linux 12.04 LTS. The services were implemented in the form of Java executables (.jar) that initiate through a function call.

B. Mobile Client Infrastructure

We emulated the vehicle OBU using a Nexus 7 tablet device that ran Android operating system. We chose this device as it almost has the same processor speed (1.3 GHz) as OBUs fit in vehicles over last 2 years. Vehicle OBU operating system also run on an equivalent Linux kernel as Android. We implemented our cloud–offloading framework as a service in the Android device and prototyped apps that will use the same. The main components in our implementation include:

1) Offload Controller: The offload controller handles the client–server interactions through a TCP socket connection over a LTE cellular network. An application profiler measures each application module's execution time, input and output data size. A network profiler, interfaced with the TCP socket module, periodically measures (through ten 1kByte probe data packets) the network bandwidth (represented as network speed in bytes/sec) between the client and the private cloud server.

2) Resource allocator: We implemented the dynamic resource allocation mechanism as a Java class and integrated the same in the resource allocator. We implemented the policy definitions and run–time adaptation strategies as Java library files (.jar) that were referenced in the main class of the resource allocator.

3) Multi–App Service Interface: The Android application executables (.apk) are structured as a set of Java classes being tied to a main Java UI (user–interface) class. We implemented a Java Interface that connected the main classes from each apk file with the resource allocator through a common main class.

C. Use–Case Applications on client device

We developed two interactive (1,2) and two non–interactive applications (3,4) for cloud offloading:
(1) Gesture : a vision based hand motion gesture recognition app that recognizes four types of motions of the palm (circle, left–right, top–down, diagonal strike).
(2) TrafficLight : a vision based traffic recognition app where the camera (pointing to the road) detects the state of traffic light and recognizes traffic signs in its field–of–view.
(3) DTW : a motion classification app using motion sensing through accelerometer data that uses discrete–time warping (DTW) tool to classify temporal waveforms.
(4) Yelp: a location based service app that uploads GPS coordinates to server and retrieves the name and location of all food, gas, hospitals and mechanic shops within 10 mile radius.

The application modules corresponding to our prototype applications are depicted in Figure 6.

VI. Evaluation

We conduct experiments using our prototype implementation to study the end–to–end application performance in terms of the response–time of applications in the workload. We evaluate our cloud–offloading system in terms of its effectiveness in scheduling a workload of applications such that they meet their deadlines.

General experiment setup and methodology. The general set–up for our experiments included an Android device (tablet) docked onto the dashboard of a car. During the course of our experiments we drove the car across local and highway roads in an urban environment, within a 5–10 mile radius of the cloud server. The car speed was maintained in $[10, 50]$ miles/hour during the experiments. The workload (4 apps) was initiated on a periodic basis (every 10 seconds). Once the workload completed execution the response time and data input and output size of each module of each application were recorded. The network profiler probed a 1kB packet over 10 trials and recorded the total round–trip time for each trial over a cellular connection (LTE). We computed the network speed as the ratio of the data size of the packet to the round–trip time as the median across the 10 trials.

A	B	C	D	E
500	350	30	300	400

TABLE I
Network speed (in kBps)

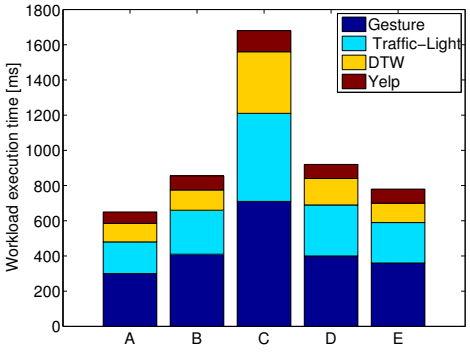


Fig. 7. Workload execution time over variable network speed (based on RTT) for 4 apps

Deadlines. Unless specified we set the deadline for each app in our evaluation as its response time considering it is the only app being executed in the OBU. To measure this response time, we profiled each app separately on the client device (see Figure 6).

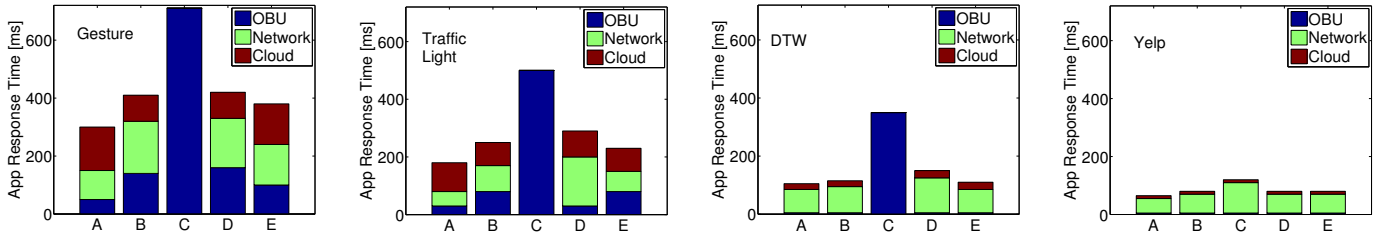
We conduct our evaluations considering to minimize the OBU CPU cycles usage and study workload execution for its adaptation to meet deadlines across different network speeds and policy selections.

A. Adapting to variable network speeds

In course of our experiments we observed that the average network speed over the LTE link between the OBU and the private cloud server, with no hops in between, was in the range of [30, 500] kBps. This was measured through the profiler that probes a 1kB packet for 10 times and computes the total RTT between cloud and OBU.

From the range of workload output samples in our experiments, we particularly pick five sample points (A, B, C, D, E) with different network speeds (Table I). We pick these points in the chronological order in which they occurred in the network profile trace. We note that these points are not contiguous in time. The goal of this evaluation is to study how our proposed system will adapt if the network bandwidth were to transition between regions shown in Table I.

In Figure 7 we plot the response time of the workload in each of the 5 cases. We can observe from Figure 7 that the response times are almost the same for A and E as well as B and D. We observed that the placement and schedule were very



(a) Gesture (deadline:650ms) (b) Traffic Light (deadline:300ms) (c) DTW (deadline:350ms) (d) Yelp (deadline:150ms)

Fig. 8. Breakdown of OBU, Networking and Cloud execution time of the four apps in the workload.

similar for (A, E) as well as for (B, D), with the difference primarily attributed to the networking times. In these cases, since the placement was planned ahead the overhead to update the placement and schedule was minimal leading to apps being able to meet their deadlines.

For case C, we observe that except for Yelp app, no app in the workload met its deadline. Our system could not find any schedule that could meet the deadline. The re–computation of the placement yielded the output that all modules of the Yelp app (since the app's data size was very small) be offloaded to cloud and the all other applications are placed in OBU.

In Figure 8 (a)–(d) we breakdown each app's response time from Figure 7 in terms of its OBU, Networking and Cloud execution times. We observe that the emphasis for offloading modules to cloud is much higher for the DTW and Yelp apps compared to Gesture and Traffic Light. This is because, the DTW app has fewer modules with long OBU execution times. So it is essential to run most of the modules of the DTW app in cloud for it to meet its deadline. On the other hand, Yelp app has fewer modules and small data size, so can effectively be placed in the cloud even at low–network speed conditions.

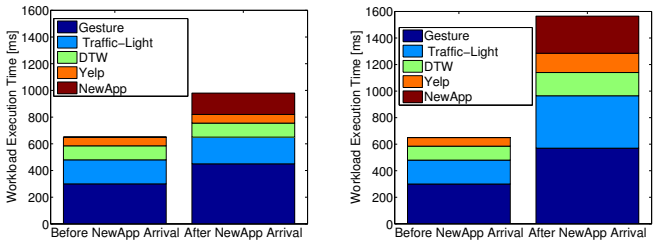
Our results from Figure 7 and Figure 8 show that our cloud–offloading system can schedule a workload with applications having different characteristics across changing network speeds.

We realize that our heuristic mechanism to plan the placements apriori is key to adapting the schedule. However, we also observe that our algorithm does not plan across all possible network conditions and that when the network condition is very poor, our system may never find a schedule to meet deadlines due to the time overhead for re-computation of the placements. One approach to address this challenge would be to predict such changes in network conditions and check if a schedule may be feasible. We hope to incorporate such predictive mechanisms in future designs of our system.

B. Policy selections when new apps arrive

For evaluation purpose, we developed an artificial app (NewApp) that we assume will be the app adding to the current workload of the 4 apps. NewApp contains 7 modules with a total execution time of 200ms. We set the data input size of all modules to 50KB.

Let us first select POLICY 1 (minimizing OBU CPU cycles usage) when executing the workload when NewApp arrives.



(a) POLICY 1 selected (b) POLICY 2 selected

Fig. 9. Workload execution time with new app arrival for different policies. Network speed corresponds to region A

We plot the workload response time before and after new app arrives considering this policy, in Figure 9 (a). We observe that the response time of the Gesture app increases after scheduling the NewApp. Since the entire network bandwidth is used, our system updates its placement such it trades off how much data it offloads from gesture recognition app and that from NewApp. In this case, all applications meet their deadlines.

Now let us select POLICY 2 (minimize network bandwidth usage) and also assume that the NewApp belongs to a class of applications that must be executed only in OBU. In this case, the OBU gets overcommitted leading to apps missing deadlines. We observe in Figure 9 (b) that all apps in the workload are late by about 100ms or less. In this case, if the policy selection were to be changed then, POLICY 3, to minimize the maximum lateness across applications, would be a good fit.

The evaluations here primarily study how our system behaves to different policy selections. We reserve the design of run–time mechanisms to adapt to different policies across the system for future work.

VII. Related Work

Code migration. Prior work in cloud offloading has proposed to offload application partitions by offloading their source–code at run–time to remote machines. MAUI [2] offloads specific annotated parts of the code at run-time to a middlebox. The machine uses an integer linear programming (ILP) solver for identifying the application partitions to be offloaded. Running ILP solvers across dynamic systems will be very challenging. Odessa [5] improves response time of perceptive

applications by leveraging parallelism in the application source code in the local and remote execution of tasks. ThinkAir [6] proposes to allot virtual machines at run–time to execute application code partitions. However, commissioning virtual machines at run–time will add high management overhead. Moreover, it makes it challenging to adapt to the dynamics of the vehicular systems. COSMOS [7] proposes to manage task allocation and offload to cloud instances running on virtual machines. However, the implementation of the framework is customized to the Android x86 processor architecture.

Offloading by replication. CloneCloud [8] proposes to optimize computation by cloning (virtualizing) the mobile device characteristics on remote machines. Such a flexibility is impractical in the vehicular context. Tango [9] proposes to replicate the code execution on both cloud and mobile device during placement selections, and picks the one that minimizes response time. While this system particularly tries to addresses the resiliency of cloud offloading across unreliable network conditions, our approach tries to conserve the resources available for offloading, and does not warrant any replication in the system.

Vehicular offloading. There has been very limited work [10], [11] and understanding of building systems for vehicular cloud offloading use–case. Carcel [12] comes closest, however, the system does not perform computational offloading. Instead it enables the cloud to have access to sensor data from autonomous vehicles as well as the roadside infrastructure for better path planning. Migration of sensory data to cloud from interactive applications can be extremely challenging in vehicular driving environments.

Placement and scheduling. [13] developed an analytical framework towards scheduling tasks for offloading, but the practicality is in question based on the assumptions about the network conditions in their model. Our design tries to address the question of how practical it is to offload application tasks when different parameters change in the system. We address the problem from a system point of view and build mechanisms that are practically viable in each step.

There has been a lot of interest in recent times in the idea of distributing computation to nearby devices through the concept of cyber–foraging [14]. The cloudlet [1] concept develops on this idea and essentially proposes to bring the cloud closer to the mobile device so as to minimize the network latency for offloading. We believe that such architectures can complement the system designs for adaptive cloud offloading systems by reducing network RTTs, however, may not address the fundamental challenge of placing and scheduling in vehicular application workloads.

VIII. Conclusion

We designed a system for offloading interactive vehicular applications with large sensor inputs where remote execution of application tasks can be availed as services. We developed a system that hosts heuristic mechanisms for partitioning applications into modules, plan placement and schedule for the modules across dynamic network conditions and variable design policies, and strategies to adapt schedule execution at run–time. Through experiments using our prototype cloud offloading system we verified that it can adapt scheduling with applications meeting their execution deadlines across variable network conditions. In future we aim to predict the network variations apriori to help simplify the dynamic resource allocation. In future we also we aim to apply the cloud computing infrastructure we developed beyond offloading applications, such as streaming, crowd—sourced based. We also aim to integrate our platform with future internet architecture (FIA) systems.

References

[1] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 153–166, New York, NY, USA, 2013. ACM.

[2] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[3] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, Hollywood, CA, 2012. USENIX.

[4] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[5] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[6] S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953, March 2012.

[7] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, and Ellen Zegura. Cosmos: Computation offloading as a service for mobile devices. In *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '14, pages 287–296, New York, NY, USA, 2014. ACM.

[8] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[9] Mark S. Gordon, David Ke Hong, Peter M. Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 137–150, New York, NY, USA, 2015. ACM.

[10] Md Whaiduzzaman, Mehdi Sookhak, Abdullah Gani, and Rajkumar Buyya. A survey on vehicular cloud computing. *J. Netw. Comput. Appl.*, 40:325–344, April 2014.

[11] H. Zhang, Q. Zhang, and X. Du. Toward vehicle-assisted cloud computing for smartphones. *IEEE Transactions on Vehicular Technology*, 64(12):5610–5618, Dec 2015.

[12] Swarun Kumar, Shyamnath Gollakota, and Dina Katabi. A cloud-assisted design for autonomous driving. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 41–46, New York, NY, USA, 2012. ACM.

[13] J. Yue, D. Zhao, and T. D. Todd. Cloud server job selection and scheduling in mobile computation offloading. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 4990–4995, Dec 2014.

[14] Rajesh Krishna Balan. *Simplifying Cyber Foraging*. PhD thesis, Carnegie Mellon University, 2006.