Computer Communications xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Vehicular Cloud Computing through Dynamic Computation Offloading

Ashwin Ashok[*,a], Peter Steenkiste[b], Fan Bai[c]

[a] Department of Computer Science, Georgia State University, Atlanta, USA
[b] Department of ECE and CS, Carnegie Mellon University, Pittsburgh, USA
[c] General Motors Research, Michigan, USA

ARTICLE INFO

Keywords:
Vehicular
Cloud computing
Offloading
Experiments
Placement
Scheduling
Real–time
Dynamic resource allocation
Interactive application

ABSTRACT

The growing demand in the number of sensor–dependent applications and infotainment services in vehicles is pushing the limits of their on–board computing resources. Today, vehicles are increasingly being connected to the Internet and becoming a part of the *Smart* Internet–of–Things (IoT) paradigm. Leveraging such connectivity, the idea of vehicular cloud–computing, where computation for vehicular applications and services are *offloaded* to the cloud, becomes an attractive proposition. However, the large sensory data inputs, strict latency requirements, and dynamic wireless networking conditions make offloading of vehicular applications to the cloud very challenging. To address this challenge, we design a dynamic approach for offloading specific vehicular application components or *modules*. We develop heuristic mechanisms for placement and scheduling of these modules in the on–board unit versus the cloud. The highlight of the proposed design is the ability to offload computation to the cloud in an elastic manner through dynamic decisions during variable network conditions. Through an experimental evaluation using our prototype system we show the effectiveness of the design in reducing the response time for compute intensive applications across variable network conditions in two urban environments.

1. 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–board cameras and motion sensors. The on–board computing resources in vehicles have been primarily allocated to accommodate the driving mechanism 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 reconfigure, service and manage such hardware over the long lifetimes of vehicles; 10–15 years.

The need for the day is the capability of vehicular computing resources to be elastic and easily reconfigurable. Improving hardware capabilities and adding IT resources in vehicles partially helps the case, but it has constraints. The vehicle has a limited amount of space to fit in more hardware components. With the notion of automated driving in picture, the sensory interfaces in vehicles in itself are becoming overwhelmingly complex. In addition, hardware additions also mean increased complexity in engineering their control and management. A potential solution to expanding the computing capabilities in vehicles is to leverage the cloud computing resources. In this regard, we promote the idea of vehicular cloud computing, where vehicular applications can benefit from *offloading* or remote execution in a cloud server. In principle, a cloud can be a remote cluster server or an proximate edge–computing unit, for example, a cloudlet [1]. The key idea here is to opportunistically utilize any remote computing resource to run vehicular applications.

Cloud–computing, while is an attractive solution for expanding vehicular information technology (IT) resources, executing vehicular applications remotely is very challenging due to some unique characteristics of the vehicular use–case and its applications. Vehicular applications are increasingly being dependent on a large array of sensory inputs. In addition, interactive applications such as those using computer vision involve large sensory inputs (e.g. high definition video) – see Fig. 1. Offloading such applications will require migrating large sensory data to the cloud which can take a lot of time depending on the network conditions. Interactive applications are particularly latency sensitive and thus large network round–trip times (RTT) will

* Corresponding author.
E-mail address: aashok@gsu.edu (A. Ashok).

https://doi.org/10.1016/j.comcom.2017.12.011
Received 8 May 2017; Received in revised form 13 December 2017; Accepted 17 December 2017

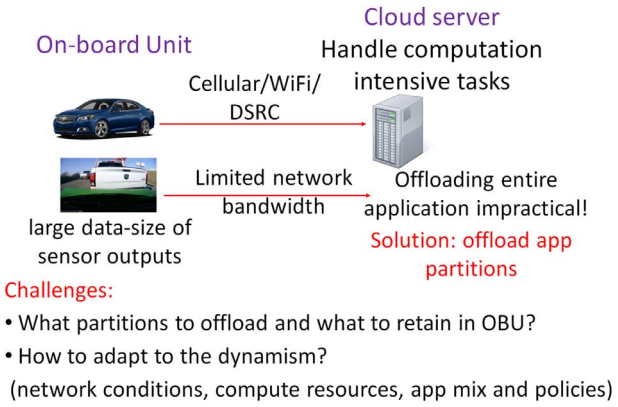


Fig. 1. Proposed offloading concept and its challenges for vehicular applications.

essentially undermine the usefulness of the fast cloud computing resource. Offloading when the vehicle is driving presents an additional challenge as the network connectivity between the vehicle and the cloud is now a variable, which may also include outages depending upon the vehicle's driving path.

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

To address the challenges in vehicular application offloading, we design a system that selectively offloads only *parts* of applications, instead of offloading the entire application. These parts, or *modules* as referred to in our design, correspond to specific set of functions as a part of the application. The design idea is to divide applications into modules and segregate a collective set of tasks from set of applications based on their computation requirements, and identify which of the modules will be offloaded to the cloud and what execute in the local on–board unit CPU. Once identified, the data corresponding to the offloaded modules is transferred to the cloud and the module computation is performed in the cloud. This design model treats that the module definitions are already available in the cloud. We introduced this treatment in this work based on our prior work that defined a *service–based offloading* [4,5] mechanism where cloud executed tasks are coded as services in the cloud server.

Designing a system for offloading specific application tasks in the vehicular context brings about key fundamental challenges in real–time resource allocation, execution and adaptation:

(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 RTT to offload application tasks over a limited network bandwidth. It must also be able to plan CPU and network bandwidth allocations for scheduling applications for different policy considerations; for example, choosing between minimizing network bandwidth expense or minimizing OBU processor cycles usage. Overall, this takes shape of a multidimensional (multiple applications and across variable constraints) scheduling problem. As we know, scheduling problems fall under NP–hard problem category. Therefore, a heuristic real–time dynamic resource allocation mechanism is required.

(ii) **Run–time Adaptation**: The system must be able to conduct the allocation and offloading in run–time as information about what applications will be initiated and the quality of the network is not known apriori. Executing such multidimensional allocation and adaption under real–time constraints brings additional challenges to the complexity of the allocation problem.**Proposed Vehicular Dynamic Computation Offloading.**To address the offloading challenges in a vehicular system, we design an adaptive cloud–offloading system that incorporates a heuristic mechanism for dynamic 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 processor cycles availability 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. The goal of our offloading system design is to ensure that the response time of embedded vehicular applications are within their stipulated deadlines, where,

Response Time is defined as the total execution time of the application measured from initiation to completion (deliver final output), and

Deadline, is defined as the maximum duration within which an application must complete execution to be regarded useful by the user.

In summary, we make the following specific contributions in this paper:

1. A framework for cloud–offloading that is flexible and can deal with vehicular applications with large sensory inputs.
2. A heuristic mechanism for partitioning and scheduling, which includes partitioning applications into tasks & identifying placement of such tasks, and computing a schedule.
3. A mechanism for adapting applications' scheduling at run–time based on the dynamics of network variations and design policy selections. The mechanism incorporates a heuristic planning–based technique that plans schedule and adapts task execution to variations in network conditions and application requirements.
4. A prototype end–end system implementation of a cloud offloading system, including four proof–of–concept cloud–computing integrated applications in an Android device.
5. Experimental evaluation of the effectiveness of the prototype offloading system, using the response time of applications as a metric. The evaluation is carried across variable network conditions, policy considerations and performed in two urban environments (Pittsburgh, USA and Atlanta, USA).

The rest of the paper is organized as follows: Section 2 motivates the importance, feasibility and challenges of cloud offloading in vehicular use–case; Section 3 presents an overview of the proposed system design; Section 4 discusses the heuristic dynamic resource allocation framework, followed by discussion of application workload structuring and placement & scheduling in Sections 5 and 6, respectively; Section 7 describes run–time adaption mechanism; Section 8 presents the prototype implementation details; Section 9 presents the macro–level evaluation results and Section 10 presents the micro–benchmarking study for the Atlanta trace data; Section 11 summarizes related work and Section 12 concludes the paper.

2. Motivation

The computational needs for vehicular applications are increasing with the growing number of sensor–based interactive applications and services. The long lifetime of vehicles (10–15 years) means however that the computational resources on most vehicles on the road are

outdated, yet, the demand for more sensor based applications and cognition in vehicular applications is fast increasing. There is a growing need for smarter applications and services in vehicles, for example, to enhance and automate driving, to improve vehicle–world interactions through sensing and communication, to provide convenience to drivers and passengers alike through cognitive applications etc. However, such applications tend to be computationally intensive thus supporting the same in the vehicle becomes a challenging.

Vehicle IT resources have improved to a large extent, however, even the state–of–the–art computing resources in vehicles lag behind the advances in smartphones, desktops, etc. by few years. Once a vehicle is manufactured it becomes extremely challenging to update in–vehicle hardware resources during its lifetime as it can incur large monetary costs and logistical management. With the increasing connectivity of vehicles to the Internet, the elastic cloud computing paradigm presents a viable resource for constantly updating the vehicle IT infrastructure and thus serving the needs of the applications using them.

2.1. Is vehicular cloud offloading feasible?

While the idea of offloading computation to the cloud is an attractive proposition, we first discuss its feasibility in a vehicular context along two dimensions: (i) offloading the entire application to the cloud, (ii) offloading only parts of the application to the cloud. In this regard, we ran two test–bench experiments where two computer vision applications, object recognition (OR) and face detection (FD), were offloaded to a cloud server using LTE connection on the phone. The experiments were conducted in an outdoor park and in the basement of a building with thick walls – the LTE signals were stronger in the park than the basement. The outdoor park location was the Woodruff Park in downtown Atlanta. The phone was handheld by an experimenter standing in the park at 50m away from the GSU CS Department building (25 Park Place, Atlanta, GA 30303). In the Basement situation, the phone was handheld by the experimenter in a room with closed glassdoors. The basement is 8 stories below the CS department at 7th floor. The cloud server was located in an office room in 7th floor of the CS department building.

We evaluate the results over the two experimental dimensions as below:

(i) **Offloading entire application.** In this setup, the camera image was directly offloaded to the cloud server for processing. We used the Microsoft Azure Cloud computing platform and directed the application to use Azure's Computer Vision API [6] for object recognition and face detection. We measured the end–to–end response time of the application when each of the use case is executed separately. In addition, we also measured the total RTT for just communicating the data associated with the offloading process. We report the total response time of the two test applications along with the RTT, averaged over 10 trials in Table 1. We clearly observe that the response times are significantly high (order of few seconds) and would be not very useful when trying to support real–time interactions and decision making. This result shows that offloading of computation of vehicular application must be done cognizant of the data load for that application and variability in network quality.

(ii) **Offloading parts of application.** In this case, we set up the test–bench system to preprocess the camera image to recognize key features (using SIFT [7]) and offload these features (as an array data structure) to the cloud. We used the Microsoft Azure Machine learning cloud service for object recognition and face detection. In this experiment we consider that the training was done apriori. We report the results of the experiments in Table 1. We can observe that the response times are reasonably within human reaction time limits (100ms) and thus can contain interactive applications.
This experimentation helps to understand the importance of judiciously offloading only parts of an interactive vehicular applications to the cloud. Such applications have strict latency requirements and can require and/or generate large amounts of data (e.g. sensor raw data, HD video and audio feed), so offloading such applications over limited wireless network bandwidths is challenging due to the large RTT involved. We refer the readers to our prior work in [4] where we have demonstrated the feasibility of offloading parts of a computer vision gesture recognition application to a cloud server with 3x performance gain over local computation and response times comparable to case (ii) discussed above.
While we have shown its beneficial to offload parts of a single application, the question of importance, and what we aim to address in this paper, is to evaluate the feasibility and develop the mechanism for offloading computation from *multiple* applications in the vehicle and yet achieving the target response times for each application. In this regard, we now highlight the key goal for this work and enlist the challenges that we aim to address through our system design.

Table 1
Response time (including RTT) for cloud offloading in different network conditions. Network RTT represents the average RTT for ten 1KB probe packets between phone and cloud. The response times for OR and FD are inclusive of the RTT. Values X/Y/Z represent the response time for each case (i) X = offloading entire application/ case (ii) Y = offloading parts of application/ case (iii) Z = running the computation on the local machine (Intel i5 CPU) respectively.

Location	Network RTT [ms]	OR [ms]	FD [ms]
Open park	400/35/0	2300/80/1500	1200/72/900
Basement	1100/100/0	8020/153/1500	6200/134/900

2.2. Design goal and challenges

The *goal of our design* is to ensure that the response time of embedded vehicular applications are within a *deadline* stipulated for the application.

As defined earlier, the *deadline* is the maximum duration within which an application must complete execution to be regarded useful by the system user. The key challenges in designing a cloud–offloading system towards achieving the goal are to:

- Designing an offloading framework that is flexible and can deal with large inputs (e.g. sensor based applications).
- Identify and schedule the different tasks from multiple applications for execution in OBU or the cloud, considering limited OBU CPU cycles and network bandwidth constraints.
- Adapt the placement, scheduling and application execution at run–time across variable network round trip times and application policies.

3. System overview

3.1. Application task model

Sensor–based applications in vehicles typically are a collection of tasks with different execution times, data size inputs, etc. We characterize the properties of these tasks through a model. We divide an application into individual units called *modules* [5]. We regard 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. Modules may execute sequentially or in parallel across an application. 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.

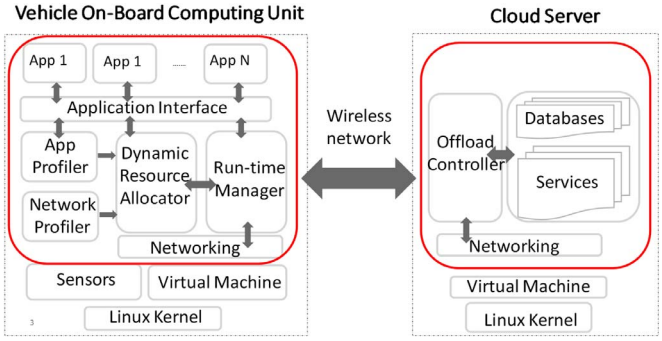


Fig. 2. Service–Oriented Cloud–Offloading System Architecture.

3.2. Cloud–Offloading system architecture

We consider that applications' modules' functionality are available in the cloud in the form of *services*. As shown in Fig. 2, the system includes a vehicle OBU (client) running multiple applications and a cluster of machines in the cloud (server) that provide application module functionality 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.

In the *service–based approach*, 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), represented as the RTT measurements of probe packets between the client and the cloud server through a cellular connection. The probing process is done periodically using fixed–size probe packets over specific time–windows. In our prototype design, the network profiler uses 10 probe packets of size 1KB each. The RTT is determined as an average of the RTT over these 10 probe packets. The probing is done on periodic basis, every 5 seconds, and following each probing the payload is delivered at the available network speed with a packet size of 1KB. The latency overhead depends on the network speed and can be of the order of 10s – 100s of ms in typical LTE setting. The probing is done in a parallel thread and thus will not interfere with the offloading process.

The core components of our cloud–offloading system include: (a) Dynamic resource allocation unit, 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.

3.3. Design scope and policies

The scope of our system design includes:

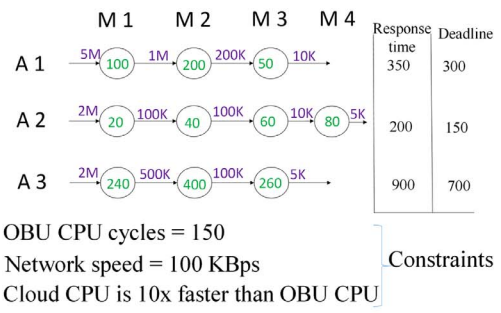
(i) Applications arriving at the resource allocation unit within a window of time (specified by designer) must be considered in the workload for resource allocation.
(ii) The resource allocation unit will consider only aperiodic or bursty application arrivals.
(iii) No part(s) of applications deemed 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.

4. Dynamic resource allocation

Our design of the dynamic offloading system is motivated by the lessons we learned through our prior work in designing a proof–of–concept service–based cloud offloading system [4]. We designed a framework for placement of application tasks or modules in OBU or cloud that used an exhaustive search mechanism for partitioning the OBU and cloud assignments based on the network bandwidth available. While this system showed about 3x improvement in application response time the framework considered only 1 application in its workload. The 3x gain may not translate linearly as we scale the workload. This is because exhaustive search based placement considering multiple application workload will incur high computation cost and may not be practical for adaptation at run–time. A judicious approach to address



A workload example with 3 applications with maximum 4 modules each

OBU CPU Cycles

	M 1	M 2	M 3	M 4
A 1	100	200	50	0
A 2	20	40	60	80
A 3	240	400	260	0

Data Input size

	M 1	M 2	M 3	M 4
A 1	5	1	0.2	0
A 2	2	0.1	0.1	0.1
A 3	2	0.5	0.1	0

Data Output size

	M 1	M 2	M 3	M 4
A 1	1	2	0.1	0
A 2	0.1	0.1	0.1	0.1
A 3	0.5	0.1	0.1	0

Tables to represent module characteristics of applications in workload

Fig. 3. Illustration of a tabular structure to represent workload.

this would be to identify all the resources and the requirements and provide a holistic and practical resource allocation strategy for cloud offloading. The goal of the resource allocation framework will then be to make decisions on what application modules are partitioned and placed in OBU or cloud, and to find a schedule for their execution.

4.1. Proposed DRA framework

One approach to addressing the resource allocation problem is to formulate it as an optimization problem. In general, both, placement and scheduling fall under the class of NP–hard problems. The additional challenge in this approach is that it is difficult to capture multiple policies for executing the vehicular applications through a single optimization goal. For example, a cost metric for minimizing the total network bandwidth usage treats the case where applications complete execution by their deadlines, however, minimizing the maximum lateness across applications treats the case when applications are late. While several heuristics for scheduling problems have been proposed before [8], the vehicular cloud–offloading context brings about the challenge of capturing the variability across multiple dimensions such as CPU cycles, network resources (bandwidth), application deadlines, policies, etc through a single optimization problem. Moreover, it is unclear as to whether addressing all such parameters in an optimization problem would converge to any solution. Testing the feasibility of such a convergence is a challenging theoretical problem.

We take an approach to address the dynamic resource allocation by developing a heuristic mechanism that disintegrates the problem of placement and scheduling from adaptation. The goal of the heuristic mechanism is to find a practical placement and schedule that the execution and adaption mechanisms can agree upon through a planning phase prior to run–time execution by the system. In particular we base our proposed heuristic approach on the following key insights:

(Insight 1) For sensor dependent and interactive applications sequential execution control flow of modules implies that the data size will shrink across the application modules. Interactive applications and cognitive applications typically depend on raw sensor data. The raw data is usually filtered and processed to infer further information or use the data in another module. This means that the largest data size in the application will be that of the raw sensor data. So, for each application, when 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, therefore avoiding the need to exhaustively check placements for each module, thus reducing the computation complexity of the placement process. The consideration of data shrinkage across modules is invalid if any of the modules in the application depend on data from another module from a different application, service or sensor. In this work, we do not consider such module dependencies and treat modules of are independent across applications.

(Insight 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 executing a schedule at run–time.

5. 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 Fig. 3 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.

The key motivation for using the tabular data structure comes from the fact that generating and navigating a matrix (table) can be done much faster than traversing well known 2D graphs and trees. The table presents a one–stop shop for storing the data as well as a way to structure the parameters. With advanced optimized mechanisms to deal with matrices today through fast mathematical shortcuts and fast computing elements (GPU), the tabular data structure is an attractive proposition for our use–case. It helps in conserving both, storage space and execution time to generate and/or navigate across the structure. The matrix size can be predetermined based on the system and no complex dynamic indexing will be required at run–time. In addition, while not addressed in this paper, the dependencies across modules can be represented by indexing the cells using markers instead of creating actual logical interconnects as in graphs or trees.

6. 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 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; minimal network bandwidth expense, minimize OBU cycles usage and minimize maximum lateness across applications.
(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). Since the response time of an application depends on the placement (OBU or cloud) and the schedule, the slack for an application is a variable quantity. 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.

6.1. 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 (n/w BW) constraint. We will refer to this placement as *primary–placement*.

Algorithm for finding *primary–placement*

Step 1: Select application with least-slack

Step 2:

(1) Select least indexed module not placed in cloud

(2) Place the module in cloud.

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

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

Step 3:

(1) Check if (Required n/w BW) ≥ (Available n/w BW). If not, go to Step 1

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

(3) Update slack for all applications

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

Step 4:

(1) If residual OBU cycles > 0 AND min slack > 0, select placement as *primary placement*

(2) If not, make workload policy changes and go to Step 1. Consider:

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

(ii) remove application(s) from the workload

6.2. 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 from the second insight that the network bandwidth and OBU CPU cycles usage follow a tradeoff, 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.

6.3. 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 Figs. 3 in 4.

Scheduling algorithm

Step 1:

Update slack for each application based on placement

Step 2:

(1) Select application with least-slack.

(2) Schedule least-indexed module. Update slack.

(3) Repeat Step 2 if no application is late.

(4) Exit if all modules have been scheduled.

Step 3:

(1) If any application is late, update placement*

(2) 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. Essentially the least–slack first heuristic inherently helps to address POLICY 3, that is, to *minimize the maximum lateness across the applications in the workload.*

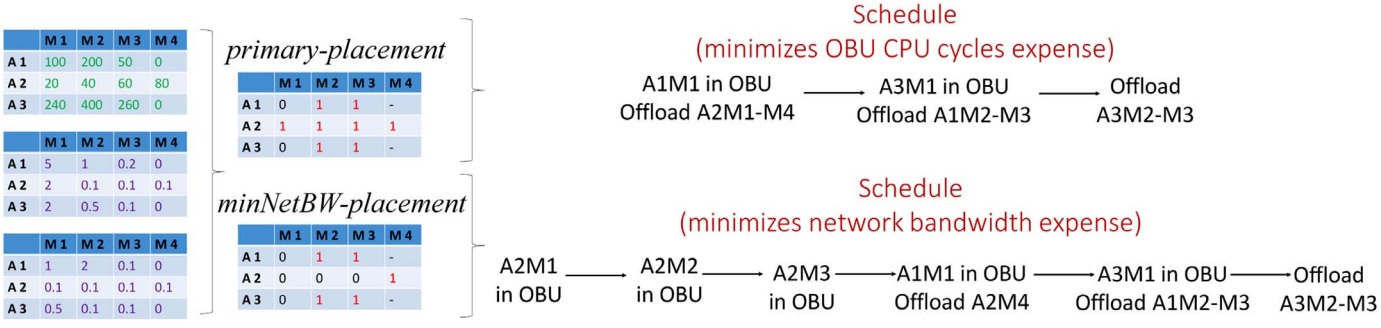


Fig. 4. Example showing the *primary–placement* and *minNetBW–placement* for workload in Fig. 3, and the schedules computed for each.

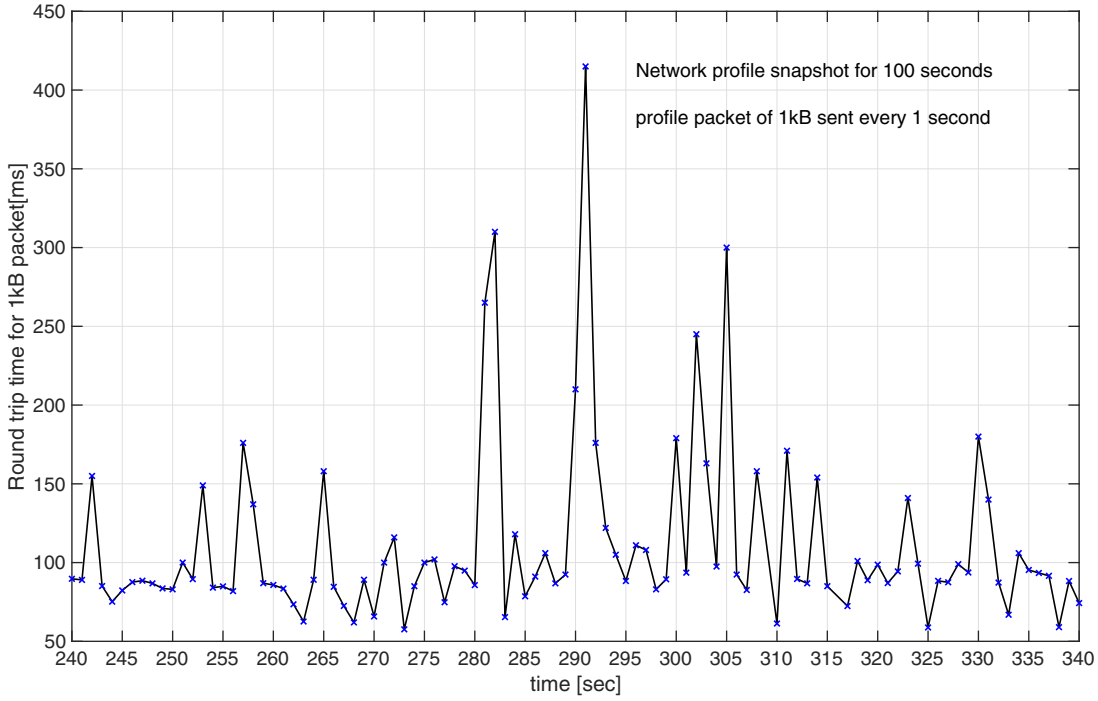


Fig. 5. Snapshot of the network profile represented in terms of RTT of a 1KB probe packet, sampled every 1sec, over a LTE wireless link between client (phone in car) and cloud (university building). The network measurement was conducted in CMU university campus.

7. Run–Time execution and adaptation

The dynamic resource allocator conducts the placement identification and scheduling process as soon as the applications arrive at the workload within an arrival time window. Only when it is ascertained that it is feasible to schedule the workload it is executed by the run–-time adaptive controller. 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 depends on the policies.

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

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

(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.

7.1. Example run–through

Let us understand the adaptation process to network variations using an example. As shown in Fig. 5 we consider a snapshot of a network profile, in terms of the RTT measured using probe packets of size 1KB between a phone docked on a driven vehicle and a static cloud server. We will study the adaption process across four contiguous 10 second snapshots with reference to the RTT across these windows from Fig. 5:

(a) [270 – 280] : Network speed is consistent. The placement process must have planned for this network speed. The pre–planned schedule is executed.
(b) [280 – 290] : Network speed significantly drops. The system selects the placement for the specifc range in bandwidth drop. In this case more modules may require to be executed locally in the OBU. However, since some modules may already have been offloaded previously, the updates may require only updating the cloud with the differential data. If the bandwidth is too low (below thresholds) the execution is moved to OBU. Such a planning is taken care in the resource allocation stage and hence incurs minimal exection time during run–time.
(c) [290 – 300] : Network speed significantly increases. The primary–placement strategy essentially plans for such a case as it is beneficial to have a high network speed to proactively offload as many modules as possible to the cloud. So, the system checks for a placement updates, if any new module can be placed in cloud, and updates the schedule and executes the same.
(d) [300 – 310] : Network speed in intermmitent. The system updates placement for the least network speed and if the network quality improves consistently in future the placement is made more agressive towards offloading. Predicting future network changes is out of scope of this paper and would be a significant contribution to the improvements in this design in future.

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 4. 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.

8. Prototype implementation

8.1. 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.

8.2. 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:

8.2.1. 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.

8.2.2. 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.

8.2.3. 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.

8.3. 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.

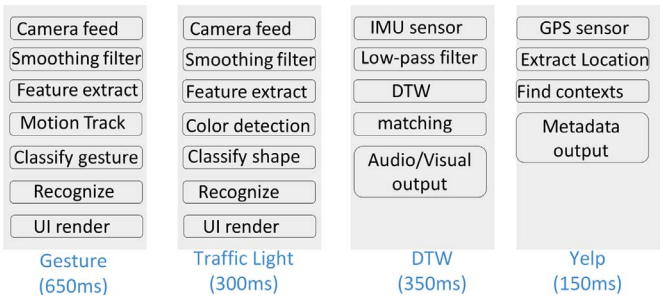


Fig. 6. Modules of our prototype apps (numbers represent the response time of each app when only one app executes in the workload). Each rectangle block is a module m_i with index starting at 1 from top to bottom.

Table 2
Execution time of each module in milliseconds.

App	G	T	D	Y
m1	10	20	10	10
m2	50	50	50	20
m3	100	50	200	110
m4	150	10	80	20
m5	270	110	20	–
m6	50	50	–	–
m7	10	10	–	–

Table 3
Data output size of each module in KB.

App	G	T	D	Y
m1	5K	5K	1	1
m2	5K	5K	1	0.5
m3	1	1	1	0.5
m4	1	1	0.5	0.1
m5	1	1	0.1	–
m6	0.5	0.5	–	–
m7	0.1	0.1	–	–

The application modules corresponding to our prototype applications are depicted in Fig. 6. We report the module execution time and data output sizes in Tables 2 and 3 respectively. The application is divided into modules based on the data input and data output of each module. The application profiler marks the size and data type and structure of data feed and generated within the application. The profiler marks a block of application code as a module when the data in and data out of the block of the code differ in size and types. The marking of modules is done offline using the application profiler for each application in consideration, and is one–time process.

9. 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 (Pittsburgh, USA), within a 5–10 mile radius of the cloud server, located at CMU campus. 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

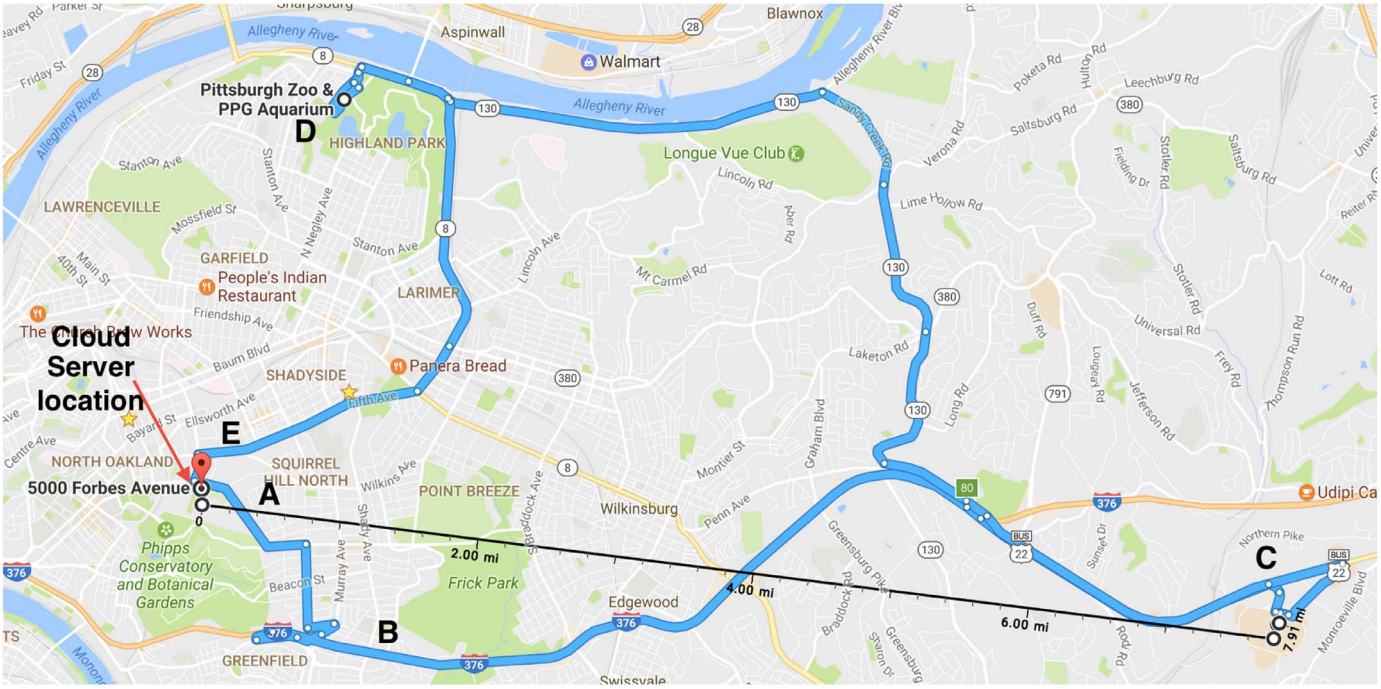


Fig. 7. Pittsburgh network profile driving map.

Table 4
Network speed at five different locations in Pittsburgh driving route (in KBps).

A	B	C	D	E
500	350	30	300	400

speed as the ratio of the data size of the packet to the round–trip time as the median across the 10 trials.

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 Fig. 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.

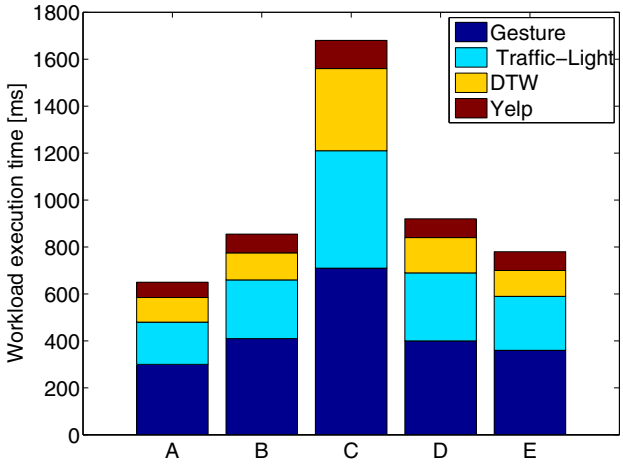


Fig. 8. Workload execution time over variable network speed (based on RTT) for 4 apps.

9.1. Adapting to variable network speeds

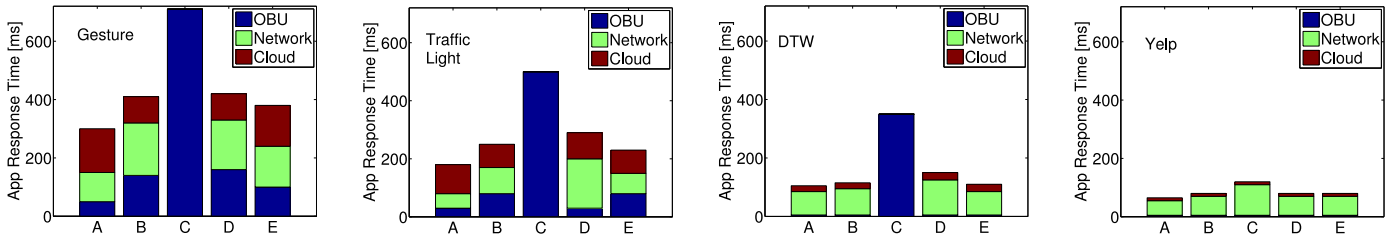
In course of our experiments we observed that the average network speed over the LTE link between the OBU and the private cloud server, with no hops in between, was in the range of [30, 500] kBps. This was measured through the profiler that probes a 1KB packet for 10 times and computes the total RTT between cloud and OBU. The driving route is as shown as in Fig. 7.

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 4). 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 4.

In Fig. 8 we plot the response time of the workload in each of the 5 cases. We can observe from Fig. 8 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 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.

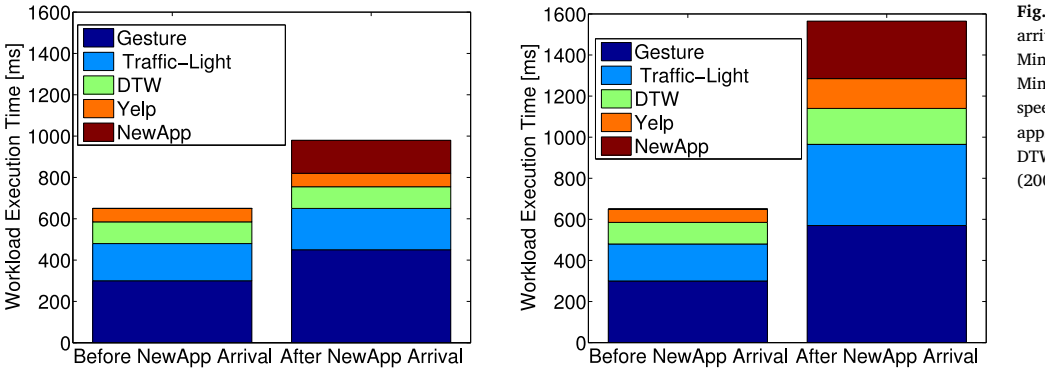
For case C, we observe that except for Yelp app, no app in the workload met its deadline. Our system could not find any schedule that could meet the deadline. The re–computation of the placement yielded the output that all modules of the Yelp app (since the app's data size was very small) be offloaded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

In Fig. 9 (a)–(d) we breakdown each app's response time from Fig. 8 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



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

Fig. 9. Breakdown of OBU, Networking and Cloud execution time of the four apps in the workload.



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

Fig. 10. Workload execution time with new app arrival for different policies. POLICY 1 = Minimizing OBU cycles usage; POLICY 2 = Minimizing network bandwidth usage. Network speed corresponds to region A. Deadlines for the apps are: Gesture (650 ms), Traffic Light (300ms), DTW (350 ms), Yelp (150 ms) and NewApp (200 ms).

modules and small data size, so can effectively be placed in the cloud even at low–network speed conditions.

Our results from Figs. 8 and 9 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.

9.2. 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. We plot the workload response time before and after new app arrives considering this policy, in Fig. 10 (a). In this case, all applications meet their deadlines as the response times are: Gesture (450 ms), Traffic Light (200ms), DTW (100 ms), Yelp (60 ms), NewApp (190 ms). 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.

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 overloaded leading to apps missing deadlines. We observe in Fig. 10 (b) that all apps in the workload are late by about 100ms. 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.

10. Microbenchmarking

10.1. Workload evaluation

To verify system performance in different geographical locations, we repeated the Pittsburgh app response time evaluation experiments in downtown Atlanta. We duplicated the Pittsburgh setup by docking the phone onto a car and drove around downtown Atlanta over 2 mile loop from Georgia State University (GSU) campus. The cloud server was located in the 7th floor of the computer science department at GSU. The driving route is shown in Fig. 11.

We observed that the workload was not being completed within the deadlines and the compute intensive computer vision apps were always late. While the network speed was acceptable for low data rate applications, it was not sufficient for offloading data intensive applications. We observed that the network latency was of the order of 500 ms as opposed to 100 ms in Pittsburgh CMU campus experiments. We believe this is due to the poor LTE cellular network connectivity in downtown Atlanta area due to the urban canyon effect.

10.2. Tracebased analysis

To study the effectiveness of our approach in downtown Atlanta we conducted a trace based analysis. We executed the workload in the Android phone by iterating over each network RTT value from the network trace shown in Fig. 12. We report the minimum and maximum values of the lateness of each application when the workload was

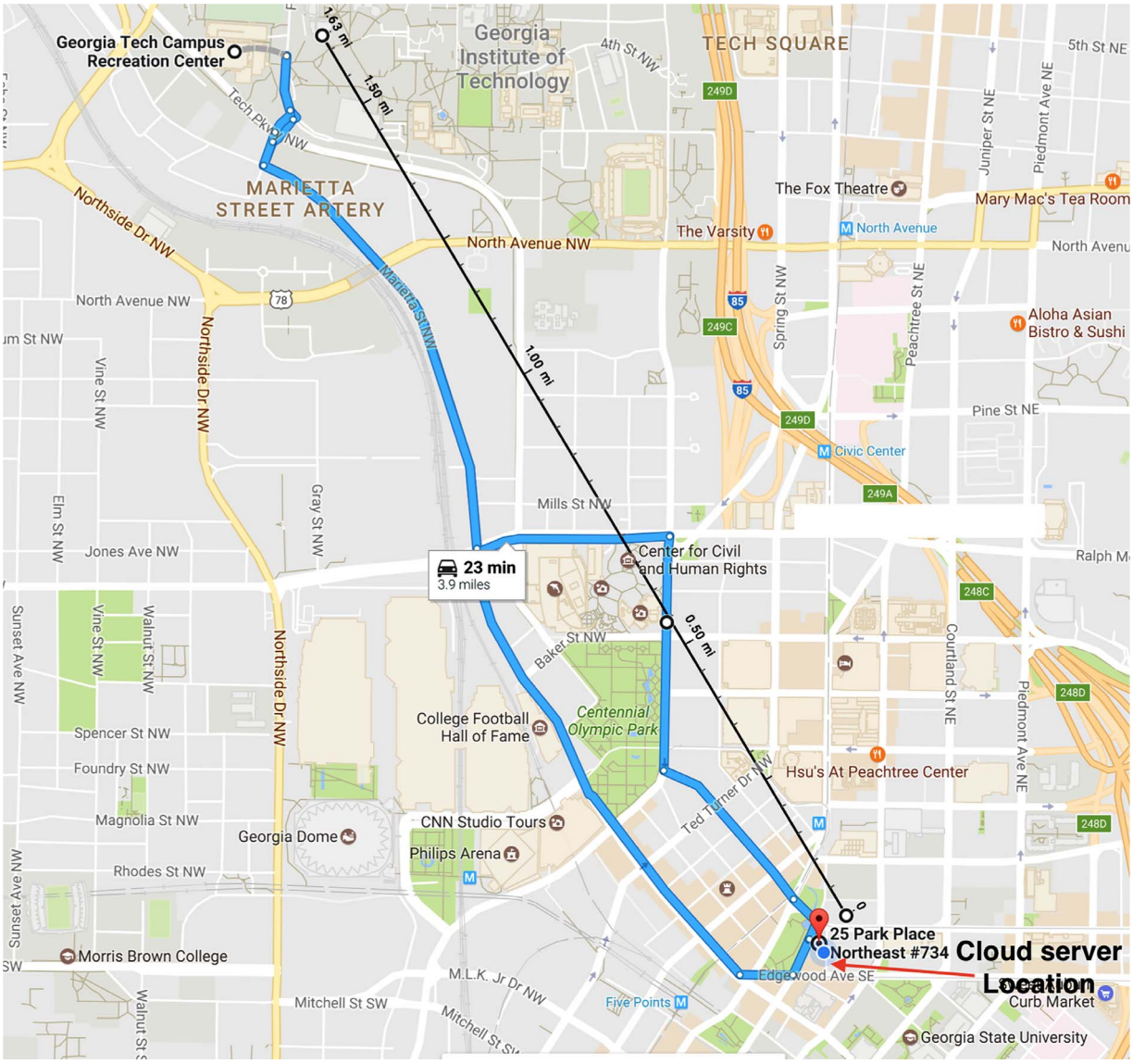


Fig. 11. Atlanta network profile driving map.

initiated periodically (every 1 s, every 10 s) in Table. 5

We observe that apps got more late if the workload was initiated every 10 s as the network speed (or RTT) variation was more significant in a 10sec window, compared to 1s window. This means that frequent adaptation of workload schedule over short time periods may not be beneficial all the time. The system must be able to adapt fast yet such adaptation must happen cognizant of the network quality changes. Clearly, knowledge of how the network will behave over a short period of future time is beneficial in this regard, however, such network predictions in vehicular settings is challenging. This motivates for future work in predicting network quality and planning the system adaptations for offloading under those predicted network conditions.

Another related question, when considering network quality, is that of scalability – *does the offloading work when there are large number of vehicles trying to benefit from the heuristic offloading mechanism*. The proposed mechanism treats that applications in a workload, whether they are part of a workload in one car or a set of workloads in multiple cars, are treated independent from one another. This means that the feasibility of offloading from a vehicle depends upon the chunk of shared network bandwidth available to the vehicle. Our mechanism treats that the numerical value of this bandwidth can be determined through profiling. However, addressing how the network can be shared across different vehicles remains out of scope of this paper. We believe solutions for the same can be inspired from the large body of work done in mobile and vehicular adhoc wireless networks. In our model, the heuristic is executed periodically in each vehicle for its workload. We consider the network bandwidth as the only external parameter. Hence, the number of vehicles does not impact the heuristic strategy but will impact the overall outcome of the offloading (e.g. total response time) as the network ravailability changes in this case.

11. 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 [9] improves response time of perceptive applications by leveraging parallelism in the application source code in the local and remote execution of tasks. ThinkAir [10] proposes to allot virtual machines at run–time to execute application code partitions. However, commissioning virtual machines at run–time will add high

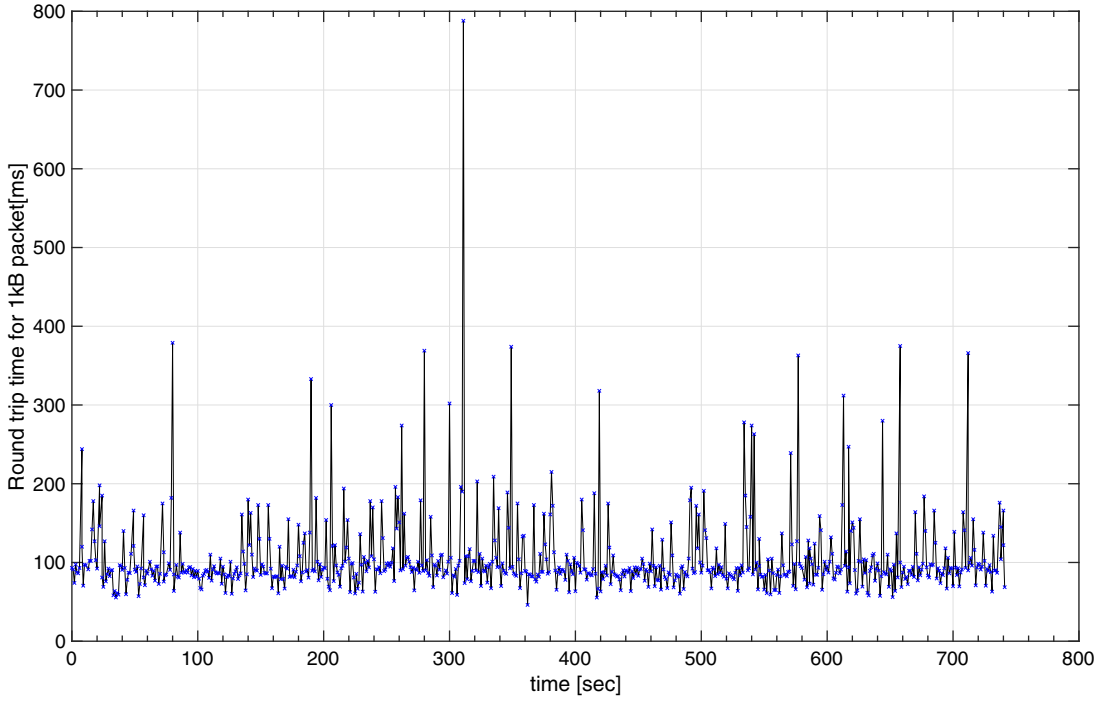


Fig. 12. Network profile represented as RTT for 1KB probe packet between client and cloud through a LTE wireless network in downtown Atlanta, USA.

Table 5
App lateness (min, max) [ms] for the network profile snapshot in Fig. 12.

App	Gesture	Traffic	DTW	Yelp
1s	(20,1010)	(40,1200)	(500,800)	(0,1250)
10s	(250,1010)	(400,1200)	(600,800)	(0,1250)

management overhead. Moreover, it makes it challenging to adapt to the dynamics of the vehicular systems. COSMOS [11] 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 [12] proposes to optimize computation by cloning (virtualizing) the mobile device characteristics on remote machines. If applied to vehicular context, each cloud server must contain the clone of the vehicle's computing unit. Virtualizing a vehicle in a cloud machine is impractical due to the sheer complexity in executing such a process. Moreover, vehicle computing units do not just process data for applications but also cater to the driving of the vehicle which is a mechanical process. Cloning the same is an impractical and inefficient process. Tango [13] 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 [14,15] and understanding of building systems for vehicular cloud offloading use–case. Carcel [16] 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. [17,18] have developed analytical frameworks towards scheduling tasks for offloading, but the practicality is in question based on the assumptions about the network conditions and the practicality of converging to an optimum solution. 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.

Vehicular cloud computing. There has been a lot of interest in recent times in the idea of distributing computation to nearby devices through the concepts of Edge/Fog computing. The concept of cyber–-foraging [19] and cloudlet [1] propose to bring the cloud closer to the mobile device so as to minimize the network latency for offloading. Recent works have explored the idea of fog computing for vehicular applications and also the idea of using vehicles as cloud infrastructures [20,21]. The idea of vehicular cloud formation by cluster of vehicles driving on the road was developed in 2012 by Prof. Gerla et al. [22]. It promotes the idea of using a vehicle and/or cluster of vehicles as a computing infrastructure. The idea of using vehicles as a cloud computing unit is interesting however the practical challenges of deployment is still a question unsolved. For example, the heterogeneity (vehicle make/model), wireless connectivity in the cluster, robustness of cluster formation, privacy/security policies across automakers are impediments in bringing this concept to reality.

In essence, fog/edge computing and vehicular clouds promote the idea of moving the cloud closer to the vehicle and opportunistically availing the nearest computing elements. Our work asks the question of how one would execute a workload of applications given there is a cloud server (a core cloud platform, an edge computing unit or a proximate device). We believe that edge computing and fog computing architectures can complement the system designs for adaptive cloud offloading systems by reducing network RTTs, however, must still address the fundamental challenge of placing and scheduling in vehicular application workloads. We strive to answer this fundamental question through our work.

12. 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 and crowd—sourcing in Internet–of–Things applications.

References

[1] K. Ha, P. Pillai, W. Richter, Y. Abe, M. Satyanarayanan, Just-in-time provisioning for cyber foraging, Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '13, ACM, New York, NY, USA, 2013, pp. 153–166, http://dx.doi.org/10.1145/2462456.2464451.

[2] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, Maui: making smartphones last longer with code offload, Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys '10, ACM, New York, NY, USA, 2010, pp. 49–62, http://dx.doi.org/10.1145/1814433.1814441.

[3] M.S. Gordon, D.A. Jamshidi, S. Mahlke, Z.M. Mao, X. Chen, Comet: code offload by migrating execution transparently, Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), USENIX, Hollywood, CA, 2012, pp. 93–106.

[4] A. Ashok, P. Steenkiste, F. Bai, Enabling vehicular applications using cloud services through adaptive computation offloading, Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services, MCS '15, ACM, New York, NY, USA, 2015, pp. 1–7, http://dx.doi.org/10.1145/2802130.2802131.

[5] A. Ashok, P. Steenkiste, F. Bai, Adaptive cloud offloading for vehicular applications, 2016 IEEE Vehicular Networking Conference (VNC), (2016), pp. 1–8, http://dx.doi.org/10.1109/VNC.2016.7835966.

[6] M. Azure, Microsoft Azure Computer Vision API, (2015).

[7] D.G. Lowe, Distinctive image features from scale-invariant keypoints, Int. J. Comput. Vis. 60 (2) (2004) 91–110, http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

[8] R.I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Comput. Surv. 43 (4) (2011) 35:1–35:44, http://dx.doi.org/10.1145/1978802.1978814.

[9] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, R. Govindan, Odessa: enabling interactive perception applications on mobile devices, Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys '11, ACM, New York, NY, USA, 2011, pp. 43–56, http://dx.doi.org/10.1145/1999995.2000000.

[10] S. Kosta, A. Aucinas, P. Hui, R. Mortier, X. Zhang, Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading, INFOCOM, 2012 Proceedings IEEE, (2012), pp. 945–953, http://dx.doi.org/10.1109/INFCOM.2012.6195845.

[11] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, E. Zegura, Cosmos: computation offloading as a service for mobile devices, Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc '14, ACM, New York, NY, USA, 2014, pp. 287–296, http://dx.doi.org/10.1145/2632951.2632958.

[12] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti, Clonecloud: elastic execution between mobile device and cloud, Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, ACM, New York, NY, USA, 2011, pp. 301–314, http://dx.doi.org/10.1145/1966445.1966473.

[13] M.S. Gordon, D.K. Hong, P.M. Chen, J. Flinn, S. Mahlke, Z.M. Mao, Accelerating mobile applications through flip-flop replication, Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '15, ACM, New York, NY, USA, 2015, pp. 137–150, http://dx.doi.org/10.1145/2742647.2742649.

[14] M. Whaiduzzaman, M. Sookhak, A. Gani, R. Buyya, A survey on vehicular cloud computing, J. Netw. Comput. Appl. 40 (2014) 325–344, http://dx.doi.org/10.1016/j.jnca.2013.08.004.

[15] H. Zhang, Q. Zhang, X. Du, Toward vehicle-assisted cloud computing for smartphones, IEEE Trans. Veh. Technol. 64 (12) (2015) 5610–5618, http://dx.doi.org/10.1109/TVT.2015.2480004.

[16] S. Kumar, S. Gollakota, D. Katabi, A cloud-assisted design for autonomous driving, Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 41–46, http://dx.doi.org/10.1145/2342509.2342519.

[17] J. Yue, D. Zhao, T.D. Todd, Cloud server job selection and scheduling in mobile computation offloading, Global Communications Conference (GLOBECOM), 2014 IEEE, (2014), pp. 4990–4995, http://dx.doi.org/10.1109/GLOCOM.2014.7037596.

[18] S.E. Mahmoodi, R.N. Uma, K.P. Subbalakshmi, Optimal joint scheduling and cloud offloading for mobile applications, IEEE Trans. Cloud Comput. PP (99) (2016) 1–1. doi:10.1109/TCC.2016.2560808.

[19] R.K. Balan, Simplifying Cyber Foraging, Ph.D. thesis, Carnegie Mellon University, 2006.

[20] S. Yi, Z. Hao, Z. Qin, Q. Li, Fog computing: platform and applications, 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), (2015), pp. 73–78, http://dx.doi.org/10.1109/HotWeb.2015.22.

[21] X. Hou, Y. Li, M. Chen, D. Wu, D. Jin, S. Chen, Vehicular fog computing: a viewpoint of vehicles as the infrastructures, IEEE Trans. Veh. Technol. 65 (6) (2016) 3860–3873, http://dx.doi.org/10.1109/TVT.2016.2532863.

[22] M. Gerla, Vehicular cloud computing, 2012 The 11th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net), (2012), pp. 152–155, http://dx.doi.org/10.1109/MedHocNet.2012.6257116.