# Mitigating Multi-tenant Interference in Continuous Mobile Offloading

Zhou Fang[1(✉)], Mulong Luo[2], Tong Yu[3], Ole J. Mengshoel[3], Mani B. Srivastava[4], and Rajesh K. Gupta[1]

[1] University of California San Diego, San Diego, USA
zhoufang@ucsd.edu
[2] Cornell University, Ithaca, USA
[3] Carnegie Mellon University, Pittsburgh, USA
[4] University of California, Los Angeles, Los Angeles, USA

**Abstract.** Offloading computation to resource-rich servers is effective in improving application performance on resource constrained mobile devices. Despite a rich body of research on mobile offloading frameworks, most previous works are evaluated in a single-tenant setting, *i.e.*, a server is assigned to a single client. In this paper we consider that multiple clients offload various continuous mobile sensing applications with end-to-end delay constraints, to a cluster of machines as the server. Contention for shared computing resources on a server can unfortunately result in delays and application malfunctions. We present a two-phase *Plan-Schedule* approach to mitigate multi-tenant resource contention, thus to reduce offloading delays. The planning phase predicts future workloads from all clients, estimates contention, and devises offloading schedule to remove or reduce contention. The scheduling phase dispatches arriving offloaded workloads to the server machine that minimizes contention, according to the running workloads on each machine. We implement the methods into *ATOMS* (Accurate Timing prediction and Offloading for Mobile Systems), a framework that adopts prediction of workload computing times, estimation of network delays, and mobile-server clock synchronization techniques. Using several mobile vision applications, we evaluate ATOMS under diverse configurations and prove its effectiveness.

## 1 Introduction

**Problem Background:** Recent advances in mobile computing have made many interesting vision and cognition applications feasible. For example, cognitive assistance [1] and augmented reality [2] applications process continuous streams of image data to provide new capabilities on mobile platforms. However, advances in computing power on embedded devices do not satisfy such growing needs. To extend mobile devices with richer computing resources, offloading computation to remote servers has been introduced [1,3–5]. The servers can be either deployed in low-latency and high-bandwidth local clusters that provide timely offloading


M. Luo and L.-J. Zhang (Eds.): CLOUD 2018, LNCS 10967, pp. 20–36, 2018.
https://doi.org/10.1007/978-3-319-94295-7_2

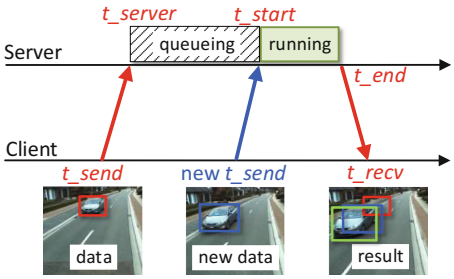



**Fig. 1.** ATOMS predicts processor contention and adjusts the offloading time (*t_send*) to avoid contention (queueing). (Color figure online)

services, as envisioned by the cloudlet [4], or the cloud that provides best-effort services.

A low end-to-end (E2E) delay on a sub-second scale is critical for many vision and cognition applications. For example, it ensures seamless interactivity for mobile applications [1], and a low sensing-to-actuation delay for robotic systems [6]. Among previous works on reducing offloading delays [1,10], a simple single-tenant setting that one client is assigned to one server is usually used to evaluate prototypes. However, in a practical scenario that servers handle tasks from many clients running diverse applications, contention on the shared server resources may raise up E2E delays and degrade application performance. Unfortunately, this essential issue of multi-tenancy is still untapped in these works.

While cloud schedulers have been well engineered to handle a wide range of jobs, new challenges arise in handling offloaded mobile workloads. First, there are stringent limits on server utilizations for conventional low latency web services [7]. However, computer vision and cognitive workloads are much more compute-intensive, which results in a large infrastructure cost to keep utilization levels low. Indeed, it is even not feasible for cloudlet servers that are much less resourceful than cloud. Second, there are many works on scheduling of batch data processing tasks with time-based Service-Level-Objectives (SLOs) [8,11,12]. However, these methods are inadequate in handling mobile workloads that desire sub-second E2E delays, compared to data processing tasks with minutes makespans and deadlines to hours.

**Our Approach:** This paper presents *ATOMS*, a mobile offloading framework that maintains low delays even under a high server utilization. Motivated by low-latency mobile applications, ATOMS consider a cloudlet setting where mobile clients connect to servers via high-bandwidth Wi-Fi networks, as in [1,4]. On the basis of load-aware scheduling, ATOMS controls future task offloading times in a client-server closed loop, to remove processor contention on servers. See Fig. 1, a client offloads an object detection task to a multi-tenant server. Due to processor contention, it may be queued before running. By predicting processor contention, the server notifies the mobile client to postpone offloading.


(a) Processor Contention (b) ATOMS

**Fig. 2.** 4 clients offload DNN object detection tasks to a 8-core server with periods 2 s to 5 s. The time slots of offloaded tasks are plotted and the load line (red) gives the total number of concurrent tasks. The dashed curve gives the CPU usage (cores) of the server. (Color figure online)

The postponed offloaded task is processed without queueing, and thus provides a more recent scene (blue box) that better localizes the moving car (green box).

Accordingly, we propose the *Plan-Schedule* strategy: (i) in the ***planning*** phase, ATOMS predicts time slots of future tasks from all clients, detects contention, coordinates tasks, and informs clients about new offloading times; (ii) in the ***scheduling*** phase, for each arriving task, ATOMS selects the machine that has minimal estimated processor contention to execute it. Figure 2 illustrates the effectiveness of ATOMS for removing processor contention. Figure 2a shows the time slots of periodically offloaded tasks. The load lines (red) give the total number of concurrent tasks, and contention (queueing) takes place when the load exceeds 1. Figure 2b shows that contention is almost eliminated because of the dynamic load prediction and the task coordination by ATOMS.

The challenge of deciding the right offloading times is that the server and the clients form an *asynchronous distributed system*. For scheduling activities, the uncertainties of wireless network delays and clock offsets must be carefully considered in the timeline of each task. ATOMS leverages accurate estimations on bounds of delays and offsets to handle the uncertainties. Variabilities of task computing times put additional uncertainties on the offloading timing, which are estimated by time series prediction. The predictability relies upon the correlation of continuous sensor data from cameras.

In addition, to ensure a high usability in diverse operating conditions, ATOMS includes the following features: (i) the support for heterogeneous server machines and applications with different levels of parallelism; (ii) the client-provided SLOs that control the offloading interval deviations from the desired period, which are caused by dynamic task coordination activities; (iii) the deployment of applications in containers, which are more efficient than virtual machines (VMs), to hide the complexities of programming languages and dependencies. ATOMS can be deployed in cloud environments and mobile networks as well, where removing resource contention is more challenging due to higher network uncertainties and network bandwidth issues. We analyze these cases by experiments using simulated LTE network delays.

This paper makes three contributions: (i) a novel Plan-Schedule scheme that coordinates future offloaded tasks to remove resource contention, on top of load-aware scheduling; (ii) a framework that accurately estimates and controls the timing of offloading tasks through computing time prediction, network latency estimation and clock synchronization; and (iii) methods to predict processor usage and detect multi-tenant contention on distributed container-based servers.

The rest of this paper is organized as follows. We discuss the related work in Sect. 2, and describe the applications and the performance metrics in Sect. 3. In Sect. 4 we explain the offloading workflow and the plan-schedule algorithms. Then we detail the system implementations in Sect. 5. Experimental results are analyzed in Sect. 6. In Sect. 7 we summarize this paper.

## 2 Related Work

**Mobile Offloading:** Many previous works are on reducing E2E delays in mobile offloading frameworks [1,9,10]. Gabriel [1] deploys cognitive engines in a nearby cloudlet that is only one wireless hop away to minimize network delays. Timecard [9] controls the user-perceived delays by adapting server-side processing times, based on measured upstream delays and estimated downstream delays. Glimpse [10] hides network delays of continuous object detection tasks by tracking objects on the mobile side, based on stale results from the server. This paper studies the fundamental issue of resource contention on multi-tenant mobile offloading servers, however, not yet considered by the previous works.

**Cloud Schedulers:** Workload scheduling in cloud computing has already been intensely studied. These systems leverage rich information, for example, estimates and measurements on resource demands and running times, to reserve and allocate resources, and reorder tasks in queue [8,11,12]. Because data processing tasks have much larger time scales of makespan and deadline, usually ranging from minutes to hours, these methods are inadequate in handling real-time mobile offloading tasks that desires sub-second delays.

**Real-Time Schedulers:** Real-time (RT) schedulers in [13–15] are designed for low latency and periodical tasks on multi-processor systems. However, these schedulers do not work in the scenario of mobile offloading. First, the RT schedulers can not handle network delays and uncertainties. Second, the RT schedulers are designed to minimize deadline miss rates, whereas our goal is to minimize E2E delays. In addition, the RT schedulers use worst-case computing times in scheduling. It results in an undesired low utilization for applications with highly varying computing times. As a novel approach for the mobile scenarios, ATOMS makes dynamic predictions and coordinations for incoming offloaded tasks, using estimated task computing times and network delays.

# 3 Mobile Workloads

## 3.1 Applications

Table 1 describes the vision and cognitive applications used for testing our work. They all require low E2E delays: FaceDetect and ObjectDetect lose trackability as delay increases; FeatureMatch can be used in robotics and autonomous systems to retrieve depth information for which timely response is indispensable. In another aspect, the three applications present differences in parallelism and variability of computing time. We use the differences to explore the design of a general and highly usable offloading framework.

**Table 1.** Test applications

| Application | Functionalities | Time | Parallelism |
|---|---|---|---|
| Face detection | Haar feature cascade classifiers [16] in OpenCV [17] | Variable | Single-threaded |
| Feature matching | Detects interest points in left and right frames from a binocular camera, extracts and matches SURF [18] features | Variable | Feature extraction on two threads, then matching on one thread |
| Object detection | Localizes objects and labels each with a likeliness score using a DNN (YOLO [19]) | Constant | Uses all cores of a CPU in parallel |

## 3.2 Performance Metrics

We denote a mobile client as $C_i$ with an ID $i$. The offloading server is a distributed system composed of resource-rich machines. An offloading request sent by $C_i$ to the server is denoted as task $T_j^i$, where the task index $j$ is a monotonically increasing sequence number. We ignore the superscript for simplicity when discussing only one client. Figure 1 shows the life cycle of an offloaded task. $T_j$ is sent by a client at $t\_send_j$. It arrives at a server at $t\_server_j$. After queueing, the server starts to process it at $t\_start_j$ and finishes at $t\_end_j = t\_start_j + d\_compute_j$, where $d\_compute_j$ is the computing time.[1] The client receives the result back at $t\_recv_j$. $T_j$ uses $T\_paral_j$ cores in parallel.

We evaluate a task using two primary performance metrics of continuous mobile sensing applications [20]. **E2E delay** is calculated as $d\_delay_j = t\_recv_j - t\_send_j$. It comprises upstream network delay $d\_up_j$, queueing delay $d\_queue_j$, computing time $d\_compute_j$ and downstream delay $d\_down_j$. ATOMS reduces $d\_delay_j$ by minimizing $d\_queue_j$. **Offloading interval** represents

[1] A symbol starting with "$t\_$" is a timestamp and "$d\_$" is a duration of time.

the time span between successive offloaded tasks of a client, calculated as $d\_interval_j = t\_send_j - t\_send_{j-1}$. Clients offload tasks periodically and are free to adjust offloading periods. Applications can thus tune offloading period for energy consumption and performance trade-off. Ideally any interval is equal to $d\_period_i$, the current period of client $C_i$. In ATOMS, however, the interval becomes non-constant due to task coordination. We desire stable sensing and offloading activities, so smaller interval jitters are preferred, given by $d\_jitter_j^i = d\_interval_j^i - d\_period_i$.

## 4 Framework Design

As shown in Fig. 3, ATOMS is composed of one *master* server and multiple *worker* servers. The master communicates with clients and dispatches tasks to workers for execution. It is responsible for planning and scheduling tasks.

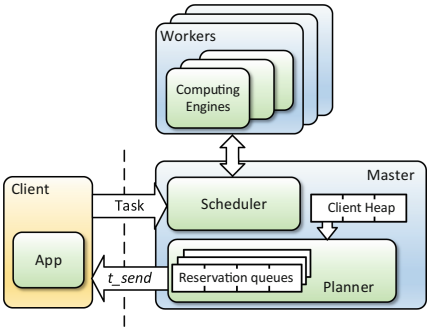



**Fig. 3.** The architecture of the ATOMS framework.

### 4.1 Worker and Computing Engine

We first describe how to deploy applications on workers. A worker machine hosts one or more *computing engines*. Each computing engine runs an offloading application encapsulated in a container. Our implementation adopts *Docker* containers.[2] We use Docker's resource APIs to set processor share, limit and affinity, as well as memory limit for each container. We focus on CPUs as the computing resource in this paper. The total number of CPU cores of worker $W_k$ is $W\_cpu_k$. The support for GPUs lies in our future work.

A worker can have multiple engines for the same application in order to fully exploit multi-core CPUs, or host different types of engines to share the machine by multiple applications. In this case, the total workloads of all engines on a worker may exceed the limit of processor resource ($W\_cpu$). Accordingly,

[2] Docker: https://www.docker.com/.

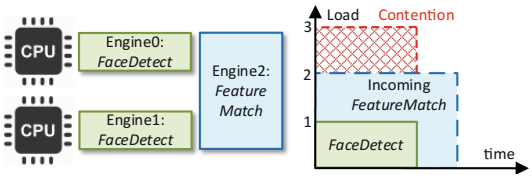



**Fig. 4.** A dual-core worker machine have two engines of FaceDetect ($T\_paral = 1$) and one engine of FeatureMatch ($T\_paral = 2$). The plot on right gives an example of processor contention.

we classify workers into two types: *reserved worker* and *shared worker*. On a **reserved worker**, the sum of processor usages of engines never exceed $W\_cpu$. Therefore whenever there is a free computing engine, it is guaranteed that dispatching a task to it does not induce any processor contention. Unlike a reserved worker, the total workloads on a **shared worker** may exceed $W\_cpu$. See Fig. 4, a dual-core machine hosts two FaceDetect engines ($T\_paral = 1$) and one FeatureMatch engine ($T\_paral = 2$). Both applications are able to fully utilize the dual-core processor. When there is a running FaceDetect task, an incoming FeatureMatch task will cause processor contention. *Load-aware scheduling* described in Sect. 4.4 is used for shared workers.

Workers measure the computing time $d\_compute$ of each task and returns it to the master along with the computation result. The measurements are used to predict $d\_compute$ for future tasks (Sect. 5.1). A straightforward method is measuring the start and end timestamps of a task, and calculating the difference ($d\_compute_{ts}$). However, it is vulnerable to processor sharing that happens on shared workers. We instead get $d\_compute$ by measuring *CPU time* ($d\_cputime$) consumed by the engine container during the computation.

### 4.2 Master and Offloading Workflow

In addition to the basic send-compute-receive offloading workflow, ATOMS has three more steps: *creating reservation*, *planning*, and *scheduling*.

**Reservation:** When the master starts the planning phase of task $T_j$, it creates a new reservation $R_j = (t\_r\_start_j,\ t\_r\_end_j,\ T\_paral_j)$, where $t\_r\_start_j$ and $t\_r\_end_j$ are the start and end times respectively, and $T\_paral_j$ is the demanded cores. As shown in Fig. 5, given the lower and upper bounds of upstream network delay ($d\_up_j^{low}$, $d\_up_j^{up}$) estimated by the master, as well as the predicted computing time ($d\_compute'_j$), the span of reservation is calculated as $t\_r\_start_j = t\_send_j + d\_up_j^{low}$ and $t\_r\_end_j = t\_send_j + d\_up_j^{up} + d\_compute'_j$. The time slot of $R_j$ contains the uncertainty of the time when $T_j$ arrives at the server ($t\_server_j$), and the time consumed by computation. Provided that the predictions on network delays and computing times are correct, the future task will be within the reserved time slot.

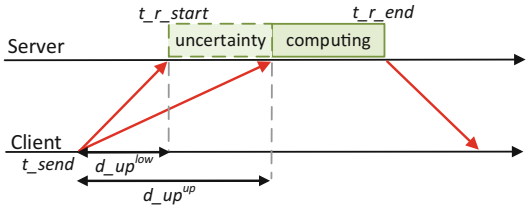



**Fig. 5.** Processor reservation for a future offloaded task includes the uncertainty of arriving time at the server and its computing time.

**Planning:** The planning phase runs before the real offloading. It coordinates future tasks of all clients to ensure that the total amount of all reservations never exceeds the limit of total processor resources of all workers. Client $C_i$ registers at the master to initialize the offloading process. The master assigns it a future timestamp $t\_send_0$ indicating when to send the first task. The master creates a reservation for task $T_j$ and plans it when $t_{now} = t\_r\_start_j - d\_future$ where $t_{now}$ is the master's current clock time, and $d\_future$ is a parameter for how far after $t_{now}$ that the planning phase covers. The planner predicts and coordinates future tasks that start before $t_{now} + d\_future$.

$T^i_{next}$ is the next task of client $C_i$ to plan. The master plans future tasks in ascending order of start time $t\_r\_start^i_{next}$. For a new task to be planned with the earliest $t\_r\_start^i_{next}$, the planner creates a new reservation $R^i_{next}$. The planner takes $R_{next}$ as input. It detects resource contention, and reduces that by adjusting the sending times of both the new task and a few planned tasks. We defer the details of planning to Sect. 4.3. $d\_inform_i$ is a parameter of $C_i$ for how early the master should inform the client about the adjusted task sending time. A reservation $R^i_j$ remains adjustable until $t_{now} = t\_send^i_j$ - $d\_inform_i$. The planner then removes $R_j$ and notifies the client. Upon receiving $t\_send_j$, the client sets a timer to offload $T_j$.

**Scheduling:** The client offloads $T_j$ to the master when the timer at $t\_send_j$ timeouts. After receiving the task, using the information of currently running tasks on each worker, the scheduler selects the worker that induces the least processor contention. The master dispatches it to the worker and gets back the result. We give the details in Sect. 4.4.

### 4.3 Planning Algorithms

The planning algorithm decides the adjustments to sending times of future tasks from each client. An optimal algorithm minimizes jitters of offloading intervals, while ensuring that the total processor usage is within the limit, and SLOs on offloading intervals are satisfied. Instead of solving this complex optimization problem numerically, we adopt a heuristic and feedback-control approach that adjusts future tasks in a fixed window from $t_{now} + d\_inform$ to $t_{now} + d\_future$. Our approach is able to improve the accuracy of computing time prediction by

using a small predicting window (see Sect. 5.1), and naturally handle changes of client number and periods.

The planner buffers reservations in *reservation queues*. A reservation queue stands for a portion of processor resource in the cluster. A queue $Q_k$ has a resource limit $Q\_cpu_k$ with cores as the unit, used for contention detection. The sum of $Q\_cpu$ of all queues is equal to the total cores in the cluster. Each computing engine is assigned to a reservation queue. The parallelism of a reservation $T\_paral$ is determined by the processor limit of computing engines. For example, $T\_paral$ of a fine-parallelized task is different for an engine on a dual-core worker ($T\_paral = 2$) and one on a quad-core worker ($T\_paral = 4$).

**Contention Detection:** When the planner receives a new reservation $R_{new}$, it first selects a queue to place it in. It iterates over all queues, for $Q_k$, calculates the needed amount of time ($\Delta$) to adjust $R_{new}$, and the total processor usage ($\Theta_k$) of $Q_k$ during the time slot of $R_{new}$. The planner selects the queue with the minimal $\Delta$. In doing so, it checks whether the total load on $Q_k$ after adding $R_{new}$ exceeds the limit $Q\_cpu$. If so, the algorithm calculates $\Delta$: the contention can be eliminated after postponing $R_{new}$ by $\Delta$. Otherwise $\Delta = 0$. We give an example in Fig. 6 that a new reservation $R_0^2$ is being inserted into a queue. The black line in the lower plot is the total load. Contention arises after adding $R_0^2$. It can be removed by postponing $R_0^2$ to the end time of $R_1^1$. $\Delta$ is thus obtained.

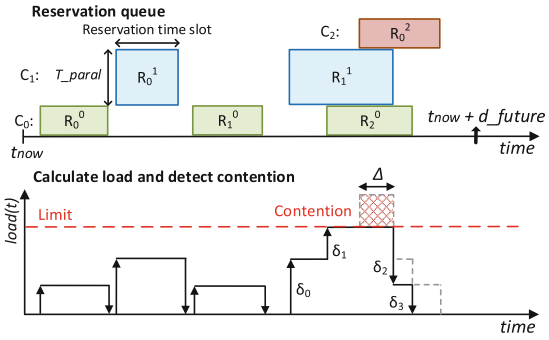



**Fig. 6.** An example of detecting processor contention and calculating required reservation adjustment. The top plot shows a reservation queue and the bottom plot shows the calculated total load *load(t)*.

If two or more planning queues have the same $\Delta$, *e.g.*, several queues are contention-free ($\Delta = 0$), the planner calculates the processor usage $\Theta$ during the time slot of $R_{new}$: $\Theta = \int_{t\_r\_start_{new}}^{t\_r\_end_{new}} load(t)dt$. We consider two strategies. The *Best-Fit* strategy selects $Q$ that has the highest $\Theta$, which packs reservations as tightly as possible and leaves the least margin of processor resources on the queue. The other strategy is *Worst-Fit* that, in contrast, selects the queue with the lowest $\Theta$. We study their difference through evaluations in Sect. 6.3.

**SLOs:** In the next *coordination* step, rather than simply postponing $R_{new}$ by $\Delta$, the planner moves ahead a few planned reservations as well, to reduce the duration to postpone $R_{new}$. The coordination process takes $\Delta$ as input and adjusts reservations according to *cost* ($R\_cost$), a metric on how far the measured offloading interval $d\_interval$ deviates from the client's SLOs (a list of desired percentiles of $d\_interval$). For example, a client with period 1 s may require a lower bound $d\_slo^{10\%} > 0.9$ s and an upper bound $d\_slo^{90\%} < 1.1$ s.

The master calculates $R\_cost$ when it plans a new task, using the measured percentiles of interval ($d\_interval^p$). For the new reservation ($R_{new}$) to be postponed, the cost $R\_cost^+$ is obtained from the upper bounds: $R\_cost^+ = \max(\max_{p \in \cup^+}(d\_interval^p - d\_slo^p),\ 0)$ where $p$ is a percentile and $\cup^+$ is the set of percentiles that have upper bounds in the SLOs. $R\_cost^+$ is the maximal interval deviation from the SLOs. For tasks to be moved ahead, deviation from lower bounds ($\cup^-$) are used instead to get the cost $R\_cost^-$. The cost is a weight between two clients to decide the adjustment on each. SLOs with tight bounds on $d\_interval$ make the client less affected during the coordination process.

### 4.4 Scheduling Algorithms

The ATOMS scheduler dispatches tasks arriving at the master to the most suitable worker machine that minimizes processor contention. The scheduler keeps a global FIFO task queue for buffer tasks when all computing engines are busy. For each shared worker, there is a local FIFO queue for each application that it serves. When a task arrives, the scheduler first searches for available computing engines on any reserved workers. It dispatches the task if one is found and the scheduling process ends. If there is no reserved worker, or no engine is free, the scheduler checks shared workers that are able to run the application. It selects the best worker based on processor contention $\Phi$ and usage $\Theta$. The task is then dispatched to a free engine on the selected shared worker. If no engine is free on the worker, the task is put into the worker's local task queue.

Here we detail the policy to select a shared worker. The scheduler uses estimated end time $t\_end'$ of all running tasks on each worker, obtained by predicted computing time $d\_compute'$. To schedule $T_{new}$, it calculates the load $load(t)$ on each worker using $t\_end'$, including $T_{new}$. The resource contention $\Phi_k$ on worker $W_k$ is calculated by $\Phi_k = \int_{t_{now}}^{t\_end'_{new}} \max(load(t) - W\_cpu_k,\ 0)dt$. The worker with the smallest $\Phi$ is selected to run $T_{new}$. For workers with identical $\Phi$, similar to the planning algorithm, we use processor usage $\Theta$ as the selection metric. We compare the two selection methods, Best-Fit and Worst-Fit, through evaluations in Sect. 6.

## 5 Implementation

In this section we present the implementation of computing time prediction, network delay estimation and clock synchronization.

### 5.1 Prediction on Computing Time

Accurate prediction of computing time is essential for resource reservation. Underestimation leads to failure in detecting resource contention, and overestimation causes larger interval jitters. We use *upper bound estimation* for applications with a low variability of computing times, and *time series prediction* for applications with a high variability. Given that $T_n$ is the last completed task of client $C_i$, instead of just predicting $T_{n+1}$ (the next task to run), ATOMS needs to predict $T_{next}$ (the next task to plan, $next > n$). $N_{predict} = next - n$ gives how many values it needs to predict since the last sample. It is decided by the parameter $d\_future$ (Sect. 4.2) and the period of the client, calculated as $\lceil d\_future/d\_period_i \rceil$.

**Upper Bound Estimation.** The first method estimates the upper bound of samples using a TCP retransmission timeout estimation algorithm [21]. We denote the value to predict as $y$. The estimator keeps a smoothed estimation $y^s \leftarrow (1-\alpha) \cdot y^s + \alpha \cdot y_i$.) and a variation $y^{var} \leftarrow (1-\beta) \cdot y^{var} + \beta \cdot |y^s - y_i|$. The upper bound $y^{up}$ is given by $y^{up} = y^s + \kappa \cdot y^{var}$, where $\alpha$, $\beta$ and $\kappa$ are parameters. This method outputs $y^{up}$ as the prediction of $d\_compute$ for $T_{next}$. This lightweight method is adequate for applications with low computing time variability, such as ObjectDetect. It tends to overestimate for applications with highly varying computing times because it uses upper bound as the prediction.

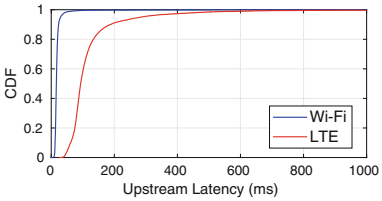
**Time Series Linear Regression.** In the autoregressive model for time series prediction problems, the value $y_n$ at index $n$ is assumed to be a weighted sum of previous samples in a moving window with size $k$. That is, $y_n = b + w_1 y_{n-1} + \cdots + w_k y_{n-k} + \epsilon_n$, where $y_{n-i}$ is the $i$th sample before the $n$th, $w_i$ is the corresponding coefficient and $\epsilon_n$ is the noise term. We use this model to predict $y_n$. The inputs ($y_{n-1}$ to $y_{n-k}$) are the previous $k$ samples measured by workers. We use a recursive approach to predict the $N_{predict}$th sample after $y_{n-1}$: to predict $y_{i+1}$, the predicted $y_i$ is used as the last sample. This approach is flexible to predict arbitrary future samples, however, as $N_{predict}$ increases, the accuracy degrades because the prediction error is accumulated. The predictor keeps a model for each client which is trained either online or offline.

### 5.2 Estimation on Upstream Latency

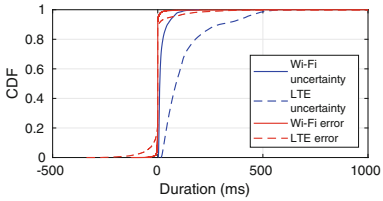
As discussed in Sect. 4.2, because network delays $d_{up}$ may have large fluctuations, we use the lower and upper bounds ($d\_up^{low}$, $d\_up^{up}$) instead of the exact value in the reservation. The TCP retransmission timeout estimator [21] described in Sect. 5.1 is used to estimate network delay bounds. We use subtraction instead of addition to obtain the lower bound. The estimator has a non-zero *error* when a new sample of $d_{up}$ falls out of the bounds, calculated as its deviation from the nearest bound. The error is positive if $d_{up}$ exceeds $d\_up^{up}$, and negative if it is smaller than $d\_up^{low}$. The estimation uncertainty is given by $d\_up^{up}$ - $d\_up^{low}$.

Because the uncertainty is included in task reservation, a larger uncertainty overclaims the reservation time slot, which causes higher interval jitters and lower processor utilizations.

We measure $d_{up}$ of offloading 640×480 frames with sizes from 21 KB to 64 KB, using Wi-Fi networks. To explore networks with higher delays and uncertainties, we obtain simulated delays of Verizon LTE networks using the Mahimahi tool [22]. The CDFs of network latencies are plotted in Fig. 7a. We demonstrate the estimator performance (error and uncertainty) in Fig. 7b. Results show that the estimation uncertainty for Wi-Fi networks is small, and it is very large for LTE (maximal value is 2.6 s). We demonstrate how errors and uncertainties influence offloading performance through experiments in Sect. 6.




(a) Upstream latency.




(b) Uncertainty and error.

**Fig. 7.** Upstream latency of Wi-Fi and LTE networks, with uncertainty and error of estimation. The maximal latency is 1.1 s for Wi-Fi and 3.2 s for LTE. The parameters of estimator are $\alpha = 0.125$, $\beta = 0.125$, $\kappa = 1$.

### 5.3 Clock Synchronization

We seek a general solution for clock synchronization without patching the OS of mobile clients. The ATOMS master is synchronized to the global time using NTP. Because time service now is ubiquitous on mobile devices, we require clients to be *coarsely* synchronized to the global time. We do *fine* clock synchronization as follows. Client sends out a NTP synchronization request to the master each time it receives an offloading result to avoid the wake-up delay [9]. To eliminate the influence of packet delay spikes, the client buffers $N_{ntp}$ responses and runs a modified NTP algorithm [23]. It applies clock filter, selection, clustering and combining algorithms to $N_{ntp}$ responses and outputs a robust estimate on clock offset. It also outputs the bounds of the offset. ATOMS uses the clock offset to synchronize timestamps between clients and the master, and uses the bounds in all timestamp calculations to consider the remaining clock uncertainties.

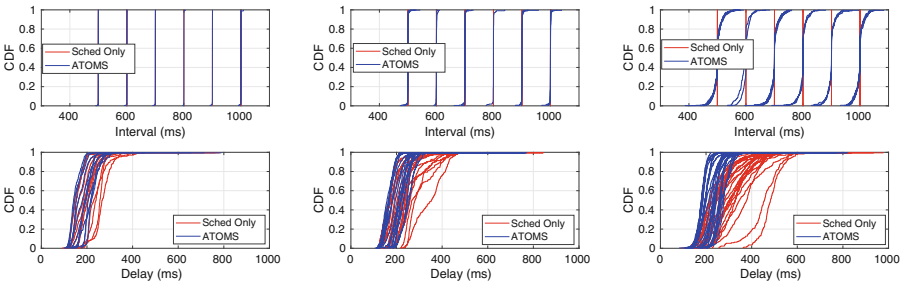
## 6 Evaluation

### 6.1 Experiment Setup

**Baselines:** We compare ATOMS with baseline schedulers to prove its effectiveness for improving offloading performance. The baseline schedulers use

load-aware scheduling (Sect. 4.4), but instead of using dynamic offloading time coordination in ATOMS, they use conventional task queueing and reordering approaches: (i) *Scheduling Only*: it minimizes the longest task queueing time; (ii) *Earliest-start-time-first*: it prioritizes the task with the smallest start time at client (*t_send*), which experiences the longest lag until now; (iii) *Longest-E2E-delay-first*: it prioritizes the task with the longest estimated E2E delay, including measured upstream and queueing delays, and the estimated computing time. Methods (ii) and (iii) are evaluated in the experiments using LTE networks (Sect. 6.2) where they perform differently from (i) due to larger upstream delays.

**Testbed:** We simulate a camera feed to conduct reproducible experiments. Each client selects which frame to offload from a video stream based on the current time and the frame rate. We use three public video datasets as the camera input: the Jiku datasets [24] for FaceDetect; the UrbanScan datasets [25] for FeatureMatch application, and multi-camera pedestrians videos [26] for ObjectDetect. We resize the frames to $640 \times 480$ in all the tests. Each test runs for 5 minutes. The evaluations are conducted on AWS EC2. The master runs on a c4.xlarge instance (4 vCPUs, 7.5 GB memory). Each worker machine is a c4.2xlarge instance (8 vCPUs, 15 GB memory). We emulate clients on c4.2xlarge instances. Pre-collected network upstream latencies (as described in Sect. 5.2) are replayed at each client to emulate the wireless networks. The prediction for FaceDetect and FeatureMatch uses offline linear regression, and the upper bound estimator is used for ObjectDetect. The network delay estimator setting is the same as in Fig. 7. We set $d\_inform$ (Sect. 4.2) to 300 ms for all clients.




(a) 12 clients. (b) 18 clients. (c) 24 clients.

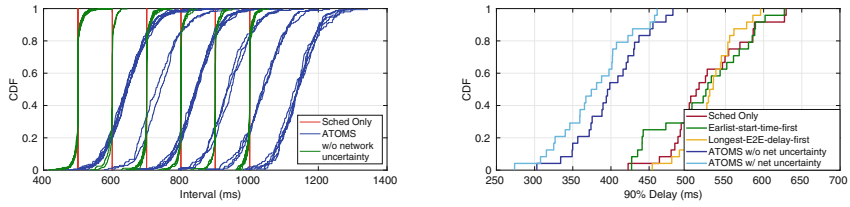
**Fig. 8.** Offloading performance (CDF of each client) of Scheduling Only and ATOMS running FaceDetect using Wi-Fi. The average CPU utilization is 37% in (a), 56% in (b) and 82% in (c).

### 6.2 Maintaining Low Delays Under High Utilization

We set 12 to 24 clients running FaceDetect with periods from 0.5 s to 1.0 s, using Wi-Fi networks. $d\_future = 2$ s is used in planning. We use one worker machine

(8 vCPUs) hosting 8 FaceDetect engines. The planner has a reservation queue for each engine with $Q\_cpu = 1$. See Fig. 8, with more clients, the interference becomes more intensive and the Sched Only scheme suffers from increasingly longer E2E delays. ATOMS is able to maintain low E2E delays even when the total CPU utilization is over 80%. Using the case with 24 clients as example, the 90% percentile of E2E delays is reduced by 34% in average for all clients, and the maximum reduction is 49%. The interval plots (top) show that offloading interval jitters increase in ATOMS, caused by task coordination.

**LTE Networks:** To investigate how ATOMS performs under larger network delays and variances, we run the test with 24 clients using LTE delay data. As discussed in Sect. 5.2, the reservations are longer in this case due to higher uncertainties of task arriving time. As a result, the total reservations may exceed the processor capability. See Fig. 9a, the planner has to postpone all reservations to allocate them, all clients hence have severely dragged intervals (blue lines). To serve more clients, we remove the uncertainty from task reservation (as in Fig. 5), and then the offloading intervals can be maintained (green lines in Fig. 9a). We show the CDFs of 90% percentiles E2E delays of 24 clients in Fig. 9b. Delays increase without including network uncertainties in reservations, but ATOMS still presents reasonable improvement: 90% percentile of delays is decreased by 24% in average and by 30% as the maximum among all clients. Figure 9b gives the performance of the reordering-based schedulers described in Sect. 6.1: Earliest-start-time-first scheduler and Longest-E2E-delay-first scheduler. The result shows that these schedulers perform similarly to Sched Only, and ATMOS achieves better performance.




(a) Offloading interval. (b) E2E delay.

**Fig. 9.** (a) Offloading interval running FaceDetect using LTE with 24 clients. The average CPU utilization is 83% for Sched Only, 59% for ATOMS, and 81% for ATOMS without network uncertainty. (b) CDFs (over all 24 clients) of 90% percentiles of E2E delay running FaceDetect using LTE networks. (Color figure online)

### 6.3 Shared Worker

Contention mitigation is more complex for shared workers. In the evaluations, we set up 4 ObjectDetect clients with periods 2 s and 3 s, and 16 FeatureMatch

clients with periods 2 s, 2.5 s, 3 s and 3.5 s. $d\_future = 6$ s is used in the planner. We use 4 shared workers (c4.2xlarge), and each hosts 4 FeatureMatch engines and 1 ObjectDetect engine.

**Planning Schemes:** We compare three schemes of planning: (i) a global reservation queue ($Q\_cpu = 32$) is used for 4 workers; (ii) 4 reservation queues ($Q\_cpu = 8$) are used and Best-Fit is used to select queue; (iii) 4 queues are used with Worst-Fit selection. Load-aware scheduling with Worst-Fit worker selection is used. The CDFs of interval (top) and E2E delay (bottom) of all clients are given in Fig. 10. It shows that Worst-Fit adjusts tasks more aggressively and causes the largest interval jitter. It allocates FeatureMatch tasks (low parallelism) more evenly to all reservation queues. Resource contention is more likely to take place when ObjectDetect (high parallelism) is planned, so more adjustments are made. The advantage of Worst-Fit is the improved delay performance. See the delay plots in Fig. 10, Worst-Fit evidently performs better for the 4 ObjectDetect clients: the worst E2E delay of the 4 clients is 786 ms for Worst-Fit, 1111 ms for Best-Fit and 1136 ms for Global. The delay performance of FeatureMatch is similar for the three schemes.

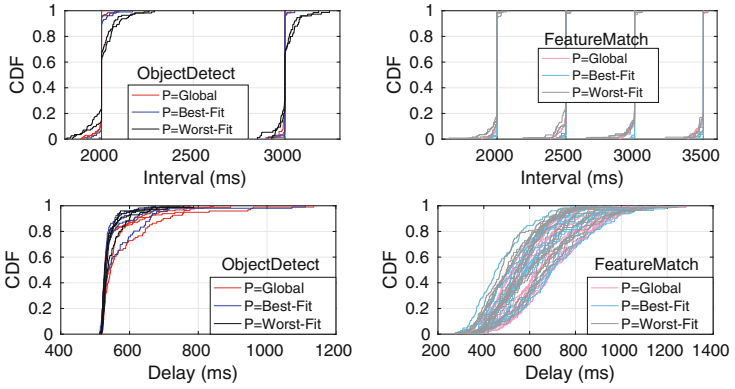



**Fig. 10.** Interval and E2E delay of 4 ObjectDetect and 16 FeatureMatch clients, using different planning schemes. The average CPU utilization is 36%.

**Scheduling Schemes:** Figure 11 shows the E2E delays using different scheduling schemes: (i) a simple scheduler that selects the first available engine; (ii) a load-aware scheduler with Best-Fit worker usage selection; (iii) a load-aware scheduler with Worst-Fit selection. The planner uses 4 reservation queues with Worst-Fit selection. For the simple scheduling, ObjectDetect tasks that can be parallelized on all 8 cores are more likely to be influenced by contention. FeatureMatch requires 2 cores at most and can get enough processors more easily. Best-Fit performs the best for ObjectDetect, whereas it degrades dramatically for FeatureMatch clients. The reason is that the scheduler tries to pack incoming

tasks as tightly as possible on workers. As a consequence, it leaves enough space to schedule highly parallel ObjectDetect tasks. However, due to the errors of computing time prediction and network estimation, there is a higher possibility of contention for the tightly placed FeatureMatch tasks. The Worst-Fit method has the best performance for FeatureMatch tasks and still maintains reasonably low delays for ObjectDetect. Therefore it is the most suitable approach in this case. Figure 12 compares the 90% E2E delay of all clients between Scheduling Only and ATOMS (Worst-Fit scheduling). In average, ATOMS reduces the 90% percentile E2E delay by 49% for the ObjectDetect clients, and by 20% for the FeatureMatch clients.

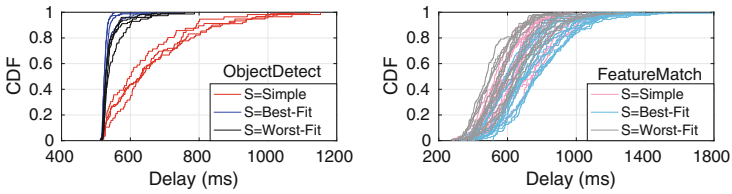



**Fig. 11.** E2E delay of ObjectDetect and FeatureMatch using different scheduling schemes. The average CPU utilization is 36% in all cases.

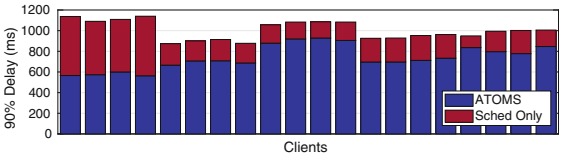



**Fig. 12.** 90% percentiles of E2E delay of ObjectDetect (bar 1 to 4) and FeatureMatch (bar 5 to 20) clients. The CPU utilization is 40% for Sched Only and 36% for ATOMS.

## 7 Conclusions

We present ATOMS, an offloading framework that ensures low E2E delays by reducing multi-tenant interference on servers. ATOMS predicts the time slots of future offloaded tasks, and coordinates them to mitigate processor contention on servers. It selects the best server machine to run each arriving task to minimize contention, based on real-time workloads on each machine. The realization of ATOMS is achieved by key system designs in computing time prediction, network latency estimation, distributed processor resource management and client-server clock synchronization. Our experiments and emulations prove the effectiveness of ATOMS in improving E2E delay for applications with various degrees of parallelism and computing time variability.

## References


1. Ha, K., et al.: Towards wearable cognitive assistance. In: MobiSys (2014)
2. Jain, P., et al.: OverLay: practical mobile augmented reality. In: MobiSys (2015)
3. Cuervo, E., et al.: MAUI: making smartphones last longer with code offload. In: MobiSys (2010)
4. Satyanarayanan, M., et al.: The case for VM-based cloudlets in mobile computing. IEEE Pervasive Comput. **8**(4), 14–23 (2009)
5. Han, S., et al.: MCDNN: an approximation-based execution framework for deep stream processing under resource constraints. In: MobiSys (2016)
6. Salmerón-Garcia, J., et al.: A tradeoff analysis of a cloud-based robot navigation assistant using stereo image processing. IEEE Trans. Autom. Sci. Eng. **12**(2), 444–454 (2015)
7. Meisner, D., et al.: PowerNap: eliminating server idle power. In: ASPLOS (2009)
8. Jyothi, S.A., et al.: Morpheus: towards automated SLOs for enterprise clusters. In: OSDI (2016)
9. Ravindranath, L., et al.: Timecard: controlling user-perceived delays in server-based mobile applications. In: SOSP (2013)
10. Chen, T.Y.H., et al.: Glimpse: continuous, real-time object recognition on mobile devices. In: SenSys (2015)
11. Tumanov, A., et al.: TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In: EuroSys (2016)
12. Rasley, J., et al.: Efficient queue management for cluster scheduling. In: EuroSys (2016)
13. Rajkumar, R., et al.: Resource kernels: a resource-centric approach to real-time systems. In: SPIE/ACM Conference on Multimedia Computing and Networking (1998)
14. Brandenburg, B.B., Anderson, J.H.: On the implementation of global real-time schedulers. In: RTSS (2009)
15. Saifullah, A., et al.: Multi-core real-time scheduling for generalized parallel task models. In: RTSS (2011)
16. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: CVPR (2001)
17. Bradski, G.: The OpenCV library. Dr. Dobb's J. Softw. Tools **120**, 122–125 (2000)
18. Bay, H., et al.: Speeded-up robust features (SURF). Comput. Vis. Image Underst. **110**(3), 346–359 (2008)
19. Redmon, J., et al.: You only look once: unified, real-time object detection. In: CVPR (2016)
20. Ju, Y., et al.: SymPhoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In: SenSys (2012)
21. Jacobson, V.: Congestion avoidance and control. In: SIGCOMM (1988)
22. Netravali, R., et al.: Mahimahi: a lightweight toolkit for reproducible web measurement. In: SIGCOMM (2014)
23. Mills, D., et al.: Network time protocol version 4: protocol and algorithms specification. RFC 5905 (Proposed Standard), June 2010
24. Saini, M., et al.: The Jiku mobile video dataset. In: MMSys (2013)
25. Raposo, C., Antunes, M., Barreto, J.P.: Piecewise-planar StereoScan: structure and motion from plane primitives. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) ECCV 2014. LNCS, vol. 8690, pp. 48–63. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10605-2_4
26. Fleuret, F., et al.: Multicamera people tracking with a probabilistic occupancy map. IEEE Trans. Pattern Anal. Mach. Intell. **30**(2), 267–282 (2008)