JAY: Adaptive Computation Offloading for Hybrid Cloud Environments

Joaquim Silva, Eduardo R. B. Marques, Luís M. B. Lopes, and Fernando Silva
CRACS/INESC-TEC & Faculty of Sciences, University of Porto, Portugal
Email: {joaquim.silva,ebmarques,lmlopes,fmsilva}@fc.up.pt

***Abstract*—Edge computing is a hot research topic given the ever-increasing requirements of mobile applications in terms of computation and communication and the emerging Internet-of-Things with billions of devices. While ubiquitous and with considerable computational resources, devices at the edge may not be able to handle processing tasks on their own and thus resort to offloading to cloudlets, when available, or traditional cloud infrastructures. In this paper, we present JAY, a modular and extensible platform for mobile devices, cloudlets, and clouds that can manage computational tasks spawned by devices and make informed decisions about offloading to neighboring devices, cloudlets, or traditional clouds. JAY is parametric on the scheduling strategy and metrics used to make offloading decisions, providing a useful tool to study the impact of distinct offloading strategies. We illustrate the use of JAY with an evaluation of several offloading strategies in distinct cloud configurations using a real-world machine learning application, firing tasks can be dynamically executed on or offloaded to Android devices, cloudlet servers, or Google Cloud servers. The results obtained show that edge-clouds form competent computing platforms on their own and that they can effectively be meshed with cloudlets and traditional clouds when more demanding processing tasks are considered. In particular, edge computing is competitive with infrastructure clouds in scenarios where data is generated at the edge, high bandwidth is required, and a pool of computationally competent devices or an edge-server is available. The results also highlight JAY's ability of exposing the performance compromises in applications when they are deployed over distinct hybrid cloud configurations using distinct offloading strategies.**

***Index Terms*—Computation Offloading, Edge Cloud, Edge Computing, Cloudlet, Cloud Computing, Machine Learning**

I. INTRODUCTION

Traditional mobile cloud computing [1] focuses on offloading computation and data generated by mobile device applications to centralized cloud datacenters. This decreases battery consumption in the devices, reduces their storage requirements and speeds-up computation, thanks to the high-performance, elastic, resource pool provided by the cloud infrastructure. The network latency between devices and a centralized cloud can be significant, especially if large amounts of data need to be transferred, offsetting the gains in computation speed-up. Under these circumstances, performing the computations closer to the data source is desirable. Moreover, reliance on a network connection to the cloud impairs applications that need to provide high availability and represents a cost to end users. From the service providers' perspective also, the cost of cloud resources is also a significant factor, and a limit on maximum resource provisioning (the "level of elasticity") is usually set to make this cost manageable and predictable. As a result, traditional mobile cloud computing alone is not a good solution for a variety of computationally intensive, high data locality, applications.

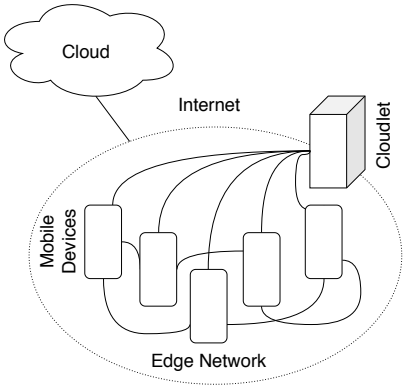


Fig. 1: A hybrid cloud environment.

To counter these limitations, edge computing tries to offload data and computation to the "edge" of the network, by pooling resources provided by mobile devices themselves and/or lightweight computing facilities near the edge, for instance through mobile edge-clouds [2] and cloudlets [3]. The processing capabilities of devices such as smartphones or tablets have been steadily increasing, making them capable of performing computationally intensive tasks with reasonable performance. In a mobile edge-cloud, nearby devices bound together by a local network or device-to-device communications form a pool of computing resources with access to local, crowd-sourced, data. Computational tasks are partitioned and schedule among the participating devices. Cloudlets add another layer of computational and storage resources closer to the edge to support local offloading of tasks and/or data from devices, reducing network latency significantly.

In this paper we consider the problem of adaptive computation offloading in mobile applications, where the decision of running a computational task locally on a mobile device or to offload it to a cloudlet or a traditional cloud infrastructure is evaluated at runtime. Thus, we consider application scenarios where 3 computational layers may coexist: mobile devices, possibly forming mobile edge-clouds; cloudlets, and; cloud servers (Fig. 1). In such hybrid environments, an adaptive offloading strategy may be informed by runtime conditions that inherently change over time - e.g., network bandwidth, computational loads across the hybrid cloud, battery status

978-1-7281-7216-3/20/$31.00 ©2020 IEEE

of the devices - that dynamically influence the performance metrics one wishes to optimize - e.g., task completion time, network traffic consumption, cloud processing cost, battery consumption.

We present JAY, a parametric framework for adaptive computation offloading that allows users to evaluate application performance according to several model parameters:

(a) hybrid cloud size and topology;
(b) data rate and size distribution;
(c) required result precision;
(c) offloading strategy, and;
(d) optimization metric.

As a proof-of-concept we use JAY with a case-study application that performs automated object detection in images sampled from videos captured and stored on devices. For object detection the application employs deep neural networks, deployed using TensorFlow [4]. We perform a benchmark evaluation for distinct configurations in terms of the task workload, hybrid cloud composition, and offloading strategies.

The results show that edge computing is competitive, with respect to infrastructure clouds, for scenarios where data is generated at the edge, high bandwidth is required, and a pool of computationally competent devices or an edge-server is available. From the framework's perspective, they also highlight JAY's capability of instrumenting apps and exposing the performance trade-offs and bottlenecks present when they are deployed over distinct hybrid cloud configurations and offloading strategies.

The remainder of this paper is structured as follows. Section II provides a description of the generic framework we employ for evaluating offloading strategies in hybrid clouds. A general rationale for the specification of the task offloading strategies, as well as concrete examples, are also presented here. We then describe our case-study application for task offloading in Section III, and present a benchmark evaluation we conducted for this application in Section IV. Related work is discussed in Section V. The paper ends with concluding remarks and a discussion of future work in Section VI.

II. THE JAY PLATFORM

JAY is a platform for the implementation and testing of computation offloading strategies in hybrid clouds. JAY is provided as services implemented in Kotlin for Android OS, or as plain Java Virtual Machines in other OSes (e.g., Linux or Windows). A hybrid cloud may be composed of mobile devices, plus servers running on cloudlets at the edge of the network or clouds accessible through the Internet. JAY instances in a hybrid cloud may host applications that generate tasks and/or serve as computational resources for offloading requests. Thus, the design makes no a priori assumptions where applications reside, even if we are are particularly interested in applications hosted on mobile devices. In any case, note that mobile devices can also serve offloading requests. Furthermore, JAY main focus is not on data security/privacy preservation, leaving this app-dependent.

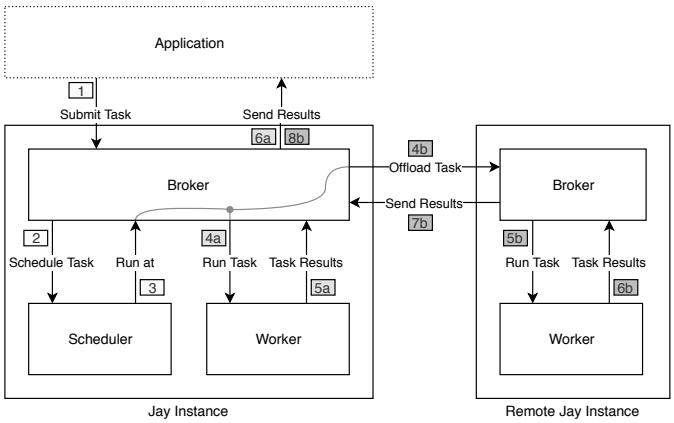


Fig. 2: JAY architecture and task life-cycle.

A. Architecture and task life-cycle

The architecture of JAY is illustrated in Fig. 2. As shown, each JAY instance runs a mandatory Broker service, plus optional Scheduler and Worker services. Each service is implemented using gRPC [1], an open-source framework by Google for remote procedure calls, that provides easy and secure data exchanges with SSL/TLS authentication. Their roles are as follows: a Broker is responsible for mediating interaction between instances, and delegating requests, responses, and state information to the Scheduler and Worker services, as required; a Worker runs tasks that are supplied to it, possibly originating in the local device or in a remote device through offloading; and, finally, a Scheduler decides where to run a given task, based on available information on JAY instances across the hybrid cloud that host a Worker service.

In Fig. 2 we also illustrate the life-cycle of tasks, with and without task offloading, and the underlying interplay between services. A task is spawned by an application, by submitting it to the local Broker (1). Next, the task is turned over to the local Scheduler (2) that decides which JAY instance in the hybrid cloud will execute the task. The Scheduler then provides the local Broker with the identity of the aforementioned JAY instance (3). If the chosen instance is the local one, the Worker executes the task (4a); in this case, when the task completes (5a), the application is notified of its results (6a). Alternatively, if the decision is to offload the task (4b), the task will run on the remote JAY instance (4b – 6b), after which its results will be transmitted to the originating instance (7b) and then, finally, also back to the application (8b).

B. Runtime adaptivity

Dynamic offloading strategies rely on information that brokers in JAY instances exchange between themselves, regarding the state of the instances. A scheduler, in turn, uses a digest of this information, provided by its broker, to guide its decisions.

Information is shared over time among brokers in the form of a state s_h per host h. The state s_h may encode distinct types of information regarding h such as: the current load of the worker running on h (if one exists), an estimate of

[1] https://grpc.io/

the communication bandwidth, and resource usage (e.g., CPU, memory, and battery). Each broker keeps a set S_H of states s_h for hosts h in H, where H is the set of hosts the broker has knowledge of; the hosts in H may include the local host h_{local} (if it hosts a worker), other hosts configured statically, typically the case of static cloudlet/cloud servers, and also hosts discovered dynamically in the case of Android devices in an edge network.

For a current state S_H and a given task T, a scheduler yields a decision $h^\star = \text{sch}(S_H, T) \in H$, indicating that T should run at host $h^\star$. JAY is parametric in the sch function. For instance, $\text{sch}(S_H, T) = h_{\text{local}}$ will mean that all tasks should run on the local host h_{local}, and no offloading takes place. In contrast, we may have $\text{sch}(S_H, T) = h_{\text{fixed}} \neq h_{\text{local}}$, meaning that every task should be offload to a fixed host h_{fixed}; this will correspond to an offloading strategy where all tasks are sent to a host that centralizes task execution, e.g., a traditional cloud server. More generally, a dynamic offloading strategy may be expressed at each host by:

$$\text{sch}(S_H, T) = \text{argmin}_{h \in H}\, f(T, S_H, h), \tag{1}$$

where f is some target function to be minimized.

Regarding the worker service, the current implementation uses a simple first-come/fist-served strategy (FCFS), i.e., it maintains a FIFO queue of pending tasks and runs one task at a time to completion. This choice let us focus on evaluating offloading strategies at the scheduler level in the current stage of JAY's design. Naturally, adaptivity could also, in principle, be applied to the way tasks are handled at the worker level, for instance the consideration of task priorities or deadlines in addition to arrival time, task preemption (provided tasks implement some sort of checkpointing), or task migration/re-offloading schemes between worker instances.

C. Example strategy for dynamic offloading

Using the current JAY version, as a proof-of-concept, we implemented a dynamic offloading strategy that seeks to minimize task execution time. The general formulation of the the target function f of Equation 1 becomes:

$$f(T, S_H, h) = c(s_h, T) + n(s_h, T) \tag{2}$$

where $s_h \in S_H$ is the known state of h. Based on s_h and T, we have estimates of computation and network overheads: $c(s_h, T)$, the estimated time for executing T on h, and; $n(s_h, T)$, the estimated time for transmitting task data from the local host to h and the task results back. When $h = h_{\text{local}}$, we fix $n(s_h, T) = 0$, i.e., to account for the case where no offloading takes place.

Let us now be more specific regarding the choice we made for c and n terms in Equation 2. Beginning with c, we must account for the FCFS policy in task execution at the worker in each host, hence we let q_h be the number of queued tasks in a worker running at h. Furthermore, under the assumption that tasks imply roughly the same raw computation effort, but allowing that variations occur in performance over time (e.g., dynamic frequency scaling in mobile devices for lower battery consumption, or due to possibly concurrent computations), we let $\overline{c}_h$ be a moving average of task computation times in h. To model n, we let $\overline{b}_h$ be a moving average of measured bandwidth performance for communications with h in a scale of bytes per second, and $d(T)$ be the amount of bytes involved in network communication with h for T. Putting all this together, letting $s_h = (q_h, \overline{c}_h, \overline{b}_h) \in S_H$ encode the known state of h and replacing the terms in Equation 2, we get:

$$f(T, S_H, h) = \overbrace{(q_h + 1) \times \overline{c}_h}^{c(s_h, T)} + \overbrace{d(T) \,/\, \overline{b}_h}^{n(s_h, T)} \tag{3}$$

Note that the $q_h + 1$ term accounts for the number of queued tasks at h (q_h) plus task T. In support of this offloading strategy, moving average settings can be calibrated in terms of the number of samples and timing requirements (cf. the sample instantiation given in Section IV).

III. CASE-STUDY APPLICATION

In recent years, we witnessed an explosion in Deep Learning (DL) research with real-world applications. DL introduces deep neural networks that can be employed in applications of diverse fields, for instance computer vision, speech recognition, text translation, or bioinformatics. DL has naturally been embraced in mobile software, e.g., for "photo beauty", face detection and augmented reality apps, among others [5, 6].

DL applications are interesting case-studies for computation offloading in hybrid clouds, since they can be computationally intensive and be handled differently in distinct execution environments. Moreover, in many cases the data (e.g., images or video) is generated at the edge and requires high bandwidth links to move around. The cloud makes it feasible to deploy heavyweight DL models that are memory and computation-intensive. A high-end cloud server may be equipped with vast amounts of RAM and many processors, and also in particular employ, for faster processing time, several graphical processing units (GPUs) or more specialized tensor processing units (TPUs) [7]. Mobile devices such as smartphones are of course resource-constrained in comparison, in spite of their ever-increasing capabilities in recent years, including the integration of specialized hardware [8]. As such, mobile devices run lightweight DL models, in particular converted models through quantization [9] that have lower (but still usable) precision for the machine learning task at stake. Popular DL libraries have "light variants", as in the case of TensorFlow (TF) [4]: the TensorFlow Lite (TFLite) library consists of a DL model converter, mapping TensorFlow models into TFLite ones through quantization and other optimizations, and a TFLite model engine.

Our case-study application is based on an Android TFLite demo for object detection[2], depicted in Fig. 3. The application identifies objects in frames extracted from real-time video captured by the device, along with bounding boxes and confidence scores per object. The application uses the MobileNet SSD

[2]https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/android

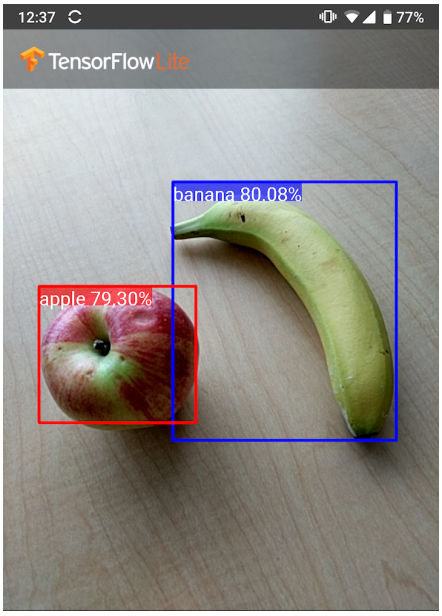


Fig. 3: Object detection Android app using TensorFlow Lite.

model [10] trained with COCO [11], a popular image dataset used for benchmarking object detection models that contains 2.5 million labeled instances for 91 object types in more than 300.000 images.

In our setup, the object detection task is incorporated into two distinct Kotlin modules, each linked with the JAY core library: one module is used in Linux servers (cloudlets and cloud) with the standard TF library, and the second is used in Android devices with TFLite. Each module implements the same DL task, i.e., given an image they run the DL model to yield object-detection outputs. The input images can have varying resolution, including High-Definition (HD) and Ultra HD (UHD), unlike the original demo that used only Standard Definition (SD) images. Both modules are parametrizable in terms of the TF (Linux version) or TFLite (Android) although here we use the same MobileNet SSD model variant for evaluation in both instances (see discussion in Section IV). Also, both also only use CPU resources, even if it is possible to parameterize TFLite and TF to use available GPUs/TPUs.

IV. EVALUATION

To evaluate JAY, we deployed the case-study application in different cloud configurations under varying workloads.

A. Hybrid cloud configurations

The cloud configurations for evaluation are depicted in Fig. 4. In all configurations, tasks are only fired by mobile devices, other (cloudlet and cloud) nodes only act as workers.

1) Device-only configurations: The top row of Fig. 4 shows two configurations that involve only mobile devices. In the first, a task always runs on the (local) device (LD) it originated from, hence no offloading takes place. In the second, a neighborhood of devices (ND), i.e., a mobile edge cloud, allows for a dynamic choice between local execution or offloading to a device in the neighborhood. The mobile devices we used were

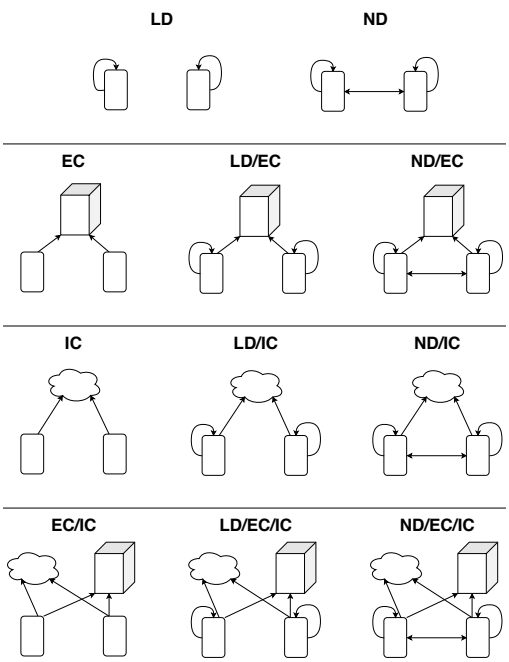


Fig. 4: Hybrid cloud configurations.

8 Google HTC Nexus 9 tablets running Android OS 7.0, each equipped with a dual-core 2.3 GHz CPU, 2 GB of RAM, and a Wi-Fi 802.11 card. To form the ND, the devices connect to a local and dedicated WiFi network established using an Asus RT-AC56U router.

2) Use of edge cloudlet and infrastructural cloud servers: The 2nd and 3rd rows of Fig. 4 show configurations that, in addition to mobile devices, deploy an edge cloudlet (EC), connected to the same WiFi network as the mobile devices, or an infrastructure cloud (IC), accessible through the Internet. Accounting for each type of server, we consider static offloading to the server (EC and IC), a dynamic choice between local device execution or server offloading (LD/EC and LD/IC), and an enhanced dynamic choice that also accounts for the ND (ND/EC and ND/IC). The EC server used was an Intel i7-6700K octa-core 4 GHz CPU with 16 GB of RAM, running Ubuntu Linux 19.04. The IC server was a Google Cloud VM hosted on Google's europe-west2-c data center in London, equipped with with 32 CPUs and 28.8 GB of RAM, accessible via Internet. Benchmarks involving the IC server ran during low network usage hours (only) in what concerns the local access point at Univ. Porto, 8pm to 8 am.

3) More heterogeneous configurations: Finally, in the last row of Fig. 4, we illustrate the most heterogeneous cloud configurations: dynamic choices between EC or IC offloading but no local device execution (EC/IC), EC/IC offloading but also local device execution (LD/EC/IC), and EC/IC but also ND (ND/EC/IC).

B. Workload parameters and setup

In addition to distinct cloud configurations, we consider workloads characterized by other JAY parameters, as summarized in Table I.

TABLE I: Workload generation parameters.

Parameter	Values
TF / TFLite model	MobileNet V1 fpn_coco (fixed)
Image resolution	SD, HD, UHD
Number of mobile devices (D)	4, 8
Task generator devices (G)	1, D
Tasks / minute (λ)	$12D$ ($G = D$); 96 ($G = 1$)

1) Object detection task: We consider the application described in Section III, configured with the MobileNet V1 fpn_coco model[3] both for TF (EC and IC servers) and TFLite (Android devices). The choice of a fixed DL model (task) in all cases, allows us to focus solely on performance comparisons.

2) Image resolution: Task inputs have 3 different image resolutions: SD, HD and UHD images. This affects computation and data transmission times. SD images are taken from the COCO test data set [11], and have an average size of 163 KB. HD and UHD images are taken from the UltraEye dataset [12], and have an average size of 717 KB and 2,210 KB, respectively. All images used were uploaded onto the mobile devices during the initialization stage by benchmark execution scripts.

3) Task generation: We have $D = 4$ or $D = 8$ mobile devices, in which either only one or all of them generates tasks, $G = 1$ or $G = D$ respectively. When $G = 1$, the remaining $D - 1$ devices merely act as workers. The TF/TFLite tasks are fired randomly with a Poisson distribution. When $G = D$ we consider scenarios in which each device independently generates tasks at rates λ equal to 12 per minute. When $G = 1$ we consider a rate λ of 96 per minute. The task generation rates were chosen to deliberately cause task processing congestion in a fair number of cloud configurations, as illustrated further on in this section.

C. Results

1) Baseline Results: In order to get a perspective of the baseline performance of different types of node in a hybrid cloud, we first measured the time it took for an individual task to finish when the task originating from one device was either: (LD) executed locally; (ND) offloaded to another device; (EC) offloaded to the edge cloudlet; and (IC) offloaded to the infrastructural cloud server. Measurements were taken for 3 batches of tasks fired in sequence (executed until completion before the next task) for 3 minutes per each type of image resolution (SD, HD, UHD). The average task execution time in seconds (with 95% Gaussian confidence intervals) are presented in Table II. Also presented (except for LD) are the times in seconds spent transmitting data over the network, the entries denoted with (n).

The first observation is that the networking overheads mostly explain the difference in completion time for the distinct image resolutions. The TF/TFLite computation time tends to be stable, regardless of image resolution, as the number of input neurons to the deep neural network is constant.

[3]https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

TABLE II: Baseline results (seconds).

	SD	HD	UHD
LD	8.70 ± 0.75	8.66 ± 0.66	8.86 ± 0.68
ND	8.94 ± 0.75	9.04 ± 0.82	9.29 ± 0.95
ND (n)	0.09 ± 0.06	0.26 ± 0.11	0.42 ± 0.26
EC	1.49 ± 0.15	1.61 ± 0.18	1.99 ± 0.22
EC (n)	0.06 ± 0.06	0.14 ± 0.08	0.32 ± 0.12
IC	1.07 ± 0.44	1.34 ± 0.23	2.43 ± 2.80
IC (n)	0.28 ± 0.21	0.50 ± 0.18	1.30 ± 2.81

(n) network transmission time.

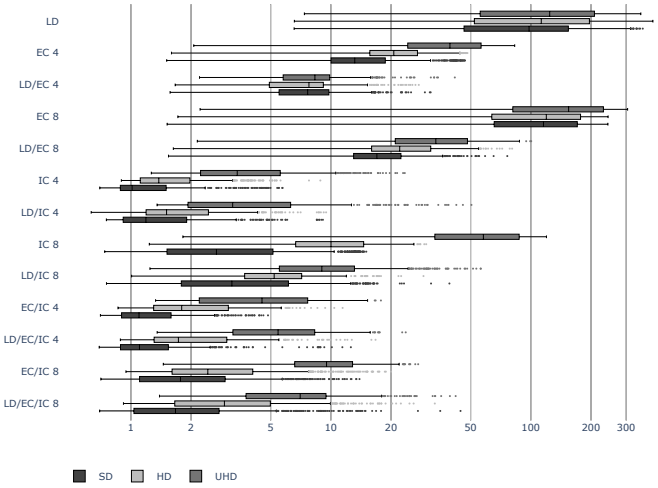
Looking at the EC and IC results, observe that EC and IC offloading significantly outperform execution on a device in all cases. Considering the execution times shown the average EC / IC speedups are, respectively: 5.8 / 8.1 for SD, 5.4 / 6.5 for HD, and 4.5 / 3.6 for UHD. Given the significant difference in EC and IC computational power compared to mobile devices, this would be expected. Note that EC is outperformed by IC in the SD and HD case, but not for UHD. IC networking overheads significantly increase for higher resolution images, and thus EC performance is overall better for UHD images. For all cases, IC networking overheads are anyway higher and more variable compared to EC, given that the EC server is accessible directly through the local WiFi network.

Finally, observe that the ND results are naturally worse than the LD ones, as offloading takes place between nodes with similar characteristics, but, similarly to EC, networking overheads are relatively small and have low variability.

2) Multi-device task generation: In Fig. 5 we depict the results obtained for scenarios with $D = 4$ and $D = 8$, in which all devices generate tasks ($G = D$) with a rate of $\lambda = 12$ tasks per minute during 3 minutes. After 3 minutes, new tasks were no longer fired, but pending ones were allowed to complete. The results are the aggregate of 3 runs, all using the same Poisson distribution seeds per device for firing tasks. The plot shows: (a) log-scale box-plots for execution time, and; (b) the distribution of tasks per type of node (LD, EC, IC) except in the cases where only one type of node executes all tasks.

We consider all cloud configurations (cf. Fig. 4) except those with ND offloading, which we discuss in a different scenario. For the current scenario, we did collect some preliminary results for ND configurations with results that were consistently worse with respect to LD configurations. The reason is that all devices generate tasks and, as such, they tend to become busy executing (mostly) their own local tasks rather than helping with tasks offloaded from other devices. Except for the "static" configurations (standalone LD, EC, IC), we employed the offloading strategy given in Equation 3 (c.f. Section II), using the moving average of the 5 latest samples completion times, reported every 5 seconds by the Worker service at each node, and similarly the 5 latest bandwidth measurements either obtained passively through the Broker service with a regular "ping" message to every node with a payload of 32 KB, or by

(a) Execution time (seconds).



(b) Task distribution (%).

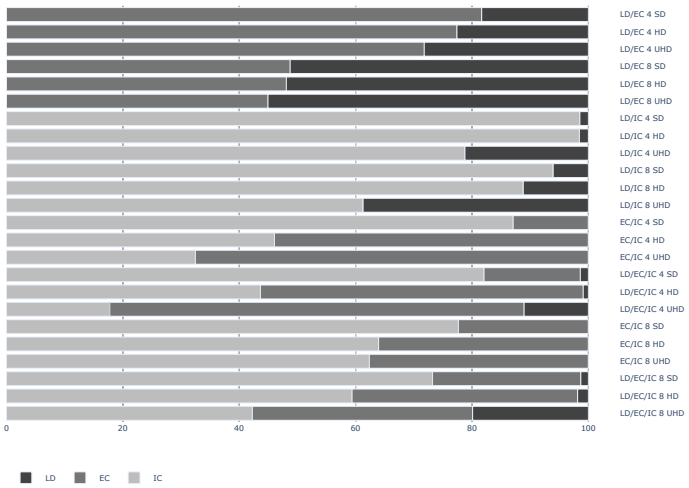
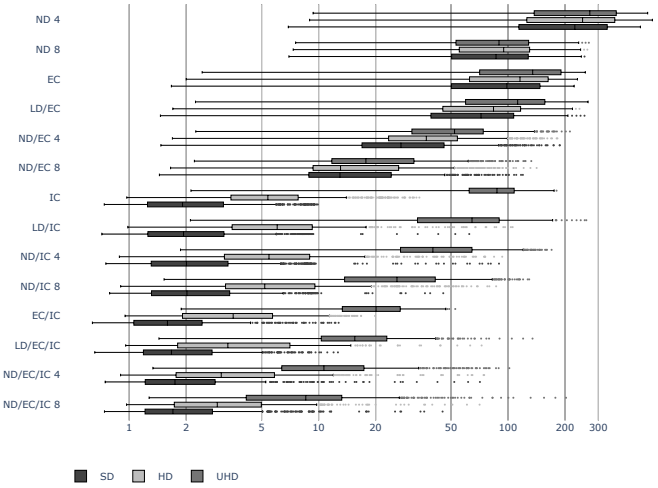


Fig. 5: Results for $G = D$, $D \in \{4, 8\}$, $\lambda = 12D$.

(a) Execution time (seconds).



(b) Task distribution (%).

Fig. 6: Results for $G = 1$, $D \in \{4, 8\}$, $\lambda = 96$.

actively measuring the response time when submitting a task.

The chosen task rate of $\lambda = 12$ tasks per minute and per device means that the average completion time for tasks should be ≤ 5s, otherwise congestion will likely result. This is observed in several cases. In the LD configuration, tasks are generated at a higher rate than they can be processed, hence they pile up in the queues of local devices that keep growing and execution times typically exceed 50s. For EC or IC, congestion becomes more pronounced when D grows from 4 to 8, and when image resolution increases from HD to UHD. Only in the case of IC with $D = 4$ we get reasonable performance for all image resolutions (most tasks take ≤ 5s).

LD/EC and LD/IC configurations have significantly better performance compare to plain EC and IC, respectively, except again in the IC case with $D = 4$. For instance, for UHD images and $D = 8$, the median execution time improves from 154.5s in EC to 33.5s in LD/EC, and from 57.9s in IC to 9s in LD/IC. With LD/EC and LD/IC, the offloading strategy is able to adjust dynamically to EC or IC congestion by running a fraction of tasks locally, especially in the case of UHD images. This can be seen in Fig. 5 (b) as in these configurations the share of tasks executed at the LD level increases with image resolution. In the EC case, this share ranges from 18% (SD) to 28% (UHD) for $D = 4$, and 51% (SD) to 55% (UHD) for $D = 8$. In the IC case, for $D = 4$ we have a significant share only for UHD images, 21%, and for $D = 8$ the share is marginal for SD, 11% for HD, and 39% for UHD.

The hybrid EC/IC and LD/EC/IC configurations are the ones that scale better with the increase in devices or image resolution, and also provide the best comparative results in almost all cases. The task distributions again show that higher image resolutions equate with similarly higher ratios of EC offloading, due to high network overheads at the IC level, and for UHD images also a relevant share of LD execution: 11% for $D = 4$ and 20% for $D = 8$. It is also worth noting that for $D = 4$, the share of EC offloading is significantly higher than for $D = 8$, and in fact exceeds the IC share for HD and UHD images. In the most extreme case the IC share is lower than 18% in the LD/EC/IC configuration for UHD images. Higher networking overheads at the IC level justify the preference for

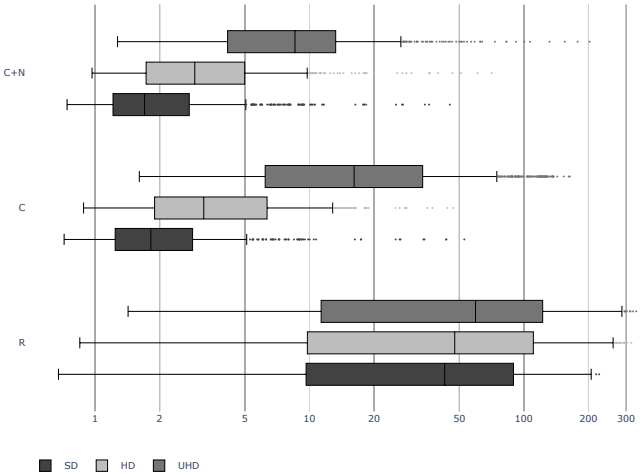


Fig. 7: Execution time (seconds) for different offloading strategies — ND/EC/IC, $G = 1$, $D = 8$, $\lambda = 96$.

EC. On the other hand, for $D = 8$, since the task workload is higher, we get a more significant share of IC offloads, ranging from 71% (SD) to 36% (UHD).

3) Single-device task generation: To evaluate the impact of ND configurations, we consider a scenario where only one device generates tasks ($G = 1$), while all others act only as workers. The results are shown in Fig. 6, with the same layout as for the multi-device scenario. We fixed a task generation rate of $\lambda = 96$ tasks per minute to match the combined rate of 8 devices in the previous scenario ($\lambda = 12 \times 8 = 96$). We also considered settings with $D = 4$ and $D = 8$, where $D - 1$ devices act as workers; in this case, more devices equate with more computational resources, whilst previously more devices equated with increased workload (as all devices generated tasks).

The results are overall similar to the previous multi-device scenario, except for the consideration of ND that offers additional insights. As shown, the ND standalone configurations offer poor performance, even if they predictably improve as the number of devices grow (from $D = 4$ to 8). However, ND-hybrid configurations clearly improve performance, particularly compared with LD-hybrid configurations, e.g., the median execution times for UHD images are 15.5s for LD/EC/IC, 10.7s for ND/EC/IC with $D = 4$, and 8.6s for ND/EC/IC with $D = 8$. The fraction of ND-offloaded tasks increases from $D = 4$ to $D = 8$ and image resolution, alleviating congestion at the EC, IC, or LD levels. For instance, in the case where images are UHD and $D = 8$, the ND share is 49% both for ND/EC and ND/IC, and 14% for NC/EC/IC.

4) Variation of offloading strategies: To explore the impact of distinct dynamic offloading strategies on performance, an additional experiment was conducted. We considered a base scenario with a single task generator and 7 workers ($D = 8$ and $G = 1$, as discussed above), using the ND/EC/IC configuration (the most heterogeneous of all considered). We compare three offloading strategies: C+N) the dynamic offloading strategy of previous scenarios, accounting for both computation and networking overhead information available at runtime (as specified by Equation 3 in Section II); C) that just accounts for computation overhead (a variation of Equation 3, but omitting the n term for networking overheads); and R) a offloading strategy that randomly assigns tasks to available nodes (LD, devices in the ND, EC, IC with a uniform distribution).

The execution times are depicted in Fig. 7. The comparison between the C+N and C results demonstrate that not accounting for network overheads (in the C strategy), specially in the UHD case when these overheads are more significant (e.g., 8.6s for C+N vs. 16.1s for C in terms of the median execution time), clearly leads to degraded performance. The results for R are very poor (the median execution time is > 40s), illustrating the extreme case where any sort of runtime adaptivity is absent.

V. Related Work

JAY follows up on previous research work in the Hyrax project[4], where we mostly dealt with data offloading applications and related challenges in edge clouds [13–15], rather than computation offloading with the exception of P3-Mobile [16], a system for computation offloading with an underlying parallel programming model.

Overall, JAY provides a framework for the evaluation of computation offloading in distinct and possibly quite heterogeneous cloud environments, including the most common research approaches for which we now provide a brief survey.

Offloading computations from mobile devices to the cloud, in line with with the Mobile Cloud Computing (MCC) paradigm [1] usually has the purpose of speeding up computation and lowering battery consumption in mobile devices, for instance as in CloneCloud [17], Cuckoo [18], or MAUI [19].

While MCC is a good fit for many scenarios, high network latencies, sub-optimal use of data locality, and the need for a permanent connection to the Internet are significant factors for many (hyperlocal) application scenarios. The increasing computing power and networking capabilities of mobile devices led to the consideration of Mobile Edge Clouds (MECs) which are formed by a neighborhood of devices [2, 20]. MECs may take advantage of data locality, as data is usually produced at the edge, and of low network latencies afforded by local WiFi networks or D2D variants like WiFi-Direct that do not require any infrastructural support. Offloading in MECs is considered under several forms in systems such as FemtoClouds [21], mClouds [22], MobileStorm [23], or P3-Mobile [16].

Cloudlets [3] are less-powerful servers that are placed at the edge that overcome the network latency associated with sending large amounts of data to cloud infrastructures, while providing computational "muscle" to edge-clouds when the hardware resources of the devices are not sufficient. Systems like CloudAP [24] or MAPCloud [25] demonstrate that two-tier cloudlet/cloud offloading can improve efficiency, and mCloud [26] also considers mobile devices as possible offloading targets. Hybrid architectures are also considered in simulation frameworks like EdgeCloudSim [27] or MobEmu [28].

[4]https://hyrax.dcc.fc.up.pt

VI. Conclusion

In this paper we described JAY, a parametric framework to explore task offloading strategies in hybrid cloud configurations and their impact on performance. As a proof-of-concept, we made use of a case-study application that concerned the use of a TF/TFLite model for object detection in images. The case-study allowed us to expose performance compromises and bottlenecks for different choices of cloud configurations, workloads, and offloading strategies.

The key highlight is that the use of computational resources at distinct networking tiers (mobile edge cloud, cloudlets, infrastructure clouds), partly or fully combined, can improve the performance of an application through task offloading, by alleviating or eliminating the bottlenecks that result from the congestion of computation and/or networking resources. In particular, the results show that edge resources, in the form of mobile edge clouds or cloudlets, are a very relevant complement or even alternative to infrastructure clouds, for scenarios where data is generated at the edge and high bandwidth is required. Moreover, through the adaptive offloading strategy we implemented in JAY and used in our case-study, we demonstrated that significant performance improvements can be achieved in terms of task completion times over static cloudlet/cloud offloading or strictly local execution.

In the future, we are interested in pursuing further work on JAY and related case-study applications by:

- evaluating different offloading strategies for other QoS metrics beyond execution time like energy efficiency, the costs of using an infrastructure cloud or mobile device network traffic, strategies that seek to balance several QoS metrics simultaneously, or fault-tolerance/resilience;
- considering more complex cloud configurations, for instance those that provide access to GPU or TPU resources in the case of DL applications, or those that have more heterogenous distribution of resources in the network, as it happens where more several cloud servers are available in different data centres or edge servers associate to different local networks, and, finally, those with more dynamic mobile edge clouds where device churn can be frequent, mobility may impair network availability, and applications may involve interaction with IoT devices;
- implementing richer task models in JAY to contemplate aspects such as QoS properties (e.g., deadlines or, for DL tasks, the precision of results), task relations (e.g., precedences, data dependencies), checkpointing to allow for preemption or migration, or task features that lead to high irregularity in computation or networking overheads;
- and, in relation to the former points, conducting real-world experiments involving users that exercise the system instead of controlled benchmark settings, e.g., following up on our previous experiment concerning video dissemination at sport venues [13], where we merely studied data offloading but not computation offloading.

Acknowledgements. This work was partially funded by project SafeCities (POCI-01-0247-FEDER-041435) from Fundo Europeu de Desenvolvimento Regional (FEDER), through COMPETE 2020 and Portugal 2020.

References

[1] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile Cloud Computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84–106, 2013.

[2] U. Drolia, R. Martins *et al.*, "The case for mobile edge-clouds," in *Proc. UIC/ATC*. IEEE, 2013, pp. 209–215.

[3] M. Satyanarayanan, Z. Chen *et al.*, "Cloudlets: at the leading edge of mobile-cloud convergence," in *Proc. MobiCASE*. IEEE, 2014, pp. 1–9.

[4] M. Abadi, P. Barham *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI*. Usenix, 2016, pp. 265–283.

[5] K. Ota, M. S. Dao *et al.*, "Deep learning for mobile multimedia: A survey," *ACM Trans. on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 13, no. 3s, p. 34, 2017.

[6] M. Xu, J. Liu *et al.*, "A first look at deep learning apps on smartphones," in *Proc. WWW*. ACM, 2019, pp. 2125–2136.

[7] N. P. Jouppi, C. Young *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ISCA*. IEEE, 2017, pp. 1–12.

[8] A. Ignatov, R. Timofte *et al.*, "AI benchmark: Running Deep Neural Networks on Android Smartphones," in *Proc. ECCV*. Springer, 2018, pp. 288–314.

[9] I. Hubara, M. Courbariaux *et al.*, "Quantized Neural Networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[10] A. G. Howard, M. Zhu *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv*, no. 1704.04861, 2017.

[11] T.-Y. Lin, M. Maire *et al.*, "Microsoft COCO: Common objects in Context," in *Proc. ECCV*. Springer, 2014, pp. 740–755.

[12] H. Nemoto, P. Hanhart *et al.*, "Ultra-Eye: UHD and HD images eye tracking dataset," in *Proc. QoMEX*. IEEE, 2014, pp. 39–40.

[13] J. Rodrigues, E. R. B. Marques *et al.*, "Video dissemination in untethered edge-clouds: a case study," in *Proc. DAIS*. Springer, 2018, pp. 137–152.

[14] P. M. P. Silva, J. Rodrigues *et al.*, "Using Edge-Clouds to Reduce Load on Traditional WiFi Infrastructure and Improve Quality of Experience," in *Proc. ICFEC*. IEEE, 2017, pp. 61–67.

[15] R. Martins, M. E. Correia *et al.*, "Iris: Secure reliable live-streaming with opportunistic mobile edge cloud offloading," *Future Generation Computer Systems*, vol. 101, pp. 272–292, 2019.

[16] J. Silva, D. Silva *et al.*, "P3-Mobile: Parallel computing for mobile edge-clouds," in *Proc. CrossCloud*. ACM, 2017, pp. 5:1–5:7.

[17] B.-G. Chun, S. Ihm *et al.*, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. EuroSys*. ACM, 2011, pp. 301–314.

[18] R. Kemp, N. Palmer *et al.*, "Cuckoo: a computation offloading framework for smartphones," in *Proc. MobiCASE*. Springer, 2010, pp. 59–79.

[19] E. Cuervo, A. Balasubramanian *et al.*, "MAUI: Making smartphones last longer with code offload," in *Proc. MobiSys*. ACM, 2010, pp. 49–62.

[20] J. Rodrigues, E. R. B. Marques *et al.*, "Towards a middleware for mobile edge-cloud applications," in *Proc. MECC*. ACM, 2017, pp. 1:1–1:6.

[21] K. Habak, M. Ammar *et al.*, "Femto Clouds: Leveraging mobile devices to provide cloud service at the edge," in *Proc. CLOUD*. IEEE, 2015, pp. 9–16.

[22] E. Miluzzo, R. Cáceres, and Y.-F. Chen, "Vision: mClouds-computing on clouds of mobile devices," in *Proc. MCS*. ACM, 2012, pp. 9–14.

[23] Q. Ning, C. Chen *et al.*, "Mobile Storm: Distributed real-time stream processing for mobile clouds," in *Proc. CloudNet*. IEEE, 2015, pp. 139–145.

[24] Y. Zhang, R. Yang *et al.*, "CloudAP: Improving the QoS of mobile applications with efficient VM migration," in *Proc. HPCC/EUC*. IEEE, 2013, pp. 1374–1381.

[25] M. R. Rahimi, N. Venkatasubramanian *et al.*, "MAPCloud: Mobile applications on an elastic and scalable 2-tier cloud architecture," in *Proc. UCC*. IEEE, 2012, pp. 83–90.

[26] B. Zhou, A. V. Dastjerdi *et al.*, "mCloud: A context-aware offloading framework for heterogeneous mobile cloud," *IEEE Transactions on Services Computing*, vol. 10, no. 5, pp. 797–810, 2017.

[27] S. Suryavansh, C. Bothra *et al.*, "Tango of edge and cloud execution for reliability," in *Proc. MECC*. ACM, 2019, pp. 10–15.

[28] R. Ciobanu, C. Dobre *et al.*, "Data and task offloading in collaborative mobile fog-based networks," *IEEE Access*, vol. 7, pp. 104 405–104 422, 2019.