Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Energy-aware adaptive offloading of soft real-time jobs in mobile edge clouds

Joaquim Silva*, Eduardo R. B. Marques*, Luís M.B. Lopes* and Fernando Silva*

Abstract

We present a model for measuring the impact of offloading soft real-time jobs over multi-tier cloud infrastructures. The jobs originate in mobile devices and offloading strategies may choose to execute them locally, in neighbouring devices, in cloudlets or in infrastructure cloud servers. Within this specification, we put forward several such offloading strategies characterised by their differential use of the cloud tiers with the goal of optimizing execution time and/or energy consumption. We implement an instance of the model using JAY, a software framework for adaptive computation offloading in hybrid edge clouds. The framework is modular and allows the model and the offloading strategies to be seamlessly implemented while providing the tools to make informed runtime offloading decisions based on system feedback, namely through a built-in system profiler that gathers runtime information such as workload, energy consumption and available bandwidth for every participating device or server. The results show that offloading strategies sensitive to runtime conditions can effectively and dynamically adjust their offloading decisions to produce significant gains in terms of their target optimization functions, namely, execution time, energy consumption and fulfilment of job deadlines.

Keywords: Computation offloading, Energy efficiency, Mobile edge clouds

Introduction

The last decade witnessed an impressive evolution in the storage and processing capabilities of mobile devices. Besides traditional processing cores, these microprocessors feature multiple GPU cores and also so called neural cores optimized for machine learning applications such as deep-learning and have reached performance levels comparable to laptop and some desktop analogs [1].

Despite these advancements, some computational jobs are too demanding for mobile devices. Mobile cloud computing [2] has traditionally tackled this problem by offloading computation and data generated by mobile device applications to cloud infrastructures. This move spares the battery in the devices and, in principle, speeds-up computation as the high-availability, elastic, cloud infra-structures can adapt to the computing and storage demands of the jobs spawned by the devices.

This offloading is, however, not without problems. Many mobile applications involve the processing of locally produced data (e.g., video) and uploading such large volumes of data to cloud infra-structures is time consuming and may not even be feasible from a QoS point of view due to the high communication latencies involved. Also, from an energy point of view, offloading jobs and/or data to cloud infrastructures is globally highly inefficient.

Mobile edge clouds [3] and cloudlets [4], on the other hand, try to harness the resources of local networks of devices and/or small servers, using device-to-device communication technologies such as Wifi and Wifi-Direct, at the edge to perform demanding computational jobs taking advantage of data locality to minimize latency and global energy consumption. In this approach, a given job is offloaded to one mobile device or a cloudlet in the network vicinity of the originating mobile device.

*Correspondence: joaquim.silva@fc.up.pt; ebmarques@fc.up.pt; lmlopes@fc.up.pt; fmsilva@fc.up.pt
CRACS/INESC TEC & Department of Computer Science, Faculty of Sciences, University of Porto, Porto, Portugal

The two approaches can be unified in a single, multi-tier architecture (Fig. 1), with: (a) local networks of devices (Tier 1), with less capable processing but fast and privileged access to raw data; (b) cloudlets directly accessible from the devices, with more processing muscle and storage (Tier 2), and; (c) traditional cloud infrastructures, featuring the highest performance and storage resources (Tier 3).

Given this architecture and a mobile application that spawns computational jobs, we consider the problem of offloading these jobs over the tiers in such a way to optimize runtime metrics such as: total execution time, global energy consumption, fulfillment of QoS requirements. In general, the decision to offload (or not) a job is supported by knowledge of observables as reported from the participating devices and servers or inferred from data exchanges, namely: available network bandwidth, computational load at each device and server, the battery status of the devices.

We previously introduced JAY [5] as tool to instantiate and experiment with different such cloud configurations, offloading strategies and mobile applications. In that paper we evaluated only latency-aware offloading strategies in several cloud configurations, from mobile edge clouds formed by Android devices up to 3-tier hybrid clouds, i.e., including also cloudlets and infrastructure cloud server instances.

In this paper we put forward a unifying model for this architecture upon which we can precisely specify the infra-structure parameters (e.g., cloud tiers and topology), the application parameters (the rate and size distribution of jobs, offloading strategy, job deadlines), the observables (as described above) and the runtime metric function to optimize. We then use JAY with the same object detection application as in [5] but with different model instances that include new QoS restrictions (jobs have deadlines) and different optimization functions such as total execution time, per device energy consumption and, total energy consumption. The cloud configurations we experiment with in this paper do not include tier-3 centralised cloud servers (e.g., Google Cloud, Amazon Web Services, or Microsoft Azure), as we would not be able to directly measure vital runtime observables such as energy consumption or at least infer them with enough confidence from the underlying virtualisation infrastructure.

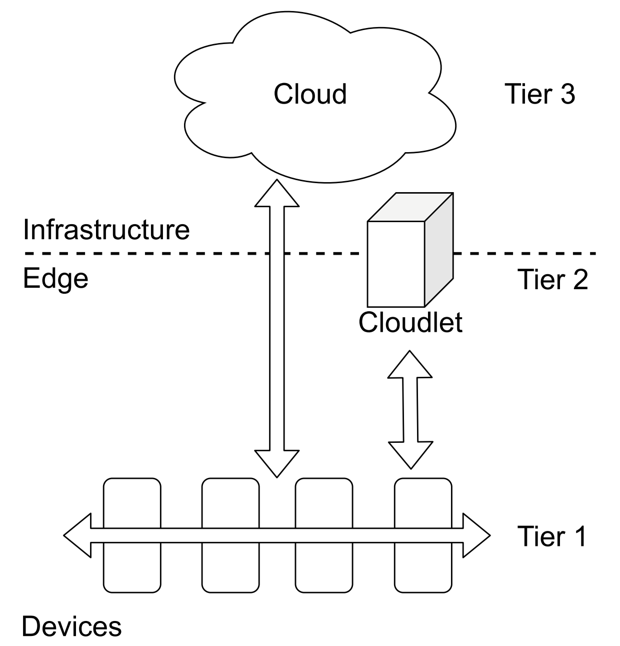


Fig. 1 A hybrid cloud environment

Thus, the main contributions of this paper are the following:

1. a model that specifies computational scenarios over hybrid edge/cloudlet/cloud topologies;
2. a complete JAY instance of the model that enables the execution of mobile applications over such network topologies through the definition of offloading strategies and optimization functions coupled with observables gathered at runtime, and;
3. a case-study with an object detection application that generates jobs with deadlines while trying to optimize execution time, energy consumption or both.

JAY and the model implementation presented here are available at Github[1].

The remainder of this paper is structured as follows. Related work is discussed in "Related work" section. "System model" section provides a description of the model we use to describe the aforementioned hybrid architecture and the computations therein. "The JAY framework" section describes the JAY framework. "Experimental setup" section presents the scenarios we model in this paper and the experimental setup. "Evaluation" section presents the results from the experiments and discusses their implications. Finally, "Conclusion" section ends the paper with concluding remarks and a discussion of future work.

Related work

The general problem of computation offloading in mobile edge clouds received considerable attention in the last two decades, as documented in recent surveys [6–8]. Our discussion of related work focuses on software systems that, like JAY, conduct adaptive offloading, and in particular those that implement energy-aware offloading policies.

[1] https://github.com/jqmmes/Jay

A number of systems focuses on semi-automated offloading to an edge cloud or centralised cloud infrastructure, without collaborative offloading between mobile devices, in line with the mobile cloud computing paradigm [2]. In some systems of this kind, e.g., Cuckoo [9] or COSMOS [10], offloading policies merely seek to minimize latency without any energy awareness, and gains in energy consumption at the mobile device level are at most a by-product of offloading computation. A number of other systems support energy-aware offloading strategies supported by runtime profiling, like AIOLOS [11], MAUI [12], Phone2Cloud [13], ThinkAir [14], or ULOOF [15].

In the system model of all the former systems, energy consumption accounts only for the (local) mobile device that hosts applications, typically the energy consumed in network transmission during offloading, rather than also the upper processing tiers in network and computation terms like JAY, which may for instance also be battery-constrained (e.g., [16, 17]) and in any case may have restrictions regarding energy consumption (e.g., monetary costs).

In any case, the offloading policies of the systems still reflect a concern for energy consumption, possibly in conjunction with latency: AIOLOS allows one of two configurable policies that either optimize for latency or energy; MAIUI minimizes energy consumption subject to a latency threshold constraint; Phone2Cloud offloads jobs whenever the estimated latency for local execution exceeds a configurable threshold, or when it perceive lower energy consumption by the mobile device is attainable; ThinkAir implements offloading policies that can seek to minimize only one of latency or energy consumption, both latency and energy consumption (offloading must pay off in both dimensions compared to local execution), and also optionally constrained by monetary posts due to the use of cloud services and; finally, ULOOF evaluates local and remote execution cost functions that are parametrised by a weight factor that can be used to attain a balance between latency and energy consumption.

Other types of systems enable collaborative offloading among mobile devices forming an edge cloud, and, in some cases, also upper cloud tiers. There are systems of this kind which merely strive to optimize latency like FemtoClouds [18], Honeybee [19], Oregano [20], and P3-Mobile [21], while others are explicitly energy-aware in diverse manners, discussed next.

CWC [22] is a system for volunteer computing, where jobs are disseminated to a pool of mobile devices. To prevent battery consumption and intrusive user experience, jobs execute only when the devices are charging their batteries and have light computational loads, and may also be paused to minimize battery charging times.

mClouds [23] works over hybrid edge clouds that, like JAY, may be composed of mobile devices in a ad-hoc wireless network, cloudlet and public clouds, offloading jobs to each of the tiers according to connectivity conditions, and also (when multiple choices are available) by a cost model with weights that balances execution time and energy consumption in a configurable manner.

MDC [24] is a system for collaborative offloading among mobile devices that seeks to maximize the battery lifetime of the set of involved devices by balancing energy consumption among them; this concern could provide an interesting refinement to the HYBRID strategy in this paper, e.g., by factoring in battery levels of devices in addition to their battery efficiency.

RAMOS [25] offloads jobs over an edge cloud formed by heterogeneous mobile and IoT devices that act as job workers, a concept borrowed from FemtoClouds [18]. As in JAY, the RAMOS scheduler can be parametrised to minimize job latency or energy consumption, jobs have deadlines and are also executed in FIFO order by workers. RAMOS' architecture is centralised, however. Jobs originate and are scheduled in batches exclusively by a centralised controller node in contrast to JAY's distributed architecture.

Synergy [26] considers collaborative offloading between devices in a peer-to-peer ad-hoc network, and, in order to maximise the devices' battery lifetime, balances latency and energy consumption by partitioning jobs among devices while at the same time scaling the devices CPU frequencies.

Summarising the above discussion, Table 1 provides a comparative overview of JAY and the other systems mentioned. The table does so first in terms of cloud architecture, making a distinction between: mobile cloud computing (MCC) systems, where devices offload jobs to a centralised cloud infrastructure; mobile edge computing (MEC) systems, where there is collaborative offloading among mobile devices; and Femtocloud systems, where a set of mobile devices is used as a worker pool for jobs fired by an external host. The table next indicates awareness to runtime information regarding time and/or energy, and the support for job deadlines in the system model. The two remaining columns characterize the scheduler component responsible for offloading decisions concerning its location and to the granularity of those offloading decisions in terms of how many jobs are accounted for at once. A scheduler may either operate locally per device or run on a central peer, and the granularity type distinguishes between single-job, multiple-job, and parallel job offloading by the scheduler. The latter is a special class of multiple-job offloading in which all jobs are bound by some type of parallel computation that is inherent to the job model.

Table 1 Comparison between offloading systems

System	Cloud architecture	Time-aware	Energy-aware	Deadlines	Scheduler	Granularity
Jay	Configurable	✓	✓	✓	Configurable	Single-job
Cuckoo [9]	MCC	✗	✗	✗	Local	Single-job
COSMOS [10]	MCC	✗	✗	✗	Local	Single-job
AIOLOS [11]	MCC	✓	✓	✗	Local	Single-job
MAUI [12]	MCC	✓	✓	✗	Local	Single-job
Phone2Cloud [13]	MCC	✓	✓	✗	Local	Single-job
ThinkAir [14]	MCC	✓	✓	✗	Local	Single-job
ULOOF [15]	MCC	✓	✓	✗	Local	Single-job
Femtoclouds [18]	Femtocloud	✓	✗	✗	Centralized	Multiple-job
Honeybee [19]	MEC	✓	✗	✓	Local	Single-job
Oregano [20]	MEC	✗	✗	✗	Centralized	Multiple-job
P3-Mobile [21]	MEC	✗	✗	✗	Centralized	Parallel jobs
CWC [22]	Femtocloud	✓	✓	✗	Centralized	Parallel jobs
MDC [24]	MEC	✓	✓	✗	Centralized	Parallel jobs
RAMOS [25]	Femtocloud	✓	✓	✓	Centralized	Multiple-job
Synergy [26]	MEC	✓	✓	✓	Local	Single-job

The main distinctive trait of Jay is that it is configurable in terms of target cloud architecture and scheduler operation. Thanks to a simple and flexible design, each of the peers in a Jay system instance may act as a scheduler, a worker, or both, as illustrated by the variety of evaluation scenarios we put forward later in the paper, meaning that one can use Jay for offloading using an MCC, MEC, or Femtocloud architecture, with per-device schedulers or a centralised one. The offloading strategies we instantiate and evaluate in Jay are partially illustrative of comparable time and/or energy-aware approaches found in other systems. However, Jay is not bound to any particular approach since offloading strategies are configurable, and there is a general design for monitoring runtime state information. Time-awareness is also reflected in Jay by the support of job deadlines, a feature supported by only a few other systems. Finally, Jay only supports single-job scheduling granularity, a characteristic that is more in line with on-the-fly offloading of independent jobs, as seen in most systems discussed. In contrast, multiple-job granularity is usually associated with the use of a centralised scheduler, a Femtocloud architecture, or a computation model that embodies parallelism.

System model

Overview

We now put forward the system model for our adaptive offloading framework.

The overall rationale is as follows. We consider a set of hosts connected over a network, such that each host may generate and/or execute soft real-time jobs over time. A host may then execute a mixture of local jobs and offloaded jobs on behalf of other hosts, but it can also be that a host only generates or executes jobs. In this setting, offloading decisions can be informed and adaptive to runtime conditions. Information broadcast amongst hosts regarding variables such as network bandwidth and latency, host job load and available energy provide the required feedback. We consider each job has a soft real-time nature, meaning that it has an associated relative deadline expressing the maximum tolerable completion time for good QoS, and also that it requires communication among hosts in the case of offloading to supply job inputs (before the job's computation can proceed) and obtain job outputs (when the computation is done).

In what follows, we first lay out the base model concerning job characteristics and the time and energy costs for offloading, and then present sample offloading strategies over that model.

Base definitions

We associate each job j with a release time $r(j)$, a termination time $t(j) > r(j)$, a relative deadline $d(j)$, an originating host h_{L} from a set of hosts $\mathcal{H}$, and, finally, a computation host h_{C} also in $\mathcal{H}$. When j is clear in context, these properties are simply denoted respectively as r, t, d, h_{L}, and h_{C}. We say that the deadline of the job is fulfilled if $t \leq r + d$. Furthermore, we say the job executes locally at h_{L} when $h_{\mathrm{L}} = h_{\mathrm{C}}$, and that it is offloaded from h_{L} to h_{C} when $h_{\mathrm{L}} \neq h_{\mathrm{C}}$.

In the scenario of runtime adaptive offloading, for job j and at time r, an offloading decision is made at time r to

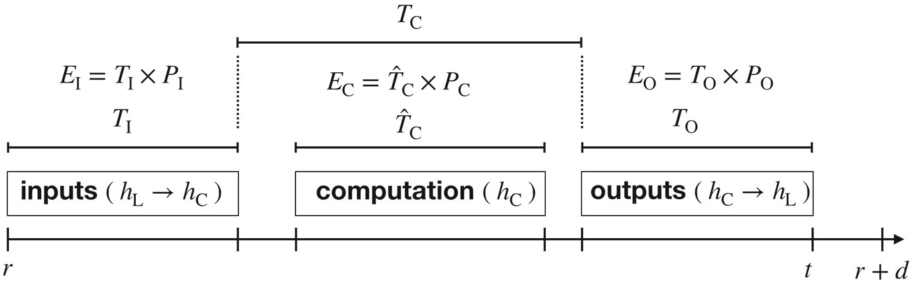


Fig. 2 Illustration of system model

determine h_C. We assume that decision to be computed locally (at h_L) and to have negligible overhead. In the case of offloading ($h_L \neq h_C$), we assume that network communication needs to take place between h_L and h_C for the inputs of j (data but possibly also code) to be available at h_C before j starts, and, later, once the computation j terminates, for the outputs to be transmitted back from h_C to h_L. This is illustrated in Fig. 2, along with the formulation for time and energy overheads during offloading.

We consider the offloading decision to be informed by estimates of completion time and energy consumption as follows. Per each host $h \in \mathcal{H}$ (including h_L) we model the estimated completion time and energy consumption of a given job j, $T(h)$ and $E(h)$, respectively as:

$$T(h) = T_I(h) + T_C(h) + T_O(h)$$

$$E(h) = E_I(h) + E_C(h) + E_O(h)$$

where time (T) and energy (E) are factored into a sum of three terms: T_I, E_I: the (time and energy) costs of input offloading job j; T_C, E_C: the (time and energy) costs of the actual computation for job j, and; T_O, E_O: the (time and energy) costs of downloading outputs of job j. Note that, given that there is no need for network communication when $h = h_L$, we should necessarily have $T_I(h_L) = E_I(h_L) = T_O(h_L) = E_O(h_L) = 0$.

Regarding energy, our aim is not merely to account for the energy consumption at the originating host (h_L) of a job, but also in the computation host in the case of offloading (h_C). This means first that network I/O expressed by the E_I and E_O should account for the energy consumption both in h_L and h_C: sending inputs from h_L to h_C requires energy to be consumed by h_L in uploading the inputs, and h_C to download, and vice-versa in the case of outputs. Since E_I and E_O are respectively dependent on the transmission times T_I and T_O and the power consumption when doing so, we model E_I and E_O as follows:

$$E_I(h) = T_I(h) \times P_I(h)$$

$$E_O(h) = T_O(h) \times P_O(h)$$

where P_I and P_O are estimates for the power consumed per time unit at both h_L and h_C[2], when, respectively, sending job inputs from h_L to h_C and receiving job outputs at h_L from h_C.

Moreover, the E_C term reflects the cost of executing the job remotely at h_C, but, as illustrated in Fig. 2, it should only account for an estimate of the energy consumption *while* h_C is effectively performing the computation for job j for an amount of time $\hat{T}_C$, rather than the energy consumption over the total period T_C, during which the job may at times be pending (e.g., waiting for other jobs in h_C to complete). Thus, we write:

$$E_C(h) = \hat{T}_C(h) \times P_C(h)$$

where P_C is an estimate for the power consumed per time unit at h due to the execution of the actual computation of j at h.

Offloading strategies

We can now express offloading strategies that may take into consideration multiple metrics to decide where a computation will take place, e.g., completion time and energy consumption. We define several such strategies for which we present a thorough evaluation later in the paper. Figure 3 illustrates the rationale in the offloading strategies. The example at stake concerns an offloading decision for a job originating at host $h_L = h_0$ in an environment with four other hosts, h_1 to h_4, such that each host may have different values regarding the estimates for time and energy consumption (T and E).

The simplest case is that of no offloading, which we designate as the LOCAL strategy (i.e., as illustrated in Fig. 3, jobs always execute locally).:

$$\text{LOCAL} \quad \equiv \quad h_C = h_L$$

The choice may also be fixed to a special host $h_S \neq h_L$ (assuming h_S does not generate jobs), for instance a cloud server that is responsible for executing all jobs (in Fig. 3, h_S is host h_1) , designated as the SERVER strategy:

[2] in these formulae, when clear from the context, the index of h is omitted for the sake of simplicity

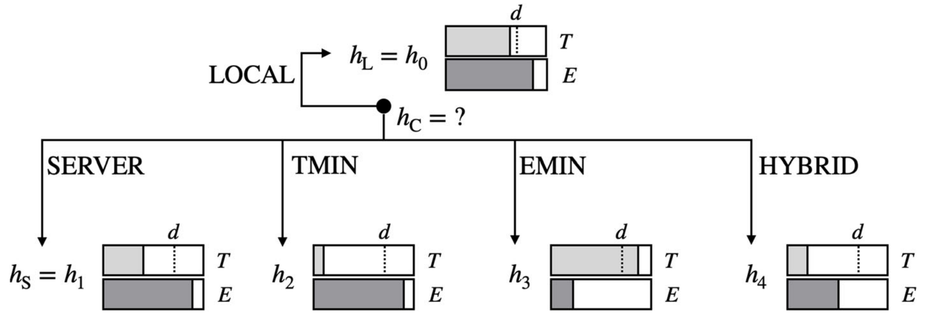


Fig. 3 Illustration of offloading strategies

$$\text{SERVER} \quad \equiv \quad h_C = h_S$$

In the case illustrated in Fig. 3, the LOCAL and SERVER strategies would not lead to optimal choices with respect to time and/or energy, as there are hosts that can execute the job faster than h_L and h_S (h_2 and h_4), or that may consume less energy than h_L and h_S to do so (h_3 and h_4). Adaptiveness comes into play if we account for the T and/or E estimates.

A strategy that seeks to minimize the completion time of job, but ignoring energy consumption, can be defined as:

$$\text{TMIN} \quad \equiv \quad h_C = \text{argmin}_{h \in \mathcal{H}} \, T(h)$$

Hence, in Fig. 3 we have $h_C = h_2$ for TMIN. In analogous manner, a strategy that seeks to minimize energy consumption can be defined as:

$$\text{EMIN} \quad \equiv \quad h_C = \text{argmin}_{h \in \mathcal{H}} \, E(h)$$

but it will not however attend to QoS requirements in terms of deadline fulfilment, i.e., h_C may be chosen regardless of whether $T(h_E) \leq d$ or not. This is illustrated in Fig. 3, where $h_C = h_3$ for TMIN but $T(h_3) > d$. Additionally, the most energy-efficient hosts will tend to be preferred. These hosts may possibly become congested with too many jobs whose execution can therefore be much delayed in time. The above strategy can be refined meaningfully to counter for these problems as:

$$\text{HYBRID} \quad \equiv \quad h_C = \text{argmin}_{h \in \mathcal{H} \,:\, T(h) \leq d} \, E(h)$$

balancing both time and energy costs and the fulfilment of d, as it expresses that h_C is chosen as the host which consumes less energy amongst those that can satisfy the job deadline ($h \in \mathcal{H} : T(h) \leq d$). This is illustrated in Fig. 3, where $h_C = h_4$ is the host with lower E value, among those with a T value lower than d (all except h_3). In the case where HYBRID yields no result, i.e., no host is estimated to be able to satisfy the job deadline, the offloading decision may for instance fallback to TMIN trying to complete the job as fast as possible anyway or to simply cancel the job altogether.

The TMIN and HYBRID strategies may lead to an imbalance between host loads, in the sense that most time-efficient and/or energy-efficient hosts will tend to have higher loads. This may be counter-productive if the hosts are stake are battery-constrained and we wish for instance to extend battery lifetime of all hosts as fairly as possible. To spread the load more evenly, a balanced selection scheme of hosts amongst those that can comply with a job deadline can be defined. For instance, a balanced selection policy can be defined as:

$$\text{BALANCED} \quad \equiv \quad h_C = \text{random}\{h \in \mathcal{H} : T(h) \leq d\}$$

i.e., h_C is randomly selected among the hosts that can comply with the deadline. In this case, the offloading choice may not be energy-optimal or time-optimal, but the random choice will tend to promote a more balanced distribution of jobs. We could also refine BALANCED to be explicitly energy-aware by refining the definition with constraints for energy consumption or battery level thresholds in addition to the job's deadline. Also, in alternative, a round-robin job distribution could be considered instead to enforce stricter load balancing.

Finally, we define a strategy that implements a form of "restricted offloading", a trait found in various systems discussed in "Related work" section, such that jobs are only offloaded if local execution is deemed unsuitable. That is, a job is only offloaded if the local host h_L is judged to be incapable of fulfilling the job's QoS, like deadlines in our case but also possibly other factors, e.g., those associated to network transmission in terms of energy, amount of data, or financial costs. In line with this rationale, we formulate the "local-first" strategy LF[f], where f is the policy to apply in the case of offloading, as:

$$\text{LF}[f] \quad \equiv \quad h_C = \begin{cases} h_L, & \text{if } T(h_L) \leq d \\ f, & \text{otherwise} \end{cases}$$

i.e., a job executes locally if the completion time estimate complies with deadline, otherwise f is evaluated to decide where the job should run, e.g., we can define the LF[TMIN], LF[HYBRID], or LF[BALANCED] strategies.

The JAY framework

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 jobs and/or serve as computational resources for offloading requests. Thus, the design makes no a priori assumptions where applications reside, even if we are particularly interested in applications hosted on mobile devices. In any case, note that mobile devices can also serve offloading requests. Furthermore, JAY's focus is not on data security/privacy preservation, leaving this app-dependent.

Architecture

The architecture of JAY is illustrated in Fig. 4. A single JAY instance runs on a network host, comprising 4 services: Broker, Scheduler, Worker and System Profiler.

The Broker service mediates network interaction with external peers, wrapping up any necessary requests and data interchange with the other internal JAY services on the same local instance. Local applications interact with the broker for job execution, and broker-to-broker interaction occurs between JAY instances for job offloading and dissemination of state information.

The internal state of each JAY instance over time is maintained by the System Profiler service, reflecting for instance energy consumption, current job workload, and network transmissions. All instances disseminate their state periodically, hence the profiler is also aware of the (latest known) state of remote JAY instances. The system profiler (on each instance) is then able to construct a global snapshot of all JAY instances in the network at any given time. The goal is to use this dynamic global snapshot to guide the offloading decisions while adapting to evolving runtime conditions.

Jobs are dealt with by the Scheduler and Worker services. The scheduler is responsible for choosing the host where to run a job submitted to the local instance by an application. In particular, it implements the offloading strategy. The scheduler's choice for assigning a job is taken from the set of all hosts having an active worker service, and can be based on state information as reported by the system profiler. Note that this set of active workers may include the local instance if it has an active worker service. Also, the local worker state is also observed by the profiler and included in the construction of the global state snapshot. The worker is in turn responsible for the actual execution of jobs, regardless of whether they are local or incoming from other hosts through offloading requests. JAY instances running only one of the scheduler or worker

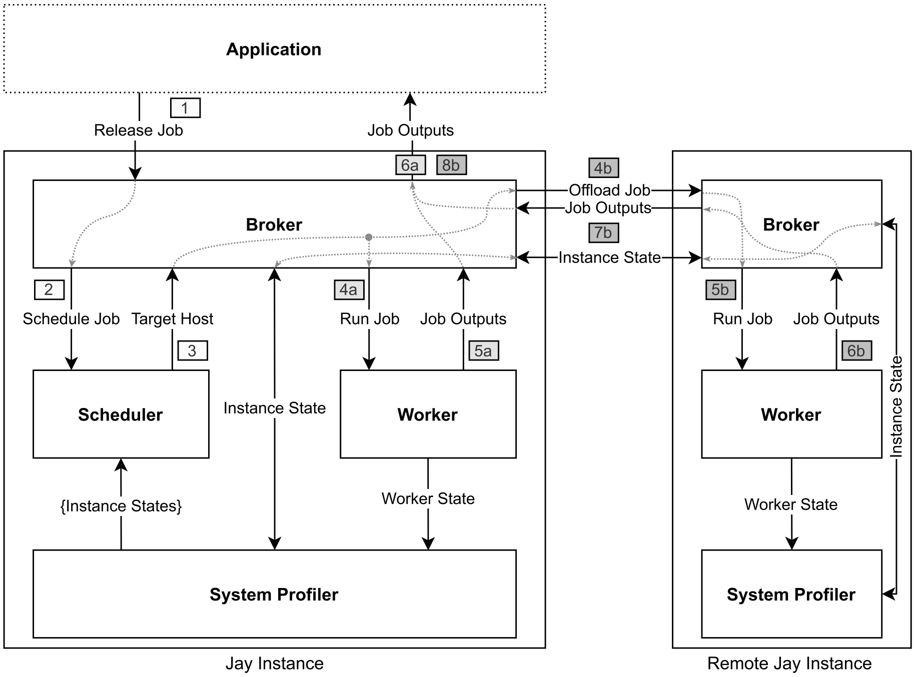


Fig. 4 Life cycle of a job in JAY

services merely act as a job execution clients or servers, respectively. On the other hand, instances may employ different implementations for the scheduler and/or worker.

Job lifecycle

In line with the interplay between Jay services just described, we can trace the lifecycle of a job in terms of the stages indicated in Fig. 4, as follows:

Job release: an application first releases the job by placing an execution request to the broker service (step 1). For simplicity, we will only consider the case where the application resides on the same host as the broker, even if Jay's architecture does not place such a constraint and other setups may be interesting from an application standpoint, e.g., to accommodate for jobs fired by IoT devices.

Offloading decision: the job execution request is passed by the broker over to the local scheduler (2) to determine the host that should execute the job. The scheduler's decision (3) may be that either the job executes locally (the local host was chosen) or needs to be offloaded to the target host.

Job execution: for local execution, the broker passes the job for execution to the local worker (4a), and when the job completes (5a) the job outputs are delivered to the application (6a). In the offloading case, the job is sent to the target host (4b) for execution (5b) and will, at some point, produce the job outputs (6b) that are then returned back to the originating host (7b) and, finally, delivered back to the application (8b).

System instantiation

We now present a sample system instantiation of Jay, later evaluated in the paper. It is composed of: a worker based on a FIFO job queue; a configurable scheduler that may implement any of the offloading strategies discussed in our system model, and; a system profiler that estimates time and energy consumption due to computation and network transmission at the local instance and aggregates it with the state information disseminated by remote instances. The result is a global snapshot of the state of the system that can be used to make adaptive offloading decisions. A summary of the model instantiation and associated notation is given in Table 2.

Profiler overview

The profiler is responsible for the state estimation driving adaptive offloading, with the functionality illustrated in Fig. 5.

The first aim of the profiler is to estimate and disseminate the state of the local instance (h_{L}), and aggregate similar state reported by the remote instances ($h \neq h_{\mathrm{L}}$), as shown lower right in the figure. The second aim is to use the state information for all available hosts (h_{L} and other hosts $h \neq h_{\mathrm{L}}$) to compute estimates for all hosts for the time (T_{I}, T_{C}, and T_{O}) and energy (E_{I}, E_{C}, and E_{O}) to run a job, and feed that information to the local scheduler, as illustrated in the lower left portion of the figure.

Table 2 Summary of model instantiation

State variables per host *h*		**Unit**
n	number of jobs at h	-
$\hat{T}_{\mathrm{C}}$	computational time of a job	s
B_{U}	bandwidth for uploading data (to h)	B / s
B_{D}	bandwidth for downloading data (from h)	B / s
P_{C}	power consumption during job computation	W
P_{D}	power consumption during data download	W
P_{U}	power consumption during data upload	W
Time estimates per host h		Unit
$T_{\mathrm{C}}(h)$	$\hat{T}_{\mathrm{C}}(h) \times (n(h) + 1)$	s
$T_{\mathrm{I}}(h)$	$B_{\mathrm{U}}(h) \times \|j\|_{\mathrm{I}}$	s
$T_{\mathrm{O}}(h)$	$B_{\mathrm{D}}(h) \times \|j\|_{\mathrm{O}}$	s
$T(h)$	$T_{\mathrm{I}}(h) + T_{\mathrm{C}}(h) + T_{\mathrm{O}}(h)$	s
Energy estimates per host h		Unit
$E_{\mathrm{C}}(h)$	$\hat{T}_{\mathrm{C}}(h) \times P_{\mathrm{C}}(h)$	W · s
$E_{\mathrm{I}}(h)$	$T_{\mathrm{I}}(h) \times (P_{\mathrm{U}}(h_{\mathrm{L}}) + P_{\mathrm{D}}(h))$	W · s
$E_{\mathrm{O}}(h)$	$T_{\mathrm{O}}(h) \times (P_{\mathrm{D}}(h_{\mathrm{L}}) + P_{\mathrm{U}}(h))$	W · s
$E(h)$	$E_{\mathrm{I}}(h) + E_{\mathrm{C}}(h) + E_{\mathrm{O}}(h)$	W · s

The locally derived information comprises three components and their estimators (modules) also shown in the figure: s_{J}, the state of local jobs, derived by the job state estimator; s_{E}, the energy consumption state, derived by the energy state estimator; and s_{B}, the bandwidth for communication between h_{L} and every other host, derived by the bandwidth estimator. To accomplish their task, the estimators feed on notification events provided by the local worker and local broker regarding job and transmission events respectively, and runtime profiling of energy consumption and bandwidth measurements. Note that only s_{J} and s_{E} need to be disseminated among instances, whereas the s_{B} information for all hosts is derived locally at each instance.

Job computation

The worker executes jobs in order-of-arrival, one at a time, and non-preemptively until completion. Pending jobs are kept on hold in a FIFO queue. This scheme is not adaptive to deadlines or other job characteristics. But, on the other hand, it allows for a simple estimation of the termination time for a released job that is not affected by the arrival of new jobs or more generally by overall variations in the system workload. Assume that job j_1 starts running at time

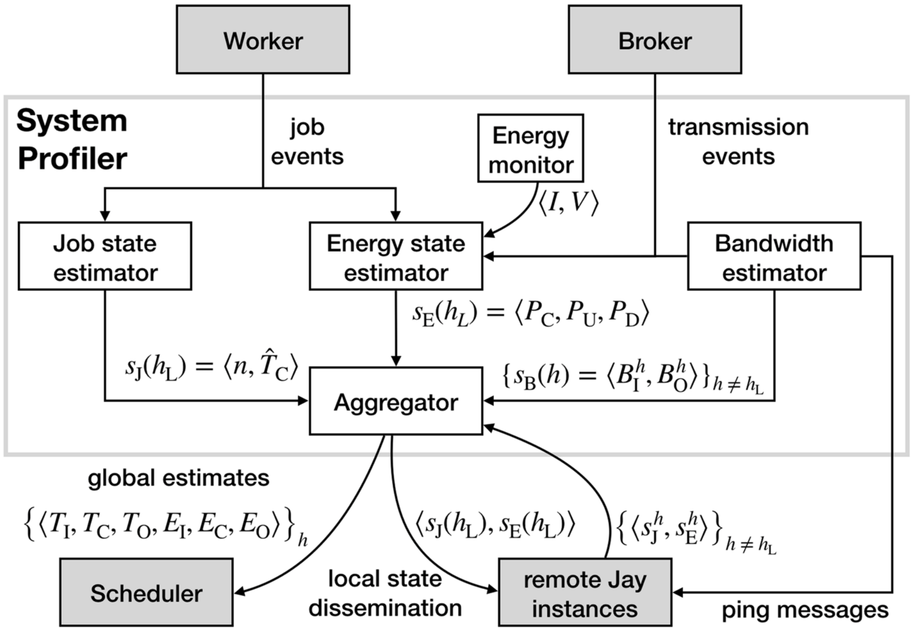

Worker
Broker
System Profiler
job events
Energy monitor
transmission events
$\langle I, V \rangle$
Job state estimator
Energy state estimator
Bandwidth estimator
$s_\mathrm{E}(h_L) = \langle P_\mathrm{C}, P_\mathrm{U}, P_\mathrm{D} \rangle$
$s_\mathrm{J}(h_\mathrm{L}) = \langle n, \hat{T}_\mathrm{C} \rangle$
Aggregator
$\{ s_\mathrm{B}(h) = \langle B_\mathrm{I}^h, B_\mathrm{O}^h \rangle \}_{h \neq h_\mathrm{L}}$
global estimates
$\{ \langle T_\mathrm{I}, T_\mathrm{C}, T_\mathrm{O}, E_\mathrm{I}, E_\mathrm{C}, E_\mathrm{O} \rangle \}_h$
$\langle s_\mathrm{J}(h_\mathrm{L}), s_\mathrm{E}(h_\mathrm{L}) \rangle$
$\{ \langle s_\mathrm{J}^h, s_\mathrm{E}^h \rangle \}_{h \neq h_\mathrm{L}}$
Scheduler
local state dissemination
remote Jay instances
ping messages

Fig. 5 State estimation by the system profiler

t, that jobs $j_2, \ldots, j_n$ are queued, and that there is an estimate Δ_i for the time j_i takes to execute. Then an estimate for the termination time of j_i is simply $t + \Delta_1 + \ldots + \Delta_i$.

Note that this "stable" estimate, derived from limited information, would be impossible to achieve if we were to resort, for instance, to an earliest-deadline first (EDF) scheme in preemptive or non-preemptive form. In this case, the arrival of new jobs could potentially invalidate a previous estimate made during an offloading decision, and raise the need to model/estimate a worst-case behavior for job arrivals.

The worker interacts with the system profiler by supplying a notification whenever a job is queued, starts, and ends. With this information, the profiler can compute an estimate of the job execution time and the worker's queue size and composition. From these quantities we can in turn derive estimates for T_C and $\hat{T}_\mathrm{C}$ (cf. Fig. 2). Feeding on the local worker information, the current profiler estimates $\hat{T}_\mathrm{C}$ using a moving average of the execution time of jobs, and T_C as:

$$T_\mathrm{C}(h) = (n(h) + 1) \times \hat{T}_\mathrm{C}(h)$$

The formula above simply expresses that the time to execute a job j will have to account for the wait for up to n jobs to complete at host h, plus the time to actually execute j. The estimate implicitly assumes however that job execution time tends to be uniform, i.e., there is only one class of job and their execution is regular. This in the case of the jobs we consider for evaluation later, but the scheme could be generalised, e.g. to handle several classes of jobs by accounting for the number of jobs per class, and irregular jobs by accounting for different job input sizes and/or considering execution time percentiles rather than a plain moving average.

Network transmission times

In order to estimate network transmission times, the profiler issues periodic ping (round-trip) messages to all hosts in the network. The information gathered from these messages allows the bandwidth estimator at each host h to maintain a moving average for uploading and downloading bandwidth measures, $B_\mathrm{U}(h)$ and $B_\mathrm{D}(h)$, respectively. These estimates are further refined with information gathered from broker notifications regarding the observed bandwidths when jobs (and their inputs) are uploaded to, or their outputs are downloaded from, a remote Jay instance. Assuming that the sizes of the inputs ($|j|_\mathrm{I}$) and outputs ($|j|_\mathrm{O}$) of a job j are known, the profiler estimates T_I and T_O as follows:

$$T_\mathrm{I}(h) = B_\mathrm{U}(h) \times |j|_\mathrm{I}$$

$$T_\mathrm{O}(h) = B_\mathrm{D}(h) \times |j|_\mathrm{O}$$

Power consumption estimates

The energy state monitor is responsible for maintaining running estimates for the power cost terms P_C, P_U, and P_D that correspond, respectively, to the power consumption per time unit when executing jobs at, uploading data from, and downloading data to the local host. In contrast to other approaches, Jay produces estimates without resorting to any a priori, usually device-specific, derived model for power consumption. As such, power consumption estimates may be more crude but, on the other hand, reflect more closely the energy dynamics of the system at any given moment.

At any given time, a JAY instance may be idle, performing computation, or transmitting data. This can be inferred by listening to job events from the worker and transmission events by the broker or bandwidth monitor. Whenever the worker starts a job, the active job computation status flag is enabled, meaning the ensuing energy consumption should reflect on the P_C estimate. The same flag is disabled whenever the job ends. The P_U and P_D estimates are derived similarly, using upload and download status flags that are enabled and disabled according to the start and end events for uploads and downloads by the broker or the bandwidth monitor. The power consumption estimates are updated as moving averages in accordance to the values of the status flags, but when only one of the flags is active. Power consumption measures are obtained in a device-specific manner through an energy monitor. For instance, by measuring the current I and the voltage V in a device, a simple estimate for the power consumption would be $P = I \times V$ (using Ohm's Law). With estimates for P_C, P_U, and P_D plus $\hat{T}_C$, T_I, and T_O we can in turn express the corresponding energy costs for jobs as follows:

$$E_C(h) = \hat{T}_C(h) \times P_C(h)$$

$$E_I(h) = T_I(h) \times (P_U(h_L) + P_D(h))$$

$$E_O(h) = T_O(h) \times (P_D(h_L) + P_U(h))$$

Note that, in line with the starting discussion for the system model, E_C depends on $\hat{T}_C$ (the effective computation time at h) rather than T_C (the entire time span the job is at h), while E_I and E_O reflect the energy costs both at h_L and h.

Experimental setup

We used the JAY framework to evaluate the algorithms presented in "System model" section, namely: LOCAL, SERVER, TMIN and HYBRID.

Devices

The experimental setup consisted of 5 Android devices and a PC-based cloudlet. In experiments detailed in the next section, Android devices are used as job generators and executors, while the cloudlet is used as job executor only. Their characteristics are summarised in Table 3. It can be seen that the Android devices are quite heterogeneous in terms of CPU, RAM and battery capacity, as well as in terms of their Android OS version. Another important aspect is that the cloudlet has significantly more RAM (16 GB) than all Android devices and, as illustrated in detail later, also uses a higher performance CPU configuration. All devices were connected to the same local network, via an ASUS RT-AC56U router, featuring a 2.4 GHz 300 Mbit/s WiFi connection for the Android devices and a 1 Gb/s Ethernet connection for the cloudlet.

Prior to each experiment, all Android devices had their batteries charged by at least 50%, to prevent interference from builtin power saving mechanisms. They were then disconnected from the power outlet using a smart plug controlled remotely by a script. For monitoring energy we used the standard Android `BatteryManager` API[3]. This API provides current intensity (I) and voltage (V) information, respectively, through the `BATTERY_PROPERTY_CURRENT_NOW` counter and the `EXTRA_VOLTAGE` notifications. The API is available and reliable across all Android versions and devices we tested, from which we estimated the instantaneous power consumption ($P = I \times V$). We used this approach uniformly for all devices, even if in some devices/Android versions the API provided a richer set of attributes such as the `BATTERY_PROPERTY_CURRENT_AVERAGE`, for average current intensity, and `BATTERY_PROPERTY_ENERGY_COUNTER`, for the remaining battery power. As for the cloudlet, it had a permanent 220 V power supply. Its power consumption was monitored using a Meross MSS310 energy plug.

Benchmark application

We used a benchmark application that fires jobs for object detection in images using deep learning, similar to one we employed in previous work [5]. As illustrated in Fig. 6, each object detection job takes an image as input and yields a set of objects detected in the image along with corresponding bounding boxes and confidence scores. Here, we used a "headless" variant of this computational job with no GUI or human intervention. Overall, this type of computation is increasingly common to classify static images or live video frames in mobile devices [27, 28]. It makes for an interesting case-study for offloading since jobs can be computationally intensive and may require high network bandwidth to transfer images. This can happen if the number of spawned jobs is large or if a QoS restriction is added to their execution (e.g., a deadline), or both.

We make use of a MobileNet SSD model variant [29] trained with COCO [30], a popular image dataset used for benchmarking object detection models that contains 2.5 million labeled instances for 80 object types in more than 300.000 images. The specific model we use is `ssd_mobilenet_v1_fpn_coco`, available in standard TensorFlow (TF) [31] and TensorFlow Lite (TFLite) format from TensorFlow's Object Detection Zoo[4]. The

[3] https://developer.android.com/reference/android/os/BatteryManager

[4] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

Table 3 Device characteristics

Device	Year	CPU	RAM	Battery	OS
Cloudlet	2015	Intel i7-6700K, 4x4.0 GHz	16 GB	N/A	Ubuntu 20.04 LTS
Google Nexus 9	2014	Nvidia Tegra K1, 2x2.3 GHz	2 GB	6.7 Ah	Android 8.1
Google Pixel 4	2019	Snapdragon 855, 1x2.84/3x2.42/4x1.78 GHz	6 GB	2.8 Ah	Android 11
Samsung Galaxy S7e	2016	Exynos 8890 Octa, 4x2.3/4x1.6 GHz	4 GB	3.6 Ah	Android 10
Samsung Galaxy Tab S5e	2019	Snapdragon 670, 2x2.0/6x1.7 GHz	6 GB	7.0 Ah	Android 10
Xiaomi Mi 9T	2019	Snapdragon 730, 2x2.2/6x1.8 GHz	6 GB	4.0 Ah	Android 10

object detection job code, adapted from a TensorFlow tutorial[5], has been incorporated into two distinct Kotlin modules, each linked with the JAY core library. One module is used in the cloudlet (Linux) and employs the standard TF library. The other is used in the Android devices and employs TFLite. Besides CPUs, TF and TFLite may employ GPUs, if available. In the case of TFLite in Android, it can also use the specialised Google Neural Networks API [6]. Nevertheless, we configured the device's Kotlin module to use only CPUs given that this basic option works for all devices and operating system versions.

The benchmark application runs on every device and fires object detection jobs according to a Poisson process with a configurable job inter-arrival time λ, and relative deadlines $d \leq \lambda$. Each job takes as input a randomly selected Ultra-HD image taken from the UltraEye dataset [32], and produces an objection detection report of at most 4 KB. Each image has a pixel resolution of 3840×2160 and an average size of 2.2 MB. All images used were uploaded to the Android devices prior to benchmark execution (as mentioned earlier, the cloudlet does not generate jobs in our experiments, it only executes them on behalf of Android devices).

Evaluation

We now present the detailed experiments we conducted and the results we obtained. Their implications are also discussed.

Experiments

Using the experimental setup described in the previous section, we conducted three sets of experiments:

1. We first measured the baseline behavior, in terms of energy and computation time, for the set of devices at hand, when executing the benchmark application in our setup without considering any type of offloading. The goal was to allow a relative comparison between devices given their heterogeneity.
2. We then considered offloading experiments for the benchmark application in a network formed only by the Android devices, where each device acts both as a job generator and worker. We compare the use the LOCAL, TMIN and HYBRID strategies for job workloads with different values for mean job inter-arrival times and job deadlines.
3. The previous experiments were repeated for the benchmark application this time using a network that also includes a cloudlet worker. Again we consider TMIN and HYBRID strategies, but also the SERVER strategy that offloads all jobs to the cloudlet server.
4. Finally, we consider again a network with mobile devices, the effect of using the BALANCED and LF[f]

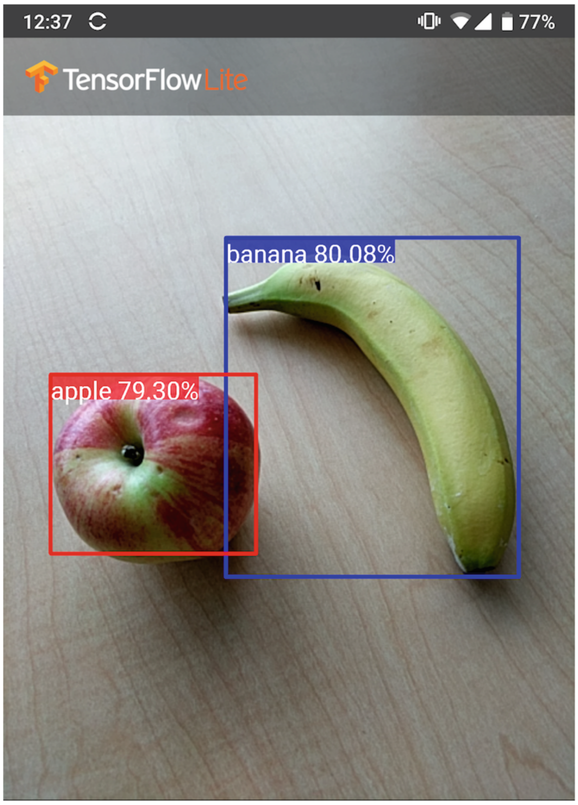


Fig. 6 Object detection in images

[5] https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/android
[6] https://developer.android.com/ndk/guides/neuralnetworks/

strategies versus TMIN and HYBRID, plus a different choice of configuration where jobs are generated and scheduled by a single external host and the mobile devices act only as workers, i.e., a Femtocloud-like configuration.

Baseline experiments

For a baseline comparison between devices, we measured power consumption and time/energy consumption during job execution for all devices. We first ran scripts to measure (instantaneous) power consumption when devices were idle, uploading data and downloading data. Each script ran for 10 minutes and average power consumption results were gathered from 3 script executions. For the uploading/downloading power measurements the scripts continuously executed plain file uploads/downloads to/from a random host in the network. For job computation behavior, we ran a script that issued local object detection jobs continuously for 10 minutes, again for 3 rounds, and computed the average power consumption and job execution time.

The results are listed, per device, in Table 4: power consumption (in Watts, when devices are idle, uploading, downloading, or computing); job execution time (seconds), and; energy consumption (in milliwatt-hour, taking into account power consumption when computing and the execution time per job).

Overall, the results clearly expose the heterogeneity of the devices used in the experiments. Looking at the power consumption results, it is clear that computation is the major factor of increase in power consumption: 2.3–4.1 times more energy is consumed than when a device is idle, compared to just 1.1–3.5 times for uploading and 1.1–1.8 times for downloading. Compared to the Android devices, power consumption numbers for the cloudlet are an order of magnitude higher (approx. 10–40 times higher). Energy-wise, the two best-performing devices while computing are Google Pixel 4 and Xiaomi Mi 9T. Samsung Galaxy S7e is the most energy conservative device when in idle mode.

Regarding the results for execution time and energy consumption per job, the cloudlet stands out again: it is both the most efficient device in computation time, and the least efficient one in energy consumption: jobs run 1.9–5.8 times faster than on the Android devices but on the other hand consuming 2.4–15.6 times more energy. Among the Android devices, and for both time and energy, Google Pixel 4 is the most efficient device, followed by Xiaomi Mi 9T, Samsung Galaxy Tab S5e, and Samsung Galaxy S7e, with Google Nexus 9 being the least efficient.

We note that the measures for energy consumption per job are more relevant for our purposes (cf. "The Jay framework" section) than those for instantaneous power consumption. Observe that Samsung Galaxy Tab S5e is more energy-efficient (consumes 5.0 mWh per job) than Samsung Galaxy S7e (which consumes 5.5 mWh per job, 10% more), even if instantaneous power consumption is higher during computation (4.5 W vs. 3.7 W, 21% higher). The reason for this is that the higher power consumption is compensated in a larger proportion by faster job execution times in Samsung Galaxy Tab S5e (4.0 s vs. 5.4 s, 33% faster).

As for the measured bandwidth during the duration of the experiment, we obtained values averaging 110 Mbit/s for download on all mobile devices and for upload we verified two distinct behaviors: Nexus 9, Pixel 4 and Samsung Galaxy S7e connected with a 300 Mbit/s connection averaging 210 Mbit/s speeds while Samsung Galaxy Tab S5e and Xiaomi Mi 9T connected to the router with a 150 Mbit/s connection leading to an average upload speed of 119 Mbit/s. As for the cloudlet, it was connected to our router via gigabit ethernet and we obtained and average of 941 Mbit/s upload speed and 946 Mbit/s download.

Offloading among android devices

We considered a network formed by the Android devices, each running the benchmark application generating jobs with mean inter-arrival times for the governing Poisson process of λ equal to 3, 6, 9 and 12 seconds (which translates to 20, 10, 6.7 and 5 jobs per minute respectively), and values of $d = 3, 6, 9, 12$ for their relative deadlines up to the value of λ (i.e., $d \leq \lambda$).

Table 4 Baseline results per device

	Energy consumption (mWh for 10 minute-intervals)				Per Job	
Device	Idle	Upload	Download	Computation	Time (s)	Energy (mWh)
Cloudlet	5783.3	6038.3	5935.0	15706.7	1.5	39.5
Google Nexus 9	433.3	476.7	556.7	1138.3	8.8	16.6
Google Pixel 4	185.0	271.7	230.0	528.3	2.9	2.5
Samsung Galaxy S7e	148.3	523.3	260.0	610.0	5.4	5.5
Samsung Galaxy Tab S5e	316.7	521.7	385.0	741.7	4.0	5.0
Xiaomi Mi 9T	151.7	250.0	178.3	535.0	3.2	2.9

In conjunction, we considered three offloading strategies, presented in "System model" section: LOCAL (local execution only, no offloading), TMIN (offloads jobs strictly seeking to minimize execution time) and HYBRID (balances QoS constraints for task deadlines with energy efficiency). The benchmark was executed 6 times for each offloading strategy with the same job generation seed, and each execution was configured to generate jobs for 10 min.

A first set of overall results for the experiment is presented in Fig. 7. We present plots for the energy consumption and execution time per job (left and middle in the figure, lower numbers are better), along with the corresponding quality-of-service (QoS) that is expressed as the percentage of jobs with a fulfilled deadline (right, higher numbers are better). The average values are plotted for each measure, along with the amplitude of the 95% gaussian confidence interval. Note that, for each configuration, the average energy consumption is obtained by measuring the total energy consumption in all of the devices, including idle time, divided by the number of jobs. Lower values of λ imply more jobs, hence the average energy consumption tends to decrease with λ (conversely, idle time grows with λ).

From the results, we can first observe that both TMIN and HYBRID generally outperform LOCAL both in energy consumption and QoS. This shows that offloading jobs pays off in both dimensions when compared to strictly local execution of jobs. The exception to this pattern is observed when the relative deadline has the tightest value, i.e., $d = 3$, and only in terms of QoS. In fact, the overall system becomes incapable of achieving reasonable QoS in all configurations when at this point: always below 30%, regardless of offloading strategy. In the more extreme case where $\lambda = d = 3$, the QoS is below 10% and there is an extremely long execution time, due to the fact that jobs simply pile up in the system. In contrast, the QoS is always higher than 50% for all configurations with $d > 3$.

Comparing TMIN and HYBRID, the results are very similar for $d = 3$ and $d = 6$ in all respects (energy, time, and QoS). Since these deadline values are the most tight, the HYBRID strategy has less scope for energy-efficient offloading choices and these tend to be similar to the choices made by TMIN. For $d > 6$ there are noticeable differences though, highlighting that gains in energy consumption can be attained by the HYBRID strategy compared with TMIN at the cost of a slight penalty in

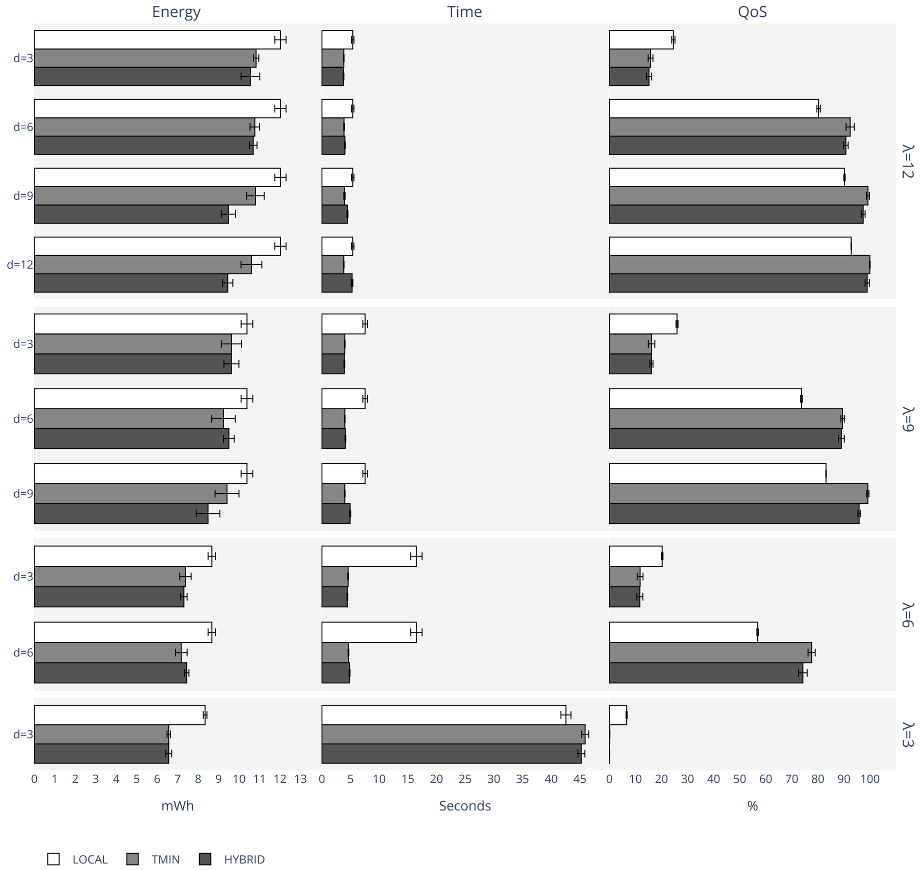


Fig. 7 Android devices scenario – energy, time, and QoS

QoS. The HYBRID strategy leads to a 10–20% decrease in energy consumption compared to TMIN, while the QoS service is only marginally higher for TMIN, at most by 5%. At the same time, the execution time is slightly higher for HYBRID, given that the strategy does not pick the device estimated to run a job faster but, rather, the most energy-efficient among those that are estimated to comply with the job deadline. For example, when $\lambda = 12$ and $d = 9$ and for HYBRID we observed: 13% less energy consumption (9.5 mWh compared to $\sim$10.8 mWh for TMIN); jobs taking 15% longer (4.5 s vs. 3.9 s), but; a QoS degradation of only 2% (97% vs. 99%).

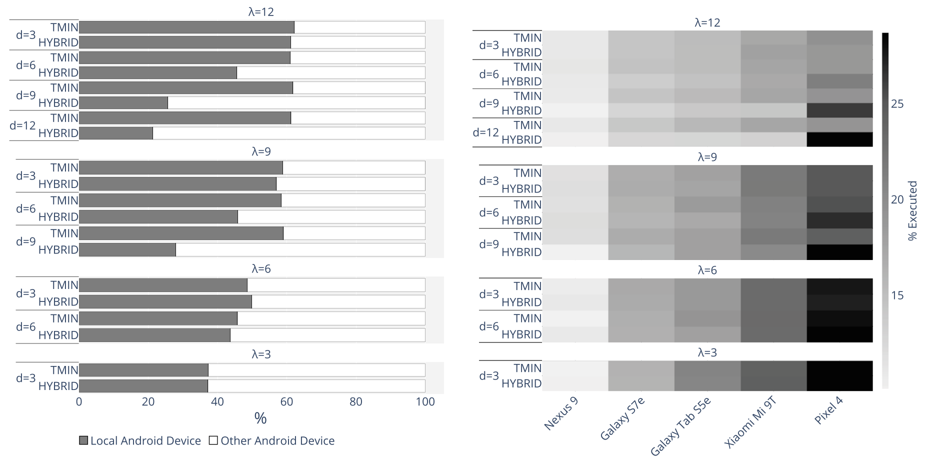
The behavior of TMIN and HYBRID is compared in more detail in Fig. 8, regarding the fraction of offloaded jobs (Fig. 8a, left) and the fraction of jobs executed per device (Fig. 8b, right). These results again illustrate that there is no significant difference between both strategies for the tighter deadline of $d = 3$. As the value of d grows, however, the offloaded job ratio tends to grow and be significantly higher for the HYBRID strategy, whereas there are only small variations for TMIN for each value of λ. When $\lambda = 12$ for instance, the offloading ratio increases progressively in the case of HYBRID as d grows from $\sim$40% when $d = 3$ up to $\sim$80% when $d = 12$, while for TMIN it is $\sim$40% for all values of d.

If we look at the fraction of executed jobs per device (Fig. 8b), we see that they are overall in line with the baseline results, i.e., faster devices (which are also more energy-efficient) execute more jobs. For instance, Google Nexus 9, the slowest device, executes the fewest jobs, while Google Pixel 4, the fastest one, executes the most jobs. The total spread of jobs is more uniform in the case of TMIN than with HYBRID, while HYBRID tends to favor Google Pixel 4 significantly for $d \geq 6$. These aspects are illustrated in particular for the flow of jobs when $\lambda = 12$ and $d = 9$, again comparing HYBRID (a) and TMIN (b), in Fig. 9. The job distribution is noticeably more biased towards Google Pixel 4 in the case of HYBRID: Google Pixel 4 executes 56% of all jobs for HYBRID compared to 33% for TMIN. TMIN offloads jobs more uniformly to the other devices (note that the size of the squares grows logarithmically), even if Nexus 9 only executes local jobs in the case of TMIN.

We finish our analysis by highlighting estimation errors by Jay's system profiler. Figure 10 depicts the average relative error in the estimated time for job execution, calculated as the difference between estimated and real execution time, expressed in percentage of the real execution time. As shown, the values are on average negative, meaning that the estimates tend to be pessimistic. In amplitude, they are less than 20% except for HYBRID when $\lambda = 12, 9$ and $d \geq 9$, and both strategies when $\lambda = 3$. This is partly explained by the fact that the estimate $\hat{T}_C$ for execution time of a job at a Jay instance accounts for the current number of jobs including the current one, but not the already spent executing the current job. This behavior, which can be mitigated in future developments of the Jay prototype, is amplified in configurations where one the devices obtains a high share of jobs (Google Pixel 4 in the case of HYBRID, for $\lambda = 12, 9$ with $d \geq 9$). On the other hand, when the system has a high load and is unable to cope (the case of $\lambda = 3$), estimates also tend to be less reliable.

Extended scenario using cloudlet

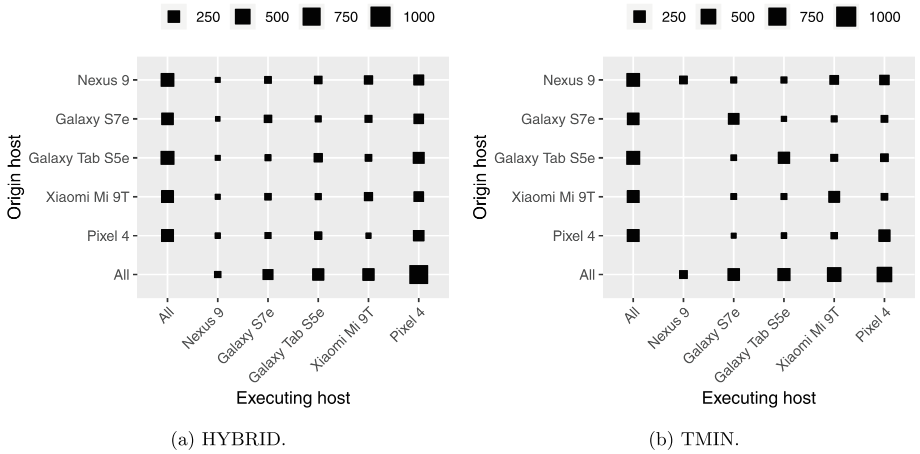
We now present results for an extension of the previous experiment that introduces a cloudlet server. The cloudlet acts only as a Jay job executor, while job



(a) Local vs. offloaded jobs.

(b) Executed jobs per device.

Fig. 8 Android devices scenario – job distribution



(a) HYBRID. (b) TMIN.

Fig. 9 Android devices scenario – flow of jobs for $\lambda = 12$ and $d = 9$

generation proceeds as before for the Android devices. As before, the HYBRID and TMIN strategies were evaluated but with the possibility of offloading from the devices to the cloudlet. We consider, in addition, the SERVER strategy that uses the cloudlet as a standalone server that executes all jobs. We present measurements similar to the previous scenario and highlight the impact of the cloudlet.

In Fig. 11 we provide plots for energy consumption, execution time and QoS. Compared to the results of the scenario without cloudlet (cf. Fig. 7) an increase in energy consumption as well as in QoS is noticeable for the TMIN and HYBRID strategies. This would be expected, given that (in line with the baseline results) the cloudlet is the most time-efficient device but also the least energy-efficient one. The energy consumption is significantly higher, something that will always be true even if the cloudlet executes no jobs (in any case it will still actively consume energy). For example, the lowest energy consumption value is 23 mWh for HYBRID and TMIN when $d = \lambda = 3$, exceeding the value of the most energy-hungry configuration of the previous scenario, 12 mWh for $\lambda = d = 12$ in Fig. 7. On the other hand, the cloudlet improves QoS for HYBRID and TMIN significantly: it is now above 90% for every configuration with $d \geq 6$, and even 46%–67% for $\lambda = 12, 9, 6$ when $d = 3$ in comparison to the 10–15% observed previously. QoS is very poor only, and again, in the extreme $\lambda = d = 3$ case.

Looking at the results for the SERVER strategy, they are generally worse than those obtained for HYBRID and TMIN. This is true for energy consumption in all configurations, and also QoS except for configurations with

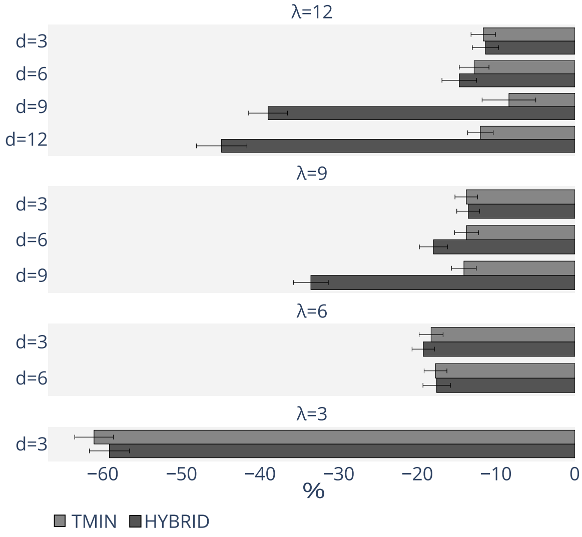


Fig. 10 Android devices scenario – average error

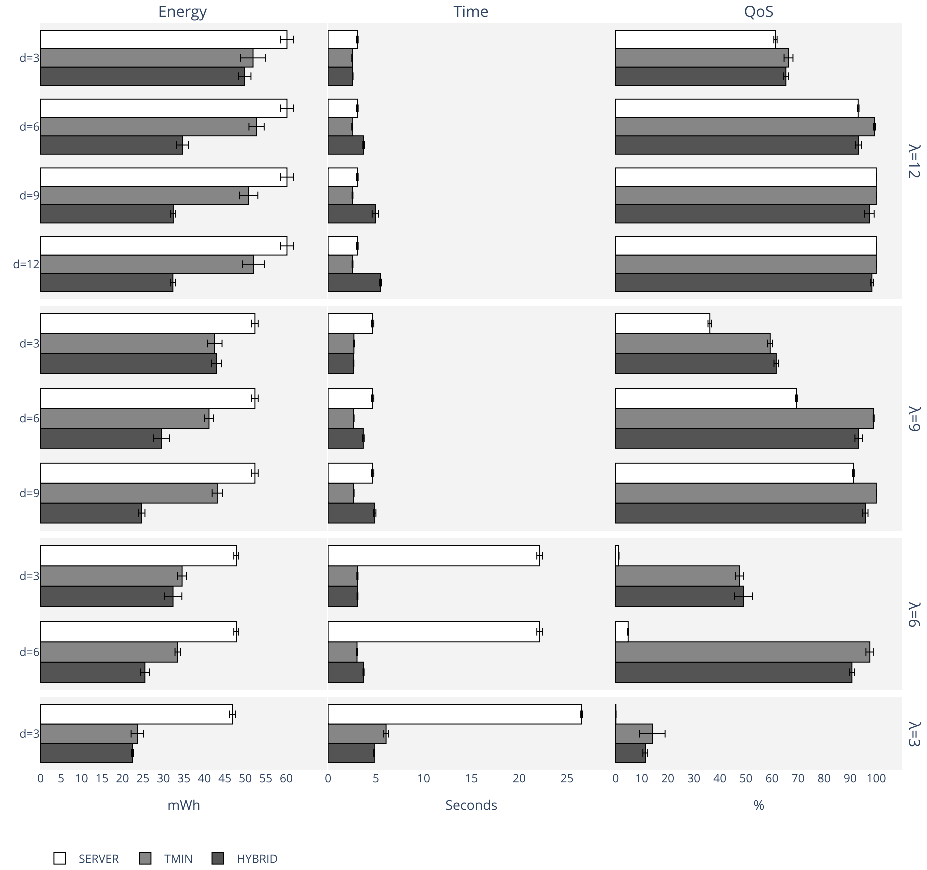


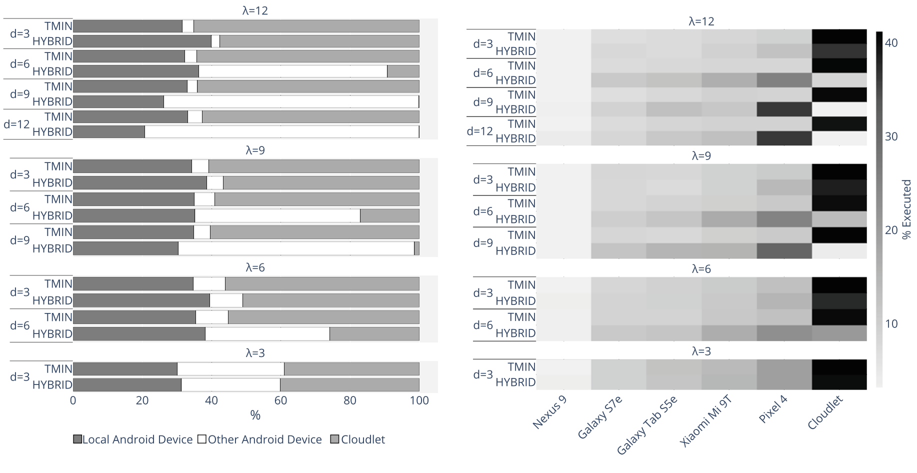
Fig. 11 Cloudlet scenario – energy, time and QoS

$\lambda = 12$ where the SERVER strategy becomes competitive. This means that job execution/offloading by the Android devices pays off compared to using the cloudlet alone, much like in the previous scenario where it payed off when compared to using local job execution only.

In this cloudlet scenario, energy consumption savings resulting from the use of HYBRID vs. TMIN can be more pronounced. In all configurations, HYBRID consumes less energy than TMIN, and the savings are noticeably more pronounced as d increases, e.g., for $\lambda = 12$, TMIN consumes just 4% more energy when $d = 3$ but 60% more when $d = 12$. On the other hand, on par with the decrease in energy consumption, HYBRID leads to noticeably longer job execution times as d grows, e.g., again for $\lambda = 12$ HYBRID causes jobs to last from 2% longer when $d = 3$ up to 214% when $d = 12$.

The difference of behavior between HYBRID and TMIN is best understood looking at the job distribution results in Fig. 12, where we depict for all configurations the fractions of: (Fig. 12a) locally executed jobs, jobs offloaded to Android devices, and jobs offloaded to the cloudlet, and; (Fig. 12b) jobs per Android device and cloudlet. Besides the fact that HYBRID tends to have a lower ratio of locally executed jobs, as in the previous Android devices only scenario, the other major difference between HYBRID and TMIN is that HYBRID tends to offload significantly less jobs to the cloudlet than TMIN. Looking at the distribution per device, it is clear that with HYBRID Google Pixel 4 is the device executing more jobs, whereas TMIN privileges the cloudlet. In fact, in some configurations, the fraction of jobs executed by the cloudlet can be residual. The job flow for both strategies when $\lambda = 12$ and $d = 9$ is illustrated in Fig. 13, and highlights this trend in one of the more extreme cases: the cloudlet executes less than 1% of all jobs for HYBRID while Google Pixel 4 executes 57%, whereas for TMIN the fractions are 64% for the cloudlet and 10% for Google Pixel 4 (note that, as before, the size of the squares grows logarithmically).

Estimation errors by JAY's system profiler are presented in Fig. 14 for the cloudlet scenario, with similar trends to the Android devices' scenario (Fig. 10). The main difference is that estimation errors are not as high for the $\lambda = 3$ case. For this configuration, the overall system copes much better with the high load scenario of in terms of job execution times even if QoS is still low, and execution estimate errors tend to be lower as a result.



(a) Local, device, and cloudlet job share.

(b) Jobs per device.

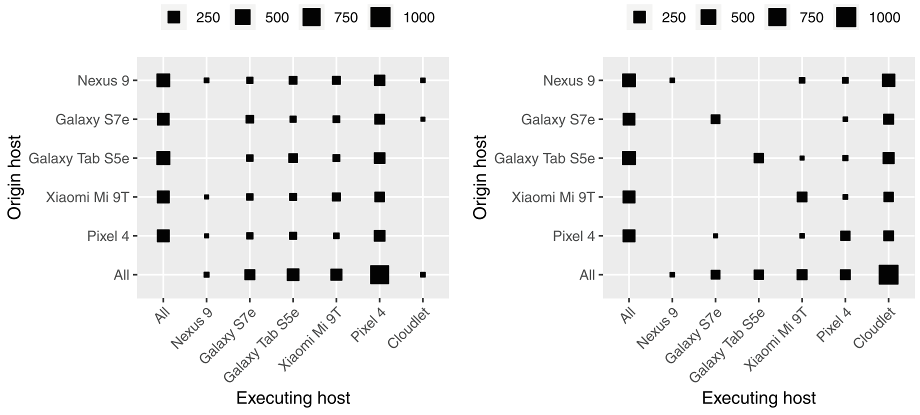
Fig. 12 Cloudlet scenario – job distribution

Additional scenarios

A final set of results is now presented, considering again a network formed by mobile devices alone. We consider the effect of having a network with a Femtocloud configuration (FC), in which jobs are generated and scheduled by a single external host and the mobile devices act only as workers, in contrast to the mobile edge cloud configuration (MEC) where all devices act as job generators and workers. Furthermore, we present results for additional offloading strategies, BALANCED and LF[f] (cf. "System model" section) that may potentially lead to different compromises in terms of time, energy, and job distribution among hosts. BALANCED applies both in the FC and MEC cases, whereas LF[f] ("local-first") by definition only applies in the MEC case (in the FC case, the external host does not act as worker, hence it does not execute jobs locally).

Jobs were generated with the same methodology as in the previous experiments the MEC configuration, but results were gathered only for a job inter-arrival time of $\lambda = 9$ and deadlines $d = 6, 9$. In the FC case, similar deadlines are considered, but the external host generates jobs with a $\lambda = \frac{9}{5}$ inter-arrival time, so that the overall workload is equivalent to the use of $\lambda = 9$ by all 5 devices in the MEC configuration. We empirically found these workload parameterisations to be illustrative of the behavior of the system for the strategies considered.

As in the previous experiments, the results are presented in terms of: average energy consumption, average



(a) HYBRID.

(b) TMIN.

Fig. 13 Cloudlet scenario – flow of jobs for $\lambda = 12$ and $d = 9$ (number of jobs)

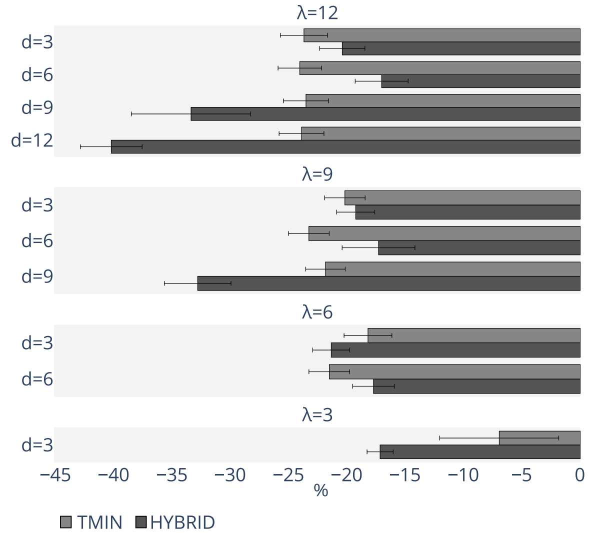


Fig. 14 Cloudlet scenario – average error

completion time and QoS (Fig. 15), and; job offloading rates and job share per device (Fig. 16).

Femtocloud setting

Looking first at the FC results, the energy consumption values are clearly the lowest, as shown in Fig. 15 (top-left). Compared to the MEC scenario, the energy consumption values for TMIN and HYBRID are $14 - 20\%$ lower for $d = 6$ and 25% lower for $d = 9$. These gains, however, come at the cost of higher execution times and lower QoS: for $d = 6$ execution times $7 - 13\%$ higher, and the QoS is $4 - 7\%$ lower; for $d = 9$ the execution time are $11 - 12\%$ higher but the QoS differences are small, lower than 2% in absolute value. Thus, the results are mixed, especially in the case of $d = 6$.

A priori, it would be expected that the centralised offloading decisions to be more reliable in the FC configuration, since it is free from the interference that arises from concurrent offloading decisions by all devices in the MEC case. However, in the MEC case jobs can execute locally, e.g. for $d = 6$ the share of local jobs is 56% for TMIN and 47% for HYBRID, as depicted at the bottom in Fig. 16a, and estimation errors tend to be lower for locally executed jobs. In the FC case (by definition) all jobs are offloaded leading to higher estimate errors. These two factors influence the behavior in different directions.

LF[f] strategies

By definition, LF[f] strategies try to execute as many jobs as possible locally, resorting to offloading through strategy f only when the local device is unable to cope with the deadline of a job. Accordingly, as shown in Fig. 16(a), the share of locally executed jobs is significantly higher for LF[f] when compared to f in almost all cases, $15 - 49\%$ more, except for TMIN when $d = 6$ where the difference is negligible ($< 1\%$).

The results for LF[f] strategies are otherwise indicative of energy/time/QoS trade-offs, as illustrated in Fig. 15. This happens especially for $d = 6$. In this case, when compared to TMIN, LF[TMIN] leads to a decrease of 9% in energy consumption but, also, an increase of 9% in execution time and a decrease of 4% in QoS. This is expected as the base strategy, TMIN, seeks to minimize execution time and thus it will tend to do better in this metric as well as in QoS. Again for $d = 6$, but using HYBRID as the base strategy this time, LF[HYBRID] degrades execution time and QoS by even more, 12% and 9% respectively, even if energy consumption is roughly the same (1% difference between both). Given that most energy-efficient devices tend to also be faster in our configuration, the degradation of execution time and QoS is expected, as with TMIN, but the difference in energy consumption is only noticeable for the larger deadline value of $d = 9$, where the energy consumption of LF[HYBRID] is 9% higher.

The BALANCED strategy

Finally, the BALANCED base strategy has the overall effect of smoothing the load distribution among devices, as intended; recall (from "System model" section) that the BALANCED strategy makes a random choice among devices that are estimated to comply with a job's deadline. Examining the numbers for the plot in Fig. 16 (b), for configurations that employ TMIN and HYBRID as a base or fallback strategy, the average shares of the jobs for the Xiaomi Mi 9T and Pixel 4 devices combined (the two devices that execute most jobs) are 64% for $d = 6$ and 63% for $d = 9$. In comparison, for configurations that employ BALANCED as a base or fallback strategy, the share of two devices is 3% lower (61%) for $d = 6$ and, more noticeably, 10% lower for $d = 9$ (53%). In more detail for $d = 9$, the average individual share grows for all of the 3 least used devices in the case of BALANCED: from 3% to 6% for Nexus 9, from 12% to 16% for Galaxy S7e, and from 21% to 25% for Galaxy Tab S5e. At the same time, the average share in the case of BALANCED drops from 26% to 24% for Xiaomi Mi 9T, and from 37% to 29% for Pixel 4.

Unlike the cases for the base strategies TMIN and HYBRID, the results for BALANCED do not exhibit a clear trend with respect to energy/time/QoS trade-offs (Fig. 15). All 6 executions per configuration use the same seed to guarantee a repeatable job generation pattern. The particular job pattern may be benefiting or hurting the behavior of BALANCED in subtle ways according to the configuration parameters, and the actual random choice made by the BALANCED strategy during execution. For instance, for $d = 6$ and the FC setting, the BALANCED strategy consumes $8 - 10\%$ more energy than TMIN and HYBRID, execution times are $5 - 6\%$ faster, and QoS is 4%

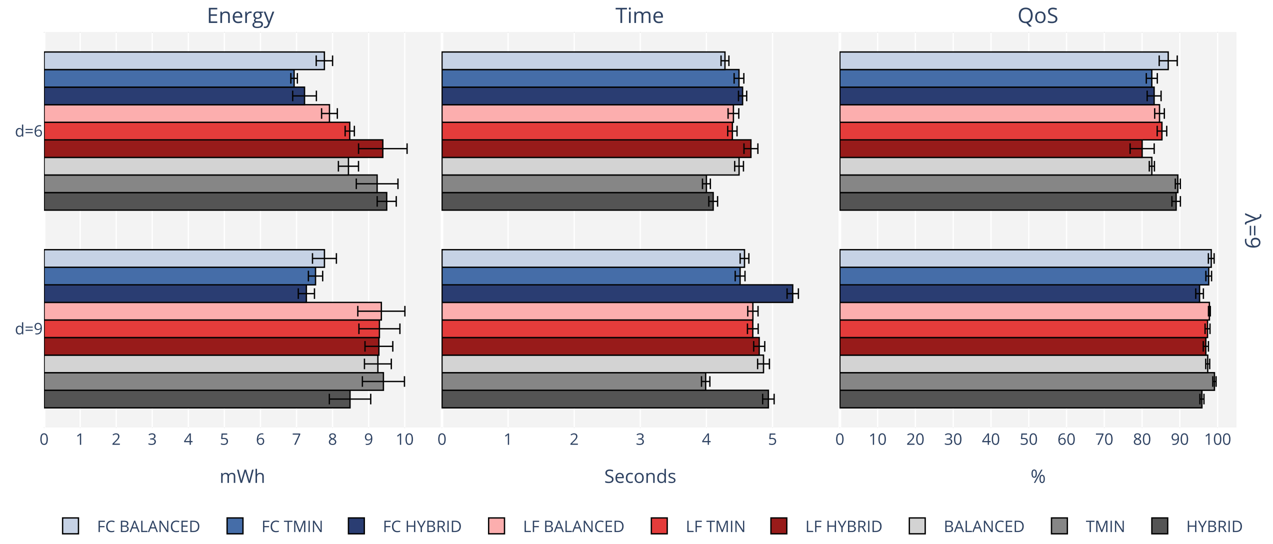
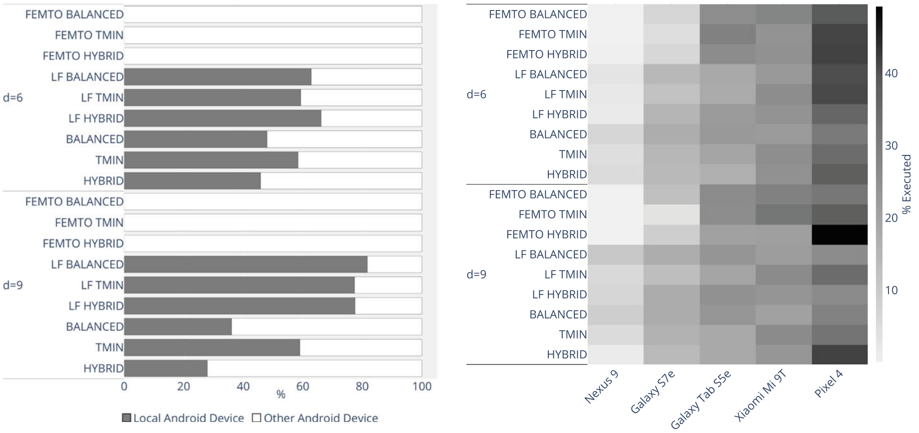


Fig. 15 Additional scenarios – energy, time and QoS

higher. The trend is however roughly symmetric for the MEC setting for instance: 9 − 11% less energy, 10 − 12% slower execution times, and a 7% lower QoS value.

Conclusion

In this paper we presented a model for soft-real time job offloading over hybrid cloud topologies, along with offloading strategies that try to optimize (either all or in part) execution time, total energy consumption and fulfill QoS requirements in the form of job deadlines. We instantiated the model in a software system, JAY, and used it to evaluate a variety of offloading strategies in clouds formed by mobile devices and two-tier hybrid clouds formed by a network of mobile devices and a cloudlet. JAY is designed with adaptive scenarios in mind. Offloading strategies are fed with the necessary runtime information to perform time and energy-aware offloading on-the-fly, Moreover, it employs a modular architecture that allows multi-tier hybrid cloud topologies to be defined with customisable roles per tier or device regarding job generation and execution. The overall system flexibility was illustrated through experiments using a benchmark application configured to spawn jobs with different rates and different soft real-time deadlines, executed over different cloud configurations and offloading strategies. The results of these experiments show that offloading strategies sensitive to runtime conditions can effectively and dynamically adjust their offloading decisions to produce significant gains in execution time, energy consumption and fulfillment of job deadlines.



(a) Local and device job share. (b) Jobs per device.

Fig. 16 Additional scenarios – job distribution

For future work, we consider two key directions:

- Regarding application scenarios, we are particularly interested in articulating computation offloading with data-placement awareness, as in systems like Oregano [20], our previous work on systems for data dissemination for hybrid edge clouds [16, 17], which are particular instances of a class of systems that have multiple users, and employ multiple mobile devices, servers, and network tiers [33]. A challenge in these scenarios is that jobs may potentially require data stored at distinct hosts and/or tiers in the cloud, hence the interplay between computation and data offloading can potentially play a key role. A different challenge is the possibility of high device churn and intermittent connectivity over heterogeneous communication links (WiFi, Bluetooth, 4G/5G, etc), requiring offloading to proceed opportunistically, to be articulated with fault tolerance mechanisms (e.g., job checkpointing or replication), and the overall handling of a more dynamic environment regarding computational resources, network bandwidth, and energy consumption.
- Regarding JAY as a system, it can be extended in a number of ways to support a richer set of offloading strategies and job workloads. Given its modular architecture, JAY can easily accommodate for other multi-objective offloading strategies, of which the hybrid latency-energy offloading strategy is just an example, that account for additional aspects beyond execution time and energy consumption, e.g., the costs of using an infrastructure cloud or mobile device network traffic. Moreover, even if JAY is adaptive over a variety of hybrid cloud architectures, we believe that awareness of the cloud system used for offloading can lead to novel adaptive offloading strategies, e.g., as in the TRACTOR algorithm [34] that accounts for aspects such as power consumption of network switches at the edge-cloud level for traffic and power-aware virtual machine placement. Finally, the system can also be improved for adaptivity in terms of resource awareness to cope with changeable cloud links due to mobility, and computational resources (e.g., GPUs could be used on the mobile devices by our deep learning benchmark). Mobile applications also commonly exhibit features that would require our job model to richer, e.g., job precedences, job aggregation and their parallel execution, checkpointing to allow migration, etc.

Acknowledgements
The authors wish to thank the anonymous reviewers for the helpful feedback.

Authors' contributions
Joaquim Silva programmed JAY and conducted the evaluation experiments. All authors have participated in the conceptual design of the JAY and associated experiments, data analysis, and manuscript writing. The author(s) read and approved the final manuscript.

Authors' information
All authors are affiliated to the Department of Computer Science, Faculty of Sciences, University of Porto (DCC/FCUP), and the Center for Research in Advanced Computing Systems at INESC TEC (CRACS/INESC-TEC).
Joaquim Silva is a PhD student in Computer Science at DCC/FCUP, Eduardo R. B. Marques is an assistant professor at DCC/FCUP, Luís Lopes is an associate professor at DCC/FCUP, and Fernando Silva is a full professor at DCC/FCUP. All authors are researchers at CRACS/INESC TEC.

Funding
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.

Availability of data and materials
JAY is available as open-source software at https://github.com/jqmmes/Jay/.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 8 January 2021 Accepted: 14 June 2021
Published online: 10 July 2021

References

1. Wikipedia (2020) Apple Designed Processors; consulted on December 1. Available at https://en.wikipedia.org/wiki/Apple-designed_processors. Accessed: 1 May 2021
2. Fernando N, Loke SW, Rahayu W (2013) Mobile cloud computing: A survey. Futur Gener Comput Syst 29(1):84–106
3. Drolia U, Martins R, Tan J, Chheda A, Sanghavi M, Gandhi R, et al (2013) The Case for Mobile Edge-Clouds. IEEE, Washington
4. Satyanarayanan M, Bahl P, Caceres R, Davies N (2009) The Case for VM-Based Cloudlets in Mobile Computing. IEEE Pervasive Comput 8(4):14–23
5. Silva J, Marques ERB, Lopes L, Silva F (2020) Jay: Adaptive Computation Offloading for Hybrid Cloud Environments. IEEE, Washington
6. Kumar K, Liu J, Lu YH, Bhargava B (2013) A survey of computation offloading for mobile systems. Mob Netw Appl 18(1):129–140
7. Mach P, Becvar Z (2017) Mobile edge computing: A survey on architecture and computation offloading. IEEE Commun Surv Tutor 19(3):1628–1656
8. Shakarami A, Ghobaei-Arani M, Masdari M, Hosseinzadeh M (2020) A survey on the computation offloading approaches in mobile edge/cloud computing environment: a stochastic-based perspective. J Grid Comput 18(4):639–671
9. Kemp R, Palmer N, Kielmann T, Bal H (2012) Cuckoo: A computation offloading framework for smartphones. In: Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST. vol. 76 LNICST. Springer Verlag, Berlin. pp 59–79
10. Shi C, Habak K, Pandurangan P, Ammar M, Naik M, Zegura ECOSMOS (2014) Computation offloading as a service for mobile devices. In: Proc. MobiHoc. ACM, New York. pp 287–296
11. Verbelen T, Simoens P, De Turck F, Dhoedt B (2012) AIOLOS: Middleware for improving mobile application performance through cyber foraging. J Syst Softw 85(11):2629–2639
12. Cuervoy E, Balasubramanian A, Cho DK, Wolman A, Saroiu S, Chandra R, et al (2010) MAUI: Making smartphones last longer with code offload. In: Proc. MobiSys. ACM Press, New York. pp 49–62
13. Xia F, Ding F, Li J, Kong X, Yang LT, Ma J (2014) Phone2Cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing. Inf Syst Front 16(1):95–111
14. Kosta S, Aucinas A, Hui P, Mortier R, Zhang X (2012) ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: Proc. INFOCOM. IEEE, Washington. pp 945–953

15. Neto JLD, Yu SY, Macedo DF, Nogueira JMS, Langar R, Secci SULOOF (2018) A User Level Online Offloading Framework for Mobile Edge Computing. IEEE Trans Mob Comput 17(11):2660–2674
16. Garcia M, Rodrigues J, Silva J, Marques ERB, Lopes L (2020) Ramble: Opportunistic Crowdsourcing of User-Generated Data using Mobile Edge Clouds. In: Proc. FMEC. IEEE, Washington. pp 172–179
17. Rodrigues J, Marques ERB, Silva J, Lopes LMB, Silva F (2018) Video Dissemination in Untethered Edge-Clouds: a Case Study. In: Proc. DAIS. Springer, Cham. pp 137–152
18. Habak K, Ammar M, Harras KA, Zegura E (2015) Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge. In: Proc. CLOUD. IEEE, New York. pp 9–16
19. Fernando N, Loke SW, Rahayu W (2012) Honeybee: A Programming Framework for Mobile Crowd Computing. In: Proc. MobiQuitous. Springer, Berlin. pp 224–236
20. Sanches P, Silva JA, Teófilo A, Paulino H (2020) Data-Centric Distributed Computing on Networks of Mobile Devices. In: Proc. EuroPar. Springer, Cham. pp 296–311
21. Silva J, Silva D, Marques ERB, Lopes L, Silva F (2017) P3-Mobile: Parallel Computing for Mobile Edge-Clouds. In: Proc. CrossCloud. ACM, New York. pp 5:1–5:7
22. Arslan MY, Singh I, Singh S, Madhyastha HV, Sundaresan K, Krishnamurthy SVCWC (2015) A distributed computing infrastructure using smartphones. IEEE Trans Mob Comput 14(8):1587–1600
23. Miluzzo E, Cáceres R, Chen YF (2012) Vision: MClouds - Computing on Clouds of Mobile Devices. In: Proc. MCS. ACM, New York. pp 9–14
24. Mtibaa A, Fahim A, Harras KA, Ammar MH (2013) Towards Resource Sharing in Mobile Device Clouds: Power Balancing across Mobile Devices. In: Proc. MCC. ACM, New York. pp 51–56
25. Gedawy HK, Habak K, Harras K, Hamdi M (2020) RAMOS: A Resource-Aware Multi-Objective System for Edge Computing. IEEE Trans Mob Comput:1–1. https://doi.org.10.1109/TMC.2020.2984134
26. Kharbanda H, Krishnan M, Campbell RH (2012) Synergy: A middleware for energy conservation in mobile devices. In: Proc. CLUSTER. IEEE, New York. pp 54–62
27. Ota K, Dao MS, Mezaris V, De Natale FG (2017) Deep learning for mobile multimedia: A survey. ACM Trans on Multimedia Computing. Commun Appl (TOMM) 13(3s):34
28. Xu M, Liu J, Liu Y, Lin FX, Liu Y, Liu X (2019) A First Look at Deep Learning Apps on Smartphones. In: Proc. WWW. ACM, New York. pp 2125–2136
29. Howard AG, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, et al (2017) MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv 1704:04861
30. Lin TY, Maire M, Belongie S, Hays J, Perona P, Ramanan D, et al (2014) Microsoft COCO: Common objects in Context. In: Proc. ECCV. Springer, Cham. pp 740–755
31. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al (2016) TensorFlow: A system for large-scale machine learning. In: Proc. OSDI. Usenix, Savannah. pp 265–283
32. Nemoto H, Hanhart P, Korshunov P, Ebrahimi T (2014) Ultra-Eye: UHD and HD images eye tracking dataset. In: Proc. QoMEX. IEEE, New York. pp 39–40
33. Huang L, Feng X, Zhang L, Qian L, Wu Y (2019) Multi-server multi-user multi-task computation offloading for mobile edge computing networks. Sensors 19(6):1446
34. Kumar K, Liu J, Lu YH, Bhargava B (2013) A survey of computation offloading for mobile systems. Mob Netw Appl 18(1):129–140

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.