EFPO: Energy Efficient and Failure Predictive Edge Offloading

Josip Zilic
Vienna University of Technology
Vienna, Austria
josip.zilic@tuwien.ac.at

Atakan Aral
Vienna University of Technology
Vienna, Austria
atakan.aral@tuwien.ac.at

Ivona Brandic
Vienna University of Technology
Vienna, Austria
ivona.brandic@tuwien.ac.at

ABSTRACT

Many researchers focus on offloading issues and challenges to improve energy efficiency and reduce application response time by employing multi-objective offloading frameworks but without considering offloading failures. Edge Computing, due to distributed architecture that contains diverse resource and reliability characteristics, is prone to server and network failures that can postpone or prevent offloading thus affecting the overall system performance. In this study, we propose a novel solution to model the energy consumption of mobile device and application response time assuming the resource and reliability diversity of the Edge Computing system. The model adopts the Markov Decision Process (MDP), which provides a formal framework for capturing stochastic and non-deterministic behavior of Edge offloading. We propose the Energy Efficient and Failure Predictive Edge Offloading (EFPO) framework based on a model checking solution called Value Iteration Algorithm (VIA). EFPO determines the feasible offloading decision policy, which should yield a near-optimal system performance. Evaluation is performed by offloading various mobile applications modeled as Directed Acyclic Graphs (DAG). Failures are emulated from the failure trace data set from Pacific Northwest National Laboratory. Results show that the proposed EFPO framework yields better time performance between 12% - 57% and better energy efficiency between 15% - 51% when comparing to other offloading decision policies from the literature.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Computing methodologies** → *Modeling and simulation.*

KEYWORDS

edge offloading; offload decision engine; model checking

1 INTRODUCTION

Modern mobile applications offer an increasing amount of features to satisfy the demands of a rising number of users, ensure revenues and preserve their position in the market. However, a rising number of features increase also resource consumption on the mobile device. Regardless of the increased computational capabilities of mobile devices, it is still difficult for them to provide enough resources to mobile applications. The results are unacceptable application run times –especially in the case of latency-sensitive applications– and subsequently shorter battery recharging intervals. The Mobile Cloud Computing (MCC) paradigm emerged to overcome the aforementioned issues. MCC is based on the offloading concept. Mobile applications are partitioned into smaller tasks that can be offloaded to a Cloud data center to decrease the resource consumption on the mobile device by executing a task in a more computationally powerful surrogate machine. Research works such as [11, 17, 24, 30] proposed offloading frameworks with various architectures to achieve energy savings through offloading code to the Cloud. However, offloading a task to the Cloud data center can sometimes do more harm than benefit. The geographical distance between Cloud data centers and mobile devices can deteriorate application performance due to network latency and bandwidth.

Edge computing bridge this gap by moving smaller-scale data centers from the Cloud to the edge of the network. Thus, the task is offloaded to the most suitable nearby Edge node instead of a distant Cloud data center to reduce application run time and prolong battery time on the mobile device. The aforementioned works in MCC were an inspiration for many works in Mobile Edge Computing (MEC) as summarized in [21], to achieve energy savings and reduce execution delays. However, all these work assume the infrastructure to be failure-free, which is not a realistic assumption Edge Computing is prone to server and network failures due to heterogeneous resources and reliability, which can postpone or prevent offloading and affect overall system performance [6, 7]. Low reliability of the remote infrastructure reduces the Quality-of-Service (QoS) and degrades the end-user experience. Failures in mobile wireless environments are considered in [22, 29, 31] but none of them were adapted for Edge Computing.

The methodology that we used in our work is the Markov Decision Process (MDP). It is a suitable solution to capture the stochastic and non-deterministic behavior of Edge offloading. It is the system where failures are occurring stochastically while offloading decisions are non-deterministic due to uncertainty. Work [3] modeled offloading as Timed Automata that captures time-critical behavior to deliver performance guarantee. Other works as [5, 26] modeled offloading as MDP to capture the stochastic behavior of wireless fading channels and non-deterministic offloading decisions to deliver optimal performance. However, all servers and nodes in simulation models are assumed to be failure-free without considering offloading failures and their stochastic predictability. Our proposed solution is the Energy Efficient and Failure Predictive Edge Offloading (EFPO) framework, which adopts MDP as a formal framework that captures the stochastic behavior of failures and non-deterministic logic of offloading decisions to deliver efficient performance in the shape of optimal decision policy. This is achieved through a model checking solution called Value Iteration Algorithm (VIA) which includes objectives like energy and time by taking offloading failure probability into account. The advantage of the model checking approach is that it exhaustively and automatically explores the state space and yields an optimal solution.

Evaluation is performed by offloading various intensive mobile applications on network infrastructure, which contains diverse resource and reliability characteristics. Failures are emulated from the failure trace dataset, which is collected from the high-performance computing system at the Pacific Northwest National Laboratory (PNNL). Results show that the EFPO framework yields better time performance and energy efficiency between 12% - 57% and 15% - 51% respectively when compared to other offloading decision policies used in the evaluation.

The paper is organized as follows. The background about mobile applications, offloading decision engine, and formal verification is described in Section 2. In Section 3, we describe our EFPO framework. Section 4 provides the evaluation results. In Section 5 related work is discussed. Finally, Section 6 briefly mentions future work and concludes the paper.

2 BACKGROUND

2.1 Mobile Applications

Mobile applications consist of multiple tasks, each of which has different requirements for computational and data storage resources. Due to this partitioned nature, a mobile application can be modeled as a Directed Acyclic Graph (DAG) [3, 14]. A DAG consists of vertices and edges, where vertices represent the application tasks specified with resource requirements and edges represent task dependencies. DAG models are used when task execution order within the application is relevant, due to the input and output dependency of tasks to others. An example of DAG is shown in Figure 1. Facerecognizer application consist of five tasks, where blue circles are non-offloadable and white circles are offloadable ones. Non-offloadable tasks can not be offloaded since they depend on specific physical device functions (e.g. camera).

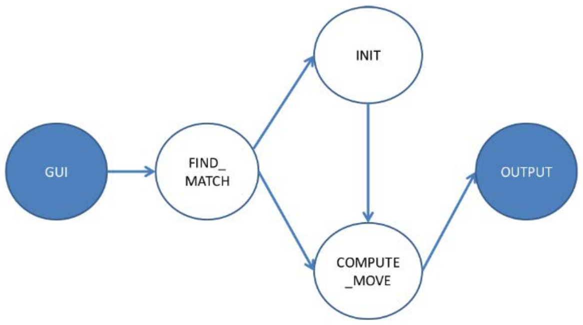

Figure 1: Facerecognizer mobile application [14]

2.2 Edge Offloading

With Edge offloading, a mobile application (or a part of it) is offloaded from a mobile device to remote network infrastructure. In Figure 2, we summarize the offloading process. We assume that offloading is performed by a software unit running on the mobile device called the offloading decision engine (ODE). ODE is responsible for offloading application tasks on remote servers. It decides on which Edge server or Cloud data center each application task shall be offloaded, taking into account applications' and infrastructures' resource requirements and capacities. Once offloading is completed, the infrastructure executes tasks and sends the results to the location where the next application task will be executed. This process repeats until the application terminates.

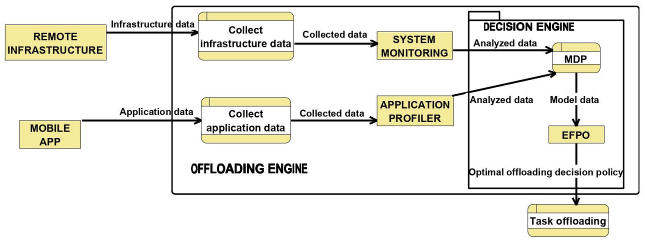

Figure 2: Edge offloading model as UML data flow diagram

Main components of offloading engine in Figure 2 are: (1) *Application profiler* that profiles and extracts DAG structure and identifies tasks requirements and dependencies; (2) *System monitoring* that monitors data about remote infrastructure, and (3) *Decision engine* that collects data from the other two and performs offloading decisions. In this work, we focus on the decision engine. We assume that the decision engine has already collected the application DAG model annotated with requirements, the offloading possibility for each task from the Application profiler, as well as the information about remote infrastructure from System monitoring. The entire system in the decision engine is viewed as an MDP model, and by verifying it, we obtain optimal decision policy that is being used in the EFPO algorithm to determine which application tasks will be offloaded on which offloading sites. MDP framework is used for modeling systems that exhibit probabilistic and non-deterministic behavior. Edge offloading fits to this scenario as offloading failures occur probabilistically and offloading decisions can be resolved in a non-deterministic manner. Executing ODE on the mobile device consumes energy that may be intolerable in some cases. The alternative is to execute ODE on a remote server and store the result on the device. In case of unstable connectivity, execution can continue on the device until the connection to the remote server is restored.

2.3 Formal Verification

Formal verification is the methodology for verifying the correctness and performance of the system, using formal methods of mathematics. A widely used technique is *model checking*, which consists of a systematical and exhaustive exploration of the mathematical model. It usually includes exploring all states and transitions in the model. One variant of model checking, which is used in this paper, is *quantitative model checking*, a mathematical technique for establishing the correctness, performance, and reliability of systems that exhibit stochastic behavior [19]. Performance (e.g. time response, energy consumption) and reliability of the system are the focus of our paper. These models are represented as labeled transition systems where each state represents some system configuration and transitions (actions) that define the behavior of the system. A general advantage of the model checking is that verification is automatic and exhaustive. However, in the case of larger systems, this may result in a state-space explosion and limit the number of formulas and properties that can be verified. Theorem proving, as an alternative approach to model checking, can work with more accurate representations of the system and express any property, but due to manual proofing, it requires more time and expertise.

3 OFFLOADING FRAMEWORK

3.1 MDP Formulation

MDP is a mathematical framework for modeling decision making in situations where outcomes are partly probabilistic and partly under the control of a decision-maker. This kind of framework can be used for modeling systems that exhibit probabilistic and non-deterministic behavior. Offloading decisions can be resolved in a non-deterministic manner as a *optimal decision policy*. An optimal decision policy describes the best action for each state in the MDP model, which yields optimal performance for the modeled system under given conditions. It can be obtained by formally verifying the MDP model using the model checking solution Value Iteration Algorithm (VIA). VIA algorithm focuses critically on expected value, in contrast to safety properties that are focused on the worst-case scenario. This allows us to exploit sampling and approximation more aggressively. There are other model checking solutions for MDP including Policy Iteration Algorithm, but VIA is preferred due to its theoretical simplicity and ease of implementation [23].

MDP is defined as a labeled transition system with state space S, where each state represents system configuration, action space A, where probabilistic transitions defines state trajectory from previous state s' to current state s and a reward function R which determines immediate reward (or cost) for taken action a while in state s. Therefore, MDP can be formally defined as a tuple $< S, A, P, R >$:

- S state space,
- A action set,
- $P(s, s', a)$ transition probability by taking action a in state s will lead to state s',
- $R(s, s', a)$ immediate reward received after transition from current state s to next state s' by taking action a.

The system architecture that we model with MDP is illustrated in Figure 3. It consists of five offloading sites, a single mobile device (MD), three Edge servers, and a single Cloud data center (CD). The scenario is that ODE is running on the mobile device and offloads the tasks from a current executed mobile application on Edge servers or Cloud data center. Alternatively, it is performed locally on the device. The mobile device has inferior computation and data storage resources comparing it with Edge and Cloud. Edge servers, on the other hand, have inferior resources when comparing to the Cloud data center. We introduce three Edge server types: (i) *Edge database server* (E1) has larger data storage capabilities and network transmission rates for faster data transfer for handling data-intensive applications such as Facerecognizer, (ii) *Edge computational server* (E2) has larger computational capabilities as CPU processing speed suitable for computational-intensive applications such as Chess, and (iii) *Edge regular server* (E3) has intermediate resources suitable more for typical applications that do not have large requirement for computation or data storage, such as social media applications. Mesh network topology is used in the architecture due to advantages such as system robustness in case of server or network failures. The system architecture is extendable for employing multiple instances of each aforementioned offloading site. MDP model checking solutions support scalability. This is an important feature which can cope with verifying larger system models. However, the price of larger system models is a larger state space that can cause state space explosion which disrupts the model checking process. In this study, we use five offloading site instances as illustrated in Figure 3.

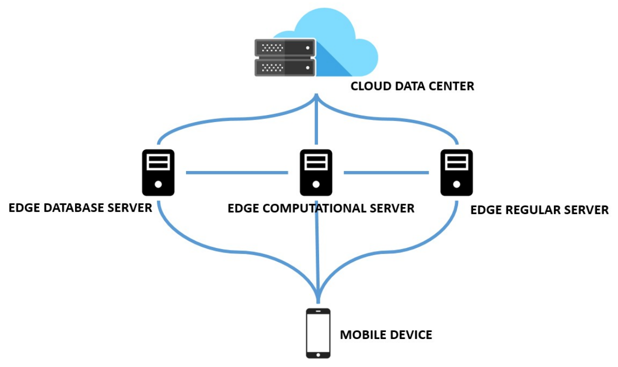

Figure 3: System architecture

Modeling of the mentioned system architecture as a MDP is as follows: (a) **The state space** S is defined as $S = \{MD, E1, E2, E3, CD\}$, where elements correspond to the aforementioned offloading sites, (b) **The action space** A is defined as $A = S$ where $a \in A$ represents offloading site decisions (c) **The discrete decision epochs** represents discrete time events when offloading actions are performed and defined as $T = \{0, 1, ..., n\}$, (d) **The transition probabilities** for each state $s(t)$ and action $a(t)$, gives quantitative information that the next state will be $s(t + 1)$, (e) **The reward function** used in this work considers two objectives, energy consumption and application response time, resulting in two reward functions, $R_e(s, a)$ and $R_t(s, a)$, which are combined in the overall reward function $R(s, s', a)$, (f) **Value iteration algorithm** (VIA) performs an approximation to the optimal value function for each state as in Equation (1) and yields optimal actions which represents our optimal offloading decision policy as shown in Equation (2). $\pi(s)$ represents offloading decision policy with initial state s and γ is discount factor $[0, 1]$ which guarantees algorithm output convergence.

$$V(s) = \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi\ (s)}(s, s') + \gamma V(s')) \quad (1)$$

$$\pi(s) = \underset{a}{\text{argmax}} \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi\ (s)}(s, s') + \gamma V(s')) \quad (2)$$

3.2 Offloading Model

Offloading sites has hardware characteristics based on which offloading and failure cost are computed. A site is defined as the vector $q = (f, ram, stor, r, l)$, where f is the CPU processing speed in millions of cycles per second, ram is the memory storage, $storage$ is the data storage capacity, r is the network bandwidth and l is the network latency. Application tasks are defined similarly, $v = (w, ram, d^{in}, d^{out}, off)$, where w is the CPU processing speed in millions of cycles per second, ram is the memory consumption, d^{in}

is the input data size, d^{out} is the output data size and off is the binary variable that indicates whether the task offloadable.

When offloading application tasks on remote infrastructure, resource constraints must be respected. Defining valid offloading, the following conditions must be satisfied:

- $\sum_{v \in O_q(t)} (\omega_v) \leq f_q$
- $\sum_{v \in O_q(t)} (ram_v) \leq ram_q$
- $\sum_{v \in O_q(t)} (d_v^{in} + d_v^{out}) \leq stor_q$
- $\phi(v) = \emptyset$

$O_q(t)$ represents a set of application tasks that are executed in time moment t on offloading site q and $\phi(v)$ represents a set of application tasks that are input dependencies for task v. First, three conditions, validate that CPU, RAM and data storage capacities of offloading site q are not exceeded. The last condition validates that all input dependable tasks are executed before task v is ready for offloading and execution. The notation used for our simulation model is listed in Table 1.

3.3 Time Response Model

Time response cost is defined as $T_v = \{t_{md}, t_{e1}, t_{e2}, t_{e3}, t_{cd}\}$, where each element denotes the time response cost to execute component v on each of the offloading sites. $f_{md}, f_{e1}, f_{e2}, f_{e3}$ and f_{cd} are defined as the CPU clock speeds (cycles/second) of offloading sites. The total CPU cycles needed to execute the application task v is ω_v and f_i denotes CPU frequency of q_i offloading site. $t_{v_i}^c$ denotes the computational time of executing task v on site q_i and defined as:

$$t_{v_i}^c = \frac{\omega_v}{f_i}, \forall v \in V, \forall i \in [0, k] \tag{3}$$

Input and output data of task v are denoted as d_v^{in} and d_v^{out}, respectively. Also, t_{vi}^{in} and t_{vi}^{out} are defined as the communication time spent for input and output data transmission between q_i and q_j offloading sites, given by both equations:

$$t_{v_i}^{in} = \frac{d_v^{in}}{r_{ij}} + l_{ij}, \forall v \in V, \forall i, j \in [0, k], i \neq j \tag{4}$$

$$t_{v_i}^{out} = \frac{d_v^{out}}{r_{ij}} + l_{ij}, \forall v \in V, \forall i, j \in [0, k], i \neq j \tag{5}$$

r_{ij} and l_{ij} represents bandwidth and latency between offloading sites q_i and q_j respectively. t_{v_i} is the total time cost of task v to be executed on site q_i and it is defined as:

$$t_{v_i} = t_{v_i}^c + t_{v_i}^{in} + t_{v_i}^{out} \tag{6}$$

In case that successive tasks are executed on the same offloading site then according to Equation (6), total time cost is $t_{v_i} = t_{v_i}^c$, without data transmission costs from Equations (4) and (5) .

3.4 Energy Consumption Model

Energy consumption cost is defined as $E_v = \{e_{md}, e_{e1}, e_{e2}, e_{e3}, e_{cd}\}$, where each element denotes the energy cost to execute task v on the offloading sites. Energy consumption is considered only from mobile device perspective. Supplies on infrastructure are perceived as unlimited. It is assumed that the energy consumption e_v is computed as the amount of energy a mobile device spends while executing the application task or waiting for the application task to be executed on remote offloading sites. Energy consumption of a task v is then defined by e_{v_i} in Equation (7) where p_c is the mobile power consumption for local computation, p_d is the mobile power consumption when downloading data, p_u is the mobile power consumption when uploading data, and p_{idle} is the mobile power consumption in idle mode when application task is executed on remote infrastructure (Edge or Cloud).

$$e_{v_i} = \begin{cases} t_{v_i}^c \times p_c + t_{v_i}^{in} \times p_d + t_{v_i}^{out} \times p_u \\ t_{v_i} \times p_{idle} \end{cases} \tag{7}$$

The first case considers offloading from the mobile device, and the second when task is migrated on the remote infrastructure. Assumption about mobile power parameters when computing total energy consumption cost is considered as $p_u > p_d > p_c > p_{idle}$ from [18], where transmission consumes more energy then local computation or idle mode.

3.5 Reward Functions

Reward functions are used to model utilities or objectives which we want maximize or minimize through state sequence sampling. We define the overall reward function $R(s, a)$ which contains reward function for energy consumption $R_e(s, a)$ and reward function for application response time $R_t(s, a)$:

$$R(s, a) = \omega_e \times R_e(s, a) + \omega_t \times R_t(s, a) \tag{8}$$

ω_e and ω_t are defined as the weight factors for energy consumption and response time. The weight factors constraints are given as $\sum_m \omega_m = 1$ where $m = \{e, t\}$ represents objectives such that $0 \leq \omega_e \leq 1$ and $0 \leq \omega_t \leq 1$. Both aforementioned reward functions are defined as follows:

$$R_e(s, a) = \frac{1}{1 + e^{e_v}} \tag{9}$$

$$R_t(s, a) = \frac{1}{1 + e^{t_v}} \tag{10}$$

In the evaluation, we used $\omega_e = \omega_t = 0.5$ which gives equal importance to both objectives. Weight factors can be altered to optimize the trade-off but this is out of this work's scope.

3.6 Failure Model

Failures can occur on Edge and Cloud infrastructure on the server or the network level. Server failures can be hardware faults (aging factor, power outage, hard disk failure, etc.) and software faults (OS failure, application crash, etc.), while network failures occur on network physical connections or network interface. Both failure types in this study are considered as an offloading failure.

Offloading failure occurrence in the simulation model is considered as a failure event $\lambda_{q_i}(t)$, which can occur in any discrete-time epoch t and offloading site q_i except mobile device which is considered as failure-free. Use case scenario of our most interest is that task offloading is performed on offloading site q_i at the same time moment t when failure event $\lambda_{q_i}(t)$ occurred. This interrupts offloading and execution process and causes additional cost in energy and time as well as forcing ODE to select other offloading sites q_j.

Table 1: Simulation parameters

Simulation parameters		
Offloading sites	q_i	i-th offloading site
Energy parameters	e_{v_i}	Energy cost of task v if executed on offloading site q_i
Time parameters	t_{v_i}	Total time cost of task v if executed on offloading site q_i
	$t^c_{v_i}$	Computational time spent executing task v on site q_i
	$t^i_{v_i}$	Time spent receiving input data to the task v on site q_i
	$t^o_{v_i}$	Time spent sending output data to the task v on site q_i
Hardware parameters	f_i	CPU clock speed (cycles/second) of offloading site q_i
	w_v	Total CPU cycles needed by the instructions of task v
	ram_q	RAM memory storage of offloading site q
	$stor_q$	Data storage capacity of offloading site q
Data parameters	d^{in}_v	Input data received by task v
	d^{out}_v	Output data sent by a task v
Power parameters	p_u	Mobile power consumption for uplink transmission
	p_d	Mobile power consumption for downlink transmission
	p_e	Mobile power consumption for local execution
	p_{idle}	Mobile power consumption at idle
Network parameters	r_{ij}	Network bandwidth rate between site q_i and q_j
	l_{ij}	Network latency between site q_i and q_j
Weight factors	ω_e	Weight factor for energy consumption reward function $R_e(s,a)$
	ω_t	Weight factor for time response reward function $R_t(s,a)$
Failure parameters	λ_{q_i}	Binary flag that indicates did failure occured on offloading site q_i
	$MTBF$	Mean time between failures
	$c^t_{v_i}$	Failure time cost on the offloading site q_i where task v is offloaded
	$c^e_{v_i}$	Failure energy cost on the offloading site q_i where task v is offloaded

Offloading failure costs for time is defined as:

$$c^t_{v_i} = \begin{cases} t^{in}_{v_i} \\ t^{in}_{v_i} + t^c_{v_i} \\ t^{in}_{v_i} + t^c_{v_i} + t^{out}_{v_i} \\ 0 \end{cases} \tag{11}$$

The first case is when a failure occurs during input data transmission, the second case is when a failure occurs during execution, thus, input data transmission and computation time cost are included, thirdly, failure occurs during output data transmission, thus, all three time cost components are included. Finally, the last case is when there are no failures observed. Next, the offloading failure cost for energy is defined as:

$$c^e_{v_i} = \begin{cases} t^{in}_{v_i} \times p_{idle} \\ (t^{in}_{v_i} + t^c_{v_i}) \times p_{idle} \\ (t^{in}_{v_i} + t^c_{v_i} + t^{out}_{v_i}) \times p_{idle} \\ 0 \end{cases} \tag{12}$$

Cases are the same as in the previous equation. All time components are multiplied with p_{idle} since all failures are occurring only on remote infrastructure and during that period failure-free mobile device is in idle mode. Simulating failure events $\lambda_{q_i}(t)$ is done by Poisson distribution similar to [29]. As a rate parameter, we use Mean Time Between Failures (MTBF). It is a measure that gives quantified information about product reliability, defined as:

$$MTBF = \frac{T}{R} \tag{13}$$

T denotes total time and R number of failures. MTBF can be expressed in hours, days or any other time unit. The longer the MTBF, the product reliability is higher. It is an opposite measure of failure rates. Using it as a rate parameter in Poisson distribution, we obtain the number of discrete epoch events until failure event $\lambda_{q_i}(t)$ occurs on offloading site q_i. The final issue is predicting failure events. Failure predictability is defined as probability estimation:

$$P(t) = 1 - e^{-t/MTBF} \tag{14}$$

3.7 EFPO Algorithm

Algorithm 1 shows the EFPO algorithm for obtaining an energy-efficient offloading decision policy with failure predictability. The algorithm obtains an optimal policy from the VIA algorithm by exploring every state in the state space and selects the action with the lowest energy and time cost to be the optimal action. It performs this operation until it finds a feasible action that can be performed on the offloading site that did not experience offloading failure. After exploring the state space, the EFPO algorithm determines the feasible offloading decision policy that is then used to make an efficient decision for every future state the system encounters. EFPO algorithm is illustrated in Algorithm 1.

In line 1 we obtain optimal offloading decision policy from the VIA algorithm. However, optimal actions from VIA does not guarantee that they are feasible due to failures that happen during the runtime. We need to iterate all states in the MDP state space to obtain feasible optimal actions. In line 5, $\lambda_{T(s,a)}$ is a boolean variable which indicates whether offloading failure occurred on the

Algorithm 1 Energy Efficient and Failure Predictive Edge Offloading Algorithm

```
1:  $< \pi^*, Q > \leftarrow VIA(S, A, P, R, s_0)$        ▷ VIA algorithm returns optimal decision policy
2:  for each state $s$ in $S$ do
3:      $a \leftarrow \pi^*(s)$        ▷ for state $s$ get optimal action $a$
4:      while True do
5:          if $\lambda_{T(s,a)}$ then        ▷ if offloading failure occurs then another $a$ action should be considered
6:              $Q \leftarrow Q - \{(s, a)\}$
7:              if $Q = \emptyset$ then return "No feasible solution"
8:              end if
9:              $a \leftarrow \text{argmax}_a[Q(s, a)]$        ▷ get next best action $a$
10:             continue
11:         else
12:             $\omega = \omega + \{(s, a)\}$        ▷ store feasible action $a$
13:             break
14:         end if
15:     end while
16: end for
17: return $\omega$        ▷ return feasible offloading policy
```

offloading site $T(s, a)$ or not. If it is true, then we consider other offloading sites from vector Q which contains all action-state values returned from the VIA algorithm. Based on those values, we obtain the next action a. The algorithm continues to iterate until it finds an action that is feasible to offload on site which did not experience failure. Otherwise, the algorithm terminates on line 7. When a feasible action is found, it is stored in ω vector in line 12 and returned in line 17.

The EFPO algorithm finds the efficient offloading decision with an algorithmic complexity of $O(SA)$ per task offloading, where S is the state space and A is the action set. This algorithmic complexity does not reflect the complexity of the VIA algorithm. Although the EFPO algorithm can be considered to be a computationally expensive operation for resource-limited mobile devices, an alternative can be made so that the feasible offloading decision policy is performed by the remote server. Therefore, mobile devices only store the matrix form of the results.

4 EVALUATION

4.1 Experimental Setup

At the time we write, there are several state-of-the-art model checker tools available such as UPPAAL [8] and PRISM [20]. UPPAAL verifies non-deterministic and time-critical systems. Moreover, UPPAAL Statistical Model Checking (SMC) [13] extension supports modeling and verification of the systems which exhibit both probabilistic and timed behavior. Nevertheless, the tool does not support MDP models. PRISM, on the other hand, is a tool for formal modeling and analysis of systems that exhibit random or probabilistic behavior. MDP modeling and verification are supported but PRISM modeling language requires that every aspect of the system, including simulation utilities (e.g. DAG models, Poisson distribution, parsing dataset, etc.) must be abstracted as state machines, which limit the expressiveness of our simulation. This can also cause time overhead in the verification process, for instance, the state space explosion. For this reason, we use our simulation framework implemented in Python which provides relatively simple syntax, diverse mathematical sampling distributions available as simple application programming interface (API) calls and MDP solver toolbox [9] for MDP modeling and verification. This framework supports DAG mobile application scheduling, energy consumption, time response, offloading failure models, simulation and distribution of failures, and Edge/Cloud infrastructure model. Moreover, it can be expended with other objectives due to modular architecture. The input of our simulation framework is the infrastructure model which includes hardware specifications of the computational nodes, network characteristics, and mobile application setup.

The evaluation scenario is as follows. ODE on the mobile device decides on which offloading site shall application task be offloaded and executed. Energy consumption and response time are affected by hardware characteristics of the site as well as network links between the sites. It is assumed that only data is offloaded from site to site, while computation is replicated on each of them. Efficient selection of replica sites in a large network is left as future work. Here, we only consider a local partition of the system. Additionally, failures on sites where tasks are offloaded prolong time and energy and ODE is forced to offload tasks on another site possibly without failure. Energy and time cost after offloading failure are defined by Equations (11) and (12). Failure can occur on links and servers. Depending on which part failure occurred, energy failure cost $c^t_{v_i}$ and time failure cost $c^e_{v_i}$ are computed accordingly.

4.1.1 *Computational nodes.*

The infrastructure model used in our simulation includes five computational nodes. We have a mobile device, three Edge servers, and a Cloud data center. The mobile device is the start and endpoint of any mobile application execution. Resources are very limited when comparing to the hardware specifications of Edge servers and Cloud data centers. We assume that the hardware specifications of the aforementioned nodes do not change during the runtime. The contemporary CPU processing power of the mobile device is typical between 1.8 - 2.2 GHz but everything above 1 GHz is acceptable [2]. Cloud data center CPU processing power with modern technological achievements can be boosted with 56 cores with a base frequency of 2.6 GHz and turbo 3.8 GHz [1]. Using 20 GHz in the simulation is to reflect the computational superiority of the Cloud server compared to other counterparts (mobile and Edge). Values from Table 2 are selected as moderate to reflect the magnitude of the computational power ratio between complementary parts of the network. Also, Edge and Cloud servers due to unreliability (server and network failures) must have larger resource capacities to stay competitive.

Concerning the mobile device, we need to consider an energy consumption model. The parameters used in energy model are p_u = 1.3W, p_d = 1.0W, p_c = 0.9W, p_{idle} = 0.3W where condition is assumed $p_u > p_d > p_c > p_{idle}$ as in [18] and used in Equations (7) and (12).

4.1.2 *Network Infrastructure.*

Network parameters that can influence offloading results are network latency and bandwidth. Network latency is the amount of time that takes the data to transmit between two points which is dependable on physical distance. This fits our model since the Cloud data center is geographically more distanced from the end-user which scale up the network latency.

Table 2: Hardware specifications

Node	CPU (GHz)	RAM (GB)	Storage (GB)
Edge database server	5	8	500
Edge computational server	8	8	250
Edge regular server	5	8	250
Cloud data center	20	128	1000
Mobile device	1	8	16

The latency on wireless links between the mobile device and Edge servers should be less due to geographical proximity. Bandwidth, on the other hand, is the rate of data transfer between the two points. This fits the DAG mobile application model where task dependency between the tasks is achieved based on input and output data transmission via network links. Overview of network latency and bandwidth distribution is shown in Table 3. Latency distribution is similar to work [14], while bandwidth for wireless links (first and the second row of the table) are actual speed limits of IEEE 802.11 wireless standard (802.11a, 802.11b, 802.11g) and for fixed network links (third and fourth row of the table) are Fast Ethernet and 1Gbit Ethernet link standards. Mentioned wireless standards are used since speed is lower when comparing to other IEEE 802.11 wireless standards which could yield to higher price and operational cost. For a fixed network, we selected Ethernet links since we are assuming that Edge servers will be localized near each other. Higher bandwidth values from the Table 3 are associated with Edge database server network connection characteristics due to high demand in data transmission.

Table 3: Network specifications

Links		Latency (ms)	Bandwidth (Mbps)
Mobile	Edge	15	5.5/20
Mobile	Cloud	$54 + \varphi(\mu, \sigma)$	20
Edge	Cloud	$15 + \varphi(\mu, \sigma)$	100/987
Edge	Edge	10	100/987

Network links between Edge servers and the Cloud data center are much faster when compared to wireless links. It is a reasonable assumption since it is well known that data transmission on telecommunication or Internet network is highly demanding, thus larger bandwidth rates should be provided, similar to the assumption in [26]. $\varphi(\mu, \sigma)$ function models Internet latency on Cloud data center due to transmission delay that according to [16] is estimated to be between 100 and 300ms. It is modeled by employing a Gaussian distribution with mean $\mu = 200$ and standard deviation $\sigma = 33.5$ to obtain the values in the aforementioned range.

4.1.3 Mobile application setup. We use DAG models of mobile applications as described in works [14, 15]. These applications are suitable for our use case scenarios since we are more interested in more typical and commercialized applications that will be more probable used by the average user. The DAG structure used for this paper comes from the description of each application in the aforementioned works. We select three applications: (i) *Facebook*, that models the behaviour of posting pictures on Facebook, which represents typical mobile application, (ii) *Facerecognizer*, models the image processing application which recognizes face on the picture, and represents data-intensive application due to large database of face images and (iii) *Chess*, that models the behavior of chess game between the user and AI software and represents computational-intensive mobile application due to large and complex computations for anticipating next game moves.

Denoting mobile application as typical, data or computational-intensive, does not imply that all tasks in the application are of the same intensity. Table 4 shows application task sizes in terms of CPU, input and output data size. CI and DI refer to as computational-intensive and data-intensive respectively. Moderate stands for application tasks that do not have emphasized computational or data components. Similar application task distribution is used in work [26]. Task specifications of aforementioned mobile applications that are used in this experiment are listed in Tables 5, 6 and 7.

Table 4: Application task specifications

Type	CPU	Input data	Output data
DI	100-200 M cycles	15-20 KB	25-30 KB
CI	550-650 M cycles	4-8 KB	4-8 KB
Moderate	100-200 M cycles	4-8 KB	4-8 KB

Table 5: Facebook task specifications

Task	Type	RAM	Offloadable
FACEBOOK_GUI	Moderate	1 GB	False
GET_TOKEN	Moderate	1 GB	True
POST_REQUEST	Moderate	2 GB	True
PROCESS_RESPONSE	Moderate	2 GB	True
FILE_UPLOAD	DI	2 GB	False
APPLY_FILTER	DI	2 GB	True
FACEBOOK_POST	DI	2 GB	False
OUTPUT	Moderate	1 GB	False

Table 6: Facercognizer task specifications

Task	Type	RAM	Offloadable
GUI	DI	1 GB	False
FIND_MATCH	DI	1 GB	True
INIT	DI	1 GB	True
DETECT_FACE	DI	1 GB	True
OUTPUT	DI	1 GB	False

4.1.4 Failure dataset. Failure dataset is vital for MTBF computing for simulating failures through sampling via Poisson distribution (as explained in subsection 3.6) and probability estimation of failures that are encoded in the probability matrix of the MDP model. There does not exist an Edge Computing failure dataset that is available for scientific research at present due to the novelty of

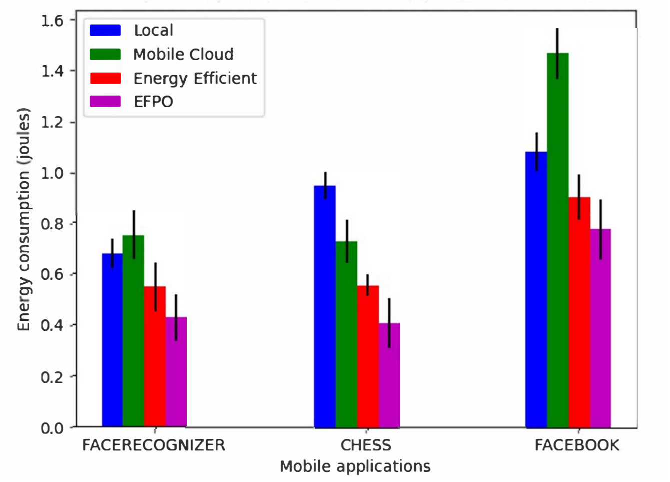

Figure 4: Energy consumption with different applications.

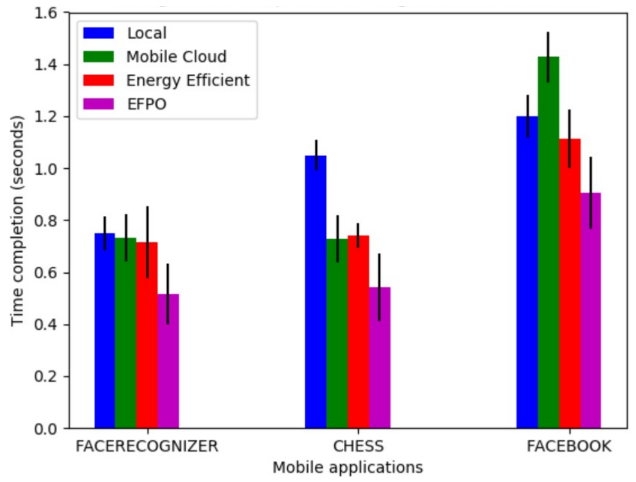

Figure 5: Response time with different applications.

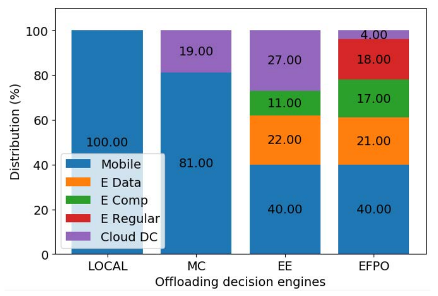

Figure 6: Offloading distribution with Facerecognizer application.

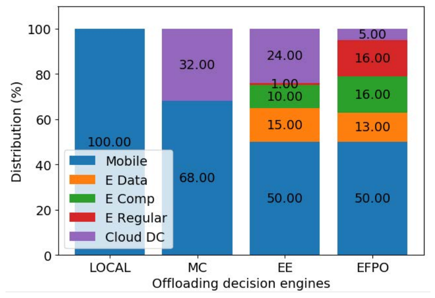

Figure 7: Offloading distribution with Chess application.

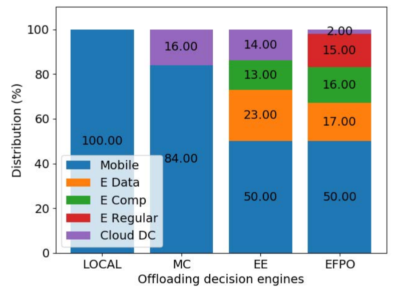

Figure 8: Offloading distribution with Facebook application.

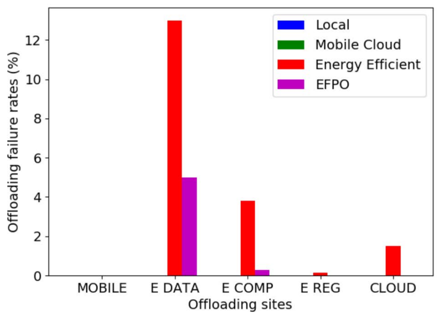

Figure 9: Offloading failure rates with Facerecognizer application.

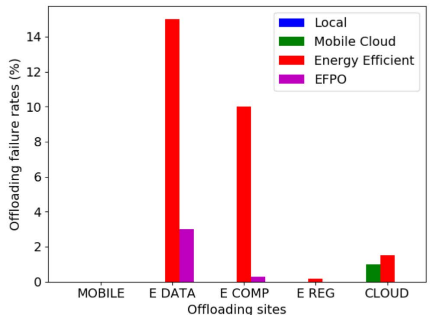

Figure 10: Offloading failure rates with Chess application.

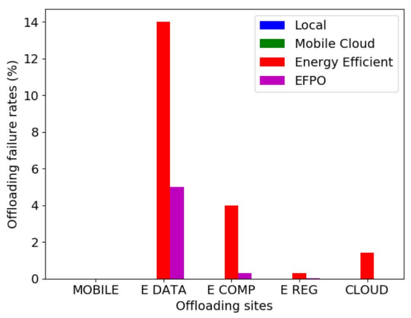

Figure 11: Offloading failure rates with Facebook application.

technology in the field. Consequently and similar to the previous work [7], we adopt failure traces from other real-world distributed systems to the edge computing scenario. Our simulation divides and maps real-world failure traces into simulation nodes that have distinctive characteristics as depicted in Figure 3. Failure dataset is needed for failure simulation and computing transition probabilities that are used for failure predictability in the EFPO algorithm. Dataset is made publicly available by Pacific Northwest National Laboratory (PNNL). Although the PNNL dataset is not collected on an Edge infrastructure, it possesses certain properties that suit our evaluation scenario. The number of computational nodes is large, they are distributed in different geographical locations, and they contain different hardware characteristics.

The dataset contains 4652 failure logs between 2003-2007. Failure logs are collected from the HPC (High-Performance Computing)

Table 7: Chess task specifications

Task	Type	RAM	Offloadable
GUI	Moderate	1 GB	No
UPDATE_CHESS	Moderate	1 GB	Yes
COMPUTE_MOVE	CI	2 GB	Yes
OUTPUT	Moderate	1 GB	No

system that consists of 980 computational nodes. Nodes are classified into several categories according to hardware characteristics. Categories are (i) *fat node*, they are 570 of them, where each contains 430 GB local disk and 10 GB RAM, (ii) *thin node*, 378 nodes, where each contains 10GB local disk and 10 GB RAM, and (iii) *Lustre servers*, they are 34 of them. Every node uses Itanium-2 processor 1.5 GHz and all are interconnected with Quadrics QsNetll.

Simulated failures are applied in the simulation model on all nodes except mobile devices, which is assumed to be failure-free. Before simulating failures, we need to map the failures of node categories from the dataset into the nodes of our simulation model. Fat nodes are suitable for Edge database server due to larger local disk capacity, Lustre servers are considered as Cloud data centers, where due to high performances are used in Cluster computing, and thin nodes are divided between Edge computational and regular server. A small portion of thin nodes is test and login nodes that have only a few failures. Those node failures are mapped on a regular server while the majority of thin nodes are mapped to a computational server. This setup gives regular servers more reliability than a computational server. With this setting, we want to explore how EFPO performs in a scenario where we have resourceful servers that are less reliable with reliable servers that are less resourceful. Another scenario is where resourceful servers are more reliable, but the EFPO algorithm could have similar performance as other state-of-the-art decision engines since offloading failures are occurring much less on resourceful servers which are more attractive for offloading. Concerning failures, server failures are identified by hardware identifier which is easy to map it on the particular nodes. Network failures, on the other hand, cannot be mapped since they do not contain information on which nodes they are connected to. Thus, network failures are distributed to node categories in proportion to the frequency of failures.

4.2 Evaluation Results

Besides the mentioned mobile applications that are used in the experiment, we implemented three additional ODEs to compare performance with EFPO. These are (i) *Local*, which considers only mobile device as an execution site, (ii) *Mobile Cloud* (MC), which considers only mobile device and Cloud data center, and (iii) *Energy Efficient* (EE), which considers offloading application tasks on all offloading sites but without considering failure probability. Considering the reliability, after mapping failures from the PNNL dataset to simulation nodes as explained in 4.1.4, the Edge database and computational server are less reliable then Edge regular and Cloud. The main goal here is to evaluate, whether EFPO boosts the system performance by offloading application tasks on more reliable servers in certain periods by mitigating offloading failures on more resourceful servers. This can extend the execution time but it is still less harmful than offloading failures.

Figures 4 and 5 show energy consumption and response time per ODE for a single mobile application execution along with the standard deviation. Single mobile application execution is sampled 100,000 times that gives validity and statistical significance to our experimental results. We also consider an experiment, where we have successive mobile application executions, but increasing the number of application executions linearly increases both energy and time. Deviation in application executions is measured and detected but does not change the conclusion of the results. This justifies that sampling a single mobile application execution is sufficient for the evaluation and less time consuming. In both figures, the EFPO algorithm outperforms all other ODE engines in all three mobile application cases. EE engine does not yield better performance since it does not contain failure predictability feature, which is shown in Figures 9, 10 and 11 where offloading failure rates are the highest. EE engine always prefers those sites that have superior resources capacities without considering failure probabilities. Thus, as shown in Figures 6, 7 and 8, EE considers Cloud as most attractive and Edge regular server as less attractive offloading site. Consequently, this yields bias in task offloading towards those sites which are resource superior but less reliable, which leads to more frequent offloading failures and increased energy consumption as well as response time. Edge regular server, on the other hand, is more attractive for EFPO due to low failure probability and forces offloading distribution to utilize Edge servers more frequently to exploit the advantages of Edge Computing in lower network latency and better network bandwidth. EFPO utilizes Edge servers in 56%, 45% and 48% of task offloading cases with Facerecognizer, Chess and Facebook application respectively. Non-offloadable tasks are executed on the mobile device, where 4%, 5%, and 2% end up in the Cloud data center in Facerecognizer, Chess and Facebook, respectively.

Local ODE outperforms MC ODE in terms of energy, in Facerecognizer and Facebook application cases, while for time response, only in the Facebook case as shown in Figures 4 and 5. This is due to high latency between mobile device and Cloud (geographical distance and Internet transmission delay) and the fact that the majority of the application tasks are DI requiring more expensive data transmissions due to larger input and output data sizes. However, in Chess application case, MC ODE outperforms Local since Chess contains CI application task *COMPUTE_MOVE* where Cloud is more suitable due to superior computational capacity and less expensive data transmission for small input and output data size. Cloud data center utilization in Chess application case for MC ODE is 32% (Figure 7), while in Facerecognizer (Figure 6) and Facebook (Figure 8) cases are 19% and 16%, respectively. However, the majority of applications tasks are executed on the mobile device. In Chess case, two out four application tasks are non-offloadable which explains the high distribution of task execution on the device. In the other two application cases, besides a high proportion of non-offloadable tasks, the majority of tasks are more data expensive due to larger input and output data size, which causes the majority of tasks to remain on the device. The mobile device is the site where the majority of application tasks are distributed, from 40% for EE and EFPO ODEs in the Facerecognizer application case up to 100% for all three application cases when Local ODE is performing.

5 RELATED WORK

Offloading was considered as a suitable solution for tackling energy efficiency and application response time issues as summarized in [4, 28] for MCC infrastructure. Most of those works introduced computation offloading frameworks and multi-objective decision-making algorithms. Similarly, in survey work [21] for MEC, a lot of literature work about offloading frameworks and architectures are systemized to overcome the offloading limitations where offloading decision-making, computation resource allocation, and mobility management are addressed as key areas. Currently, some researchers in the Edge Computing area are coping with offloading challenges through multi-objective optimization algorithms as [14, 15] inspired by offloading frameworks in MCC as [10, 11, 17] where energy consumption, application run time and/or monetary costs are considered as primary objectives. None of these works considers the effect of offloading failures on systems' performance.

Research works about fault-tolerant offloading systems that exist for mobile wireless environments such as [22, 29] using M/M/1 queue model and checkpointing mechanism respectively. Work as [27] performs a trade-off between local re-execution and offloading on remote infrastructure in case of offloading failure using the timeout mechanism, while work [25] considers recovery mechanism by finding alternative paths via ad-hoc relay nodes through Fyold-Warshall algorithm in case of offloading failure occurrence on the shortest path. None of this works provides formal verification of performance and reliability for Edge Computing. A desirable solution for achieving both goals is formal verification. Work [31] was using the MDP algorithm for obtaining optimal offloading decision policy in a wireless mobile environment where uncertainty in wireless connections and user mobility can cause offloading failures. This work was adopted for Cloudlet systems. There exist works [26] and [5] for MCC and Edge Computing, which use the MDP algorithm to obtain optimal offloading decision policy but without considering offloading failures. Also, the MDP reward optimization technique is used for Edge/Cloud offloading but in the context of data stream analytics [12].

6 CONCLUSION AND FUTURE WORK

In this work, we show that Edge offloading failures can impact the energy consumption of the mobile device and the time response of mobile applications dramatically. Unreliability expressed as failure probability can be used for the probability estimation of an offloading failure for certain offloading sites. Based on that, we proposed the EFPO framework with a failure predictability feature that comes in great benefit to mitigate offloading failures to boost system performance. Failure predictability is combined with objective functions as energy consumption of the mobile device and application time response. MDP is used as a formal modeling framework to construct the EFPO framework, which consists of states to represent offloading sites and actions that represent offloading decisions. Value Iteration Algorithm as a model checking solution is used to obtain optimal offloading decision policy that consists of a set of optimal actions for certain offloading sites. However, the optimal policy may not be feasible due to offloading failures. Obtained a feasible offloading decision policy contains offloading sites that did not experience failure in the current discrete epoch. For evaluation purposes, we used Facerecognizer, Chess and Facebook mobile applications modeled as DAGs. Failures are simulated through Poisson distribution as well as MTBF values obtained from the PNNL failure dataset. This dataset fits our model due to resource heterogeneity and distributed architecture. Our solution outperforms all ODE engines, namely EE, MC, and Local ODE both in energy and time aspects. As future work, we will focus our research activities in provisioning Edge resources for application replicas. Several application replicas should be positioned on determined offloading sites, which yields better reliability and performance.

ACKNOWLEDGEMENTS

This work is partially funded through the Rucon project (Runtime Control in Multi Clouds), FWF Y 904 START-Programm 2015.

REFERENCES

[1] [n.d.]. Intel's new assault on the data center: 56-core Xeons, 10nm FPGAs, 100gig Ethernet. https://arstechnica.com/gadgets/2019/04/intels-new-assault-on-the-data-center-56-core-xeons-10nm-fpgas-100gig-ethernet/. Accessed: 2019-09-05.
[2] [n.d.]. The Specs That Really Count When Buying a Phone. https://smartphones.gadgethacks.com/how-to/specs-really-count-when-buying-phone-0171678/. Accessed: 2019-09-05.
[3] Luca Aceto et al. 2015. Decision support for MCC applications via model checking. In *IEEE Int'l. Conf. on Mobile Cloud Computing, Services, and Engineering*. 199–204.
[4] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. 2018. Mobile cloud computing for computation offloading. *Applied Comp. and Inf.* 14, 1 (2018), 1–16.
[5] Khalid R Alasmari, Robert C Green, and Mansoor Alam. 2018. Mobile edge offloading using MDP. In *Int'l. Conf. on Edge Computing*. Springer, 80–90.
[6] Atakan Aral and Ivona Brandic. 2017. Quality of service channelling for latency sensitive edge applications. In *IEEE Int'l. Conference on Edge Computing*. 166–173.
[7] Atakan Aral and Ivona Brandic. 2018. Dependency mining for service resilience at the edge. In *IEEE/ACM Symposium on Edge Computing (SEC)*. 228–242.
[8] Gerd Behrmann et al. 2006. *A tutorial on UPPAAL 4.0.* Technical Report. Department of computer science, Aalborg university.
[9] Iadine Chadès, Guillaume Chapron, et al. 2014. MDPtoolbox: a multi-platform toolbox to solve stoch. dyn. prog. problems. *Ecography* 37, 9 (2014), 916–920.
[10] Byung-Gon Chun et al. 2011. Clonecloud: elastic execution between mobile device and cloud. In *ACM Conference on Computer systems*. 301–314.
[11] Eduardo Cuervo et al. 2010. MAUI: making smartphones last longer with code offload. In *Int'l. Conf. on Mobile Systems, Applications, and Services*. ACM, 49–62.
[12] da Silva Veith et al. 2019. Multi-Objective Reinforcement Learning for Reconfiguring Data Stream Analytics on Edge Computing. In *International Conference on Parallel Processing*. 106.
[13] Alexandre David, Kim G Larsen, et al. 2015. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer* 17, 4 (2015), 397–415.
[14] Vincenzo De Maio and Ivona Brandic. 2018. First hop mobile offloading of dag computations. In *IEEE/ACM Int'l. Symp. on Cluster, Cloud and Grid Comp.* 83–92.
[15] Vincenzo De Maio and Ivona Brandic. 2019. Multi-Objective Mobile Edge Provisioning in Small Cell Clouds. In *ACM/SPEC Int'l. Conf. on Perf. Eng.* 127–138.
[16] Mark DeVirgilio, W David Pan, et al. 2013. Internet delay statistics: Measuring internet feel using a dichotomous hurst parameter. In *IEEE Southeastcon*. 1–6.
[17] Sokol Kosta, Andrius Aucinas, et al. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offl.. In *IEEE Infocom*. 945–953.
[18] Karthik Kumar and Yung-Hsiang Lu. 2010. Cloud computing for mobile users: Can offloading computation save energy? *Computer* 4 (2010), 51–56.
[19] Marta Kwiatkowska. 2007. Quantitative verification: Models, techniques and tools. In *ACM SIGSOFT Symp. on the Foundations of Software Engineering*. 449–458.
[20] M. Kwiatkowska et al. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *International Conference on Computer Aided Verification*. 585–591.
[21] Pavel Mach and Zdenek Becvar. 2017. Mobile edge computing: A survey on architecture and computation offloading. *arXiv preprint arXiv:1702.05309* (2017).
[22] Shumao Ou, Yumin Wu, Kun Yang, and Bosheng Zhou. 2008. Performance analysis of fault-tolerant offloading systems for pervasive services in mobile wireless environments. In *IEEE Int'l. Conf. on Communications*. 1856–1860.
[23] Martin L Puterman. 2014. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.
[24] Hao Qian and Daniel Andresen. 2015. Jade: Reducing energy consumption of android app. *Int'l. J. of Networked and Distributed Computing* 3, 3 (2015), 150–158.
[25] Dimas Satria, Daihee Park, and Minho Jo. 2017. Recovery for overloaded mobile edge computing. *Future Generation Computer Systems* 70 (2017), 138–147.

[26] Mati B Terefe, Heezin Lee, et al. 2016. Energy-efficient multisite offloading policy using MDP for MCC. *Pervasive and Mobile Computing* 27 (2016), 75–89.

[27] Qiushi Wang, Huaming Wu, and Katinka Wolter. 2013. Model-based performance analysis of local re-execution scheme in offloading system. In *IEEE/IFIP International Conference on Dependable Systems and Networks*. 1–6.

[28] Huaming Wu. 2018. Multi-objective decision-making for mobile cloud offloading: A survey. *IEEE Access* 6 (2018), 3962–3976.

[29] Huaming Wu. 2018. Performance Modeling of Delayed Offloading in Mobile Wireless Env. With Failures. *IEEE Comm. Letters* 22, 11 (2018), 2334–2337.

[30] Feng Xia et al. 2014. Phone2Cloud: Exploiting computation offloading for energy saving on smartphones in MCC. *Inf. Systems Frontiers* 16, 1 (2014), 95–111.

[31] Yang Zhang, Dusit Niyato, and Ping Wang. 2015. Offloading in mobile cloudlet systems with intermittent connectivity. *IEEE Transactions on Mobile Computing* 14, 12 (2015), 2516–2529.