A cost/reward method for optimal infinite scheduling in Mobile Cloud Computing

Luca Aceto[1], Kim G. Larsen[2], Andrea Morichetta[3], and Francesco Tiezzi[4]

[1] Reykjavik University, Iceland
[2] Aalborg University, Denmark
[3] IMT Institute for Advanced Studies Lucca, Italy
[4] Università di Camerino, Italy

Abstract. Computation offloading is a key concept in Mobile Cloud Computing: it concerns the capability of moving application components from a mobile device to the cloud. This technique, in general, improves the efficiency of a system, although sometimes it can lead to a performance degradation. To decide when and what to offload, we propose the use of a method for determining an optimal infinite scheduler, which is able to manage the resource assignment of components with the aim of improving the system efficiency in terms of battery consumption and time. In particular, in this paper we define a cost/reward horizon method for Mobile Cloud Computing systems specified in the language MobiCa. By means of the model checker UPPAAL, we synthesize an optimal infinite scheduler for a given system specification. We assess our approach through a case study, which highlights the importance of a scheduler for reducing energy consumption and improving system performance.

1 Introduction

In the last few years, the ubiquity of the Internet and the increasing use of mobile devices has changed the mobile application market. The majority of these applications are available everywhere at any time with a heavy traffic over the network and a high computation demand. These characteristics require a large amount of resource usage, especially concerning battery lifetime, in limited devices. The current technological constraints have led to the emergence of a new concept called Mobile Cloud Computing (MCC) [1, 2]. MCC is a new paradigm given by the combination of mobile devices and cloud infrastructures. This mix exploits the computational power of the cloud and the ubiquity of mobile devices to offer a rich user experience. It relies on the *offloading* concept, that is the possibility of moving the computation away from a mobile device to the cloud.

The evolution of power-saving approaches at hardware level has led to mobile devices that can adapt the speed of their processors. This allows devices to save energy, but it is not powerful enough in case of power hungry applications. Since battery life is the most important feature in mobile devices according to their users [3], computation offloading is clearly a cheaper alternative or a valid partner to the hardware solutions to improve performance and efficiency.

Unfortunately, the use of remote infrastructures does not always come without a cost, as sometimes computation offloading may degrade the application's performance. In general, a good rule of thumb is to offload an application component only if the cost of its local execution is greater than the cost for the synchronization of the input and output data plus that of the remote execution. This could be highly influenced by network latency, bandwidth, computational power of the mobile device, and computational requirements of the code to offload. Notably, in the issues discussed above, when we talk about cost we consider both time and the battery energy required to execute the application component. Therefore, in this field it is critical for the developer to assess the cost of the application execution during the development phase, in order to identify the best trade-off between energy consumed and performance.

The decision of whether to offload a given application component can be taken by means of a set of rules, i.e. the so-called *scheduler*. By applying the scheduler rules, we obtain a sequence of offloading choices, called *schedule*, that should allow the system to reach the desired goals while improving performance and reducing battery consumption.

In this paper, we take into consideration schedulers that produce infinite schedules ensuring the satisfaction of a property for infinite runs of the application. This is particularly useful for MCC, where applications are supposed to provide permanent services, or at least to be available for a long period. In particular, considering that our model is equipped with constraints on duration, costs and rewards, we are interested in identifying the *optimal* schedulers that permit the achievement of the best result in terms of energy consumption and execution time. In fact, over infinite behaviors, it is possible to recognize a cyclic execution of components that is optimal and is determined by means of the limit ratio between accumulated costs and rewards. Consequently, an optimal scheduler is given by maximizing or minimizing the cost/reward ratio.

We propose here a cost/reward horizon method for MCC systems. We focus on a domain specific language (DSL), called MobiCa [4], that has been devised for modelling and analysing MCC systems. The use of a DSL increases the productivity for non-experts as, due to its high-level of abstraction, it keeps MCC domain concepts independent from the underlying model for the verification. Since the semantics of MobiCa is given in terms of a translation into networks of Timed Automata (TA), we show how the problem of designing and synthesising optimal schedulers can be solved by using the well-known model checker UPPAAL and the cost/reward method. Moreover, our approach also allows the developer of MCC systems to define a custom scheduler and compare its quality vis-a-vis the optimal one. In particular, by performing analysis with the statistical model checker UPPAAL-SMC [5] we are able to precisely quantify how much the custom scheduler differs from the optimal one, to understand if the custom scheduler is more suitable for time or energy optimization, and to simulate its behavior in order to study how it scales as system executions grow longer.

Although the optimal scheduling research field already provides many different techniques, we believe that model checking is an interesting approach for our

Systems:	$Sys ::= (c, b, n, m) \triangleright \tilde{A} \mid c \triangleright \tilde{A} \mid Sys_1 \vert Sys_2$
Applications:	$A ::= \langle \tilde{F}; S \rangle$
Fragments:	$F ::= f[i, m, s, o]$
Structure:	$S ::= f_1 \; Op \; \tilde{f}_2 \mid S_1 \, ; S_2$
Operators:	$Op ::= \longrightarrow \mid \dashrightarrow \mid \twoheadrightarrow$

Table 1. MobiCa syntax

purposes, due to the flexibility provided by the use of logics for characterizing different system properties and to its capability of performing analysis through optimization techniques already implemented in model verification, which can be fruitfully exploited for designing schedulers for MCC systems.

We illustrate our approach through a simple running example, and show its effectiveness and feasibility by means of a larger case study.

The rest of the paper is structured as follows. Section 2 presents syntax and semantics of the MobiCa language. Section 3 introduces our method for synthesising optimal schedulers for MCC systems, while Section 4 illustrates how statistical model checking can be used for evaluating the performance of schedulers. Section 5 shows our approach at work on a navigator case study. Finally, Section 6 reports our conclusions and describes future work.

2 The MobiCa language

In this section we recall syntax and semantics of the MobiCa language [4]. It is designed to model MCC systems and, in particular, permits to specify both the contextual environment where an MCC application will run and its behavior.

2.1 Language syntax

Table 1 shows the syntax of MobiCa given as BNF grammar. A system in MobiCa is expressed by a set of mobile devices and cloud machines composed in parallel. A typical example of mobile device is a smartphone, a tablet or any kind of device with limited computational and resource capabilities. A *mobile device* $(c, b, n, m) \triangleright \tilde{A}$ consists of a container of applications $\tilde{A}$ (where $\tilde{x}$ denotes a tuple of elements of kind x) and of a tuple of device information *(c,b,n,m)*, which in order denote: computational power (that is the number of instructions executed per second), battery level, network bandwidth and used memory. A *cloud machine* $c \triangleright \tilde{A}$ is also a container of installed applications $\tilde{A}$, but as device information it only specifies the number c of instructions executed per second. An *application* A is organized in components $\tilde{F}$, called 'fragments', whose composition is described by a structure S. A *fragment* F can be a single functionality, a task or an action, derived by partitioning the application in parts. It is described as a name f that uniquely identifies the fragment, the number i of instructions composing

it, the amount m of memory required at runtime, the amount s of data to be transferred for synchronization in case of offloading, and finally a boolean value o indicating whether the fragment is designed to be offloadable or not. A *structure* is a collection of terms of the form $f_1\ Op\ \tilde{f}_2$, where from the source fragment (f_1) on the left of the operator Op the execution can proceed with one or more target fragments ($\tilde{f}_2$) defined on the right, according to three types of operators:

- *Non-deterministic choice* ($\longrightarrow$) indicates that the execution will progress from the source fragment to *one* of the target fragments, which is non-deterministically selected.
- *Sequential progress* ($\dashrightarrow$) allows the computation to sequentially progress from the source fragment (on the left of $\dashrightarrow$) to each fragment in the ordered tuple (on the right of $\dashrightarrow$). If the tuple contains more than one fragment, after the execution of each of them the computation should go back to the source fragment.
- *Parallel execution* ($\twoheadrightarrow$) allows the execution to progress from the source fragment to *all* the target ones, by activating their parallel execution.

If we have more operators for the same source fragment, the system will non-deterministically choose among them. Notably, self-loops are disallowed.

Below we show a scenario where an optimal infinite scheduling is necessary for minimizing energy consumption and improving system performance.

Example 1 (A simple application). The example in Figure 1 is inspired by one from [6] and graphically describes a possible MobiCa application A. The application is composed of three fragments, f_0, f_1 and f_2, connected by means of the non-deterministic operator ($\longrightarrow$) and by the sequential operator ($\dashrightarrow$). Since the application behavior is deterministic in this case, the unique run is composed by an initialization phase $f_0 \to f_2$, followed by an infinite loop $f_2 \to f_0 \to f_2 \to f_1 \to f_2$. Each fragment of the sequence, can be executed either on the mobile or in the cloud, with the only requirement of maintaining the data consistent. For consistency we intend that either a fragment is executed on the same location of its predecessor or at a different location only after the result of the predecessor has been synchronized.

In the figure, the fragments are annotated with 4 parameters; in order, we have: the execution time on the mobile device (given by the number of instructions divided by the mobile computation power, i.e. i/c), the execution time on

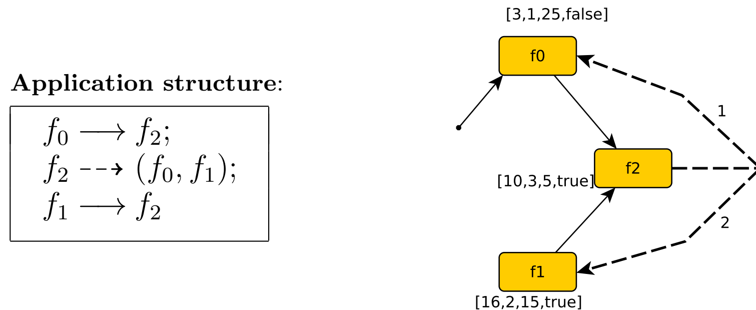

Fig. 1. A simple example of a MobiCa application

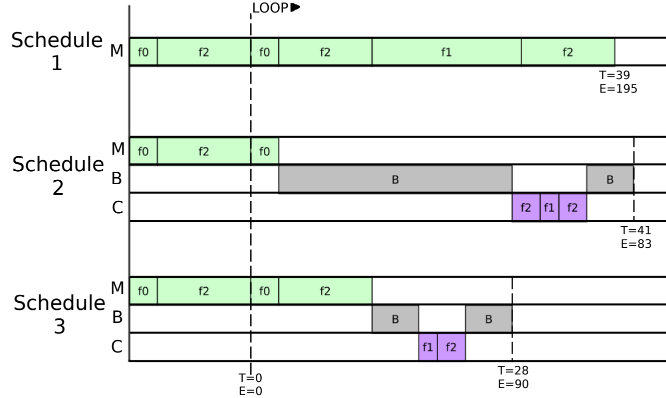

Fig. 2. Schedules for the simple application

the cloud, the synchronization time of the results on the bus (given by s/n) and a boolean value representing the offloadability of the fragment (a false value indicates that only the local execution is admitted, as in the case of f_0). The graphical notation in Figure 1 is formalized in terms of the so-called System Graph in Section 3 (Definition 1). Notably, the memory parameters introduced in MobiCa are not considered in this particular formalization.

An infinite scheduler for the simple application shown in Figure 1 should provide a sequence of execution choices for each of the three fragments between the available resources. A schedule is optimal if the total execution time or cost is minimum, considering that the energy consumption per time unit for the mobile device is 5 when it is in use, 1 in the idle state, and 2 for the synchronization.

The Gantt chart in Figure 2 depicts three possible schedules for the proposed example application. For each of them, we indicate the location of execution between mobile and cloud, and the use of the bus. The values of T and E at the end of the sequence are the time and the energy required by the scheduler for computing a complete loop cycle. In the first schedule, the computation is maintained locally for all fragments; this behavior is reasonable when the network is not available. Another approach might be to maintain the computation locally only for the non-offloadable fragments (in our case only f_0) and to try to move the computation remotely as soon as possible; this allows one to manage the task congestions in the mobile device. The third schedule instead takes into consideration the sequence of offloadable fragments and executes the computation remotely only when the synchronization of data is minimal. □

2.2 TA-based semantics

We describe here the semantics of MobiCa, given in terms of a translation to networks of Timed Automata (TA). Such a semantics can indeed be used to solve the previously described scheduling problems, by resorting to the UPPAAL model checker. We refer the interested reader to [4] for a more complete account of the MobiCa semantics, and to [7] for the presented UPPAAL model.

$\longrightarrow$	f_0	f_1	f_2
f_0	0	0	1
f_1	0	0	1
f_2	0	0	0

$\twoheadrightarrow$	f_0	f_1	f_2
f_0	0	0	0
f_1	0	0	0
f_2	0	0	0

$\dashrightarrow$	f_0	f_1	f_2
f_0	0	0	0
f_1	0	0	0
f_2	1	2	0

Table 2. Operators translation

The translation is divided in two parts: the *passive* part, which focusses on resources, and the *active* one, which focusses on the applications. Thus, the TA corresponding to a given MobiCa system is the composition of the results of the passive and active translations merged together by means of a global declaration. Below we describe the details of the translation in terms of UPPAAL models.

Global declaration. The global declaration consists of all the shared variables used for the synchronization of fragments, clocks for keeping track of time, and variables stating the internal state of the resources. In the global declaration we find also the structure S of the application declared as an adjacency matrix. A structure consists of three $n \times n$ matrices, one for each transition operator, where n is the length of the $\tilde{F}$. Let m_{ij} be the (i,j) entry of a matrix, $m_{ij} \geqslant 1$ if the ith and j^{th} fragments are connected, and 0 otherwise. Notably, the diagonal of each matrix is always zero, as self-loops are not admitted. Table 2 shows the corresponding three adjacency matrices, related to the example shown in Figure 1. In particular, we have:
($\longrightarrow$): the non-deterministic transition for fragment f_i is activated if row_i has non-zero cells, and the next fragment to be activated is non-deterministically selected in $\{f_j \mid m_{ij} = 1\}$;
($\twoheadrightarrow$): the parallel transition is similar to the non-deterministic one, with the difference that the fragment f_i activates all the fragments f_j with $m_{ij} = 1$;
($\dashrightarrow$): the sequential operator matrix is slightly different from the previous ones, as the values are not only 0 or 1. These values must be interpreted as a sequence defining the order in which the target fragments are activated for each execution of the source fragment. The activation of the sequential operator on a fragment excludes the other operators until the sequence of activation is terminated. In our example, fragment f_2 activates first the execution of f_0 and then the execution of f_1 (see the last row of the matrix at the right-hand side in Table 2).

Fragments. The TA for a generic fragment is depicted in Figure 3; the template is parametric, so that it is a valid representation for each fragment of the application. The execution of the fragment starts from the initial location where it is waiting for the activation. The activation is managed by the array *activated[]* as follows: whenever the element in the array corresponding to the fragment index becomes *true*, the corresponding fragment can move to the *ready* location. In this latter location, it can continue its execution on the mobile device or the cloud, depending on the evaluation of the guards on the transitions. They state that the fragment can be executed locally only if the the results of the previous fragment are updated locally (*result[previous[id]][0]==1*), or remotely only if they are updated remotely and the fragment is offlodable (*result[previous[id]][1]==1 and Info[id].isOffloadable==true*). When the execution of the fragment is completed, it can proceed towards either the *network* location, in order to synchronize the results locally and remotely (*result[id][0]=1, result[id][1]=1*), or the initial loca-

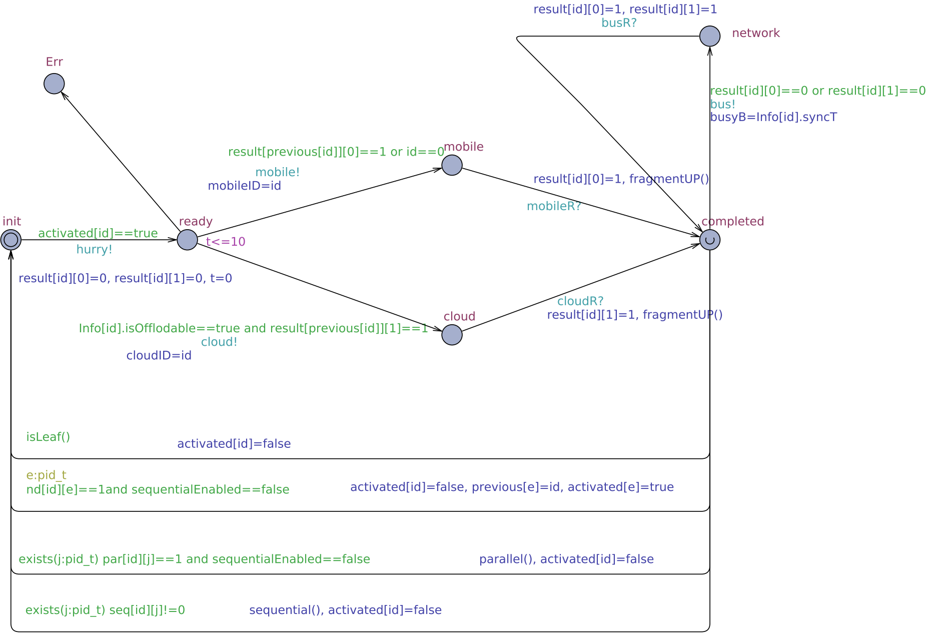

Fig. 3. Fragment translation

tion by following one operator in the structure. Indeed, the use of each operator is rendered as an outgoing transition from the *completed* location to the *init* one; these transitions are concurrent and enabled according to the corresponding adjacency matrix, defined in the global declaration.

Resources. Each kind of resource (i.e., mobile device, cloud and bus) is translated into a specific TA; since these TA are similar, we show here just the one for the mobile device (Figure 4) and, for the sake of presentation, we describe it in terms of a general resource. A resource can be in the *idle* state, waiting for some fragment, or *inUse*, processing the current fragment. When the resource synchronizes with a fragment, it resets the local clock and remains in the *inUse* state until the clock reaches the value corresponding to the occupation time for the current fragment. Before releasing the resource, the battery level of the mobile device is updated according to the permanence time and the energy consumed by the resource. In this model, we assume that no energy is consumed if there is nothing to compute, and the energy power consumed by the cloud during its execution corresponds to the energy used by the mobile in the idle state waiting for the results.

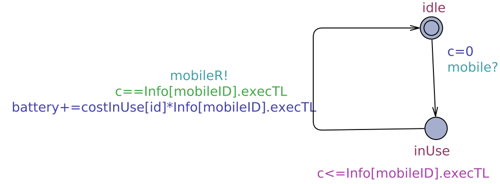

Fig. 4. Mobile translation

3 Synthesis of optimal infinite schedulers for MCC

In this section, we formalize the notion of optimal infinite scheduler in terms of two cost functions on a System Graph (SG). A SG provides a graphical represen-

tation of a MobiCa application, which is useful as an intermediate model between the specification and the resulting network of TA generated by the translation.

Definition 1 (System Graph). *Given an application $\langle \tilde{F}; S \rangle$ installed in a system with a mobile device defined by information (c, b, n, m) and a cloud machine defined by c', its* system graph *SG is a tuple $\langle N, \longrightarrow, \dashrightarrow, \twoheadrightarrow, T, E, O \rangle$ where:*

- *$N = \{f \mid f[i, m, s, o] \in \tilde{F}\}$ is a set of fragment names.*
- *$\longrightarrow$, $\dashrightarrow$, $\twoheadrightarrow$ $\subseteq N \times N$ are three transition relations defined as $f \longrightarrow f'$ (resp. $f \dashrightarrow f'$, $f \twoheadrightarrow f'$) iff $f' \in \tilde{f}$ for some $\tilde{f}$ such that $f \longrightarrow \tilde{f} \in S$ (resp. $f \dashrightarrow \tilde{f} \in S$, $f \twoheadrightarrow \tilde{f} \in S$). We use $f \rightarrowtail f'$ to denote either $f \longrightarrow f'$ or $f \dashrightarrow f'$ or $f \twoheadrightarrow f'$.*
- *$T : N \times \{M, C, B\} \rightarrow \mathbb{N}$ gives the execution time of a fragment on a resource (where M is the mobile device, C the cloud, and B the bus); given $f[i, m, s, o] \in \tilde{F}$, we have: $T(f, M) = \lfloor i/c \rfloor$, $T(f, C) = \lfloor i/c' \rfloor$, and $T(f, B) = \lfloor s/n \rfloor$.*
- *$E : \{M, C, B\} \rightarrow \mathbb{N}$ is the energy, expressed as a natural number, consumed per time unit by the mobile device when a given resource is in use.*
- *$O : N \rightarrow \{0, 1\}$ represents the possibility of offloading a fragment (value 1) or not (value 0); given $f[i, m, s, o] \in \tilde{F}$, we have $O(f) = o$.*

Notably, an SG is completely derived from the information specified in the corresponding MobiCa system except for the energy function E. Energy consumption information, indeed, is added to that provided by MobiCa specifications in order to enable the specific kind of analysis considered in this work.

A *path* on SG is a finite sequence $\eta = f_0 \rightarrowtail f_1 \rightarrowtail \ldots \rightarrowtail f_k$ $(k \geq 0)$. Notably, in a path, parallel activations of fragments are interleaved.

Definition 2 (Scheduler). *Given a system graph SG, a* scheduler *is a partial function $\Theta : N \times Op \times N \rightarrow \{0, 1\}^2$ that, given a transition $f \rightarrowtail f'$ in SG, returns a pair of values $\pi_s, \pi_t \in \{0, 1\}$ which indicate the execution location of the source fragment f and of the target fragment f', respectively, where* 0 *denotes a local execution and* 1 *a remote one.*

When a scheduler is applied to a transition of the corresponding SG, it returns information about offloading decisions for the involved fragments. By applying a scheduler Θ to each transition of a sequence of transitions, i.e. a path η, we obtain a *schedule* δ, written $\Theta \cdot \eta = \delta$. A schedule consists of a sequence of triples of the form (f, π, β), each one denoting a fragment f, belonging to the considered path, equipped with its execution location π and the synchronization flag β. Parameters $\pi, \beta \in \{0, 1\}$ indicate the local $(\pi = 0)$ and remote $(\pi = 1)$ execution and the need $(\beta = 1)$ or not $(\beta = 0)$ of data synchronization for f. Formally, $\Theta \cdot \eta = \delta$ is defined as follows: let f and f' being two consecutive fragments in the path η, there exist in δ two consecutive triples $(f, \pi, \beta) \rightarrowtail (f', \pi', \beta')$ iff $\Theta(f, \rightarrowtail, f') = (\pi_s, \pi_t)$ s.t. $\pi = \pi_s$, $\pi' = \pi_t$ and $\beta = |\pi_s - \pi_t|$. Notice that, as Θ is a partial function, there may be transitions in η that are not in δ; for such transitions the schedule does not provide any information about the offloading strategy to apply, because they are not considered by the scheduler.

Taking inspiration from the approach in [8], we define two cost functions. In particular, we consider the cost of executing a given path in the considered SG using the scheduler, i.e. the cost functions are defined on schedules.

Definition 3 (Time and energy costs). *The time and energy costs of a schedule δ for a given $SG = \langle N, \longrightarrow, \dashrightarrow, \twoheadrightarrow, T, E, O\rangle$ are defined as follows:*

$$Time(\delta) = \textstyle\sum_{(f,\pi,\beta)\in\delta} \left((1-\pi)\times T(f,M) \ + \ \pi\times T(f,C) \ + \ \beta\times T(f,B) \right)$$

$$Energy(\delta) = \textstyle\sum_{(f,\pi,\beta)\in\delta} \left((1-\pi)\times T(f,M)\times E(M) \ + \ \pi\times T(f,C)\times E(C) \right. \\ \left. + \ \beta\times T(f,B)\times E(B) \right)$$

The function $Time(\delta)$ calculates the total time required by the schedule δ, i.e. the time for executing a path of SG according to the scheduler that generates δ. For each fragment f in the system, we add the time $T(f,M)$ if the fragment is executed locally ($\pi = 0$), or the time $T(f,C)$ if the fragment is executed remotely ($\pi = 1$). The function considers also the synchronization time $T(f,B)$ if two consecutive fragments are executed at different locations ($\beta = 1$).

The function $Energy(\delta)$ calculates the total energy required to complete the scheduled path. The difference with respect to the previous function is that here the time of permanence of a fragment in a resource is multiplied by the corresponding energy required per time unit.

Relying on the cost functions introduced above, we can have the time-optimal scheduler Θ_T and the energy-optimal scheduler Θ_E for a given SG, which determine the sequence of actions that generates the less costly combination of resources, in terms of $Time(\delta)$ and $Energy(\delta)$ respectively, for a path in SG.

Example 2 (Time and battery costs for the small application). We evaluate here the schedules proposed in Example 1 (Figure 2) using the cost functions introduced above. Notice that in the calculation we consider only the cyclic part of the application omitting the initialization that is not relevant in terms of an infinite scheduler. Table 3 reports the time and energy consumed for the three schedules calculated according to Definition 3.

Evaluating the results, the time-optimal scheduling for the application is achieved in Schedule 3, that is $(f_0, 0, 0) \rightarrowtail (f_2, 0, 1) \rightarrowtail (f_1, 1, 0) \rightarrowtail (f_2, 1, 1)$, with a total time cost $T_3 = 28$. The offloading choices for achieving such result are formalized in terms of the scheduler (written here using a notation based on triples) Θ_T={($f_0 \longrightarrow f_2$,0,0),($f_2 \dashrightarrow f_1$,0,1),($f_1 \longrightarrow f_2$,1,1),($f_2 \dashrightarrow f_0$,1,0)}. On the other hand, Schedule 2, that is $(f_0, 0, 1) \rightarrowtail (f_2, 1, 0) \rightarrowtail (f_1, 1, 0) \rightarrowtail (f_2, 1, 1)$, is the energy optimal one, with a total energy consumption $E_2 = 83$. The corresponding scheduler is Θ_E={($f_0 \longrightarrow f_2$,0,1),($f_2 \dashrightarrow f_1$,1,1),($f_1 \longrightarrow f_2$,1,1),($f_2 \dashrightarrow f_0$,1,0)}. From this example it is clear that there may not be a correspondence between energy and time consumption, since we have different cost results. Hence, defining a scheduler optimized for more resources is not always a simple task. □

Sched.	*Time*	*Energy*
1	$T_1 = (3 + 10 + 16 + 10) = 39$	$E_1 = (3 + 10 + 16 + 10) \times 5 = 195$
2	$T_2 = (3 + 25 + 3 + 2 + 3 + 5) = 41$	$E_2 = (3 \times 5 + 25 \times 2 + 3 \times 1 + 2 \times 1 + 3 \times 1 + 5 \times 2) = 83$
3	$T_3 = (3 + 10 + 5 + 2 + 3 + 5) = 28$	$E_3 = (3 \times 5 + 10 \times 5 + 5 \times 2 + 2 \times 1 + 3 \times 1 + 5 \times 2) = 90$

Table 3. Time and energy costs of the schedules for the simple application

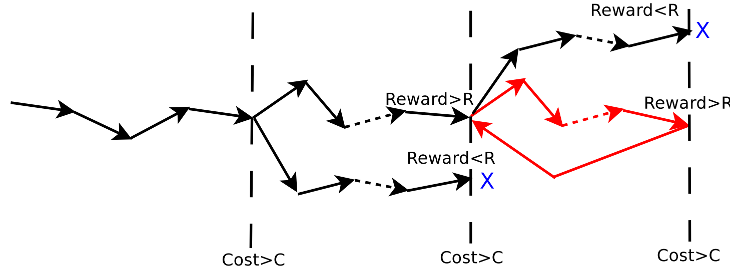

Fig. 5. cost/reward Horizon Method

3.1 Cost/reward horizon method

In order to find the optimal scheduler for an application with infinite behavior, as discussed above, we propose a cost/reward horizon method. From the literature [9, 10], we know that the optimal ratio is computable for diverging and non-negative rewards systems.

In what follows we first present the basic concepts behind our cost/reward method and then we show how the optimal infinite scheduling problem can be solved using TA and the UPPAAL model checker. The behavior of the application is the starting point for defining an optimal infinite scheduler. It can be described as a set of paths. For each path UPPAAL will generate all possible schedules and will choose the best one according to a specific ratio (clarified below). The chosen schedule is indeed the less costly one and, hence, it can be used to synthesize the rules that compose the optimal scheduler.

Let's start defining the ratio of a finite path η of a SG as follows:

$$Ratio(\eta) = Cost(\eta)/Rewards(\eta)$$

where $Cost()$ and $Rewards()$ are two functions keeping track of the accumulated cost/reward along the path η. Now, we extend this ratio to an infinite path γ=$f_0 \rightarrowtail$, ... , $\rightarrowtail f_n \rightarrowtail$ ..., with γ_n the finite prefix of length n; the ratio of γ is:

$$Ratio(\gamma) = \lim_{n\to\infty}(Cost(\gamma_n)/Rewards(\gamma_n))$$

An optimal infinite path γ_o is the one with the smallest $Ratio(\gamma)$ among all possible schedules.

Finding the smallest ratio is not always a tractable problem, but it is possible to improve its tractability reducing the problem to a given horizon. From this new point of view, we want to maximize the reward in a fixed cost window. Notice that, the cost window should be of an appropriate length, in order to complete the execution of at least one application cycle.

This technique can be implemented in UPPAAL considering the query:

$$\texttt{E[] not(}f_1\texttt{.Err}, ..., f_n\texttt{.Err) and (Cost}\geq\texttt{C imply Reward}\geq\texttt{R)} \quad (1)$$

This query asks if there exists a trace were the system keeps running without errors and whenever the predefined cost `C` is reached, the accumulated reward should be at least `R` (Figure 5).

For verifying the satisfaction of the above formula, the TA model includes an additional template (Figure 6) implementing the cost window using the reset mechanisms. It consists of one state and a self-loop transition, where each time the simulation reaches the cost C the transition will reset the accumulated Costs and Rewards. In this way, the behavior of the application is split in cost windows and in each of them the accumulated rewards should satisfy the formula Reward≥R.

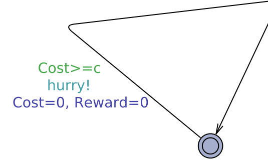

Fig. 6. Reset TA

Since we are looking for the maximum reward given a predefined cost, for finding the optimal scheduler it is necessary to discover the maximum value of R for which the formula (1) holds. The resulting trace generated by a satisfiable formula has the structure depicted in Figure 5. The trace starts with some initial actions corresponding to the application start-up and leads to its cyclic behavior. As shown in the figure, the approach does not consider all possible traces, but only the ones that satisfy the constraints of the query. The candidate schedule is the piece of trace that is highlighted in red, which means that UPPAAL has found a cyclic behavior in the application whose execution satisfies the formula forever. This means that we have found an optimal schedule from which it is possible to derive the set of rules that will generate the optimal scheduler.

3.2 The horizon method at work

In this section we show how the cost/reward horizon method can be applied to MCC systems and, in particular, to the example presented in Figure 1.

We are interested in finding a time-optimal and/or battery-optimal scheduler. By applying the method presented in Section 3.1 to a MCC system, given an infinite path γ, the time- and energy-based ratios become $r_T = \lim_{n\to\infty}(Time(\gamma_n)/Fragments(\gamma_n))$ and $r_E = \lim_{n\to\infty}(Energy(\gamma_n)/Fragments(\gamma_n))$, respectively. Thus, the accumulated costs are calculated by the functions $Time()$ and $Energy()$ given in Definition 3. The rewards are instead defined by a function $Fragments() : \eta \to \mathbb{N}$ which counts the number of fragments executable in the fixed window. The more fragments we are able to execute with the same amount of time or energy, the better the resources are used.

To find the minimum time-based ratio using the UPPAAL model checker we can ask the system to verify a query of the following form:

```
E[] forall(i:pid_t) not(Fragment(i).Err) and (GlobalTime≥300 imply
                                (fragments>41))
```

In this specific case we want to know if, in a fixed window of 300 units of time, it is always possible to execute 41 fragments. To find the minimum ratio we have to iterate on the number of fragments in the denominator in order to find the maximum number for which the query holds. In our running example, the maximum value that satisfies the query is 41, giving a ratio $300/41 = 7.31$.
The resulting trace generated by the presented query results in an execution sequence that can be synthesized as the Schedule 3 shown in Figure 2.

The query for determining the energy-based ratio is defined as:

```
E[] forall(i:pid_t) not(Fragment(i).Err) and (battery≥900 imply
            (fragments>43 and battery≤930))
```

In this case, the resulting ratio is $900/43 = 20.93$. thus, the system requires 20.93 units of battery per fragment. Notice that in this query there is an extra constraint defined as an upper bound on the right side of the `imply` keyword. This is because we can have different schedules satisfying the formula, but we consider only the ones that exceed the battery threshold as little as possible. The resulting trace from the energy query gives us the energy optimal schedule, that in this case can be synthesized as the Schedule 2 in Figure 2.

To assess the truthfulness of the cost/horizon method, we can compare the obtained results with the ones calculated directly in the Gantt chart in Figure 2. The energy ratio for Schedule 2 on one loop is given by $83/4 = 20.75$. The slight difference in the results is due to the size of the cost window. In the Gantt chart we are considering a window that fits perfectly four fragments and, hence, we do not have any incomplete fragment that affects the final result as in the case of the horizon method, but the approximation is really close to the best case.

4 Evaluating performance of a custom control strategy

In this section, we present a technique for evaluating the performance of a custom scheduler using the Statistical Model Checking facilities [5, 11] provided by UPPAAL (the model checker is called UPPAAL-SMC, or SMC for short).

Let's suppose now that a developer wants to define his own scheduler for an application and to know how much the resulting custom scheduler is close to the optimal one or to calculate some statistics. Possible reasons for customizing a scheduler could be problems in the development phase related to hardware cost or less quantitative issue, such as security and privacy that force the developer to introduce a static scheduler.

The new personalized scheduler in UPPAAL is modeled as a TA template called Manager. The duty of this manager is to decide on which resource each fragment should be executed. Considering the model presented in Figure 3, here we do not have anymore the decision in the *ready* location between mobile and cloud transition, but just a transition that is synchronized with the manager. The manager operates as a mediator, between the fragments and the resources. Once the manager receives notice of a new fragment to execute, it decides according to some rules in which resource's waiting list to move it. The resources are modeled following a busy waiting paradigm, where every time the queue is not empty, a new fragment is consumed. Before executing the assigned fragment, the resource checks the execution location of its predecessor; if data synchronization is not required, it just executes the fragment, otherwise it synchronizes the data and then processes it. Once the computation is completed, the resource returns the control back to the executed fragment and passes to the next one.

4.1 A custom scheduler

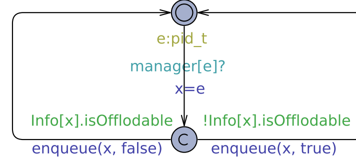

Fig. 7. Manager TA

The manager can contain different kinds of rules. One possibility is to define a rule for each fragment or to specify a more general rule that can be applied to the whole application. For example, to execute a fragment remotely when it is offloadable is a very simple rule that a developer can consider to implement in a MCC system. Figure 7 depicts the manager implementing this custom strategy. In detail, after a fragment is synchronized with the manager, the latter decides to enqueue the fragment on the corresponding waiting list according to the guard `Info[x].isOffloadable`. In this way, if the fragment `x` is offloadable it is queued locally, otherwise remotely.

Looking more closely at the custom scheduler, we notice it realises the same behavior of Schedule 2 in Figure 2, which we already know to be energy-optimal.

4.2 Evaluation via SMC

As previously said, usually a customized scheduler is defined for reasons related to particular conditions of the environment. Here, we perform some statistical analysis to quantify how much the customized scheduler above is far from the optimal one, in order to have a quantitative measure of any performance loss. In particular, below we present some verification activities and compare the obtained results with the time/energy-optimal scheduling we found in Section 3.2.

By evaluating the following queries using the SMC tool, we can determine the expected maximum value for the number of fragments that can be executed in a given temporal window:

```
E[time<=300;2000](max:fragments)      E[battery<=900;2000](max:fragments)
```

These two queries aim at finding the expected value over 2000 runs in a window of 300 units of time and 900 units of battery, respectively. The results are: 29 fragments for the first query and 41 for the other one. Comparing these results with those of optimal ones, we can clearly see that the scheduler defined by the developer is almost as efficient as the energy-optimal one. Indeed, they differ only for 2 fragments in the energy case. Instead, the performance of the custom scheduler is very far from that of the time-optimal scheduler, as they differ for 14 fragments.

The proposed strategy can be also evaluated to see if it is closer either to the energy-optimal scheduler or to the time-optimal one. This can be achieved by checking if the probability to reach the time-optimal scheduler is greater than the probability to reach the energy-optimal scheduler.

```
Pr[time<=300](<> fragments>=43)>=Pr[battery<=900](<> fragments>=41)
```

The result of this query is `false` with probability `0.9`, meaning that the probability of reaching the energy-optimal scheduler is greater than the one for the time-optimal scheduler.

We can also simulate the system behavior executing the following commands:

```
simulate 1[time<=300]{battery,fragments}
simulate 1[battery<=900]{time,fragments}
```

Their results are shown in Figure 8. On the left-hand side, we have the number of fragments compared with the consumed battery. On the right-hand side, instead, we have the ratio of executed fragments and required time.

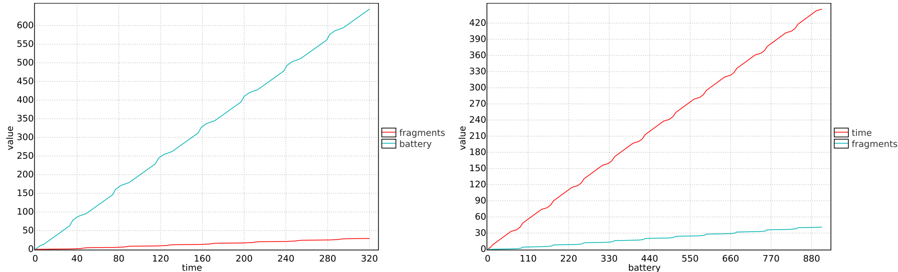

Fig. 8. Simulation results

5 Experiments with a navigator case study

In the previous sections we have illustrated the proposed approach by means of a simple application. In this section, we aim at showing the effectiveness and feasibility of the approach by means of a larger case study, drawn from [4], concerning a navigator application. This kind of application is one of the most complex and used in mobile devices. This is an interesting case study for this work from the point of view of its complexity and its strong real-time requirements to be satisfied at runtime. In particular, the greatest challenge for navigation system developers is to provide an application that is able to find the right route, and recalculate it as quickly as possible in case of changes, considering the current traffic condition.

The corresponding MobiCa system is represented in Figure 9. The system starts when the user inserts the destination in the Configuration panel that consequently activates the Controller. The Controller in turn asks the GPS for the current coordinates and forwards them to the Path calculator. The Path calculator, interacting with the Map and the Traffic evaluator, will provide a possible itinerary. The itinerary is processed by the Navigator, which forwards information to the Navigation Panel. This latter component, with the help of the Voice and Speed Trap Indicator, provides the navigation service to the end user. The Navigator is also responsible for reactivating the Controller in order to check possible updates of the route.

We present now the results obtained using the cost/reward horizon method applied to this case study. The complexity of this example is a good test bed for our method. Notice that the values of the parameters used in the example are generated ad-hoc as a proof of concept. From real life we expect that the developer can determine information about fragment instructions by performing experimentation, statistics or simply studing the complexity in the code.

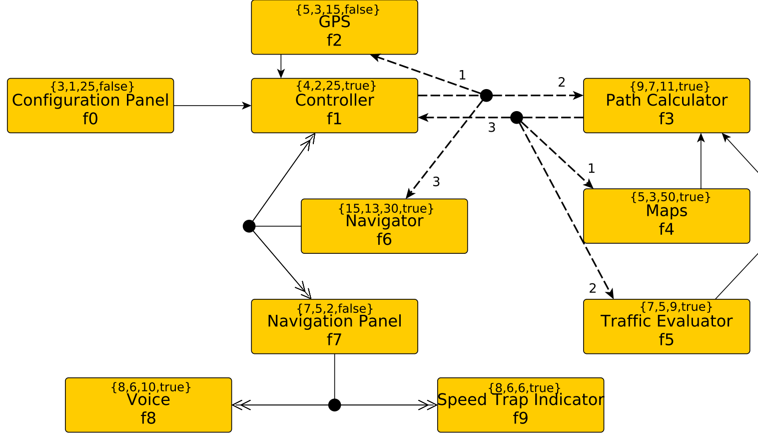

Fig. 9. Navigator case study: MobiCa specification

The diagram in Figure 10 shows the resulting schedules synthesized from the verification of the following queries:

```
E[] forall(i:pid_t) not(Fragment(i).Err)
    and (time≥100 imply (fragments>16 and time<120))

E[] forall(i:pid_t) not(Fragment(i).Err)
    and (battery≥400 imply (fragments>21 and battery<419))
```

The first query defines the time-optimal schedule with respect to a time window of 100 units and with a maximum number of executed fragments equal to 16. The ratio $r_T = 100/16 = 6.25$ was reached keeping the execution local for almost all fragments except for f_8 and f_9, which are executed in parallel remotely.

The opposite behavior is identified in the verification of the second query for the energy-optimal case, where only three fragments are executed locally and all the others remotely. Since the fragments f_7 and f_2 are not offloadable, they are maintained locally together with the fragment f_1. The choice to execute f_1 locally is given by the necessity of the scheduler to wait for a suitable moment to move the computation remotely. Clearly, moving the computation between two non-offloadable fragments is not convenient; furthermore, sometimes it is better to anticipate or postpone the offloading when the data synchronization is minimal or less costly. The ratio of this scheduler is $r_E = 400/21 = 19.05$ with a final energy consumption equal to 272 units per cycle.

The cost/reward horizon method fits MCC systems perfectly. In particular, during the sequential behavior of the application it tries to find the best moment for moving the computation remotely, defining also different strategies according to the role of the fragment. Instead, during parallel behavior, where there are no direct relations between fragments, it tries to exploit the benefit derived by allocating the computation both on the mobile and on the cloud.

As a final evaluation we present the results obtained verifying the custom scheduler described in Section 4 on the navigator case study using SMC. Verifying the expected maximum reward using the query `E[battery<=400;2000](max: fragments)`, we obtain a cost energy ratio $r_E = 400/16 = 25$. Even worse is the score obtained by trying to optimize the performance using the query

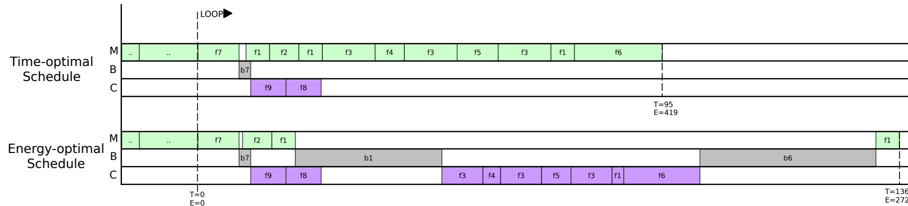

Fig. 10. Navigator case study: optimal schedules

`E[time<=100; 2000](max: fragments)`, which achieves a ratio $r_T = 100/5 = 20$. Thus, comparing the obtained values, it is possible to notice a substantial growth of the ratio for the custom scheduler. Since a higher ratio means a decrease in performance, we can claim that the strategy defined by the developer is not a good approximation of the optimal one. Furthermore, analyzing the results in more detail, we notice that the custom scheduler is very far from the time optimal, with a ratio that is four time larger than the one achieved by the optimal scheduler. Considering instead the energy case, it is possible to reach a ratio of 25 against the 19.05 of the optimal one. Looking at these results, the developer is aware that using his custom scheduler he can achieve a good performance if he is interested in energy optimization, although this is not optimal.

By performing a simulation (we omit the picture due to lack of space), we can see a significant gap between the number of executed fragments and the elapsing of time according to the consumed battery power; there is indeed a symmetric increase of values generated by the cyclic behavior of the application. The plot in Figure 11, instead, represents a scheduler synthesized using a histogram. Using an appropriate simulation query, which takes into account the fragments in execution on the resources, it is possible to represent each fragment as a column of the same height of its identifier in the specific resource. For the sake of readability, columns referring to cloud (red lines) and mobile (green lines) are depicted on the same level of the graph, while the network columns (blue lines) are reported just below. A peak in the blue line means that the corresponding fragment above requires the synchronization on the bus before its execution.

6 Concluding remarks

We provide an approach for designing schedulers for MCC systems specified in MobiCa. Using UPPAAL, and relying on a cost/reward horizon method introduced here, we are able to synthesize an optimal infinite scheduler for a mobile application. This scheduler defines offloading choices that allow the system to reach the best results in terms of performance and energy usage.

Related work. Optimization is a topic that is considered in many application fields. Also in the MCC literature there is a significant effort on the optimization of a utility function or specific metrics for the offloading technique. Among the most significant works, we mention RPF [12], which derives its strategy using the direct observation of the system. It runs processes alternatively between local and remote machines in order to determine the best choices. This technique is not optimized for highly dynamic systems, where the parameters of the resources change constantly, but it can be a good solution for more static environments. Another approach based on direct observation is MAUI [8], where information

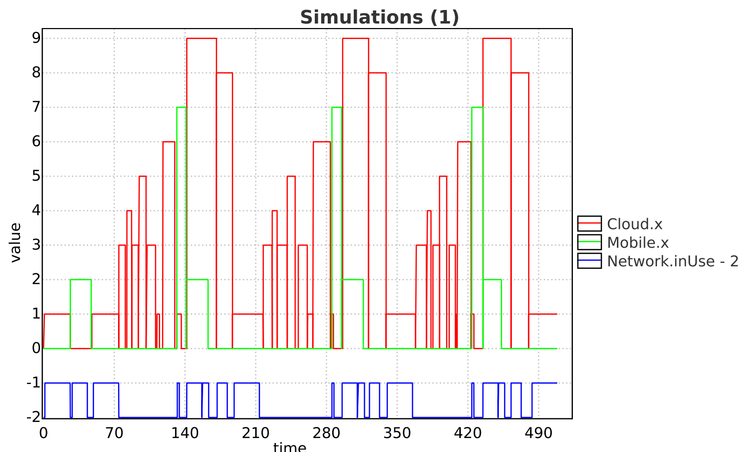

Fig. 11. Navigator case study: fragments execution

about the environment is collected and used to formulate the problem as an optimization problem. The proposed optimization function compares the time required for executing a process locally against the time for the synchronization of data plus the remote execution.

The limitations of methods based on the direct observation have been addressed using the past history. The resulting systems, like Spectra [13], Chroma [14] and Odessa [15], build a model on past inputs and use it to make predictions or decisions rather than to observe the current system configuration.

Our approach is similar to the ones mentioned above, with the main difference that we provide a language that describes the system environment. This language, called MobiCa, was presented for the first time in [4], and here is used to generate an optimal infinite scheduling using the UPPAAL model checker. Compared with the other works, we are able to foresee all the possible configurations of the system at design time, by providing a scheduler that is optimal for a certain interval of parameters. The optimal infinite scheduler is generated using the cost/reward horizon method implemented with timed automata and solved verifying a simple query by means of the model checker. The closest related work, from which we take inspiration for the cost/reward horizon method is presented in [9, 10], where a general version of this method was applied to the priced automata formalism to find the best configuration of the considered system. The flexibility of this method has permitted to obtain good results in the MCC field, confirmed also by reasonable performances that are in the order of seconds for the considered models.

Future work. There are still several issues which are open for future work. A possibility would be to extend our work in order to have an automatic procedure for obtaining the maximum number of fragments in a given time window. Indeed finding the optimal ratio requires one to consider several computations, that may become unfeasible in case of high numbers. Thus, a possible idea to optimize the methodology is to develop heuristics that allow one to reach the best result in a faster way. Another aspect to consider is how the proposed approach can be

transferred to the technology. A possibility is to include the decision support as part of a middleware that can provide to the developer an optimal scheduler derived through our method. This middleware could be integrated also with the runtime decision support proposed in [4]. Another interesting point of extension is the re-scheduling at runtime. Indeed, a small variation of the environmental parameters can bring to different results in the system optimization leading to an obsolete scheduler. Thus, we need to consider its recalculation at runtime.

ACKNOWLEDGEMENTS: Luca Aceto has been supported by the projects 'Nominal Structural Operational Semantics' (nr. 141558-051) of the Icelandic Research Fund and 'Formal Methods for the Development and Evaluation of Sustainable Systems', grant under the Programme NILS Science and Sustainability, Priority Sectors Programme of the EEA Grants Framework. Kim G. Larsen is supported by the SENSATION FET project, the Sino-Danish Basic Research Center IDEA4CPS, the Innovation Fund Center DiCyPS and the ERC Advanced Grant LASSO. Andrea Morichetta and Francesco Tiezzi have been supported by the EU projects ASCENS (257414) and QUANTICOL (600708) and by the MIUR PRIN project CINA (2010LHT4KM).

References

1. Fernando, N., Loke, S.W., Rahayu, W.: Mobile cloud computing: A survey. Future Generation Computer Systems **29**(1) (2013) 84 – 106
2. Flinn, J.: Cyber foraging: Bridging mobile and cloud computing. Synthesis Lectures on Mobile and Pervasive Computing **7**(2) (2012) 1–103
3. Kumar, K., Lu, Y.H.: Cloud computing for mobile users: Can offloading computation save energy? Computer **43**(4) (2010) 51–56
4. Aceto, L., Morichetta, A., Tiezzi, F.: Decision support for mobile cloud computing applications via model checking. In: MobileCloud. Volume 1., IEEE (2015) 296–302
5. Bulychev et al.: Uppaal-smc: Statistical model checking for priced timed automata. arXiv preprint arXiv:1207.1272 (2012)
6. Gruian, F., Kuchcinski, K.: Low-energy directed architecture selection and task scheduling for system-level design. In: EUROMICRO, IEEE (1999) 1296–1302
7. MobiCa/UPPAAL model: http://www.amorichetta.eu/MobiCa/m_u_model.zip
8. Cuervo et al.: Maui: making smartphones last longer with code offload. In: MobiSys, ACM (2010) 49–62
9. Bouyer, P., Brinksma, E., Larsen, K.G.: Optimal infinite scheduling for multi-priced timed automata. Formal Methods in System Design **32**(1) (2008) 3–23
10. Rasmussen, J.I., Larsen, K.G., Subramani, K.: On using priced timed automata to achieve optimal scheduling. Formal Methods in Syst. Design **29**(1) (2006) 97–114
11. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Van Vliet, J., Wang, Z.: Statistical model checking for networks of priced timed automata. In: Formal Modeling and Analysis of Timed Systems. Springer (2011) 80–96
12. Rudenko, A., Reiher, P., Popek, G.J., Kuenning, G.H.: The remote processing framework for portable computer power saving. In: SAC, ACM (1999) 365–372
13. Flinn, J., Park, S., Satyanarayanan, M.: Balancing performance, energy, and quality in pervasive computing. In: Distributed Computing Systems. (2002) 217–226
14. Balan, R.K., Satyanarayanan, M., Park, S.Y., Okoshi, T.: Tactics-based remote execution for mobile computing. In: MobiSys, ACM (2003) 273–286
15. M. Ra et al.: Odessa: Enabling interactive perception applications on mobile devices. In: MobiSys, ACM (2011) 43–56