# PSRL: A New Method for Real-Time Task Placement and Scheduling Using Reinforcement Learning

Bakhta Haouari ★☆ Rania Mzid ★✚ Olfa Mosbahi ☆

★ ISI, University Tunis-El Manar, 2 Rue Abourraihan Al Bayrouni, Ariana, Tunisia
☆ LISI Lab INSAT, University of Carthage, INSAT Centre Urbain Nord BP 676, Tunis, Tunisia
✚ CES Lab ENIS, University of Sfax, B.P:w.3, Sfax, Tunisia

## Abstract

*Modern real-time system development methodologies describe a stage in which application tasks are deployed onto an execution platform. The deployment process is divided into two steps: (i) task placement on processors and (ii) task scheduling to determine their execution order. The overall performance of the deployment model depends on the two steps, which are interdependent. In this paper, a new method based on reinforcement learning techniques, called PSRL, is proposed. PSRL explores all the feasible placements in the first step. In the second step, an optimal schedule is considered for each feasible placement. PSRL generates the optimal deployment, which corresponds to the placement and scheduling that minimize task response times. Application to case studies shows the applicability and quality of the obtained solutions when compared to related work.*

## 1 Introduction

Today, embedded systems can be found in a wide range of applications, including avionics, trains, and medical equipment. Embedded systems are frequently real-time as they must perform certain tasks in a specific amount of time (i.e., deadline); failure to satisfy the timing constraints might be essential for human safety [1]. Currently, real-time embedded systems are becoming more complex, requiring more computational power. As a result, to be executed, a system's functionalities (i.e., tasks) may be distributed on different execution units called processors.

The development of Real-Time Embedded Systems (RTES) requires the definition of the deployment model. The deployment stage involves two main steps: placement and scheduling [2]. The task placement problem is concerned with task-to-processor mappings. The scheduling problem, on the other hand, aims to define the execution order of tasks in each processor. To guarantee the respect of real-time properties, design-time verification using schedeuleability analysis techniques [3] is typically employed.

Producing a deployment model has been proved to be NP-hard problem even for small-size systems [3]. Due to their inherent complexity, placement and scheduling problems should be automated by solving an optimization problem with respect to real-time constraints. This problem has been largely considered in the literature. Some works consider partially the problem by concentrating solely on the placement or scheduling issue. In [4], the authors use genetic algorithms (GA) to solve the task placement problem in distributed systems to minimize communication cost. The authors in [5] provide a Mixed Integer Linear Programming (MILP)-based technique to minimize the blocking time between tasks in the placement phase. Producing the optimal scheduling of tasks is not considered in this work; however, the authors use the Rate-Monotonic algorithm [6] for priority assignment and focus on minimizing the number of tasks to improve the overall performance of the system. The scheduling problem is addressed in [7] where a new algorithm based on Reinforcement Learning (RL) is proposed for global fixed-priority task assignment in multi-processor real-time systems. In [8], the authors propose a new scheduling algorithm that maximizes the magnitude of safety margins while respecting the engineering constraints. In [9], new scheduling policies are proposed for heterogeneous platforms equipped with a hardware accelerator. When dealing with both problems (i.e., placement and scheduling), studies in the literature consider, in general, sub-optimal staged approaches. Existing approaches use optimization techniques to optimize independently the placement and the scheduling steps. For instance, in [2], the authors propose a GA-based approach where they minimize the number of processors in the placement step and the response time of tasks in the scheduling step. However, the authors claim that the obtained solution is near-optimal since the optimization performed in the first step can hide a better solution in the second phase.

To address this problem, we propose in this paper an RL-based technique called PSRL (i.e., Placement and Scheduling using Reinforcement Learning). The PSRL method defines two steps: The first stage involves an exhaustive RL-based search of all feasible placements and which serves as input to the second stage. The second step determines an RL optimal scheduling for each placement. The global optimal scheduling is then the best among all the alternatives. The optimal scheduling in this paper is the one that optimizes the response time of tasks. The placement and scheduling

DOI reference number: 10.18293/SEKE23-178

problems are thus modeled as a Markov Decision Problem (MDP) [6] and the Q-learning algorithm [10] is used in the placement and scheduling steps. The originality of this paper is the use of Q_learning algorithm, typically used to generate an optimal solution, in the placement phase to conduct an exhaustive search of feasible solutions through the exploitation of the Q_table. As a result, the scheduling step explores all the optimal solutions without being constrained by a given placement, thus avoiding producing sub-optimal solutions.

The rest of the paper is organized as follows: Section 2 gives the system formalization. Section 3 describes the proposed PSRL method and details RL algorithms. Section 4 illustrates experimentation, and Section 5 concludes the paper and discusses future directions.

## 2 System formalization

The placement problem considers as inputs: (i) the task model that we denote by $\tau$ in this work. We assume that this model is composed of $n$ synchronous, periodic, and independent tasks (i.e.,$\tau = \{T_1, T_2 \ldots T_n\}$). Each task $T_i$ is characterized by static parameters $T_i = (C_i, pr_i, d_i)$ where $C_i$ is an estimation of its worst case execution time, $pr_i$ is the activation period of the task $T_i$, and $d_i$ is the deadline that represents the time limit in which the task $T_i$ must complete its execution, (ii) and the hardware model that we denote by $\mathcal{P}$, represents the execution platform of the system. We assume that this model is composed of $m$ homogeneous processors (i.e., $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$).

The placement step produces a set of possible placements that we denote $PM$. Each placement $PM_k \in PM$ defines a way to place tasks among the different processors in the hardware model. For RTES, the placement model $PM_k$ must be feasible. Feasibility means that the placement of the real-time tasks on the different processors must guarantee respect for the timing requirements of the system. In that context, Audsley [3] developed a necessary and sufficient schedulability test. This test is based on the computation of the processor demand factor $U_p$ and is defined as follows:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{d_i} \leq n(2^{\frac{1}{n}} - 1) \quad (1)$$

For each feasible placement, $PM_k$, the scheduling step defines, in each processor, the execution order of tasks to which they are assigned. As a result, a deployment model $DM_k$ is produced. We denote by $DM$ the set of deployment models produced in the scheduling step. Each task $T_i$ in the deployment model $DM_k \in DM$ will be mapped to a particular processor and will be assigned a priority $p_i$. Priority assignment in the scheduling stage must guarantee the feasibility checked in the placement step. Real-time feasibility at this level guarantees that the response times of the different tasks are lower or equal to their deadlines (i.e., $\forall T_i \in \tau, RT_i \leq d_i$). The response time of a task $T_i$ is given in [3], and is computed according to the following expression:

$$RT_i = C_i + \sum_{s \in hp(i)} \left\lceil \frac{RT_i}{pr_s} \right\rceil C_s \quad (2)$$

Where $hp(i)$ refers to the tasks with priorities higher or equal to the priority of the task $T_i$

An optimal deployment model $DM{*}_k$ is one that minimizes the *Sum RT-ratio* defined by [9] as a measure of performance for scheduling models, which is computed as follows:

$$\sum_{i=1}^{n} \frac{RT_i}{d_i} \quad (3)$$

We identify by $DM*$ the deployment model that has the lowest *Sum RT-ratio* among all produced $DM{*}_k$, such as

$$\sum_{T_i \in DM*} \frac{RT_i}{d_i} = \min_{DM{*}_k \in DM} \sum_{T_i \in DM{*}_k} \frac{RT_i}{d_i} \quad (4)$$

## 3 PSRL description

In this section, we describe the proposed method for real-time task placement and scheduling. The PSRL method is schematically described in Figure 1; the specific steps are shown in Algorithm 1. The PSRL technique considers as entries: (i) the task model, which describes the application functions, and (ii) the hardware model, which describes the execution processors. The initial stage is to generate possible feasible placements to the given problem. After producing all solutions, the second phase iterates on each placement to define the optimal scheduling. In fact, the second phase generates one deployment model $DM{*}_k$ for each placement $PM_k$ which minimizes the *Sum RT-ratio* (see expression 3). Once all of the deployment models for each placement have been generated, the PSRL algorithm selects the optimal solution from among those generated ($DM*$) (as defined in the expression 4).

### 3.1 Step 1: Generate feasible placement models

As previously mentioned, the objective of this step is to produce placement models for a given task and hardware models. As shown in Figure 1, when no solution is found, the designer has to adjust the parameters of the entry models. Otherwise, this step generates all feasible placement models. In this step, we are particularly referred to the Q_learning algorithm (i.e., a free model RL algorithm) for the exploration of the design space. Indeed, during its processing, Q_learning generates a Q_table as a workspace where it stores all its knowledge about the task placements. At the end of the agent learning, the Q_table is used to generate a unique solution, the optimal one. In this paper, since

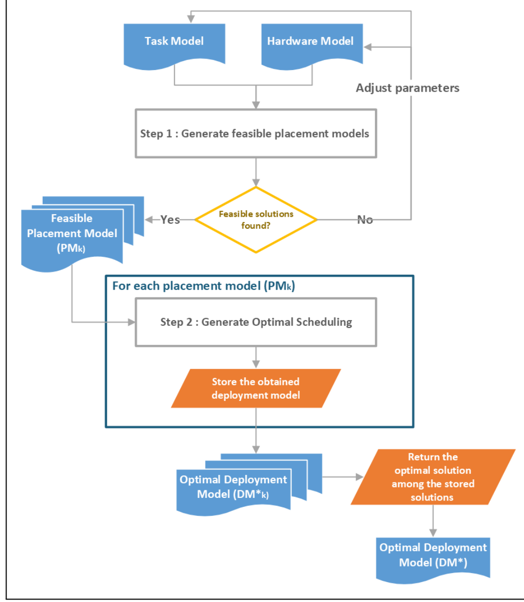


Figure 1: PSRL overview

---

**Algorithm 1:** PSRL Algorithm

---

**Input:** $\tau$: List of tasks
$\mathcal{P}$: List of processors
**Output:** $\mathcal{DM}*$: The optimal deployment model
**Notations:**
$\mathcal{PM}$ : Feasible placement models
$\mathcal{DM}$ : Optimal deployment models
$DM*$ : Best deployment model
$\mathcal{PM} \leftarrow$ Generate feasible placement models ($\tau$,$\mathcal{P}$);
**foreach** $PM_k \in \mathcal{PM}$ **do**
  $DM*_k \leftarrow$ Generate Optimal Scheduling ($PM_k$);
  $DM \leftarrow$ Update($DM*_k$);
**end**
$DM* \leftarrow$ Select_Best_Solution ($DM$);
**return** $DM*$;

---

optimal placement does not guarantee an optimal deployment model, we propose to use Q_learning for an exhaustive search of the feasible placements (the optimal and the sub-optimal ones). In fact, we believe that sub-optimal placements can hide good scheduling, even the optimal one. As a result, we use the Q_table to extract all the placement models. The final states in the Q_table refer to possible placement models that may be feasible or not following the initial designer-specified constraints. Application of RL techniques to the placement problem requires the refinement of its key elements as the following:

**State :** It is a snapshot of the system state at time step $t$, it is represented by the collection of tasks already placed on the processors with respect to their deadlines, as well as the list $L_t$ of tasks that have not yet been placed

**Decision epoch :** Matches the placement of all the tasks on the different processors

**Agent :** It is the decision maker, it has to be learned and then used to decide a processor for a given task

**Reward (R) :** It is the award given by the system to the agent in return to the agent's action and it is computed as Equation 5.

$$R = \begin{cases} U_j - U_i & \text{When there is enough} \\ & \text{space on } P_j \text{ to support } T_i, \\ -n & \text{Otherwise} \end{cases} \quad (5)$$

Where $U_j$ denotes the utility of processor $P_j$ and $U_i = C_i/di$, denotes the utility of the task $T_i$. Algorithm 2 describes the first step. As an initial step, the algorithm builds the Q_table then the agent starts its learning through the assignment of a given task to a processor. If the assignment helps to yield a feasible placement, the agent is rewarded; if not, it is penalized. Equation 5 summarizes the reward compute. The process of placement selection is iterated until the Q_table values become invariable. At this step, the totality of the system's states are visited, and the majority of solutions are built in the Q_table final states, i.e states where all the tasks are placed. Note that a feasible placement is one in which each task from the task model is assigned to a processor with respect to the processor utility constraint (see equation1).

### 3.2 Step 2: Generate Optimal Scheduling

The set of possible placement models generated in the first stage is used as input for the second. Q_learning is then reused in the second stage to build an optimal deployment model $DM_k$ for each placement $PM_k$. For the scheduling problem, key RL elements are also refined as the following:

**State:** At a time step t a system state $S_t$ is represented by the set of tasks already ordered and the ones waiting for a priority assignment

**Decision epoch:** Matches the ordering of all the tasks on a given processor

**Agent:** The decision maker, it chooses the action to move from $S_t$ to $S_{t+1}$ will maximizing its reward following a policy $\pi$

**Reward (R):** The bonus that the agent attempts to maximize or the penalty that the agent attempts to avoid following the execution of an action. In the scheduling problem, the agent must select the job that generates the minimum *Sum RT-ratio* (see equation3). However, because the agent seeks to maximize R, R is defined as the inverse of the *RT-ratio* and is calculated as:

$$R = \begin{cases} \frac{1}{\frac{RT_i}{d_i}}, & \text{when } RT_i \leq d_i \\ -m & \text{Otherwise} \end{cases} \quad (6)$$

**Algorithm 2:** Step1: Generate feasible placement models

```
Input: τ: List of tasks; P: List of processors;
Output: PM: The feasible placement models
Notations:
α: The learning rate
ϵ: The ϵ-greedy value
γ: The discount factor
ϵ_decease_factor: The ϵ degradation factor
Q : The Q-table
L_t : List of unplaced tasks
PM_t : The placement model at time step t (i.e., list of
  (processor, tasks) pairs)
S : The state at the time step t
S′ : The state at the time step t + 1
States ← Generate_PlacementStates (τ, P);
Actions ← Generate_PlacementActions (τ, P);
Q ← Create_Q-table (States, Actions);
while Q(S, a) still moving do
    reset S;
    while L_t is not empty do
        a ← Select_Placement (ϵ, L_t);
        Take_Action a then Compute (R) ;          /* compute
          reward using expression 5 */
        Observe Q(S′, a′);
        Q(S, a) =
          Q(S, a) + α[R + γmax_a′ Q(S′, a′) − Q(S, a)] ;
          /* Bellman's equation [10] */
        Update Q(S, a);
        Remove (T_i, L_t) ;        /* remove T_i from L_t */
        S ← S′;
    end
    ϵ ← max(ϵ - ϵ_decease_factor, 0);
end
PM ← Extract_feasible_final _states_from Q ;
return PM;
```

**Algorithm 3:** Step2: Generate Optimal Scheduling

```
Input: PM_k: A feasible placement model
Output: DM∗_k: The optimal deployment model for the
        placement PM_k
Notations:
L: The list of tasks to be scheduled in the processor P_j; α: The
  learning rate; ϵ: The ϵ-greedy value; γ: The discount factor
ϵ_decease_factor: The ϵ degradation factor
Q : The initial Q-table for the processor P_j
L_t : List of unscheduled tasks
Sch_t : The scheduling model at time step t
S: The state at the time step t
S′ The state at the time step t + 1
foreach P_j ∈ PM_k do
    S ← Generate_SchedStates (PM_k);
    A ← Generate_SchedActions (PM_k);
    Q ← Create_Q-table (S, A);
    while Q(S, a) still moving do
        reset S;
        while L_t is not empty do
            a ← Select_Scheduling (ϵ, L_t, Sch_t);
            Take_Action a then Compute (R); ;   /* compute
              reward using expression 6 */
            Observe Q(S′, a′);
            Q(S, a) =
              Q(S, a) + α[R + γmax_a′ Q(S′, a′) − Q(S, a)]
              ;          /* Bellman's equation [10] */
            Update Q(S, a);
            Remove (T_i, L_t) ;   /* remove T_i from L_t */
            Sch_t ← Update (Sch_t) ;   /* schedule T_i */
            S ← S′;
        end
        ϵ ← max(ϵ - ϵ_decease_factor, 0);
    end
    DM∗_k ← Update (DM∗_k) ;  /* ADD Sch_t to DM∗_k
      */
end
return DM∗_k;
```

The RL model for the scheduling stage is described in the Algorithm 3. After creating the scheduling Q_table, the agent is trained how to select the task to be ordered with the aim to maximize its profit as described in the expression 6. This phase is repeated for each processor in the current $PM_k$ to finally produce the optimal deployment model $DM{*_k}$.

The proposed algorithms (Algorithm 2 and Algorithm 3) have some initialization parameters, such as $\alpha$, which represents the learning rate to moderate the speed of learning and the update of Q-values we assume $\alpha$ = 0.5), and $\gamma$, which represents the discount factor that quantifies the importance given to future rewards (in our approach we consider that future task placement are important thus we attribute an enough great value to $\gamma$ = 0.9). To choose an action (i.e., for the placement or scheduling), Q-learning uses an $\epsilon$-greedy policy [11]. $\epsilon$-greedy policy is an efficient random approach that selects with a probability $\epsilon$ a random action and with a probability (1- $\epsilon$) the action with the highest estimated reward $Q(S, a)$. An $\epsilon$ of zero means that the agent will never explore new states (i.e., choose action with the best Q-value), whereas an $\epsilon$ of one forces the agent to only explore (i.e., choose only random action). As a result, a well-studied value of $\epsilon$ is necessary to capture a trade-off between exploration and exploitation.

## 4 Evaluation

This paper considers two case studies to assess the applicability and effectiveness of the proposed PSRL method. The first is a subsystem of Continental AG's Cruise Control System (CCS), which is used on AUTOSAR-compliant architectures [12] and allows a car's speed to be maintained regardless of the surface form (flat or sloping) on which it is driving. The second case study is the Unmanned Air Vehicle (UAV), which is an autonomous plane with a camera dedicated to dynamically defined waypoints that was constructed as part of the AMADO project [13].

It is expected that the two case studies will run on two processors, $P_1$ and $P_2$. Based on the data set defined in [14], Table 2 and Table 1 describe, respectively, the related task models composed of eight periodic tasks for CSS and

Table 1: UAV Taks model description

| Task | $C_i$ (m) | $pr_i$ (m) | $d_i$ (m) |
|---|---|---|---|
| $T_1$ | 1 | 100 | 10 |
| $T_2$ | 1 | 25 | 25 |
| $T_3$ | 1 | 30 | 30 |
| $T_4$ | 5 | 125 | 80 |
| $T_5$ | 1 | 50 | 7 |

five periodic tasks for UAV with their real-time parameters.

Table 2: CCS Taks model description

| Task | $C_i$ (m) | $pr_i$ (m) | $d_i$ (m) |
|---|---|---|---|
| $T_1$ | 3 | 125 | 20 |
| $T_2$ | 3 | 125 | 80 |
| $T_3$ | 2 | 125 | 20 |
| $T_4$ | 4 | 125 | 80 |
| $T_5$ | 2 | 125 | 20 |
| $T_6$ | 4 | 125 | 80 |
| $T_7$ | 3 | 125 | 80 |
| $T_8$ | 2 | 125 | 80 |

Since the objective of the first step is to perform an exhaustive search of feasible placements, we perform, as a first evaluation, a sensibility analysis of the number of feasible placement models. We maintain a set of experiments on the epsilon value ($\epsilon$ in Algorithm 2), which guides the exploration/exploitation process to fill the Q_table and subsequently identify the feasible placements. While the value of epsilon is increased at each step, the placement generation algorithm (Algorithm 2) is run and the number of feasible placement models is tracked. We remark that the number of placement models increases from one step to another until it converges and reaches a maximum value (i.e., the number of feasible placement models for the considered application). However, this comes at the expense of the number of iterations required for the agent to reach convergence. In fact, a low epsilon value forces the agent to rely on its limited experience, decreasing the possibility of many states in the Q_table being visited and, as a result, reducing the number of feasible solutions generated. A large epsilon value allows the agent to visit more states in the Q_table, but also causes the time of convergence to be delayed. A trade-off between the episode number and the epsilon value is required to generate the set of possible placement models in a reasonable number of iterations.

Figure 3 and Figure 2 show the evaluation results for respectively the UAV and the CCS case study. Figure 3 shows two curves. The green corresponds to the feasible placement models number for various epsilon values, whereas the blue corresponds to the total number of placement possibilities (i.e., feasible and non feasible placement models). As the number of placement models is the same for epsilon $\in [0.6..1]$, it is unnecessary to run PSRL with epsilon

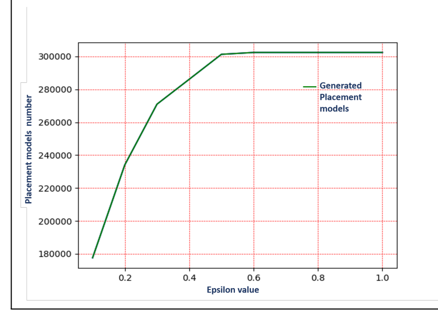


Figure 2: CCS placement models

greater than $0.6$ for the UAV case study. An epsilon value greater than $0.6$ results in more useless episodes. We also note that the number of feasible placement models is close to $1200$, which is relatively low when compared to the total number of placement models. Indeed, this is explained by the fact that, due to the real-time parameters considered for the UAV case study, solutions that result in tasks running in one processor ($P_1$ or $P_2$) are all rejected. Figure 2, in contrast to Figure 3, shows only one curve. Indeed, for the CCS case study, the real-time parameters considered lead to a scenario in which all possible placement models are feasible. This explains why we get the same curves.

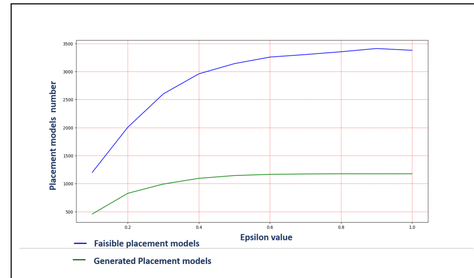


Figure 3: UAV placement models

We conduct a second set of experiments on the two case studies, where we set the epsilon value to 0.6 for the UAV and to 0.5 for the CCS, respectively (i.e., results of the first experiments). The goal of this evaluation is to compare the quality of the derived optimal model to the approach described in [2], where the deployment model is generated in two steps: the first seeks to minimize the number of processors, while the second aims to decrease task response time. Figures 4 and 5 demonstrate, for the CCS and UAV case studies, the set of deployment models represented by the *Sum RT-ratio* values for all viable placement models, with extreme values matching the best schedule if it is a global minimum and the worst one if it is a global maximum. The figures show that the optimal solution for both case studies is far from matching the processor number minimization [2], thus a thorough search of all conceivable

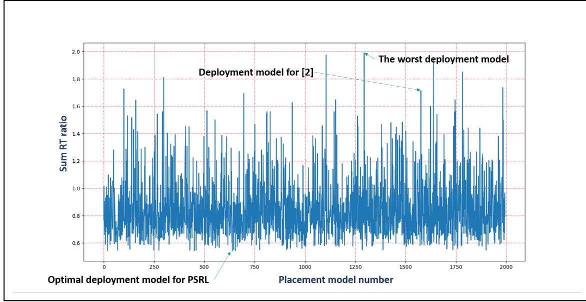


Figure 4: CCS Deployment Models

placement is quite important for absolute optimal schedule generation. For the UAV case study, the optimal deployment model produced by the PSRL method is $DM* = \{P_1\{T_3, T_1, T_5, T_2\}, P_2\{T_4\}\}$, where $T_1$, $T_2$, $T_3$, and $T_5$ are assigned to the processor $P_1$ and $T_3$ has the highest priority value, however $T_2$ has the lowest one. $T_4$ in turns is assigned to the processor $P_2$. For the CCS case study, the optimal deployment model produced by the PSRL method is $DM* = \{P_1\{T_1, T_2, T_7, T_6\}, P_2\{T_4, T_3, T_5, T_8\}\}$, where $T_1$, $T_2$, $T_6$, and $T_7$ are assigned to the processor $P_1$ and $T_1$ has the highest priority value, however $T_6$ has the lowest one. Tasks $T_3, T_4, T_5$, and $T_8$ are placed in the processor $P_2$ where $T_4$ is assigned the highest priority and $T_8$ the lowest one.

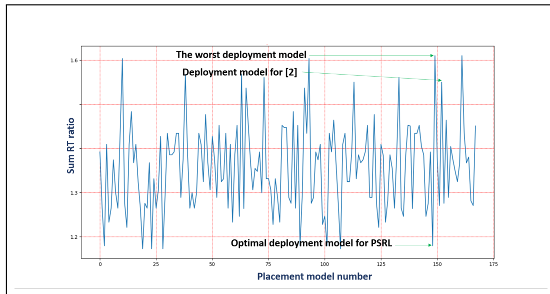


Figure 5: UAV Deployment models

## 5 Conclusion

In this paper, we propose a new method called PSRL for real-time task placement and scheduling. The proposed solution was based on reinforcement learning techniques, particularly the Q-learning algorithm, to solve the deployment problem for real-time systems in two stages. In the first stage, placement, an exhaustive search for all feasible solutions was performed. For each feasible placement, the scheduling stage produces the deployment model, which minimizes the response times of tasks. The best solution among all the ones generated is considered an optimal deployment for the given problem. PSRL was tested on two case studies with different properties, and the results prove the efficiency of PSRL compared to related work. In future work, we aim to extend the proposed approach by considering more complex systems. In addition, we intend to consider other optimization techniques. Thus, an extension of this technique to be multi-objective will be addressed.

## References

[1] W. Lakhdhar, R. Mzid, M. Khalgui, and N. Trèves, “Milp-based approach for optimal implementation of reconfigurable real-time systems.” in *ICSOFT-EA*, 2016, pp. 330–335.

[2] M. Salimi, A. Majd, M. Loni, T. Seceleanu, C. Seceleanu, M. Sirjani, M. Daneshtalab, and E. Troubitsyna, “Multi-objective optimization of real-time task scheduling problem for distributed environments,” in *Proceedings of the 6th Conference on the Engineering of Computer Based Systems*, 2019, pp. 1–9.

[3] N. C. Audsley, “On priority assignment in fixed priority scheduling,” *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[4] M. H. Kashani, H. Zarrabi, and G. Javadzadeh, “A new metaheuristic approach to task assignment problem in distributed systems,” in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*. IEEE, 2017, pp. 0673–0677.

[5] W. Lakhdhar, R. Mzid, M. Khalgui, G. Frey, Z. Li, and M. Zhou, “A guidance framework for synthesis of multi-core reconfigurable real-time systems,” *Inf. Sci.*, vol. 539, pp. 327–346, 2020. [Online]. Available: https://doi.org/10.1016/j.ins.2020.06.005

[6] Y. Manabe and S. Aoyagi, “A feasibility decision algorithm for rate monotonic scheduling of periodic real-time tasks,” in *Proceedings Real-Time Technology and Applications Symposium*. IEEE, 1995, pp. 212–218.

[7] H. Lee, J. Lee, I. Yeom, and H. Woo, “Panda: Reinforcement learning-based priority assignment for multi-processor real-time scheduling,” *IEEE Access*, vol. 8, pp. 185 570–185 583, 2020.

[8] J. Lee, S. Y. Shin, S. Nejati, and L. C. Briand, “Optimal priority assignment for real-time systems: a coevolution-based approach,” *Empirical Software Engineering*, vol. 27, no. 6, p. 142, 2022.

[9] D. Casini, P. Pazzaglia, A. Biondi, and M. Di Natale, “Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration,” *Journal of Systems Architecture*, vol. 124, p. 102416, 2022.

[10] A. G. Barto, “Reinforcement learning: An introduction. by richard s. sutton,” *SIAM Review Vol. 63, Issue 2 (June 2021)*, vol. 63, no. 2, p. 423.

[11] V. Bulut, “Optimal path planning method based on epsilon-greedy q-learning algorithm,” *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 44, no. 3, p. 106, 2022.

[12] S. Anssi, S. Tucci-Piergiovanni, S. Kuntz, S. Gérard, and F. Terrier, “Enabling scheduling analysis for autosar systems,” in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2011, pp. 152–159.

[13] K. Traore, E. Grolleau, and F. Cottet, “Simpler analysis of serial transactions using reverse transactions,” in *International Conference on Autonomic and Autonomous Systems (ICAS’06)*. IEEE, 2006, pp. 11–11.

[14] S. D. Alesio, L. C. Briand, S. Nejati, and A. Gotlieb, “Combining genetic algorithms and constraint programming to support stress testing of task deadlines,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–37, 2015.