IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 18, 2018, accepted December 5, 2018, date of publication December 13, 2018,
date of current version January 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2885950

Parallel Implementation of Reinforcement Learning Q-Learning Technique for FPGA

LUCILEIDE M. D. DA SILVA[1], MATHEUS F. TORQUATO[2], AND MARCELO A. C. FERNANDES[3]

[1]Department of Computer Science and Technology, Federal Institute of Rio Grande do Norte, Santa Cruz 59200 000, Brazil
[2]College of Engineering, Swansea University, Swansea SA2 8PP, U.K.
[3]Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte, Natal 59078 970, Brazil

Corresponding author: Marcelo A. C. Fernandes (mfernandes@dca.urn.br)

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)—Finance Code 001.

ABSTRACT Q-learning is an off-policy reinforcement learning technique, which has the main advantage of obtaining an optimal policy interacting with an unknown model environment. This paper proposes a parallel fixed-point Q-learning algorithm architecture implemented on field programmable gate arrays (FPGA) focusing on optimizing the system processing time. The convergence results are presented, and the processing time and occupied area were analyzed for different states and actions sizes scenarios and various fixed-point formats. The studies concerning the accuracy of the Q-learning technique response and resolution error associated with a decrease in the number of bits were also carried out for hardware implementation. The architecture implementation details were featured. The entire project was developed using the system generator platform (Xilinx), with a Virtex-6 xc6vcx240t-1ff1156 as the target FPGA.

INDEX TERMS FPGA, Q-learning, reinforcement learning, reconfigurable computing.

I. INTRODUCTION

Reinforcement learning (RL), is an artificial intelligence formalism that allows an agent to learn from the interaction with the environment where it is inserted [1]. This approach is indicated for situations in which there is not enough information about the behavior that the agent must take to reach its objective, that is, the agent without previous knowledge learns through interaction with the environment, receiving rewards for his actions and finding, the optimal policy [2].

The development of the Q-learning reinforcement learning technique in hardware enables designing faster systems than their software equivalents, thus opening up possibilities of its use in problems where meeting tight time constraints and/or processing a large data volume is required. It is also possible to reduce power consumption by reducing clock cycles in applications where processing speed is not relevant or less limiting than the need for low power consumption. Navigation algorithms on mobile robotics applications, in general, respond in hundred of the milliseconds and this property enables solutions on dedicated hardware work with a low clock frequency regards the other software solutions embedded on micro-controllers and microprocessors.

Real-time applications may have different time restrictions. Some examples of applications with the greatest restrictions are: systems for monitoring signals in health facilities, industrial systems control, digital communication systems, robots and even cars and aircraft. Traditional mechanisms and methods are not always able to overcome the barriers imposed by the more challenging time constraints. The research and development of artificial intelligence hardware algorithms for real-time applications has grown significantly in recent years due to their sampling time performance potential [3]–[8]. One of the purposes for the Q-learning technique implementation in hardware is to accelerate the algorithm processing and to obtain a faster optimal policy so that it can be used in high demanding applications.

Another motivation for the development of this work is the possibility to accelerate applications with great data flow such as in Big Data processing. Another application with the same burden of handling large amounts of data is Bioinformatics which, usually, needs to handle a large amount of genomic sequencing data [9]. It is also possible to use this approach in Data Mining applications to discover relevant information, which happens to be masked in large amounts of data [10].

Unlike general-purpose processors that usually have their clock at the maximum throughput, on field programmable gate arrays (FPGAs) the clock depends on what is running on it. Using a clock rate less than the maximum theoretical operating frequency causes the dynamic power consumed to decrease. The lower the clock, the lower the

consumption [11], making it suitable as well for low consumption systems.

For this work, FPGA was chosen because it provides high performance with a low operating frequency through the exploration of parallelism [12]. The latest FPGAs can deliver ASIC-like performance and density with the advantages of reduced development time, ease and speed of reprogramming, not to mention its flexible architecture [13].

Thus, this work presents a modular and parallel architecture proposal for the Q-learning technique implementation on FPGA reconfigurable hardware with the purposes of reducing processing time, allowing the algorithm to be used both in high dynamic systems and large data flow as well as low power consumption applications. The development of this work, as well as all simulations and results, was carried out using the development platform Xilinx System Generator [14] configured to work with a FPGA Virtex 6 xc6vcx240t 1ff1156 [15].

A. RELATED WORK

Machine learning, artificial intelligence and signal processing have been widely used in many recent applications. It is important to note two important new features for these types of applications: the amount of data that needs to be processed is constantly growing and mobile and robotic systems are becoming increasingly important [16]. As a result, several machine learning algorithms and artificial intelligence hardware implementations can be found in the related literature.

In [3], an overview of hardware implementations of artificial neural networks and fuzzy systems is presented, highlighting the main limitations, advantages and disadvantages of various application techniques. The author also performs an analysis of various hardware performance parameters, bottlenecks and and the cost-benefit intrinsic to the various implementation methodologies. In [5] the implementation of an FPGA hardware architecture for a neural network of associative memory applied to image recognition systems is described, where a detailed study of the network performance is conducted, including data such as occupancy rate, processing speed and consumption of the system in hardware.

However, little can be found regarding hardware implementation of fixed and programmable architecture for the Q-Learning reinforcement learning technique.

In the work of [17], a hardware pipeline architecture was described for the selection mechanism of the best action in the Q-learning state. According to the author, the algorithm delay increases with the number of actions, being the bottleneck of the system and it is possible to reduce this delay with the implementation of a pipeline architecture to select the best action in the state. It has also been proved, through a consistent mathematical reasoning, that the value function converges to an optimal policy, despite the pipeline architecture implementation. However, no hardware solutions were presented for the other mechanisms inherent to the Q-learning algorithm for reinforcement learning, nor were the details of the implementation, occupation analysis or tool used for the hardware design of this system mechanism presented. In [18], a hardware architecture of the SARSA (or On-line Q-learning) learning algorithm for reinforcement learning was developed for dynamic power management application. The main difference between SARSA and Q-learning is that SARSA is on-policy, that is, it learns action values related to the policy it follows, while Q-learning is off-police, not depending on the policy which is being used. The author converted the SARSA algorithm into its equivalent hardware modeling the architecture in VHDL (modelsim). The architecture implements a power management system that is able to change policy according to its workload. The proposal was simulated and synthesized in the Xilinx Spartan 2E. Implementations of Deep Q-Learning algorithms on FPGA are presented in [19]–[21] where the results show the of using FPGA compared to GPU and CPU. However, the works [19]–[22] are applying a semi-parallel implementation technique which differ from the approach proposed in this work that it uses a full-parallel implementation. The full-parallel approach allows a high throughput performance when compared with another implementation techniques.

Certain applications in signal processing and machine learning impose hardware technical limitations. In addition, the amount of data that needs to be processed is constantly growing [16]. A practical alternative for the design of hardware architectures is the use of reconfigurable tools such as FPGAs which provides a density performance similar to an ASIC (Application Specific Integrated Circuit) with the advantage of using rapid and flexible prototyping [4]. Due to its reconfigurable nature, new functions can always be added and the system upgrade can be performed as needed [13]. Another relevant advantage for the use of FPGA is the possibility of designing hardware modules that work in parallel, then, increasing the system processing capacity [23], which allows different parts of the algorithm to be executed simultaneously in order to reduce the overall processing time. This reduction results in an interesting alternative with a better performance than conventional microprocessors such as CPUs or GPUs, especially for applications where there are severe time restrictions.

Some papers found in the literature point out the design of parallel algorithms to increase their processing capacity. In [24], a decomposition technique for the Markov Decision Process (MDP) is approached in sub-problems, presenting a structure for the parallelization of reinforcement learning techniques. This technique, according to the work, is able to decrease processing time by up to ten times. In [25], an implementation of the Q-learning algorithm is proposed in a massively parallel machine using a Parallel Virtual Machine (PVM) message exchange paradigm with a cache-based communication scheme. This work presents significant convergence results and increase of speed, pointing the parallelization as an interesting training time reduction alternative for the policy learning. The work shown in [26] presents a comparative study of several parallel

implementations of the Q-learning algorithm with computer clustering architecture. The parallel Q-Learning (PQL) methods studied were the State Division Learning Method (SDLM), the Prioritized Field Learning Method (PFLM) and the Parallel Fuzzy Q-Learning (PFQL). A parallel algorithm with multi-agent learning using Q-learning is proposed in [27]. The algorithm is called PQL with Co-allocation of Storage and Processing (PCSP), and it uses a table partition strategy for sharing Q-table information among the processing nodes. The works presented in [24]–[27] have as the objective the improvement in the Q-learning processing using the High-Performance Computing (HPC), however this alternative has been identified as a costly solution regards to power consumption per processing.

Other works support using FPGA by presenting some advantages over other platforms for applications with time restrictions. In [12] a comparative performance study involving FPGA, GPU and CPU in image processing problems is conducted. Despite the possibility of using parallelism in multi-cores microprocessors, which improves performance for a large number of applications, cores are all grouped, and data transfer between them is very limited. Exclusively for some simple problems (e.g. naive algorithms) the GPU is able to achieve a performance similar to the FPGA. For more sophisticated algorithms (e.g. shared arrays), GPUs do not demonstrate the same performance since they have memory access limitations as a result from its architecture.

FPGAs provide hardware platforms suitable for deploying software algorithms [23]. From the theoretical basis presented, it is possible to conclude that the low execution time of FPGA devices in comparison with its software counterparts is the main reason for its use as a platform for the development of the Q-learning Reinforcement Learning Technique.

B. MAIN CONTRIBUTIONS

This work presents as contribution a hardware parallel architecture on FPGA of the Q-learning reinforcement learning technique. The main idea is based on the development of a modular and parallel architecture to enable an increase in the algorithm execution speed or lower power consumption by decreasing the clock frequency. The intrinsic properties of the FPGA, such as: flexibility and parallel processing, were fundamental to achieving this goal. The parallelization of the data flow on FPGA allows the Q-learning technique to be used in applications where there are a significant data flow and strict processing time restrictions. Another possibility of application of this architecture is in low consumption systems, where the system clock can be reduced in way to reduce the power consumption.

C. PAPER ORGANIZATION

This paper is organized as described in the following paragraphs.

In this first section a brief introduction was presented, in which the problem to be approached was contextualized. A bibliographical and state of the art review was also conducted as well as the main objectives to be achieved were presented.

In section II a theoretical foundation on reinforcement learning will be presented, exploring the main characteristics and advantages of the algorithm that was implemented in Hardware, the Q-learning.

In section III, a detailed description of the architecture development and implementation will be explained, describing the various modules used to build the algorithm in hardware.

In Section IV, the system validation will be performed by simulating few problems in the architecture presented, comparing with results obtained by simulating the same problems in software. The hardware synthesis analysis for different implementation scenarios was also performed, alongside the evaluation of parameters such as occupation area and throughput (or sampling frequency).

The section V will present the final considerations.

II. Q-LEARNING TECHNIQUE

This section aims to discuss the concepts and uses of reinforcement learning, emphasizing the technique used in this work, Q-learning.

Reinforcement learning is the maximization of numerical rewards by mapping events defined by states and actions [1]. The agent does not receive the information of what action to take as in other forms of machine learning, instead, it must find out which actions will produce the best reward in each state from interactions with the environment. As described in [1], the agent determines an action to be performed from situations encountered in the environment. The executed action transforms the environment and disturbs the state in the inverse of reaching the goal. Modifications are transmitted to the agent through a reward and next state.

The algorithm goal is to find a sequence of actions that determines an optimal policy, defined as the state mapping in actions that maximize the sum of the reinforcement values. Figure 1 summarizes the described agent-environment interaction process, where:

- s_k is a representation of the environment state where $s_k \in S$ and S is the set of possible states;
- a_k is an action representation, where $a_k \in A$ and A is the set of possible actions in the state s_k;
- r_{k+1} is a numerical reward, a consequence of the action a_k taken;
- s_{k+1} is the new state.

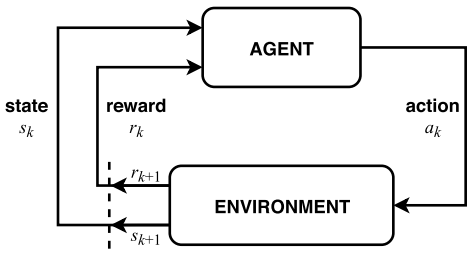

FIGURE 1. Interaction agent-environment in reinforcement learning [1].

Q-learning [28] is one of the learning techniques classified as an off-policy time difference method, since the convergence to optimal values of Q does not depend on the policy being used. The future reward function in the s state when performing an a action, denoted as $Q(s, a)$, is assimilated by interactions with the environment. It is considered as the most popular RL algorithm and has been proposed as a way to iteratively learn optimal policy when the system model is not known.

The equation for updating the value function of the state-action pairs $Q(s, a)$ is based on the value-action function expressed as:

$$Q_{k+1}(s_k^n, a_k^z) = Q_k(s_k^n, a_k^z) + \alpha[r(s_k^n, a_k^z) + \gamma \max(Q(s_{k+1}^n, a_{k+1}^z)) - Q_k(s_k^n, a_k^z)] \quad (1)$$

where

- k is the discretization instant, with sampling period T_s.
- s_k^n is the n-th environment state in the k-th iteration;
- a_k^z is the z-th action taken in the n-th state s_k^n also in the k-th iteration;
- $Q_k(s_k^n, a_k^z)$ is the accumulated result for the agent having chosen the action a_k^z in the state s_k^n in the instant k;
- $r(s_k^n, a_k^z)$ is the immediate reinforcement received in s_k^n for taking action a_k^z;
- s_{k+1}^n is the future state;
- $\max(Q(s_{k+1}^n, a_k^z))$ is the value Q corresponding to the maximum value function in the future state.
- α and γ are positive constants of value less than the unit that represent the learning coefficient and discount factor, respectively.

The learning coefficient determines to what extent new information will replace the previous ones, while the closest discount factor reduces the influence of immediate rewards and considers those in the long run. The reward function r indicates the immediate promising actions and the value function Q indicates the total accumulated gain. When the agent changes from a state to a future one, Q-learning updates the new Q value function estimate from the new state to the previous state.

A relevant aspect of the Q-learning reinforcement learning technique is that the choice of actions to be performed during the process of estimating the $Q(s, a)$ value function can be performed by any method of exploration/exploitation or even randomly. As demonstrated by [28], if each action-state pair is visited an infinite number of times the value function Q will converge with probability 1 to its optimal value, using a sufficiently small alpha learning coefficient.

Figure 2 shows the Q-learning algorithm pseudocode. It has as inputs the learning coefficient, the time discount rate and the reward function. It begins with the initialization of the Q values function matrix and initial state s_0. The algorithm chooses an action from among the possible ones for the current state and observes the next state and reward. The value of Q is updated, the new state is defined and then the process is repeated until it returns the updated Q matrix.

1: $Q \leftarrow Q_0(s_0^n, a_0^z)$
2: $s \leftarrow s_0$
3: **while** $k <$ steps limit **do**
4: Drawn an action a_k^z for the state s_k^n
5: Execute the action a_k^z
6: Observe the state s_{k+1}^n and update $Q_k(s_k^n, a_k^z)$ according with:
7: $Q_{k+1}(s_k^n, a_k^z) = Q_k(s_k^n, a_k^z) + \alpha[r(s_k^n, a_k^z) + \gamma \max Q(s_{k+1}^n, a_{k+1}^z) - Q_k(s_k^n, a_k^z)]$
8: $s_k \leftarrow s_{k+1}$
9: **end while**
10: **return** matrix Q

FIGURE 2. Q-learning algorithm.

Although the Q-learning convergence criterion requires state-action pairs to be visited infinite times, in practice it is possible to reach quite relevant values when executing a sufficiently large number of iterations (considering the task to be learned). For a problem of 18 states and 2 actions, as described in [2], where Q-learning was used to determine an optimum selection policy between beam conformation and power control of an adaptive arrangement of antennas, the matrix Q convergence happens after approximately 2500 iterations. In the work of [29], Q-learning was used to make adaptive thermal management of multicores systems, in order to improve the reliability and extend their useful life. In this second work, the RL algorithm learned the relation between the core mapping, the core frequency and its temperature, defining the thermal stress intervals as states and the threads, voltages and operating frequencies as actions. For a quad-core Intel, the system was modeled by 12 state and 8 actions, requiring a total of 5500 iterations for the Q value function convergence. In the work presented in [30] the Q-learning algorithm was used to determine the policy optimization of three controllers applied to a tank system in order to take advantage of the positive characteristic of each of them, thus optimizing the system output. In this case the error signal was discretized in 41 intervals, characterizing the admissible states, and the choice among one of the three controllers to the actions of the problem. A system with 41 state and 3 actions, randomly choosing the actions, converges to an optimal policy in approximately 8000 iterations.

III. IMPLEMENTATION DESCRIPTION

In this section the implementation details of the developed architecture are described. In III-A is presented an FPGA Hardware overview of the Q-learning architecture, where the notation used to describe the structure is defined. In the following topics, particularities of each of the system modules are discussed, detailing the mechanisms used for the hardware implementation of the algorithm shown in Figure 2.

A. PROPOSED ARCHITECTURE OVERVIEW

An overview of the developed hardware architecture is presented in Figure 3. The system receives the FPGA clock as

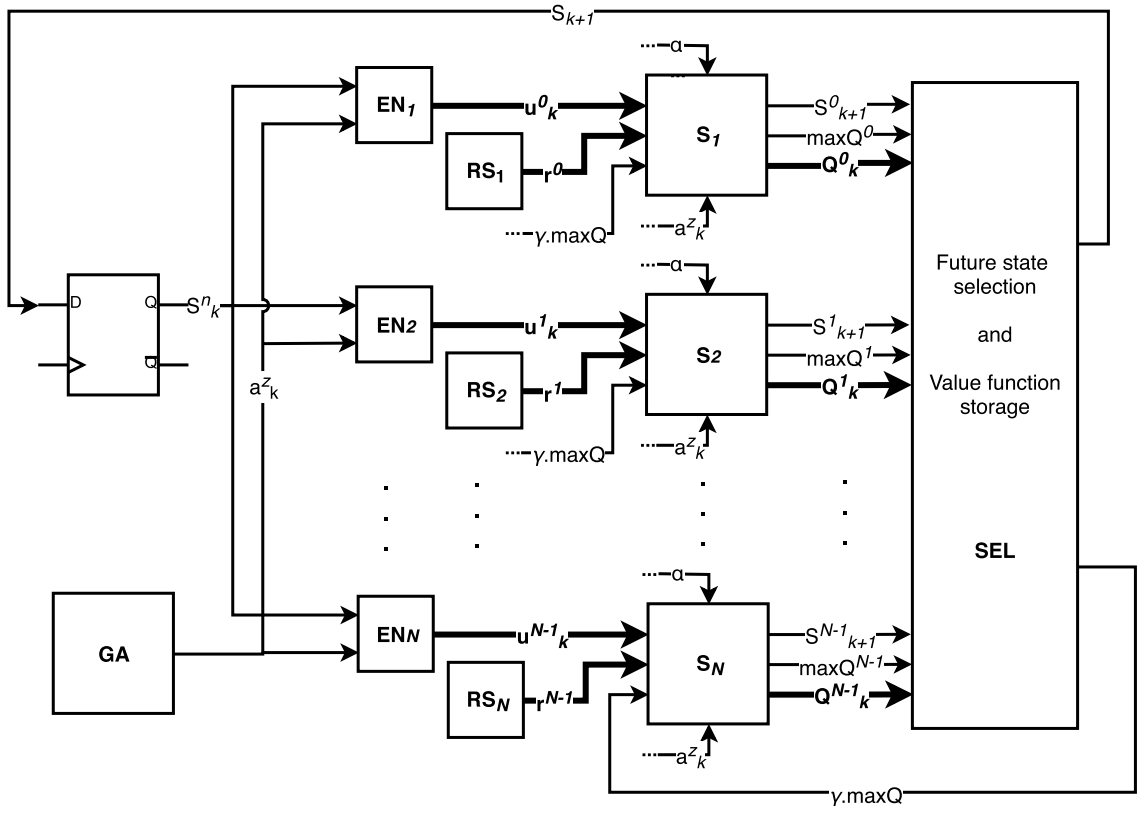

FIGURE 3. Overview of the proposed architecture.

input and the initial state value, s_0, must be randomly initialized in the REG1 register. The whole system is detailed from this diagram, where the main mechanisms of action choice, value function calculations, state-action pairs update and the mechanisms of future state selection will be explained. The system is designed to operate with N states and Z actions and therefore a combination of $N \times Z$ possible state-action pairs. The architecture was developed in an attempt to parallelize as much as possible the algorithm execution in order to decrease the Q-learning processing time.

The notation used in the figures is described below:

- k is the discretization time, with sampling time T_s, in which one can measure the transfer rate, known by throughput. The throughput (or sampling frequency) can be expressed as $F_s = 1/T_s$ in samples per seconds (Sps) or iterations per seconds (Ips);
- s_k^n is the n-th environment state in k-th iteration;
- a_k^z is the z-th action taken in the n-th state s_k^n also in the k-th iteration;
- $\mathbf{u}_k^n$ is a Z vectors elements that will enable or disable the registers that store the value assigned to the value function in the n-th state;
- $\mathbf{r}^n$ is a constant value vector with the immediate reinforcement for the Z actions of the n-th state;
- s_{k+1}^n is the n-th future state;
- max Q^n is the Q value corresponding to the action with higher reinforcement value in the state, updated at each iteration;
- $\mathbf{Q}_k^n$ is a vector containing the elements of the value function assigned to the Z actions of the n-th state;
- α and γ are positive constants of value less than the unit that represent the learning coefficient and the discount factor, respectively.

The architecture is composed by five main modules types: The GA module, responsible for randomly choosing the actions of the algorithm; The EN modules, which determine which state-action pair should be updated; The RS modules, responsible for storing the reward values; The S modules, responsible for the calculation of the Q value function; And the SEL module, where the future state selection and the storage of the Q value function are made. Each of the system modules is detailed individually in the following sections.

B. GA - ACTION DRAW

As seen in the Q-learning pseudo-code shown in Figure 2 (Section II) it is necessary to draw the a_k^z action for the s_k^n state. For this purpose, a Pseudo Random Number Generator (PRNG) was implemented. The generator draws from all possible actions (0 to $Z-1$) what action will be taken. Each z-th action is formed by one word of $\log_2(Z)$ bits. The first s_0 state is randomly initialized, among all possible states (0 through $N-1$), in the REG1 register, and has a size of $\log_2(N)$ bits.

The pseudo random number generator is the starting point for executing the algorithm. From the second iteration, the following states are defined by feedback, as a

consequence of the system actions and the actions continue to be randomly defined.

For the algorithm convergence, it is necessary for all state-action pairs to be visited a sufficiently large (ideally infinite) number of times. And for this it's used the pseudo-random number generator based on the numerical congruence described in [31]. The expression used to implement the PRNG is presented as

$$a_k^z = P_1 \times a_{k-1}^z + P_2 \ (\text{mod}\ Z), \tag{2}$$

where the values P_1 and P_2 are constants, a_k^z is an integer between 0 and $Z-1$ and a_{k-1}^z is the value from the previous instant of the pseudo-random number series. The internal architecture of the random number generator is illustrated in Figure 4.

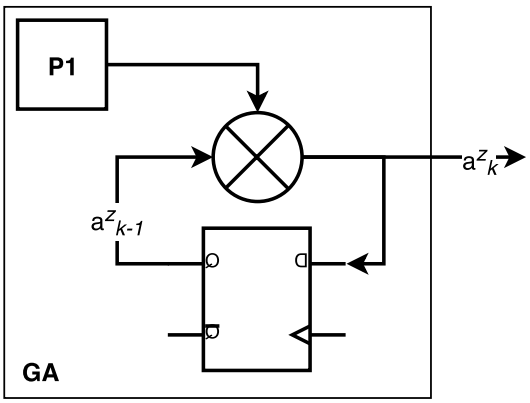

FIGURE 4. Pseudo random number generator architecture.

Here, the constant P_2 was adopted as zero. The modulo operation (mod Z) is performed from an intrinsic multiplier overflow function, the wrap-around (i.e. values that exceed the maximum number of bits are bypassed within the representable range by saving only the least significant bits).

However, problems can occur where the number of possible actions are not multiple of two. To solve this limitation, an artifice shown in Figure 5 was adopted. A PRNG with a number of possible combinations ($N \circ \max$) much larger than the number of desired actions is utilized. Then, the interval containing all combinations is divided into Z equal intervals, where Z is the desired number of actions. If the number drawn is $0 < x < C_1$ the action a_k^z will be a_k^0. In case the number drawn is $C_1 < x < C_2$ the action a_k^1 and so on until $x > C_{Z-1}$, when the action will be a_k^{Z-1}. This division is made using comparators and combinational logic.

Once determined the action-state pair at each iteration, it is known which of the elements of the value function matrix $Q_k(s_k^n, s_k^z)$ must be updated. As the actions are randomly chosen (pseudo-random), the architecture performs only the exploration (training) of the environment by the agent to obtain the optimal policy, not worring about the exploitation.

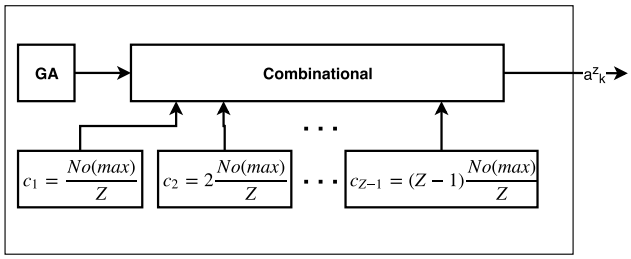

FIGURE 5. Hardware implemented of the random number generator.

C. EN - UPDATE MODULE

The update modules called EN are responsible for selecting which state-action pair (s_k^n, a_k^z) will be updated. Each n-th module, ENn, is a combinational logic block that has as inputs the k-th state, s_k^n, action a_k^z, and its output is a vector of Z elements, here denoted as $\mathbf{u}_k^n$ and represented by

$$\mathbf{u}_k^n = \begin{bmatrix} u_k^{n,0} \\ u_k^{n,1} \\ \vdots \\ u_k^{n,Z-1} \end{bmatrix} \tag{3}$$

where $u_k^{n,z}$ is a bit that when at high logic level represents that the matrix element of the Q value function referring to the n-th state and the z-th action must be updated. Considering the outputs of all N modules, there are $N \times Z$ outputs, the same number of state-action pairs. However, only one output from one of these modules will have high logic level for each iteration. The logical operation that determines $\mathbf{u}_k^n$ is expressed as

$$\mathbf{u}_k^n = \begin{cases} (1 >> a_k^z) || A & \text{if } s_k^n = n \\ A \end{cases} \tag{4}$$

where $>>$ is a logical shift operator to the right and A is a Z zeros vector.

The values for each value function $Q_k(s_k^n, s_k^z)$ action-state pair are stored in $N \times Z$ registers which have their enable inputs connected to the outputs of the EN modules. Therefore, the value function is only updated when one of the elements of one of the n-th $\mathbf{u}_k^n$ vectors is at the high logic level.

D. RS - REWARD FUNCTION MODULE

The values of the immediate reinforcements, or reward, are stored in the N modules called RS. The reward function $\mathbf{r}^n$ is a vector of Z elements, represented as

$$\mathbf{r}^n = \begin{bmatrix} r^{n,0} \\ r^{n,1} \\ \vdots \\ r^{n,Z-1} \end{bmatrix}, \tag{5}$$

which indicates the immediate promising actions in that state. Each n-th RS module has Z constant, $r^{n,z}$, associated with each of the Z actions of the n-th state. These constants express

the goal the agent wants to achieve. Each z-th $r^{n,z}$ variable consists of a word of B bits. Actions leading to the target state have a positive numerical $r^{n,z}$ reinforcement value. Undesired actions in the state receive a negative numeric boost $r^{n,z}$. The actions that lead to other states receive $r^{n,z} = 0$.

E. S - VALUE FUNCTION CALCULATION MODULE

The Q-learning hardware architecture, as observed in the main diagram shown in Figure 3, is paralleled regarding its states (s_k^n). The nth-state module, Sn, is subdivided into two other different functions modules. Its configuration is illustrated in Figure 6.

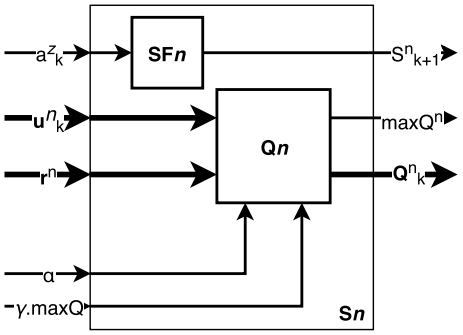

FIGURE 6. Sn module architecture.

In the SFn module, the future state, s_{k+1}^n, is determined locally, from the draw action information only, a_k^z. In the Qn module, the calculation of the value function vector elements Q_k^n is performed, and the value function corresponding to the action with the highest value, maxQ^n, is determined for the n-th state.

1) Qn - VALUE FUNCTION CALCULATION

Each n-th module Qn computes the vector

$$\mathbf{Q}_k^n = \begin{bmatrix} Q_k^{n,0} \\ Q_k^{n,1} \\ \vdots \\ Q_k^{n,Z-1} \end{bmatrix}. \tag{6}$$

$\mathbf{Q}_k^n$ is a vector of Z elements, where each $Q_k^{n,z}$ element is formed by B bits. The set of N vectors, $\mathbf{Q}_k^n$, forms the matrix value function $Q_k(s_k^n, a_k^z)$ that can be expressed as

$$\mathbf{Q}_k(s_k^n, a_k^z) = \begin{bmatrix} \mathbf{Q}_k^0 & \mathbf{Q}_k^1 & \dots & \mathbf{Q}_k^{N-1} \end{bmatrix} = \begin{bmatrix} Q_k^{0,0} & Q_k^{1,0} & \dots & Q_k^{N-1,0} \\ Q_k^{0,1} & Q_k^{1,1} & \dots & Q_k^{N-1,1} \\ \vdots & \vdots & \dots & \vdots \\ Q_k^{0,Z-1} & Q_k^{1,Z-1} & \dots & Q_k^{N-1,Z-1} \end{bmatrix}. \tag{7}$$

The inputs for this module are the enable vector $\mathbf{u}_k^n$, the $\mathbf{r}^n$ reward function, the learning coefficient α and the Q value corresponding to the action with the greatest future reinforcement value in the future discounted state of γ (γ.max$Q$).

Part of the internal architecture of the Sn modules is illustrated in Figure 7, which is also a parallel architecture, paralleled regarding the system actions. At each iteration the Q matrix is updated. In order to do so, $2 \times Z$ adders (SUM1nz and SUM2nz), Z subtractors (SUBnz), and Z multipliers (MULTnZ) are used in each of the n-th Sn modules that implement Equation 1 which is the fundamental Q-learning equation. Additionally, Z registers are used for the $\mathbf{Q}_k^n$ storage.

In addition to calculating the value function, in module Sn is also calculated the Q value corresponding to the action with the highest value in the n-th state. This variable is illustrated in Figure 7 as maxQ^n and it is obtained from the comparison (COMPn) of all z-th elements of vector $\mathbf{Q}_k^n$ in the n-th state.

2) SFn - FUTURE STATE

There is also a third functionality implemented in the Sn module. In it is determined what would be the future S_{k+1}^n status for the a_k action drawn by GA. The structure shown in Figure 8 represents the portion of the architecture of the module Sn, the internal module SFn, responsible for executing this functionality. Since it is a parallelized architecture, a future state is determined in each of the N modules Sn taking into account only the information of the action a_k drawn. In the SEL module it is decided which n-th future state S_{k+1}^n will continue in the algorithm and become the current state in the next iteration.

Therefore, the Sn block delivers three information to the system: the value function vector for the n-th state $\mathbf{Q}_k^n$ actions, the value function correspondent to the action with the highest value maxQ^n and the n-th future state $s_{k+1}^n(s)$ determined from the action taken.

F. SEL - FUTURE STATE SELECTION MODULE

The last module from the architecture is the SEL module. Its structure is shown in Figure 9. It is the algorithm junction point, where information parallel computed in previous modules meet. It is in this module that it is determined which will be the next state to be explored by the architecture. It is also where the action with greater value in the future state max$Q$ is determined and where the N vectors $\mathbf{Q}_k^n$ are assembled to construct the system value function matrix $Q_k(s_k^n, a_k^z)$.

In order to determine the future state s_{k+1}, all n-th future s_{k+1}^n states from the N modules Sn are placed in a MUX2 multiplexer, which selects the current state s_k. This future state value is fed back to the beginning of the architecture and becomes the current state in the next iteration.

In an effor to determine the action with the highest value in the future state max$Q$, the future state s_{k+1} is used as the selector of the MUX3 multiplexer that has as input the N actions with the highest value of the N states from the maxQ^n architecture. The value of max$Q$ is multiplied by the time discount factor γ and fed back to the inputs of the Sn modules for the calculation of the $\mathbf{Q}_k^n$ vectors that make up the value function.

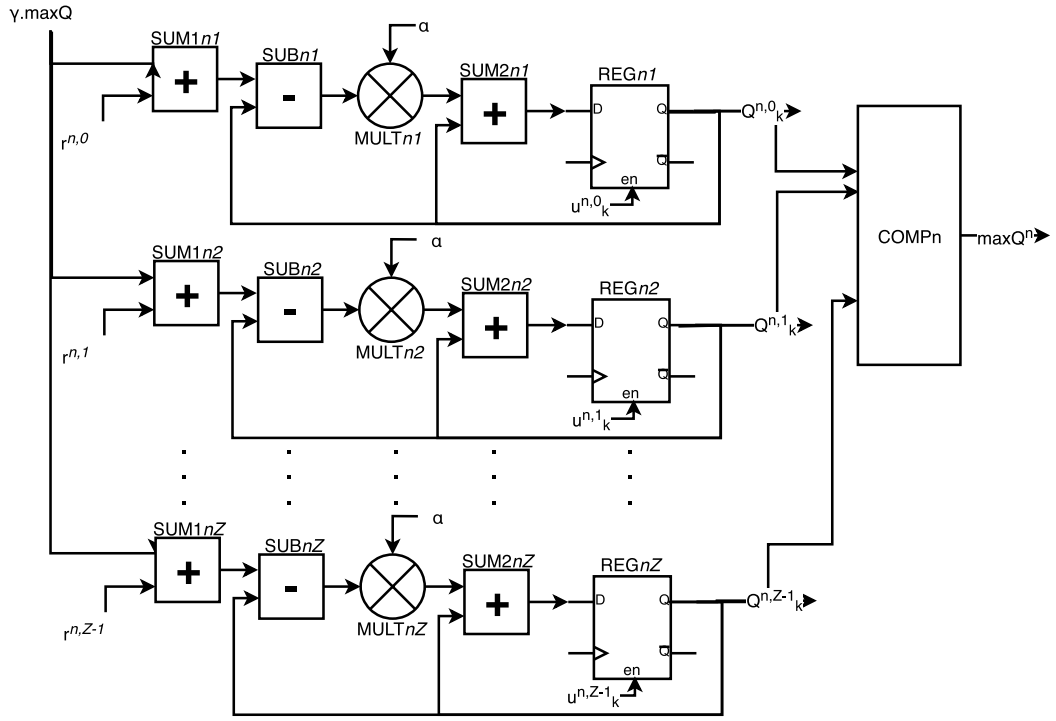

FIGURE 7. Q*n* module architecture - function value calculation.

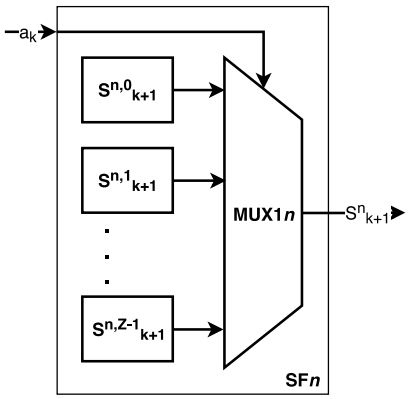

FIGURE 8. SF*n* module architecture - future state choice.

IV. RESULTS

In this section, simulation and hardware synthesis results for the architecture proposed in this work are presented. Simulations and syntheses for different scenarios were carried out and the numbers of states and actions were varied. All scenarios were simulated and synthesized for different bits resolutions. The simulation results were used to validate the hardware architecture and to evaluate the resolution error from the bits number. The synthesis results allowed the system analysis regarding important parameters for the design of hardware architectures, such as occupation rate and sampling time.

The applicability of the proposed architecture to real problems is also analyzed in this section. Applications found in the literature, using the Q-learning algorithm for training the agent were synthesized on FPGA in order to obtain the throughput, F_s, and the time of convergence for the optimal policy.

A. SIMULATION RESULTS

For the the Q-learning algorithm architecture simulation and validation, a scenario in which a robot moved in an arena, aiming to reach a certain region of it was analyzed. The number of states in this problem represents the granularity of the arena, while the actions represent the robot possible movements. The arena was divided into six regions (states: s_1, s_2, s_3, s_4, s_5 and s_6) and the robot had four possible directions of movement (actions: a_1 - up, a_2 - down, a_3 - left, a_4 - right). The problem described is illustrated in Figure 10 and was simulated with fixed-point digital representation for five different resolutions. The notation used is $[n.b]$ where n is the total number of bits, b bits represent the fractional part and $(n - b)$ bits represent the integer part.

It was desired that the agent could reach room 6 (s_6), regardless of the room in which it was in. To define room 6 as a goal, an immediate $r^{n,z} = 100$ reinforcement was associated with actions that directly lead to the desired region. Blue arrows were used in Figure 10 to illustrate these actions. If the agent performed an action that resulted in collision with the edges of the arena, it received an immediate negative reward ($r^{n,z} = -500$). These actions are represented by red arrows. In all other transitions, the received reward is zero

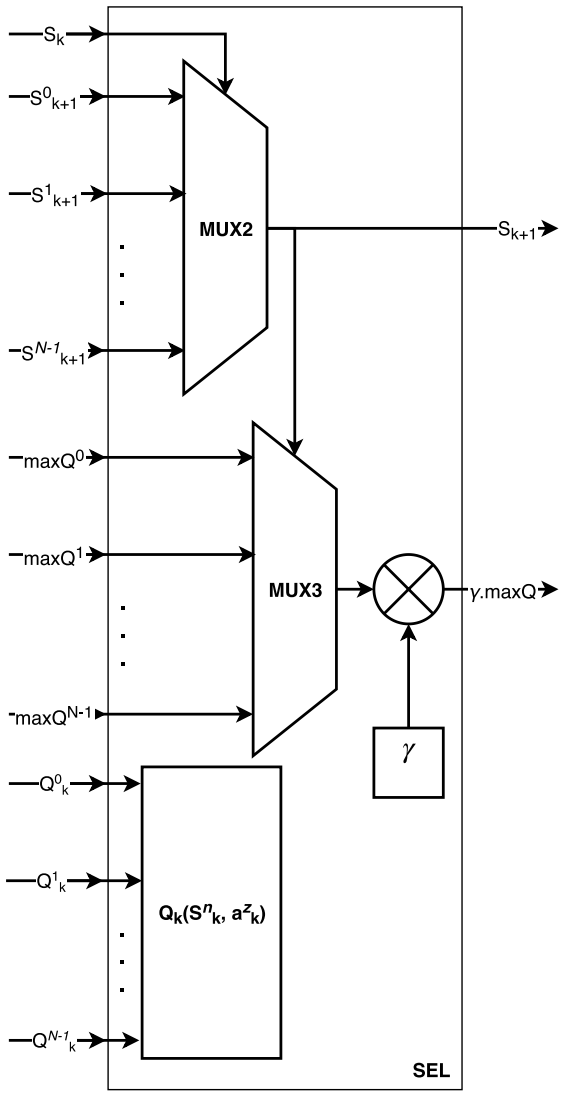

FIGURE 9. SEL module architecture.

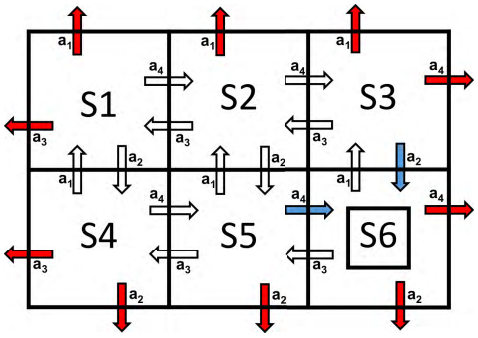

FIGURE 10. Example used for simulation and validation of the hardware implementation.

$(r^{n,z} = 0)$, represented by the white arrows. The $\mathbf{r}$ matrix

$$\mathbf{r} = \begin{bmatrix} -500 & 0 & -500 & 0 \\ -500 & 0 & 0 & 0 \\ -500 & 100 & 0 & -500 \\ 0 & -500 & -500 & 0 \\ 0 & -500 & 0 & 100 \\ 0 & -500 & 0 & -500 \end{bmatrix} \quad (8)$$

shows all rewards for all state-action pairs, where states $(s_1, s_2, s_3, s_4, s_5, s_6)$ are represented in the array rows, while the (a_1, a_2, a_3, a_4), are represented by the columns. Each element of the $\mathbf{r}$ array is formed by one word of $[n.b]$ bits.

The numerical results of the function value $Q_k(s_k^n, a_k^z)$ after the developed architecture simulation were compared with results obtained through an Matlab floating point implementation, IEEE 754 standard. To simulate the example described, a learning coefficient $\alpha = 0.8$ and a discount rate $\gamma = 0.8$ were used as parameters. These parameters were used both in the Matlab floating-point simulation and in the parallel hardware architecture simulation performed using System Generator.

The floating-point value function matrix in the IEEE 754 standard is shown below.

$$\mathbf{Q}(s_k^n, a^z) = \begin{bmatrix} -357.8 & 177.8 & -357.8 & 177.8 \\ -322.2 & 222.2 & 142.2 & 222.2 \\ -277.8 & 277.8 & 177.8 & -277.8 \\ 142.2 & -322.2 & -322.2 & 222.2 \\ 177.8 & -277.8 & 177.8 & 277.8 \\ 222.2 & -322.2 & 222.2 & -322.2 \end{bmatrix} \quad (9)$$

The hardware architecture simulation results are shown in Table 1. The hardware architecture was simulated with digital fixed-point representation in four different scenarios. In the first scenario, 24 bits were used, 14 bits for the binary part (Table 1(a)). In the second, 20 bits, being 10 for the binary part (Table 1(b)). In the third one, 16 bits with 6 bits in the binary part were used (Table 1(c)). In the last scenario, 12 bits were used with 2 bits in the binary part (Table 1(d)).

After the simulations, it was possible to observe from the optimal policy results obtained, the lower the number of bits, the greater the resolution error (e) obtained regarding the floating point. However, it is important to emphasize that the resolution of the Q matrix is not as significant as long as its optimal policy is well defined. Figure 11 is the representation of the value function obtained in the floating-point simulation from a color matrix where the values of the actions are codified in a linear scale from -400 to 400. The smallest value of the scale is represented by the darkest blue while the highest value is represented by lightest shade or red. Figure 12 is the representation, in color matrices, of the functions values obtained from the fixed-point architecture simulation using the same scale as reference. From this it is possible to observe that, despite the resolution error associated with the decrease on the number of bits, the policy is well characterized. Even in the worst simulated case, with only 12 bits of resolution where an error greater than 30% was obtained, the best and worst actions, despite having different colors of the reference matrix, are well defined and present the same policy when compared with the floating point and with the other cases simulated in different resolutions.

The resolution error can always be improved by increasing the number of bits. However, as will be detailed in the following sections, this directly implies the increase of the

TABLE 1. Value function for different binary representations.

$$\mathbf{Q} = \begin{bmatrix} -357.8 & 177.8 & -357.8 & 177.8 \\ -322.2 & 222.2 & 142.2 & 222.2 \\ -277.8 & 277.8 & 177.8 & -277.8 \\ 142.2 & -322.2 & -322.2 & 222.2 \\ 177.8 & -277.8 & 177.8 & 277.8 \\ 222.2 & -322.2 & 222.2 & -322.2 \end{bmatrix}$$

(a) [24.14] bits and $e = 0.01\%$

$$\mathbf{Q} = \begin{bmatrix} -358.0 & 177.5 & -358.0 & 177.5 \\ -322.5 & 222.0 & 142.0 & 222.0 \\ -277.0 & 277.5 & 177.5 & -278.0 \\ 142.0 & -322.5 & -322,5 & 222,2 \\ 177.5 & -278.0 & 177.5 & 277.5 \\ 222.0 & -322.5 & 222.0 & -322.5 \end{bmatrix}$$

(b) [20.10] bits and $e = 0.17\%$

$$\mathbf{Q} = \begin{bmatrix} -361.5 & 173.9 & -361.5 & 173.9 \\ -326.1 & 218.2 & 138.5 & 218.2 \\ -281.8 & 273.9 & 173.9 & -281.2 \\ 138.5 & -326.1 & -326.1 & 218.2 \\ 173.9 & -281.8 & 173.9 & 273.9 \\ 218.2 & -326.1 & 218.2 & -326.1 \end{bmatrix}$$

(c) [16.6] bits and $e = 2.61\%$

$$\mathbf{Q} = \begin{bmatrix} -405.0 & 127.2 & -405.0 & 127.2 \\ -372.7 & 170.0 & 95.0 & 170.0 \\ -330.0 & 227.2 & 127.2 & -330.0 \\ 95.0 & -372.7 & -372.7 & 170.0 \\ 127.2 & -330.0 & 127.2 & 227.2 \\ 170.0 & -372.7 & 170.0 & -372.7 \end{bmatrix}$$

(d) [12.2] bits and $e = 33.20\%$

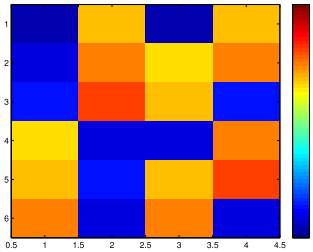

FIGURE 11. Color matrix for function value obtained at floating point representation.

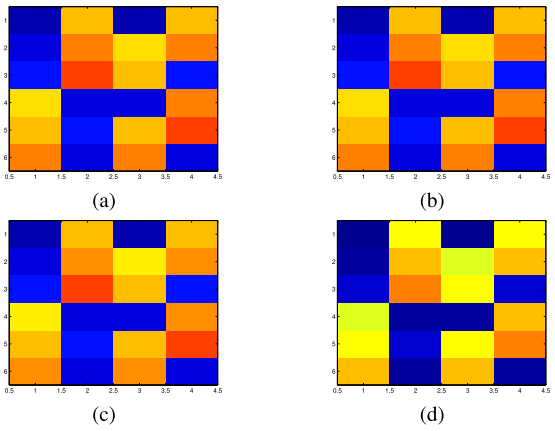

FIGURE 12. Color matrices for value functions obtained at fixed-point representation. (a) [24:14] bits. (b) [20:10] bits. (c) [16:06] bits. (d) [10:00] bits.

occupied area in the FPGA and increase in the processing time.

B. SYNTHESIS RESULTS

For the hardware architecture synthesis analysis, in addition to the scenario presented in Figure 10, nine other distinct scenarios were analyzed, with different numbers of states and actions. The scenarios were characterized using also as problem the robot, described in section IV-A, that moves in an arena. The greater the granularity of the arena, the greater the number of states. Six scenarios were determined for four possible actions: a_1 - up, a_2 - down, a_3 - left, a_4 - right, where the agent moves from a position in the direction indicated by the action. Other four scenarios were characterized for eight possible actions: a_1 - up, a_2 - down, a_3 - left, a_4 - right, a_5 - up2, a_6 - down2, a_7 - left2, a_8 - right2, where the behavior of the agent is the same as in the previous scenarios for the first four actions while in the other actions it moves two positions in the direction indicated by the action. The size of all examples implemented and simulated are presented in Table 2, where N represents the number of states and Z the number of actions. The parameters used in the tests of these 10 scenarios are shown in Table 3. All results were obtained for the Xilinx Virtex-6 FPGA [15].

TABLE 2. Synthetized cases.

Case	I	II	III	IV	V	VI	VII	VIII	IX	X
N	6	12	12	20	20	30	30	56	56	132
Z	4	4	8	4	8	4	8	4	8	4

TABLE 3. Synthesis parameters.

Parameters	Values
Number of States	N
Number of Actions	Z
Learning Coefficient (α)	0.8
Discount rate (γ)	0.8
Digital Representation	Fixed-point
Number of Bits	$[n.b]$

Figure 13 shows the hardware setup for the experiments. It was used the Virtex-6 FPGA ML605 Evaluation Kit by Xilinx [15], [32]. The architecture was developed

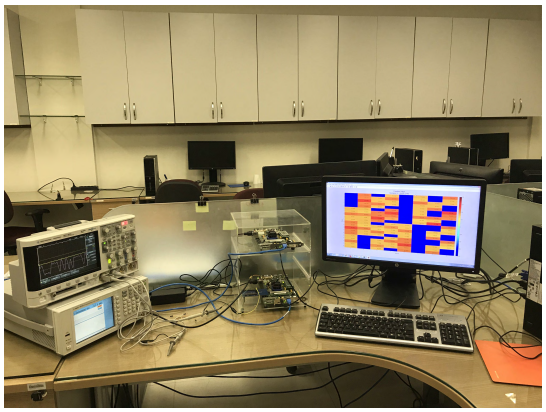

FIGURE 13. Hardware setup for the experiments with the Xilinx Virtex-6 FPGA ML605 Evaluation Kit.

TABLE 4. Hardware synthesis - Scenario I ($N = 6$, $Z = 4$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	34 (4%)	788 (< 1%)	2597 (2%)	23.02	5
[20.10]	34 (4%)	669 (< 1%)	2159 (1%)	21.80	4
[16.06]	34 (4%)	548 (< 1%)	1734 (1%)	23.54	3
[10.00]	34 (4%)	367 (< 1%)	1086 (< 1%)	26.42	3

TABLE 5. Hardware synthesis - Scenario II ($N = 12$, $Z = 4$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	106 (13%)	1509 (< 1%)	5070 (3%)	20.83	8
[20.10]	58 (7%)	1270 (< 1%)	4222 (3%)	22.23	6
[16.06]	58 (7%)	1029 (< 1%)	3387 (2%)	22.27	6
[10.00]	58 (7%)	668 (< 1%)	2133 (1%)	24.76	6

using structural modeling with the System Generator for DSP™ [14]. The system generator is the architecture-level design tool used to create high-performance algorithms on Xilinx devices using Matlab/Simulink (license number 1080073) [33] together with the Xilinx Vivado Design Suite (license number 505318) [34].

All scenarios were synthesized with all variables at fixed-point. The Tables 4 - 13 illustrate the results obtained both in terms of occupancy rate and throughput, F_s, in Mega-Samples per second (MSps) or Mega-Iterations per second (MIps). In the first columns is indicated the bit resolution synthesized for the variables $\mathbf{r}^n$ and $\mathbf{Q}_k^n$, the other variables have its resolution fixed. For this synthesis analysis, four different representations were implemented.

TABLE 6. Hardware synthesis - Scenario III ($N = 12$, $Z = 8$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	202 (26%)	2661 (< 1%)	10379 (7%)	18.08	15
[20.10]	106 (13%)	2230 (< 1%)	8639 (5%)	20.71	15
[16.06]	106 (13%)	1797 (< 1%)	6960 (4%)	20.67	10
[10.00]	106 (13%)	1148 (< 1%)	4415 (3%)	23.45	10

TABLE 7. Hardware synthesis - Scenario IV ($N = 20$, $Z = 4$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	170 (22%)	2470 (< 1%)	8349 (5%)	17.82	14
[20.10]	90 (11%)	2071 (< 1%)	6966 (4%)	19.87	9
[16.06]	90 (11%)	1670 (< 1%)	5594 (3%)	20.16	10
[10.00]	90 (11%)	1069 (< 1%)	3541 (2%)	21.23	9

TABLE 8. Hardware synthesis - Scenario V ($N = 20$, $Z = 8$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	330 (42%)	4390 (1%)	17188 (11%)	16.74	23
[20.10]	170 (22%)	3671 (1%)	14373 (9%)	15.96	20
[16.06]	170 (22%)	2950 (< 1%)	11561 (7%)	17.66	21
[10.00]	170 (22%)	1869 (< 1%)	7321 (4%)	19.79	20

TABLE 9. Hardware synthesis - Scenario VI ($N = 30$, $Z = 4$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	250 (32%)	3670 (1%)	12379 (8%)	16.57	19
[20.10]	130 (16%)	3071 (1%)	10299 (6%)	20.19	13
[16.06]	130 (16%)	2470 (< 1%)	8272 (5%)	19.67	13
[10.00]	130 (16%)	1569 (< 1%)	5206 (3%)	21.29	13

C. ANALYSIS OF HARDWARE OCCUPATION RESULTS

In Tables 4 - 13, the second column was used to display the number of multipliers used. The multipliers are used

TABLE 10. Hardware synthesis - Scenario VII ($N = 30$, $Z = 8$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	490 (63%)	6550 (2%)	25627 (17%)	15.53	32
[20.10]	250 (32%)	5471 (2%)	21406 (14%)	16.48	25
[16.06]	250 (32%)	4390 (1%)	17208 (11%)	16.45	26
[10.00]	250 (32%)	2769 (< 1%)	10915 (7%)	16.29	28

TABLE 11. Hardware synthesis - Scenario VIII ($N = 56$, $Z = 4$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	490 (59%)	6792 (2%)	23117 (15%)	15.90	55
[20.10]	234 (30%)	5673 (2%)	19252 (14%)	15.59	28
[16.06]	234 (30%)	4552 (1%)	15526 (10%)	15.15	29
[10.00]	234 (30%)	2875 (< 1%)	9808 (6%)	20.20	28

TABLE 12. Hardware synthesis - Scenario IX ($N = 56$, $Z = 8$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	746 (97%)	12248 (4%)	90917 (60%)	13.75	62
[20.10]	378 (49%)	10153 (3%)	55573 (36%)	14.89	54
[16.06]	378 (49%)	8136 (2%)	48536 (32%)	14.41	47
[10.00]	378 (49%)	5115 (2%)	28849 (19%)	17.88	47

TABLE 13. Hardware synthesis - Scenario X ($N = 132$, $Z = 4$).

$[n.b]$ bits	Multipliers	Registers (flip-flops)	LUTs	F_s (MSps)	Power (mW)
[24.14]	730 (95%)	15958 (5%)	142778 (94%)	12.23	91
[20.10]	370 (48%)	13175 (4%)	77574 (51%)	13.40	78
[16.06]	370 (48%)	10533 (3%)	70311 (46%)	13.35	67
[10.00]	370 (48%)	6626 (2%)	40764 (27%)	14.00	60

in the PRNG (GA module), as well as in the S modules where the value function is calculated, specifically for the multiplication of the learning coefficient (α) (MULTnz) and for the multiplication of the factor (γ) for the actions with the highest future reinforcement value (max$Q$) (SEL module). An increase in the number of multipliers, in all scenarios, is observed for configurations larger than 20 bits. This is due to the size of the hardware multipliers built into the used FPGA, which is of 48 bits. Therefore, arithmetic operations performed with a multiplier now need more than one unit. All the multiplications carried out in the first 8 scenarios (I-VIII) were implemented through embedded multipliers (DSP48E1) from the Virtex-6 FPGA. In scenarios IX and X, due to a greater systems complexity, some multiplication operations were implemented through logical cells (lookup table - LUTs) as the chosen FPGA did no have enough embedded multipliers to implement these scenarios when synthesized in higher resolution.

The third column displays the number of registers for the implementation. The area occupied by the registers is due to the storage of the $Q_k(s_k^n, a_k^z)$ (REGnz) value function which is calculated for each of the state-action pairs. Registers have also been used to store the value of the function corresponding to the action with the highest value maxQ^n for each state and to store the future state (s_{k+1}) during an iteration before it is fed back to the beginning of the architecture and becomes the current state (s_k)(REG1).

The fourth column shows the number of logical cells used. The occupation of the logical cells is related to the arithmetic operations implemented in the N S blocks to enable the calculation of the value function Q. In the more complex scenarios, IX and X, this occupation is directly related to the use of LUTs to carry out the multiplication operations in place of the embedded multipliers.

Figures 14, 15 and 16 illustrate the occupation varying with the states and actions for the scenarios already characterized. Figures 14(a), 15(a) and 16(a) illustrate the occupation for the highest synthesized resolution, [24.14] bits. Figures 14(b), 15(b) and 16(b) were constructed from the lowest resolution data, [10.0] bits. It is possible to notice that the occupied area increases almost linearly with the number of state-action pairs. The discontinuities present in Figures 14 and 16 therefore appear when replacing some embedded multipliers by LUTs, in the scenarios of greater complexity, consequently increasing the proportional number of LUTs and decreasing the number of multipliers. When comparing the occupied area for the case of higher resolution with the case of lower resolution, it is possible to observe that the behavior is practically the same regarding the area occupied by multipliers (Figure 14) and by registers (Figure 15). However, it is possible to notice that for LUTs this does not happen. The growth in LUTs is much higher when the number of bits is increased and LUTs are used for multiplication operations in place of the embedded multipliers.

It is observed that occupation area is determined both by the number of bits and by the complexity of the problem, i.e. the higher the resolution used in the problem and the more state-action pairs (n, z) the more space occupied in the FPGA. In situations where it is necessary to reduce the FPGA

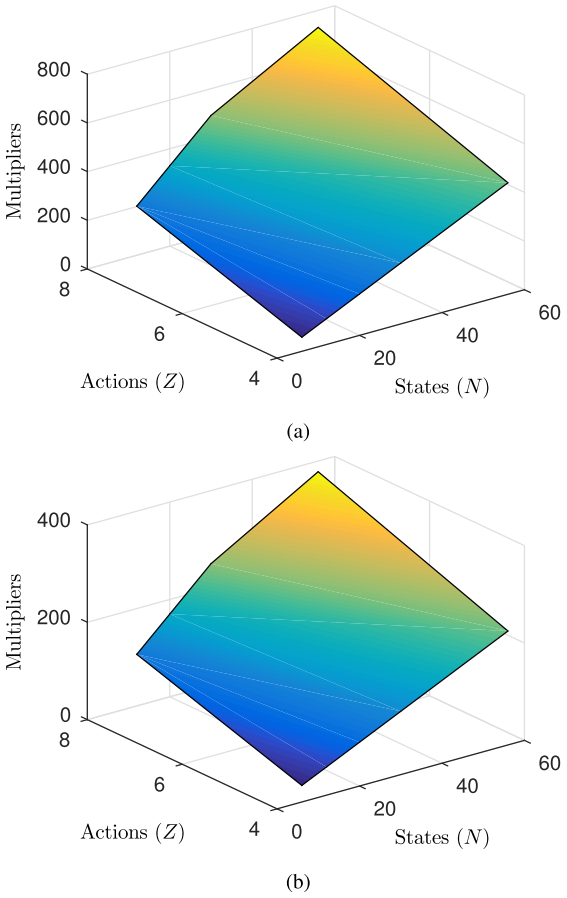

FIGURE 14. Occupied area in multipliers for different scenarios. (a) [24.14] bits. (b) [10.00] bits.

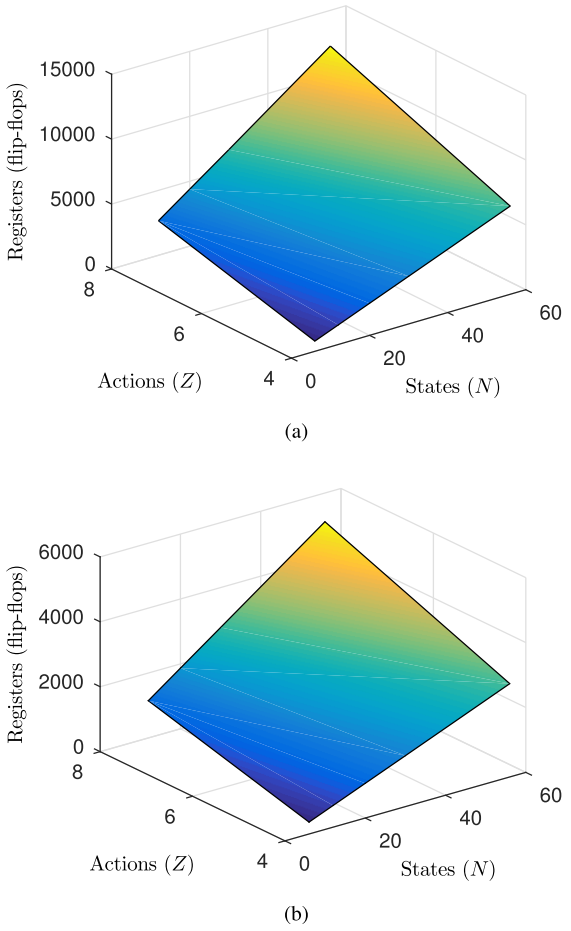

FIGURE 15. Occupied area in register for different scenarios. (a) [24.14] bits. (b) [10.00] bits.

occupancy, it is possible to modify the resolution, because as demonstrated in section IV-A (Figure 12), it is possible to obtain the optimal policy even though there is an associated resolution error.

D. ANALYSIS OF SAMPLING RESULTS

In Tables 4 - 13, the throughput, F_s, is observed in before the last column. Since the system was developed in such a way as to parallelize the data stream to the maximum, the sampling rate does not vary significantly, maintaining a very high throughput, F_s, in the order of 10 MSps.

From the sampling data, a decrease in the maximum throughput is observed with the increase in the number of states of the problem. It is also possible to note that this throughput decreases with the increase in the number of possible actions per state. This is explained by the increase in the complexity of the problem which results in a greater amount of data to be processed. Figures 17(a) and 17(b) present the throughput for different state numbers and actions with [24.14] and [10.00] bits, respectively. As the system was developed in such a way as to parallelize data flow to the maximum, this variation is not so significant. Since even increasing the number of state-action pairs, the path traveled during the processing of the information is not significantly modified. A small reduction in the throughput, F_s, occurs because by increasing the number of actions per state, it is necessary to make the value function $Q(s, a)$ comparison (COMP n) of all possible actions in order to determine the value function maximum $\max Q(s_{t+1}, a)$ for the best action in the nth-state. By increasing the number of states, consequently the number of multiplexers inputs that select the next state (MUX1 n) in the modules S n also grows. The same occurs in the multiplexers that select the next state (s_{t+1}) (MUX2), and the maximum function value of the future state $\max Q(s_{t+1}, a)$ (MUX3). These are the main bottlenecks for the complete parallelization of the system and the factors that influence the decrease of the sampling rate as the complexity of the problem increases. Despite bottlenecks, it is observed that between the biggest ($N = 132, Z = 4$) and the smallest case ($N = 6, Z = 4$) the variation of the sampling period is less than 50 ns.

E. POWER CONSUMPTION ANALYSIS

Tables 4 - 13 show the dynamic power (last column) for each of the scenarios and the four different representations

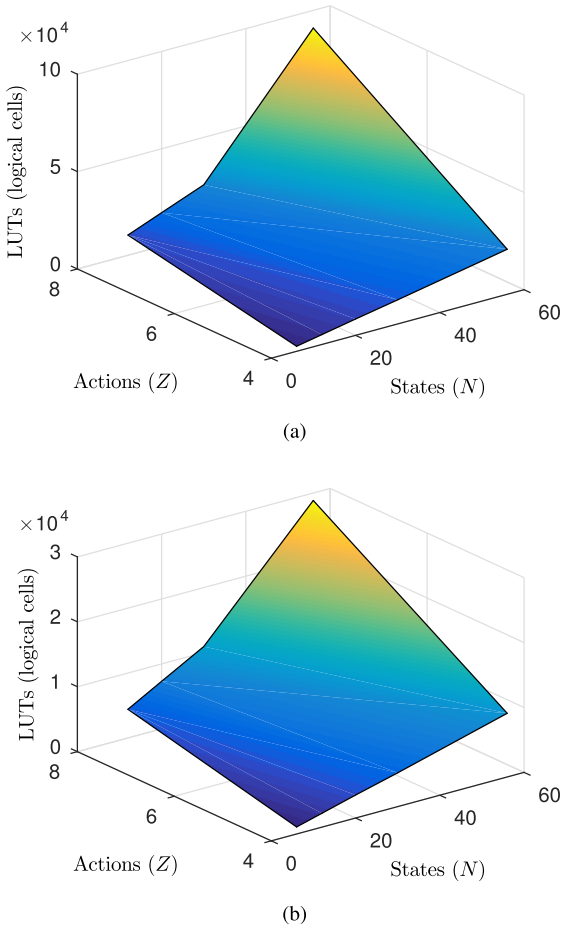

FIGURE 16. Occupied area in LUTs for different scenarios. (a) [24.14] bits. (b) [10.00] bits.

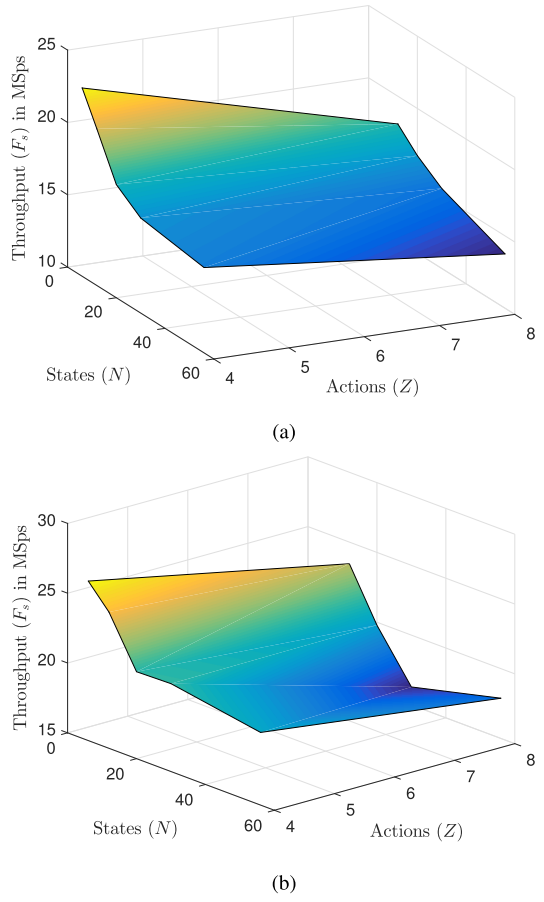

FIGURE 17. Throughput (F_S) in MSps for different scenarios. (a) [24.14] bits. (b) [10.00] bits.

in bits (first column). Figure 18 shows the dynamic power as a function of the number of bits for all scenarios (see Table 2), and it is possible to observe that the power consumption increase with the number of bits and the size of matrix **Q** ($N \times Z$) for all scenarios. However, it is possible to notice that the dynamic power consumption is small, in the order of 90 mW for the scenarios with greater complexity and higher resolution ($N = 132, Z = 4$).

The Figure 19 shows the dynamic power consumption as a function of the size of matrix **Q** ($N \times Z$) and also according to the throughput (color bar) and the number of bits (size of circle). All situations were plotted with four different resolutions (or the number of bits), an they were [10.00], [16.06], [20.10]] and [24.14]. There are always four circles with the same size of matrix **Q** (there are ten different sizes of matrix **Q**).

Analyzing the Figure 19, it can be inferred that the increase of the **Q** matrix size, $N \times Z$, reduces the throughput, F_s, and increases the power consumption. Another critical point, it is the non-linear power growth as showed in Figure 19. The number of bits has a minor impact on power consumption for low **Q** matrix sizes ($N \times Z < 200$) however it has a high nonlinear impact for large **Q** matrix size ($N \times Z > 200$). When there is an increase in the number of states (N) or a problem with a greater number of actions (Z), for the same amount of states, it implies an increase in power consumption and a reduction of the throughput (F_s). That is, the power consumption and the throughput (F_s) are determined by the complexity of the problem.

F. REAL-WORLD EXPERIMENTS

Table 14 shows some practical applications of Q-learning found in the literature. The first column contains the reference. In the second and third columns are presented the dimensions of the problem (number of states and actions). In the fourth column are presented the number of iterations necessary for the problems, according to those dimensions, to converge to the optimal value function if implemented in the Q-learning hardware architecture proposed in this work. In the following column is presented the sampling rate that the problem could reach if Q-learning were implemented in the proposed architecture. In the last column, from the number of iterations and the maximum throughput, an estimation of the convergence time for the optimal policy of the mentioned problems is calculated. In a problem like the one

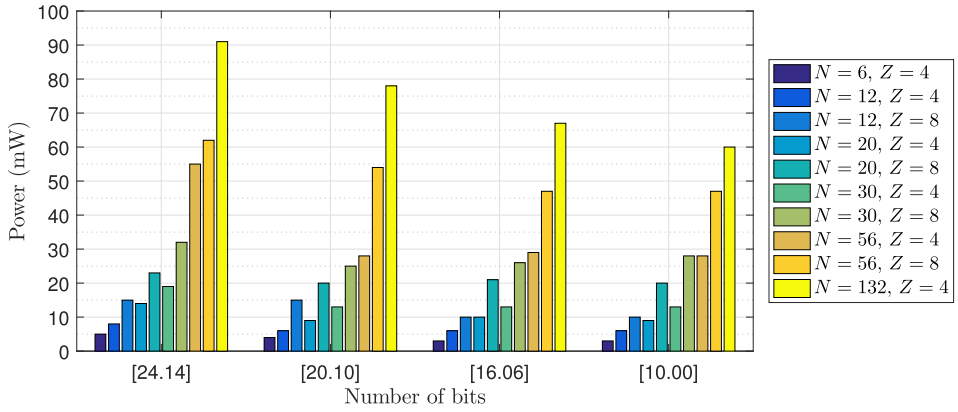

FIGURE 18. The dynamic power as a function of the number of bits for all scenarios (see Table 2).

TABLE 14. Convergence time and sampling Rate of literature applications using the Q-learning hardware architecture.

Reference	Number of states	Number of actions	Number of iterations	Sampling throughput (F_s)	Convergence time
[30]	$N = 41$	$Z = 3$	~ 8000	~ 15 MSps	~ 0.5 ms
[29]	$N = 12$	$Z = 8$	~ 5500	~ 23 MSps	~ 0.2 ms
[2]	$N = 18$	$Z = 2$	~ 2500	~ 25 MSps	~ 0.1 ms

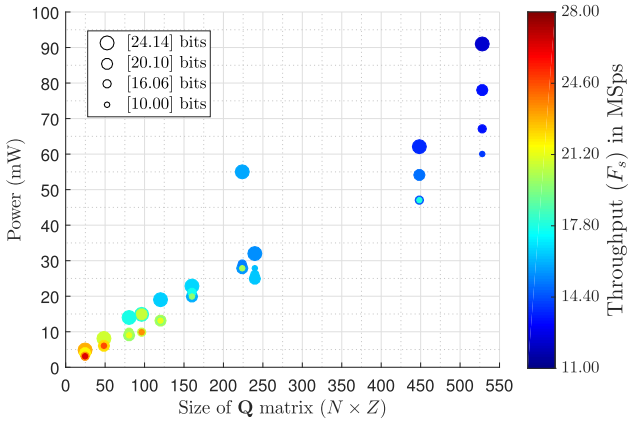

FIGURE 19. The power consumption as a function of the size of matrix Q ($N \times Z$) and also according to the throughput (color bar) and the number of bits (size of circle).

presented in [30], which has the same order of complexity as scenario VIII, it is possible to execute the Q-learning with a sampling rate of approximately 15 MSps. As seen in Section II, it takes 8000 iterations for the convergence of the Q matrix, which means that the optimal policy would be available to the system after 0.5 ms. In the problem presented in [29], which has the same number of states and actions of scenario III as the value function matrix, which converges with 5500 iterations, it would need 0.2 ms. In the problem presented in [2], with 2500 iterations, it would take 0.1 ms. If there is a time restriction to acquire the necessary data for the calculation of the policy, it is also possible to reduce the system clock so that, as demonstrated in [11], the system consumption is reduced and the execution time is adjusted for the data acquisition.

G. COMPARISON WITH OTHER PLATFORMS

In [25] was propose a parallel implementation for a PVM platform. A two-dimensional maze problem was studied with 135 states and 4 actions each of them. The maximum iterations (or episodes) used was of 32000 and the time for 1, 2, 4 and 8 processors were 4.32 s (throughput about 7400 samples or iterations per second), 3.4 s (throughput about 9411 samples or iterations per second), 1.85 s (throughput about 172971 samples (or iterations) per second) and 2.65 s (throughput about 12075 samples or iterations per second), respectively. The hardware proposed in this work has a throughput about 12.23 MSps (or 12.23 Mega-iterations per second) in the scenario with $N = 132$ states, $Z = 4$ actions and this is equivalent of 2.61 ms for 32000 iterations. For this case, the hardware implementation proposed here reaches a speed up about 1655×, 1302×, 708× and 1015× for 1, 2, 4 and 8 processors used in [25], respectively.

The Q-learning FPGA implementation is shown in [19] and [22] where it is proposed a semi-parallel approach. For a scenario where the **Q** matrix has about 240 elements ($N \times Z$), the works [19], [22] achieved a throughput of about 2.34 MSps (or 2.34 Mega-iterations per second). On the other hand, the throuhput achieved by the present implementation as 15.53 MSps (or 15.53 Mega-iterations per second), i.e., a speed up of about 6.63×. Another FPGA implementation of the Q-learning is shown in [21] where for the scenario with $N = 27$ states and $Z = 5$ actions (**Q** matrix with 135 elements). In this case, it was obtained a throughput of

TABLE 15. Comparative results regarding the speed up with other platforms.

Reference	Hardware	**Q** matrix size ($N \times Z$)	Reference throughput	Obtained throughput	Speedup
[25]	CPU - 1 processor	528	7400 Sps	12.23 MSps	1655×
	CPU - 2 processors		9411 Sps		1302×
	CPU - 4 processors		172971 Sps		708×
	CPU - 8 processors		12075 Sps		1015×
[19], [22]	FPGA	240	2.34 MSps	15.53 MSps	6.63×
[21]	FPGA	160	25000 Sps	16.74 MSps	669.6×

about 25000 Sps (or 25000 iterations per second) for the best example. In the similar scenario with $N = 20$ and $Z = 8$ (**Q** matrix with 160 elements), the architecture here proposed achieved a throughput of about 16.74 MSps (or 16.74 Mega-iterations per second), i.e., speed up about 669.6×.

Table 15 shows the speedups achieved by the approach proposed in the work here presented in comparison with the schemes presented in [19], [22], and [25].

V. CONCLUSION

This work illustrated a Hardware parallel architecture proposal of the Q-learning technique on FPGA. The state of the art of implementations of machine learning techniques in hardware was presented, giving emphasis to RL techniques. Applications using use Q-learning as a technique were listed, thus illustrating the main motivations for the development of this work. The choice of FPGA was justified due to its high performance, low operation frequency and to have several processing cores.

Q-learning is a reinforcement learning technique that has as its main advantage the possibility of obtaining an optimal policy interacting with the environment without any knowledge about the system model needed. Developing this technique in hardware allows shortening the system processing time. The FPGA was used due to the possibility of rapid prototyping and flexibility, parallelism and low power consumption, some of the main advantages of FPGA. Details of the implementation of the Hardware architecture have been described. It was also discussed details of individual system modules and what hardware mechanisms were used to implement the Q-learning algorithm. It was proposed a generic architecture of N states and Z actions that makes the data processing from a parallel and distributed implementation so that the processing time of Q-learning is optimized.

A detailed analysis of the implementation was conducted, in addition to the analysis of simulation and synthesis data. From the simulation data, the architecture was validated. It was also investigated the impacts of the resolution error to obtain the problem optimal policy. It is possible to observe that obtaining the optimal policy can also happen even for low resolutions in bits that imply in a smaller area of occupancy. The analysis of the synthesis data allowed to verify the behavior of the system regarding important parameters, such as occupancy rate and sampling time. By observing FPGA synthesis performed in this work it was observed that with the development of this algorithm, directly in hardware, it is possible to reach high performance, especially in terms of processing time and/or low power consumption when compared with their counterparts in software. These characteristics allow their use in more complex practical situations and with the most diverse types of applications, mainly time-constrained applications such as real-time applications, telecommunications applications where data flow needs to be handled very quickly; or in applications where there are power restrictions and low power consumption is required for these devices.

REFERENCES

[1] R. S. Sutton, Ed., *Reinforcement Learning: A Special Issue of Machine Learning on Reinforcement Learning*, 8th ed. Norwell, MA, USA: Kluwer, 1992.

[2] N. C. de Almeida, M. A. C. Fernandes, and A. D. D. Neto, "Beamforming and power control in sensor arrays using reinforcement learning," *Sensors*, vol. 15, no. 3, pp. 6668–6687, 2015.

[3] L. M. Reyneri, "Implementation issues of neuro-fuzzy hardware: Going toward HW/SW codesign," *IEEE Trans. Neural Netw.*, vol. 14, no. 1, pp. 176–194, Jan. 2003.

[4] A. C. D. de Souza and M. A. C. Fernandes, "Parallel fixed point implementation of a radial basis function network in an FPGA," *Sensors*, vol. 14, no. 10, pp. 18223–18243, 2014.

[5] B. J. Leiner, V. Q. Lorena, T. M. Cesar, and M. V. Lorenzo, "Hardware architecture for FPGA implementation of a neural network and its application in images processing," in *Proc. Electron., Robot. Automot. Mech. Conf. (CERMA)*, Morelos, Mexico, 2008, pp. 405–410.

[6] F. Mengxu and T. Bin, "FPGA implementation of an adaptive genetic algorithm," in in *Proc. 12th Int. Conf. Service Syst. Service Manage. (ICSSSM)*, Guangzhou, China, Jun. 2015, pp. 1–5.

[7] M. F. Torquato and M. A. C. Fernandes. (2018). "High-performance parallel implementation of genetic algorithm on FPGA." [Online]. Available: https://arxiv.org/abs/1806.11555

[8] S. Usenmez, R. A. Dilan, M. Dolen, and A. B. Koku, "Real-time hardware-in-the-loop simulation of electrical machine systems using FPGAs," in *Proc. Int. Conf. Elect. Mach. Syst.*, Nov. 2009, pp. 1–6.

[9] H. Saldanha *et al.*, "A cloud architecture for bioinformatics workflows," in *Proc. 1st Int. Conf. Cloud Comput. Services Sci.*, Noordwijkerhout, The Netherlands, 2011, pp. 477–483.

[10] L. A. V. de Carvalho, *Datamining: A Mineração de Dados no Marketing, Medicina, Economia, Engenharia e Administração*. São Paulo, Brazil: Editora Ciência Moderna, 2005.

[11] L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic power consumption in Virtex-II FPGA family," in *Proc. ACM/SIGDA 10th Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2002, pp. 157–164.

[12] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *Proc. Int. Conf. Field Program. Logic Appl.*, Prague, Czech Republic, 2009, pp. 126–131.

[13] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[14] Xilinx. (2018). *System Generator for DSP*. Accessed: Nov. 17, 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html
[15] Xilinx. (2018). *Virtex-6 Family*. Accessed: Nov. 17, 2018. [Online]. Available: https://www.xilinx.com/support/documentation-navigation/silicon-devices/fpga/virtex-6.html
[16] H. Woehrle and F. Kirchner, "Reconfigurable hardware-based acceleration for machine learning and signal processing," in *Formal Modeling and Verification of Cyber-Physical Systems*. Wiesbaden, Germany: Springer, 2015.
[17] Z. Liu and I. Elhany, "Large-scale tabular-form hardware architecture for Q-learning with delays," in *Proc. Midwest Symp. Circuits Syst.*, Montreal, QC, Canada, Aug. 2007, pp. 827–830.
[18] V. L. Prabha and E. C. Monie, "Hardware architecture of reinforcement learning scheme for dynamic power management in embedded systems," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, p. 065478, 2007.
[19] P. R. Gankidi and J. Thangavelautham, "FPGA architecture for deep learning and its application to planetary robotics," in *Proc. IEEE Aerosp. Conf.*, Mar. 2017, pp. 1–9.
[20] S. Shao *et al.*, "Towards hardware accelerated reinforcement learning for application-specific robotic control," in *Proc. IEEE 29th Int. Conf. Appl.-Specific Syst., Archit. Process. (ASAP)*, Jul. 2018, pp. 1–8.
[21] J. Su, J. Liu, D. B. Thomas, and P. Y. K. Cheung, "Neural network based reinforcement learning acceleration on FPGA platforms," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 4, pp. 68–73, 2017.
[22] P. R. Gankidi, "FPGA accelerator architecture for Q-learning and its applications in space exploration rovers," Ph.D. dissertation, SpaceTREx Lab., Univ. Arizona, Tucson, AZ, USA, 2016.
[23] R. Faraji and H. R. Naji, "An efficient crossover architecture for hardware parallel implementation of genetic algorithm," *Neurocomputing*, vol. 128, pp. 316–327, 2014.
[24] J. T. Barron, D. S. Golland, and N. J. Hay, "Parallelizing reinforcement learning," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., UC Berkeley, Berkeley, CA, USA, 2009.
[25] A. M. Printista, M. L. Errecalde, and C. I. Montoya, "A parallel implementation of Q-learning based on communication with cache," *J. Comput. Sci. Technol.*, vol. 1, no. 6, p. 11, 2002.
[26] M. Kushida, K. Takahashi, H. Ueda, and T. Miyahara, "A comparative study of parallel reinforcement learning methods with a PC cluster system," in *Proc. IEEE/WIC/ACM Int. Conf. Intell. Agent Technol.*, Dec. 2006, pp. 416–419.
[27] M. Camelo, J. Famaey, and A. S. Latré, "A scalable parallel Q-learning algorithm for resource constrained decentralized computing environments," in *Proc. 2nd Workshop Mach. Learn. HPC Environ. (MLHPC)*, Nov. 2016, pp. 27–35.
[28] C. J. C. H. Watkins and P. Dayan, "*Q*-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, 1992.
[29] A. Das, R. A. Shafik, and G. V. Merrett, "Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems," in *Proc. ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2014, pp. 1–6.
[30] A. A. R. Diniz, A. J. J. L. Filho, P. R. da Motta Pires, S. M. Kanazava, J. D. de Melo, and A. D. D. Neto, "Application of q-learning to define optimal policy for triggering pid, neural and fuzzy controllers in a level control process," in *Proc. Brazilian Congr. Autom.*, 2010, pp. 3270–3277.
[31] P.-C. Wu, "Multiplicative, congruential random-number generators with multiplier $\pm 2^{k1} \pm 2^{k2}$ and modulus $2^p - 1$," *ACM Trans. Math. Softw.*, vol. 23, no. 2, pp. 255–265, 1997.
[32] Xilinx. (2018). *Virtex-6 FPGA ML605 Evaluation Kit*. Accessed: Nov. 17, 2018. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html
[33] MathWorks. (2018). *MATLAB/Simulink*. Accessed: Nov. 17, 2018. [Online]. Available: http://www.mathworks.com
[34] Xilinx. (2018). *Vivado Design Suite*. Accessed: Nov. 17, 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

LUCILEIDE M. D. DA SILVA was born in Natal, Brazil. She received the B.S. degree in electrical engineering from École Nationale Supérieure d'électronique, d'Electrotechnique, d'Informatique et d'Hydraulique de Toulouse (ENSEEIHT), Toulouse, France, in 2011, and the M.S. degree in computer engineering from the Federal University of Rio Grande do Norte, Natal, Brazil, in 2012 and 2016, respectively. She is currently a Professor with the Informatics Group, Federal Institute of Rio Grande do Norte, Santa Cruz, Brazil. She is part of the Robotics Group's at the Institute, training teenagers in different competitions and developing assistive technology to children with special needs. She is also part of the Research Group on Embedded Systems and Reconfigurable Hardware, Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte, where the main research topics are the acceleration of articial intelligence algorithms through reconfigurable computing on FPGA. Her research interests include artificial intelligence, embedded systems, reconfigurable hardware, tactile Internet, educational robotics for teenagers and specially children.

MATHEUS F. TORQUATO was born in Natal, Brazil. He received the B.Sc. degree in science and technology, in 2013, the B.E. degree in computer engineering, in 2015, and the M.Sc. degree in computer engineering from the Federal University of Rio Grande do Norte, Natal, Brazil, in 2017. He is currently part of the Research Group on Embedded Systems and Reconfigurable Hardware, where the main research topics are the acceleration of artificial intelligence (AI) algorithms through reconfigurable computing (RC) on FPGA. Apart from his main research topic of AI and RC at the Federal University of Rio Grande do Norte, he has other research experiences, such as human–computer interaction at the Future Interaction Technology Lab, Swansea University, Wales, U.K., and computer vision at the Sensing and Machine Vision for Automation and Robotic Intelligence Lab, University of Ottawa, Ottawa, ON, Canada. His research interests include artificial intelligence, embedded systems, reconfigurable hardware, human–computer interaction, and tactile Internet.

MARCELO A. C. FERNANDES was born in Natal, Brazil. He received the B.S. and M.S. degrees in electrical engineering from the Federal University of Rio Grande do Norte, Natal, Brazil, in 1997 and 1999, respectively, and the Ph.D. degree in electrical engineering from the University of Campinas, Campinas, SP, Brazil, in 2010. From 2015 to 2016, he was a Visiting Researcher with the Centre Telecommunication Research, King's College London, London, U.K. He is currently an Adjunct Professor with the Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte. He is also the Leader of the Research Group on Embedded Systems and Reconfigurable Computing and the Coordinator of the Laboratory of Machine Learning and Intelligent System. He has authored or co-authored many scientific papers and practical studies on reconfigurable computing on FPGA to accelerate artificial intelligence algorithms. His research interests include artificial intelligence, digital signal processing, embedded systems, reconfigurable hardware, and tactile internet.