Applying Game-Learning Environments to Power Capping Scenarios via Reinforcement Learning

Pablo Hernández, Luis Costero, Katzalin Olcoz, and Francisco D. Igual

Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
{pherna06,lcostero,katzalin,figual}@ucm.es

Abstract. Research in deep learning for video game playing has received much attention and provided very relevant results in the last years. Frameworks and libraries have been developed to ease game playing research leveraging Reinforcement Learning techniques. In this paper, we propose to use two of them (RLlib and Gym) in a very different scenario, such as learning to apply resource management policies in a multi-core server, specifically, we leverage the facilities of both frameworks coupled to derive policies for power-capping. Using RLlib and Gym enables implementing different resource management policies in a simple and fast way and, as they are based on neural networks, guarantees the efficiency in the solution, and the use of hardware accelerators for both training and inference. The results demonstrate that game-learning environments provide an effective support to cast a completely different scenario, and open new research avenues in the field of resource management using reinforcement learning techniques with minimal development effort.

Keywords: Reinforcement Learning · RLlib· Gym· Resource Management · Power Capping · DVFS.

1 Introduction

Artificial Intelligence (AI) has been widely used in video games for quite a long time, both for improving the user experience playing the game, and also developing automatic systems that learn how to win them [11]. In October 2015, AlphaGo, an algorithm based on Deep Reinforcement Learning (RL), beat one of the best human players, reaching a milestone in AI research for video game playing [12]. Nowadays, deep learning for video game playing is an active area of research [9]. The goal of these systems is to automatically learn policies that win the game, with *agents* interacting with an *environment* and the achieved score as a *reward* to be maximized.

In response to the increasing interest in learning environments for games, different platforms and software ecosystems have emerged to improve both the quality and the efficiency of the learning process. Among them, the tuple RLLIB–GYM[1] is one of the most successful efforts. GYM provides an abstraction for the construction of *environments* that can be interacted externally by means of the application of actions, and sensed afterwards. RLLIB is a complete distributed infrastructure that provides a plethora of RL agent development primitives that, together with a GYM environment, can yield policies in an easy way, leveraging high performance computing strategies.

Environments provide an abstraction to build ad-hoc *black boxes* that mimic the behaviour of virtually any stateful scenario, with the possibility of receiving actions, modifying (and observing) the internal state of the environment and receiving back a proper reward depending on the target optimization objective. The behaviour of an environment can be easily personalized, maintaining a common external interface for an easy interaction and deployment. In this paper, we leverage this abstraction to propose the application of the aforementioned game-specific frameworks, to a dramatically different scenario: *power capping* on modern multi-core servers. *Power capping* [15] is a strategy typically applied in different datacenter-level scenarios to limit instantaneous power consumption at different levels (chip, server, facility, etc.), in order to fulfill user-specific limits, or system-wide power restrictions.

Modern multi-core servers exhibit a number of knobs that can directly impact power consumption; one of the most effective is DVFS (*Dynamic Voltage-Frequency Scaling*) that, assisted with the appropriate operating system support, can adapt the effective operating frequency at different granularities. Additionally, modern processors are armed with different mechanisms to measure the instantaneous power consumption at different levels (usually per core, socket and complete system). All in all, a computing server equipped with DVFS and capabilities to measure power consumption can be considered as an ideal scenario for a complete interaction with a RL agent, as it:

- Provides a stable *observation space*, in terms of sensed power consumption.
- Features a set of *actions* with impact on the observed power consumption.
- In a *power capping* scenario, allows the design and implementation of a convenient *reward* strategy that penalizes power states above the specified power cap, and positively rewards power states below the power cap.

We explore the feasibility of extending state-of-the-art RL libraries widely used in the video game arena, to a power capping scenario. Specifically, we will show how a power capping scenario can be easily modelled as a RL problem, and how little effort is required to implement it in terms of existing software infrastructure. A discussion of how state and action definition impacts on the outcome policy will also be presented. Additionally, we will show how the described approach can deal with different workloads (in terms of different power demands) and with different power capping limits.

[1] https://docs.ray.io/en/latest/rllib.html – https://gym.openai.com/

2 The RLlib and Gym frameworks

Reinforcement Learning [14] is one of the three basic pillars of Machine Learning, together with *Supervised* and *Unsupervised learning.* It is oriented towards training a system by interacting with its environment, to automatically determine its ideal behaviour in a specific context, so that its profit is maximized. This general goal is usually tackled by means of agent-environment paradigms, in which one (or multiple) agent decides the actions that the environment needs to perform, based on its state or observation. The agent receives a numeric value (*reward*), calculated as a function of the previous environment state, and the current state after applying an action. This way, the agent can *learn* which actions must be taken by the environment to maximize the cumulative reward. Different RL algorithms implement different methods to obtain these policies (e.g., table-based solutions as Q-Learning, or Neural Network based solutions as DQN [10]). The benefit of using Neural Netwoks as a function approximator is three-fold: *(i)* they scale in performance for large action-state spaces; *(ii)* they can leverage software frameworks (e.g. Tensorflow) for efficient implementations; and *(iii)* domain-specific hardware can be used to accelerate their execution.

RL has been successfully applied to game-learning systems in order to infer, learn and apply game rules by means of observation and interaction with existing game engines with minimum level of environmental knowledge. A representative example is DQN, a complex Deep Reinforcement Learning architecture that is able to learn policies directly from high-dimensional sensory inputs; specifically, DQN receives only the pixels and the game score as inputs, and is able to surpass a human games tester on a wide range of games.

2.1 RLlib

Ray is an open-source framework that aims at creating a universal API for *distributed applications.* Within the framework, the Python library RLlib is focused on supporting Reinforcement Learning applications at all necessary levels. Ray/RLlib offer facilities to parallelize code across shared- or distributed-memory architectures, optionally equipped with accelerators (GPUs). Internally, RLlib uses Tensorflow to model the complete structure of a RL problem and to integrate models (neural networks) that mimic the approximation functions necessary on RL problems.

2.2 Gym

Gym is an open-source library designed to develop and compare RL algorithms. Gym includes a rich collection of *environments* that allow an interaction with agents trained via reinforcement learning. The interaction with these environments mimics the general idea of reinforcement learning: the agent can apply a number of actions on an environment (the exact set of actions is defined in the Gym environment) and, as a result, the Gym environment returns a tuple of values, including the obtained *reward* and the new *observation.* Gym is mainly focused on game learning, but offers mechanisms to define new environments based on an API that allows a straightforward and portable interaction with the environment from an external agent. This API orbits around two main routines:

- `reset()`: that obtains an initial *state observation* from the environment.
- `step( action )`: invoked by an agent, and receiving an action to apply (an integer within a range of pre-determined values).

The `step()` routine returns information of the effect of the action on the environment, including:

- `observation`: object representing a new observation after the application of an action. Its type and range of values depend on the specific environment.
- `reward`: reward obtained after the application of a new action on the environment in a given state. It is represented as a floating point number and its range is also defined by the specific environment.
- `done`: optional value that indicates the end of the environment lifetime.
- `info`: dictionary with additional debugging information, including information regarding the evolution of the learning process.

Each environment features an *action space* ($\mathcal{A}$) and an *observation space* ($\mathcal{S}$), that define the values that can be taken as actions (hence, the shape of the output tensor of the underlying neural network) and the values that can be obtained as an observation (hence, the shape of the input tensor). Additionally, a reward function ($\mathcal{R}$) that gives a score to each action applied by the agent is used to train the system and obtain the desired policy ($\mathcal{R} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$).

Obtaining a suitable RL policy is just the task of training the underlying neural network. This task is transparently carried out by the RLlib framework by means of interacting with the Gym environment.

3 RL for Resource Management

Computer architectures have evolved drastically in the last decades, seeking an optimal combination of performance and energy efficiency. The strategies followed in early-2000s, mainly dictated by a constant increase of the frequency, cannot be further pursued due to the lack of technological support, keeping core frequencies around 2-4 GHz to keep heat and power under control [1], leading to a shift towards multi-core architectures. Together with the increase in the number of compute units, the addition of different technologies that allow processors to adapt dynamically to the changes in the environment and running applications has accompanied the processor evolution. DVFS [2], Power Capping [5] or Cache Partitioning [6] are only three examples of hardware-assisted support to increase performance and/or energy efficiency. In addition, this type of systems usually expose a number of mechanisms to measure (or estimate) different metrics with different granularity (e.g., modern processors offer mechanisms to measure energy at a core, socket, and system level).

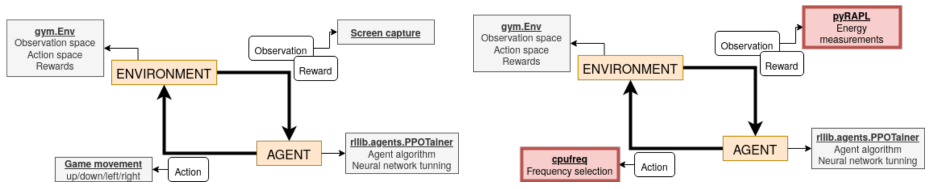
This evolution has provoked a change in how resources are managed. In the past, Resource Managers had to deal with simple scenarios with a limited number of parameters to configure and metrics to monitor. Nowadays, architectures offer a plethora of different metrics to observe, and parameters to tune (possibly simultaneously), leading to more complex scenarios. Specifically, modern resource managers need to deal with:

1. *Malleability at system-level:* Modern platforms support a plethora of different mechanisms to adapt themselves to the running applications and environments. Examples are DVFS capabilities, support for core disabling/enabling, or cache partitioning. In addition, one metric can be affected by multiple parameters, and one parameter can have effects on multiple metrics at the same time, making the process of designing the policy an arduous task.
2. *Malleability at application-level:* Modern applications expose a number of parameters that can be configured statically and dynamically, and affect directly to different application- and system- metrics (e.g., changes on the number of threads used during the computations).
3. *Multiple optimization goals:* As platforms and applications have evolved, their requirements have evolved too. Modern resource managers should seek to fulfill multiple optimization goals at the same time (e.g., energy efficiency, performance, Quality of Experience (QoE), etc.). However, designing a multi-objective system is not a trivial task.
4. *Additional restrictions:* Apart from the optimization goals defined by the system designer, applications and platforms can present additional restrictions that the resource manager has to fulfill (e.g., power capping limits, minimum Quality of Service (QoS) requirements, etc.).

In this scenario, a main question arises: *how can we obtain a multi-objective resource management policy able to modify multiple parameters concurrently at the same time it deals with the previous problems?*

Traditionally, complex heuristics have been used as the *de-facto* solution to manage shared resources in computing platforms [7, 13]. On one hand, heuristics are simple solutions to these problems, yielding easy-to-understand policies. On the other hand, a deep knowledge of the problem is required to design effective heuristics. In addition, the obtained policies are usually valid only for a specific set of fixed conditions, and dependent on the problem input.

With the increasing interest on Machine Learning, RL has been proved as a valid alternative to tackle these scenarios [4, 8], as it presents several advantages over the traditional approaches. Among others, a less deep knowledge of the problem is required to formulate the solution, as well as the ability to obtain input-independent policies. As drawbacks, ML-based approaches can lead to long training periods, as well as to solutions that, being valid, are difficult to understand by a human compared with heuristics. However, the application of these techniques to resource management is far from being trivial, and typically requires ad-hoc implementations [4, 8].



(a) Game-learning environment (b) Power capping environment

Fig. 1: RLlib-based implementation applied to video games (left) and power capping (right). Note that minimum changes are needed (marked in red).

4 Casting a power capping scenario with Gym

Our contribution lies on the design of Gym environments that tackle the aforementioned *power capping* scenario. The proposed environments implement *(i)* an observation space that can be filled by ad-hoc power measurement mechanisms *(ii)* an action space, in which actions can range from selecting specific frequencies between those offered by the processor, to increase/decrease on an individual step basis; and *(iii)* a specific *reward* strategy, that implements techniques to fulfill a specific goal (e.g. maximizing performance under a power cap).

All this logic is encapsulated within a Gym environment that keeps the same interface as that previously described. Armed with this type of environments, we integrate them in the RLlib environment and apply training procedures to determine the ability of the system to extract efficient policies that, deployed on a real multi-core server, can keep the power consumption under an established cap for different workloads. Figure 1 shows the original game-learning environment and the modifications that are needed in order to obtain a power capping policy. Only two python modules are needed: `pyRAPL`, that obtains energy measurements used as observations, and `cpufreq`, that applies the changes in frequency that correspond to the different actions chosen. The remainder of this section shows how to create a new environment, defining the set of actions, states and rewards.

4.1 Defining states

The observation space is formed by the different values of power consumed by the system. Even if this value is a real number, we will use a discrete number of states, each one comprising all the power consumption values in a given range. However, there is not any golden rule about how states, actions and rewards have to be defined, and expert knowledge of the problem is required. Their definition, and specially the number of actions and states will ultimately determine the quality of the learned policy as well as the training time required to obtain a functional policy [3]. On one hand, if each state covers a wide range of power values, the learning time will decrease as the number of states to explore will be lower. However, the quality of the obtained policy may be negatively affected as the system will not be able to apply different actions to different values in the same interval. On the other, increasing the number of states (i.e., decreasing the power interval covered by each state) will improve the quality of the policy as the agent will be able to apply actions in a finer granularity, at the expenses of longer learning times, since the agent will need to explore more states.

So, different alternatives for the definition of states and actions will be explored in this work. In our first formulation of the problem, there will be one state for all the power values lower than a certain power value, another for all the power values greater than another certain value, and the interval between those two values will be divided in equally sized states of a specific size.

In order to choose these three values (minimum power value, maximum power value, and interval size), some profiling is required. After that, the minimum power value will be the consumption when the system is in idle, and the maximum power value will correspond with the TDP of the machine (or maximum power measured). Additionally, the size of the interval will be certain value that guarantees that the power measurements obtained at different frequencies during the profiling phase are not classified into the same state.

For running the experiments, different programs were run at each of the available frequencies of the system, obtaining the results in Figure 2 and that will be explained in the next section. Thus, the idle power (15W) and the TDP (115W) will be chosen as the minimum and maximum values respectively. Based on the results, a size of 3W will be chosen for creating the intervals in the first set of experiments, since it is similar to the difference in power values consumed by two adjacent frequencies. Sizes of 2W and 4W will be also explored. Finally, other configuration with non-uniform distribution of states will be tried.

4.2 Defining actions and rewards

Similar to the state definition, the number of actions will ultimately determine the quality of the obtained policy, as well as the learning time (greater number of actions usually means greater quality at expenses of longer training periods). In our first environment, we consider the actions limited to increase one step the current frequency, and to decrease one step the frequency. Other definitions are possible, like considering all the available frequencies as possible actions for the agent. Nevertheless, a different set of actions will be considered later, that includes the posibility of maintaining the frequency at the same level.

The reward function is defined based on how good or bad is the action taken compared with the previous step. More specifically, the reward given at step $t+1$ (R_{t+1}) is defined as follows:

$$R_{t+1}(s_t, s_{t+1}) = \begin{cases} -1 & \text{if} \quad |s_{t+1} - s_{goal}| > |s_t - s_{goal}| \\ +1 & \text{if} \quad |s_{t+1} - s_{goal}| < |s_t - s_{goal}| \\ +2 & \text{if} \quad s_{t+1} - s_{goal} = 0 \end{cases} \tag{1}$$

where s_{t+1} and s_t are the states the system is in the current and previous steps respectively, and s_{goal} is the state containing the established power cap.

This reward definition guarantees that the system will move closer to the desired power consumption at each step. Indeed, the reward will penalize the

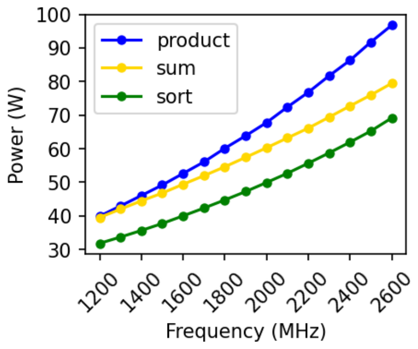


Fig. 2: Power consumption profile for the different workloads.

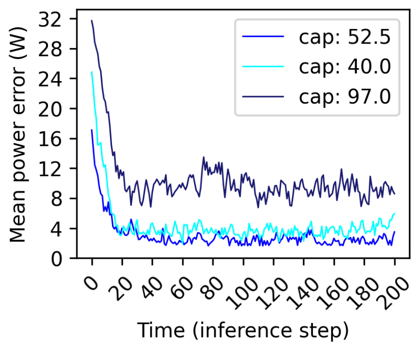


Fig. 3: Mean power error for different power cap limits.

agent if the new power observation is farther from the cap than the previously observed power (reward of -1), will give a reward of +1 if the new power observation is closer than the previous one, and will grant a maximum reward of +2 if the power observation is in the same interval as the established cap.

5 Experimental results

All experiments were implemented on a real server comprising two Intel Xeon CPUs E5-2670 with a total of 16 physical cores and 64GB of DRAM. Thermal Design Power (TDP) is 115W for each processor. Available frequencies range from 1200MHz to 2600MHz, selectable with a 100MHz granularity. The following libraries (and versions) were used in the experiments: `Python (3.8.5)`, `cpufreq` (0.3.3), `pyRAPL` (0.2.3.1), `GYM` (0.17.3), `Ray` (1.0.1), and `NumPy` (1.18.5).

All environments have been tested via RLlib with different background workloads: matrix-matrix multiplication (called `product` in the following), vector sort (`sort`) and matrix-matrix addition (`sum`), all using floating point elements. Each experiment was initialized with a different frequency, covering all the possible values. Each experiment was repeated 15 times with different initial frequency values. Figure 2 shows the different power values measured for the previous workloads at different frequencies. Matrix-based operations (`product` and `sum`) were configured to use $1,000 \times 1,000$ elements, while a vector of $1,000,000$ elements was used for the sorting operation. Observe how these workloads cover all the possible range of power values for the different frequency values tested, and how the benchmarks exhibit a different behaviour as the frequency increases, covering different situations present on real-life scenarios.

Power measurements correspond to average power consumption measured during 150 ms. In addition, the following 100 ms after a frequency change were not measured to not introduce noise in the system. For training the system, a total of 5 epochs were run, with 4000 steps in each epoch, each of 250 ms. These values guarantee that the Reinforcement learning process does not influence on the learning process. The `product` workload was used on this phase. For testing each learned policy, 15 iterations of 200 steps each were run. For each iteration, the environment was initialized with a different frequency to test all the scenarios. Numerical results of each tested policy are presented in terms of mean error ($\epsilon(t)$) and mean accumulated error (E), defined as follows:

$$\epsilon^j(t) = |x^j(t) - \overline{x}|, \qquad \epsilon(t) = \frac{1}{15}\sum_{j=1}^{15} \epsilon^j(t), \qquad E = \frac{1}{201-T}\sum_{t=T}^{200} \epsilon(t) \tag{2}$$

where $\overline{x}$ is the power target, $x^j(t)$ is the power value at step t and iteration j, and T is the step number at which the iteration starts to converge. In our experiments T was set to 75 experimentally.

5.1 Analysis under different power caps

Figure 3 shows a detailed trace of our approach when executing a matrix-matrix `product` workload in the background under three different power caps: 97W ($\approx$ 84% TDP), 52.5W ($\approx$ 46% TDP) and 40W ($\approx$ 35% TDP). This three values mimic possible values present in real-life scenarios. The simple definition of states (fixed size of 3W) and actions (increase/decrease frequency) was used.

As observed, our approach exhibits a similar behaviour for all the tested power caps: a first phase with constant decreasing error values, and a second phase where the system has converged and the error keeps relatively constant ($t \geq 75$). This is the result of how the experiments were carried out. Indeed, as the initial frequencies of the 15 testing iterations cover the whole frequency spectrum, the agent needs to modify the frequency several times until an ideal frequency producing power values near the required value is reached. However, when comparing the three approaches, the error obtained when the power cap is close to the limit is significantly higher than for the other values. This behaviour is a direct consequence of the available actions of the agent. For a high power cap, the optimal policy should maintain the frequency to the maximum most of the time. However, our agent definition does not consider this option, being constantly oscillating between the two highest frequencies. We show how this behaviour can be improved in the next subsection. Nevertheless, our approach is able to maintain the system with a power consumption close to the power cap, with an average accumulated error (E) of 3.68W, 2.42W and 9.49W for 40W, 52.5W and 97W power cap values respectively.

Figure 4 offers an intuitive vision of the convergence of the training algorithm, when establishing a power cap of 52.5W. Observe how, as the training process proceeds, the agent takes less random decisions (in terms of frequency selection), and the observed power converges to the target cap.

5.2 Impact of the state and action definitions

Figure 5 shows the mean power error (ϵ) for three different state definitions, each with a different power interval size (2W, 3W and 4W), configured in the

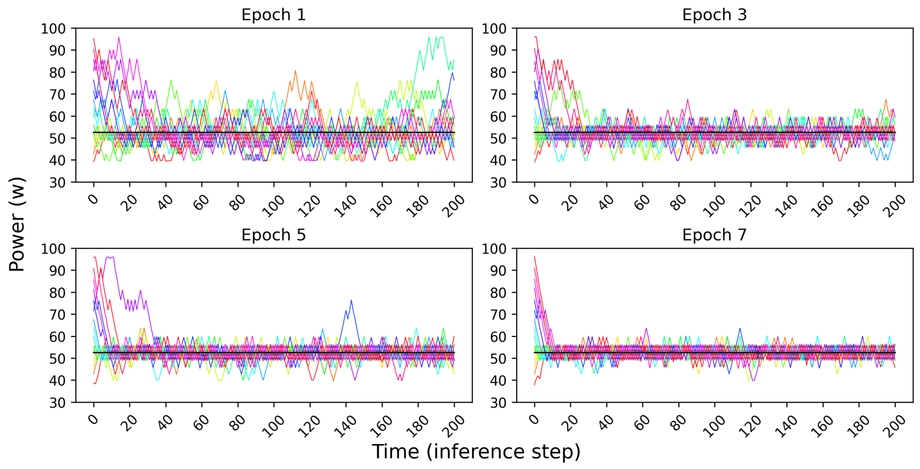


Fig. 4: Power consumption for different training epochs, using a matrix-matrix `product` workload. The black line represents the target power cap. Each other color represent a different iteration with a different initial frequency.

same environment described before with a target power cap of $52.5W$. The mean accumulated error (E) is $2.54W$, $2.42W$ and $2.71W$ respectively. Observe how, for this specific scenario, the policy obtained in each case produces similar results. This is due to the reduced number of actions the agent has to consider at each state. Indeed, the agent will potentially take the same action for all the power values in the same interval, producing the same results independently on how big or small is each one. This behaviour will be different only in those states close to the power cap imposed. However, the number of these states is negligible in comparison with the amount of states far from this value.

To test how different state/action definitions impact our scenario, two additional environments (apart from the one defined) were defined:

– `Env2`: same strategy as `Env1`, but an additional action to maintain the frequency at the same level was added. Thanks to this modification, the agent can potentially learn to maintain the frequency when the power measurements are close to the power cap, decreasing the error.
– `Env3`: An extension of `Env2`, but performing a non-uniform distribution of the observation space. This distribution is based on the power profiling from a specific workload (matrix-matrix `product`). For each power value measured, a new state was created containing this value but not any other value measured. Considered actions are the same as in the previous environment.

Table 1 summarizes the configuration of the aforementioned environments, and the parameters of the underlying neural network in terms of number of inputs, neurons per layer, number of outputs and trainable parameters.

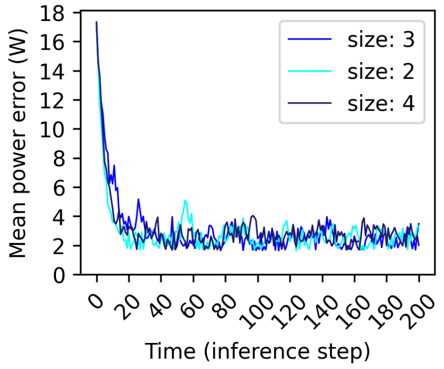


Fig. 5: Mean power error for different state definitions (interval sizes) with a power cap of 52.5W for the matrix-matrix multiplication workload.

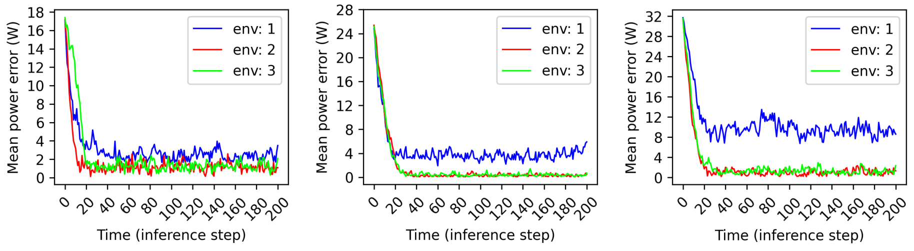
Table 1: Overview of the different tested environments.

			Neural network				
	State definition	*Actions*	*Inputs*	*L1*	*L2*	*Outputs*	*Trainable parameters*
`Env1`	Uniform (3W)	↑/↓ freq.	37	256	256	2	151,811
`Env2`	Uniform (3W)	↑/↓/↔ freq.	37	256	256	3	152,068
`Env3`	Non-uniform	↑/↓/↔ freq.	16	256	256	3	141,316

Figure 6 and Table 2 report the mean power error (ϵ) and mean accumulated error (E) respectively for the three different power caps tested. Overall, observe that `Env2` and `Env3` obtain low and similar error values (ϵ and E), while the policy associated to `Env1` produces worse results. This is a direct consequence of adding to the agent the option to maintain the frequency, that improves drastically the results for all the tested power caps, with improvements of 2×, 11× and 10× for 52.5W, 40W and 97W respectively in the mean accumulated error (E) when compared with `Env2`, and 2×, 9×, and 8× when compared against `Env3`. Lastly, observe that there is not too much difference between environments `Env2` and `Env3`. However, using custom intervals for the state definition (`Env3`) allow us to reduce the number of states drastically from 36 in `Env2` to 15 in `Env3`, and therefore, reduce the learning time (or equivalently, obtain a policy with greater quality for the same learning time) at the cost of loosing generality.

5.3 Behaviour under different workloads

To show the effectiveness of our approach under different workloads, the environment `Env2` was trained running matrix-matrix multiplications (`product`) in the background and tested against the other two operations described above (i.e., matrix-matrix addition -`sum`- and vector sorting -`sort`-). Figure 7a shows the



(a) Pcap=52.5W (b) Pcap=40.0W (c) Pcap=97.0W

Fig. 6: Mean power error for different power cap limits and actions when running the matrix-matrix multiplication workload in background.

Table 2: Mean accumulated error (E) under different power caps.

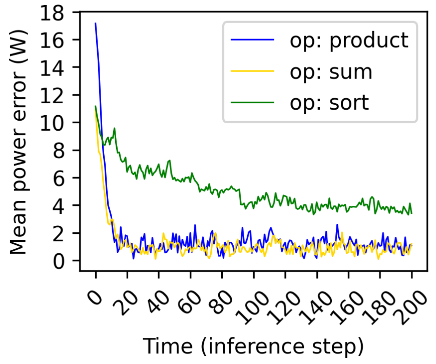
	`Env1`			`Env2`			`Env3`		
Pcap	52.5W	40.0W	97.0W	52.5W	40.0W	97.0W	52.5W	40.0W	97.0W
	2.42W	3.68W	9.49W	1.17W	0.34W	0.98W	1.25W	0.39W	1.25W

mean power error (ϵ) produced for the different workloads with a power cap of 52.5 W. As observed, for both matrix-matrix multiplications and matrix-matrix addition, the results are similar, producing an average error of 1W respect to the power cap. However, a greater error is obtained when the `sort` workload is run in the background. This error is the consequence of the different power consumption each workload has (see Figure 2). Indeed, the `sort` operation presents power values that are not produced by any of the other benchmarks at any frequency. As a consequence, because the system was trained using the `product` operation as workload, the agent still has some states not explored when running with the `sort` operation in background, leading the agent to take actions randomly. This behaviour can be seen more clearly in Figure 7b, that shows the frequency taken by the agent at each iteration. As observed, although most of the actions are centered around 2200MHz, there are noisy actions far from this value. This behaviour ultimately produces the high error values described before.

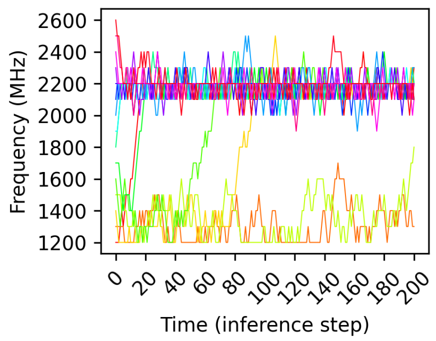
These results reveal the importance of the input used for training, and how if a workload able to produce all the different power values (or a mixed of multiple workloads) is used, the obtained policy will yield better results.

6 Conclusions

In this paper, we have given clues and evidences towards the integration of power capping mechanisms within frameworks (RLlib and *Gym*) that are conceived and designed for other type of domains. By means of abstracting observations, actions and rewards, we have shown how existing Reinforcement Learning frameworks



(a) Mean power error



(b) Actions applies for the `sort` workload

Fig. 7: Behaviour of the system under different workloads and a power cap of 52.5 W. On the left, mean power error. On the right, actions taken by the agent (i.e., frequency set) for the `sort` workload at epoch 5.

can obtain efficient policies to automatically apply power capping techniques by software. This idea opens a plethora of research lines, including the acceleration of the training processes via multiple hardware accelerators, provided the underlying infrastructure supports this kind of functionality.

We have experimentally proven how our proposal is able to control the power consumption of a system running different workloads, each with a different power-consumption profile. All the experiments were run considering different power cap values, making them generic enough to be applicable to other scenarios. Additionally, we have shown how the state and action definitions influence in the quality of the policy obtained. In particular, adding an action to the original environment has proved to decrease the obtained error by 2×, 11× and 10× for executions with a power cap of $52.5W$, $40W$ and $97W$ respectively.

Acknowledgements

This work was supported by the EU (FEDER) and Spanish MINECO (RTI2018-093684-B-I00), and Comunidad de Madrid under the Multiannual Agreement with Complutense University in the line Program to Stimulate Research for Young Doctors in the context of the V PRICIT under projects PR65/19-22445 and CM S2018/TCS-4423.

References

1. Luiz André Barroso. The price of performance: An economic case for chip multiprocessing. *Queue*, 3(7):48–53, 2005.

2. Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. *CC:PE*, 29(12):e4143, 2017.
3. Luis Costero, Francisco D. Igual, Katzalin Olcoz, and Francisco Tirado. Leveraging knowledge-as-a-service (KaaS) for QoS-aware resource management in multi-user video transcoding. *The Journal of Supercomputing*, 2020.
4. Luis Costero, Arman Iranfar, Marina Zapater, Francisco D. Igual, Katzalin Olcoz, and David Atienza. Resource Management for Power-Constrained HEVC Transcoding Using Reinforcement Learning. *IEEE Trans. on Parallel and Distributed Systems*, 31(12):2834–2850, 2020.
5. Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *ACM Int. Conf. on Supercomputing*, 2005.
6. Adrián Garcia-Garcia, Juan Carlos Saez, José Luis Risco-Martin, and Manuel Prieto-Matias. PBBCache: An open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies. *Journal of Computational Science*, 42:101102, 2020.
7. Vinay Hanumaiah, Digant Desai, Benjamin Gaudette, Carole Jean Wu, and Sarma Vrudhula. STEAM: A smart temperature and energy aware multicore controller. *ACM Trans. on Embedded Computing Systems*, 13, 2014.
8. Arman Iranfar, Marina Zapater, and David Atienza. Machine Learning-Based Quality-Aware Power and Thermal Management of Multistream HEVC Encoding on Multicore Servers. *IEEE Trans. on Parallel and Distributed Systems*, 29(10):2268–2281, 2018.
9. Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *IEEE Trans. on Games*, 12(1):1–20, 2020.
10. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.
11. Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games, 2019.
12. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, L. Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.
13. Amit Kumar Singh, Alok Prakash, Karunakar Reddy Basireddy, Geoff V. Merrett, and Bashir M. Al-Hashimi. Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs. *ACM Trans. on Embedded Computing Systems*, 16(5s), 2017.
14. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An introduction*, volume 1. MIT press Cambridge, 1998.
15. Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *21st Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 545–559, New York, NY, USA, 2016. ACM.