Why do we need to use Reinforcement Learning?

Consider the following System on Chip shown in Fig. 1, which has a CPU and 4 IO peripherals (keyboard, mouse, monitor and speaker) which are connected to the System memory using an NOC. The NOC has a bi-direction port per component and routes traffic between to and fro from System memory. Each interface has buffering to account for latency of data transfer to and from System memory.

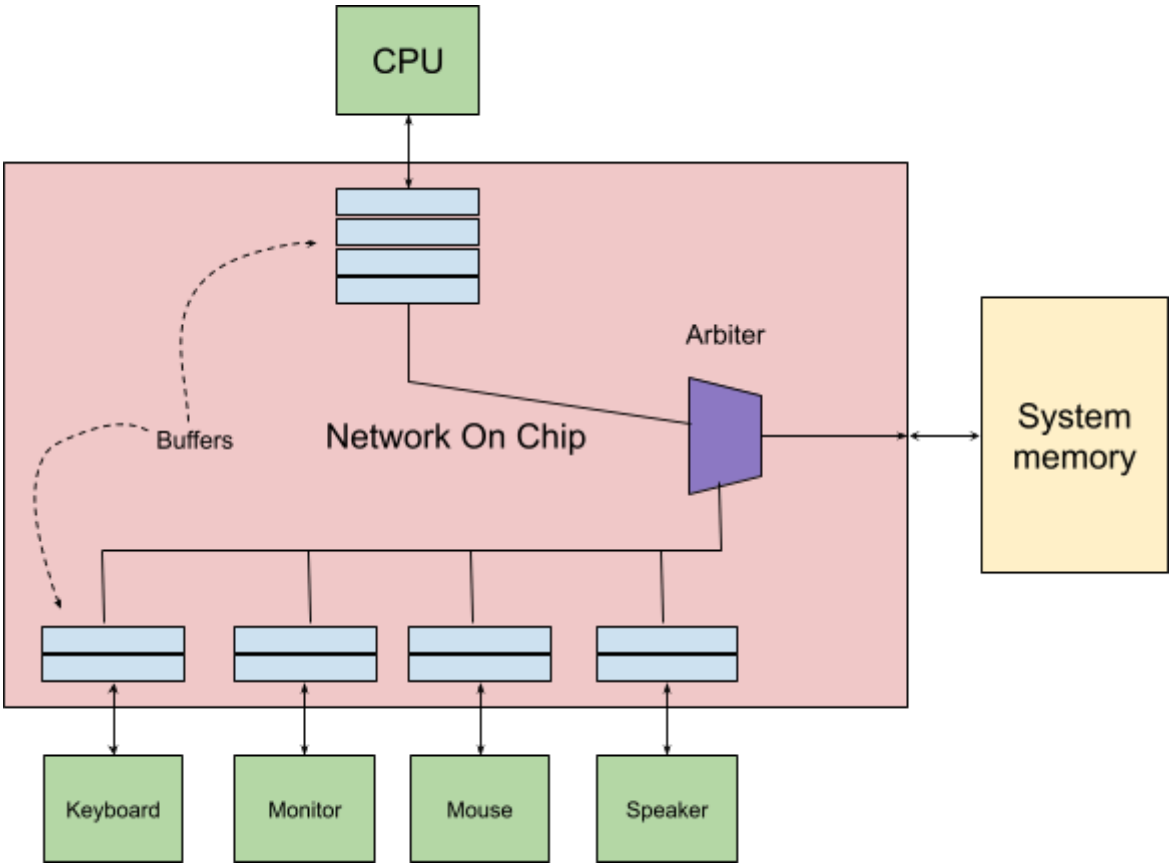

Fig. 1 Block diagram of an SoC

Suppose you are using this system to write a report while listening to music. Since we know what workload is running in the system, we can pre-assign the arbiter weights and average buffer occupancy.

Suddenly, you have copied a huge amount of text from a website and have pasted it in your document or you keep changing the songs because they are not matching the vibes. Such actions, we can guess and write our code accordingly.

But there are infinitely possible workloads that can run on the system and it is quite difficult to guess each one of them and put them in our code. There are also other difficulties such as:

1. It takes a lot of time to find every possibility and coding accordingly.
2. This increases the size of the code and the area of the NOC to accommodate every possibility.
3. Power management is not flexible.

This is where Reinforcement Algorithms step in. In an essential way, these algorithms are closed-loop because the learning system's actions influence its later inputs which is the feature that is required in designing the NOC, adjusting the NOC parameters like weights and buffer size according to the simulator provided monitor output.

Another key feature of reinforcement learning is that it explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment. A system where there are infinitely possible workloads, the algorithm model that assigns the values must be able to learn and understand its uncertain environment. This is possible with the help of Reinforcement Learning.

How to use Reinforcement Learning?

The framework of an RL algorithm consists of states, actions and rewards. Fig. 2 shows the flowchart of how, in general, a Reinforcement Learning algorithm works.

States: States or behaviours represent the current situation or configuration of the NOC system. They include various parameters and metrics that describe the state of the system. In the context of NOC optimization, states may include:

1. **Buffer Occupancy:** Occupancy levels of buffers for CPU and each IO peripheral.
2. **Traffic Patterns:** Current traffic patterns in the NOC, including read and write requests from CPU and IO peripherals.
3. **Arbitration Rates:** Arbitration rates at the input of the arbiter.
4. **Latency:** Latency of pending transactions in the system.
5. **Power Consumption:** Power consumption of the NOC system.

Actions: Actions are the decisions or changes that the RL agent can make to modify the NOC configuration. In the context of NOC optimization, actions may include:

1. **Buffer Sizing:** Increase or decrease the size of buffers for CPU and each IO peripheral.
2. **Arbitration Weight Adjustment:** Adjust the arbitration weights for CPU and IO peripherals.
3. **Throttling:** Adjust the operating frequency of the NOC to manage power consumption.

Rewards: Rewards are the feedback that the RL agent receives for its actions. They indicate how good or bad the actions were in achieving the desired objectives.

1. **Latency Reward:** Reward for minimising transaction latency.
    - Positive reward for reducing latency below a certain threshold.
    - Negative reward for exceeding latency limits.
2. **Bandwidth Reward:** Reward for maximising the throughput or bandwidth of the NOC.
    - Positive reward for achieving high bandwidth utilisation.
    - Negative reward for underutilization or congestion.
3. **Buffer Occupancy Reward:** Reward for maintaining buffer occupancy within acceptable limits.
    - Positive reward for achieving target occupancy levels (e.g., 90%).
    - Negative reward for buffer overflows or underutilization.

4. **Power Efficiency Reward:** Reward for maintaining power efficiency.
    - Positive reward for operating within power limits.
    - Negative reward for exceeding power constraints.

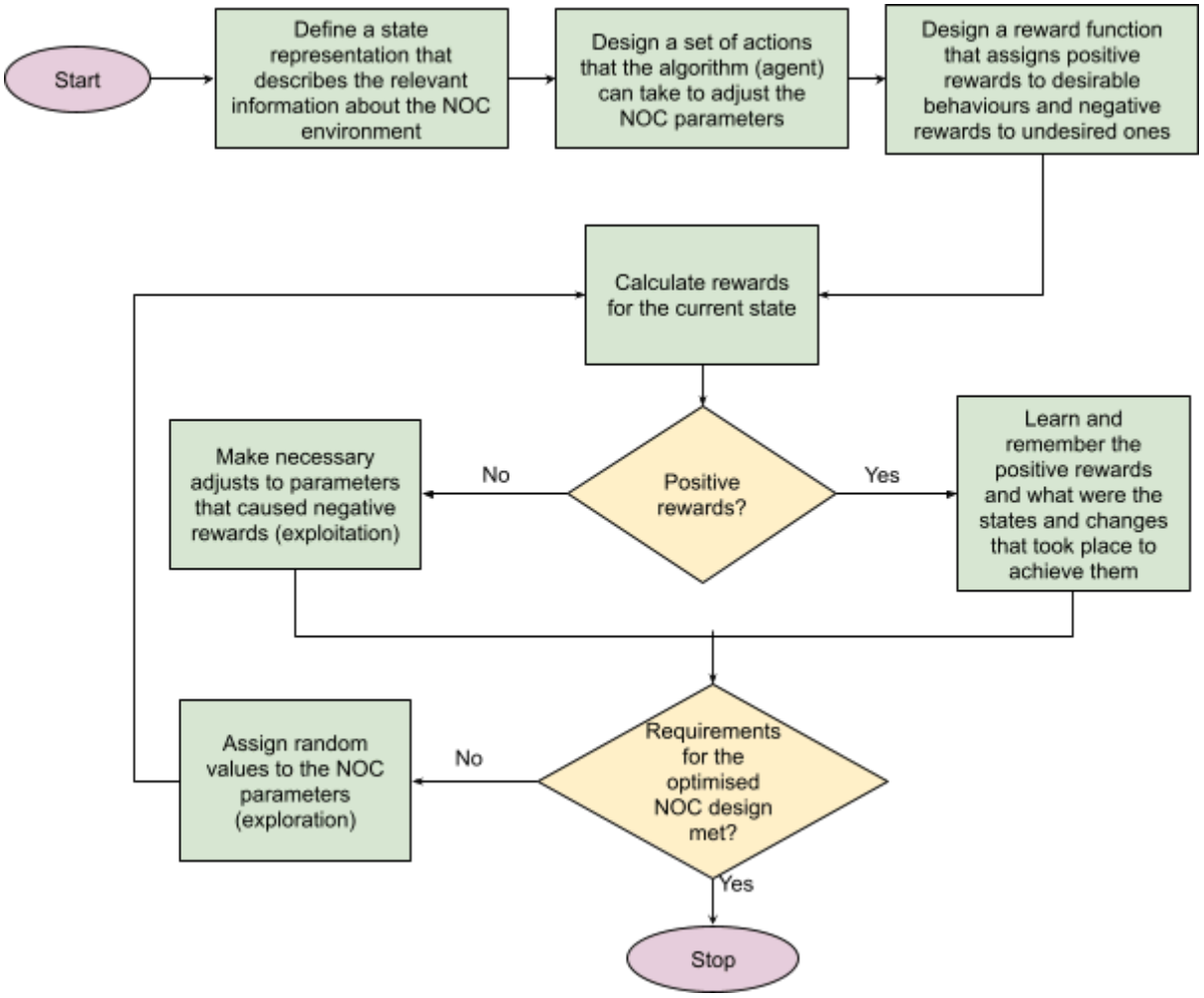

Fig. 2 Typical steps involved in using Reinforcement Learning algorithm

Which Reinforcement Learning algorithm is the best suited for the optimization of NOC design?

Given in the problem statement, the requirements are:

- Measured latency is **<= min_latency**
- Measured bandwidth is at **95% of max_bandwidth**
- Buffer sizing to support 90% occupancy
- Throttling happens only 5% of the time. Ex: for every 100 cycles, throttling should happen for 5 cycles on an average. Throttling is based on get_powerlimit_theshold() being 1 or 0.

A suitable Reinforcement Learning algorithm to meet the above requirements would be **Soft Actor-Critic (SAC)**.

Features of Soft Actor-Critic (SAC):

1. **Policy Optimization:** SAC optimises a stochastic policy in an off-policy manner, allowing for continuous action spaces. This is suitable for adjusting parameters like buffer sizes, arbitration weights, and throttling levels.
2. **Exploration:** SAC uses entropy regularisation to encourage exploration, ensuring that the agent explores different configurations of the NOC effectively while still exploiting learned policies.
3. **Continuous Action Space:** SAC can handle continuous action spaces, which is crucial for adjusting parameters like buffer sizes and arbitration weights to fine-tune the NOC design.
4. **Stability:** SAC is known for its stability in learning, which is important for achieving reliable performance in optimising NOC parameters.

Advantages of SAC:

- **Continuous Action Space:** SAC can handle continuous action spaces efficiently, allowing for fine-grained adjustment of NOC parameters.
- **Stability:** SAC is known for its stable training process, making it suitable for complex optimization tasks like NOC design.
- **Exploration:** SAC's entropy regularisation encourages exploration while still exploiting learned policies effectively.
- **On-Policy Learning:** SAC learns directly from its own experience, which can adapt to changes in the environment and make real-time adjustments.

In conclusion, Soft Actor-Critic (SAC) is a suitable algorithm for optimising the NOC design for the System on Chip described. Its ability to handle continuous action spaces, stability in learning, and effective exploration make it well-suited for the task. By defining appropriate state representations, action spaces, and reward functions, SAC can learn to optimise NOC parameters to meet the specified requirements of latency, bandwidth, buffer occupancy, and throttling efficiently.