Adaptive Hybrid Synchronization Primitives: A Reinforcement Learning Approach

Fadai Ganjaliyev
School of IT and Engineering, ADA University
Baku, Azerbaijan

Abstract—**The choice of synchronization primitive used to protect shared resources is a critical aspect of application performance and scalability, which has become extremely unpredictable with the rise of multicore machines. Neither of the most commonly used contention management strategies works well for all cases: spinning provides quick lock handoff and is attractive in an undersubscribed situation but wastes processor cycles in oversubscribed scenarios, whereas blocking saves processor resources and is preferred in oversubscribed cases but adds up to the critical path by lengthening the lock handoff phase. Hybrids, such as spin-then-block and spin-then-park, tackle this problem by switching between spinning and blocking depending on the contention level on the lock or the system load. Consequently, threads follow a fixed strategy and cannot learn and adapt to changes in system behavior. To this end, it is proposed to use principles of machine learning to formulate hybrid methods as a reinforcement learning problem that will overcome these limitations. In this way, threads can intelligently learn when they should spin or sleep. The challenges of the suggested technique and future work is also briefly discussed.**

Keywords—Spinning; sleeping; blocking; spin-then-block; spin-then-park; reinforcement learning

I. Introduction

While multicore architectures bring new opportunities for parallel software, they also present certain challenges, such as the choice of contention management strategy, which is crucial for the performance and scalability of parallel applications. The diversity of computing environments and unpredictability of application behavior makes this issue even more severe.

General synchronization techniques used to provide concurrent access of threads to shared objects are spinning (busy waiting) or blocking (descheduling the waiting thread). The other approaches are some combinations of the two, such as spin-then-block or spin-then-park. When spinning, to get the ownership of the shared resource, a thread continuously polls the resource until it becomes free, while in case of blocking, the thread relinquishes processor, thereby allowing other threads to utilize CPU. Spinning provides very quick lock handoff and is preferred in undersubscribed scenarios. However, in oversubscribed cases, every type of spinning can create scalability bottlenecks because it is highly CPU intensive by design. To mitigate this issue exponential backoff technique [1-3] inserts random delays between consecutive spins and queue-based protocols [4-9] spread contention among different memory locations. Still, in an overloaded system, spinning is inefficient because it wastefully burns CPU cycles. Blocking, on the contrary, saves processor cycles by descheduling the contending thread even though context switches associated with the lock handoff phase (one to park out and another one to wake up) significantly add up to the critical path. Besides, frequent sleeps and wakeups can make the scheduler very busy and deteriorate its performance.

To balance tradeoffs between spinning and blocking, hybrid, spin-then-block, and spin-then-park strategies are used where threads spin at low and average contentions and block when contention rises [2, 10-13]. These techniques provide quick lock handoff at moderate contentions, meanwhile avoids waste of CPU time because when contention rises, threads suspend themselves, and other threads can do useful work. However, these strategies do not eliminate parking out and waking up from the lock handoff phase. To remove scheduler interaction from the critical path, previous [14] and recent [15] research suggests to maintain system load and to park and wake up threads in bulk as load changes. In summary, these works address two main problems: 1) whether a thread should spin or sleep, and 2) how a thread should make sleeping decisions.

Threads can address these issues more elegantly. Instead of acting in a certain way deemed efficient at particular states of the system, a thread can take action (for example, sleep for a specific duration) and evaluate it by peeking at its outcome, which drives the thread to the goal it is trying to achieve. Eventually, a thread will have a set of state-action pairs that will not only allow it to act optimally at any state but also will help it to predict optimal behavior for future unseen cases.

In summary, this paper makes the following contributions:

- We show that system load cannot serve as the only criteria in sleeping decisions, as previous and recent research states.
- We show that a thread that follows either of the hybrid methods can be treated as an entity capable of learning optimal actions (spin or take a timed sleep) via interaction with the system.
- We formulate both spin-then-block and spin-then-park strategies as a reinforcement learning (RL) problem, which allows a thread to 1) learn when it should spin or sleep 2) adapt its behavior to changes in the system and 3) utilize learned experience to future cases.

The rest of this work is organized as follows. The next section briefly describes the hybrid methods and motivates the described approach. Section 3 presents the suggested method. Section 4 discusses the opportunities and challenges of the approach and future work. Finally, Section 5 summarizes conclusions.

II. Background and Motivation

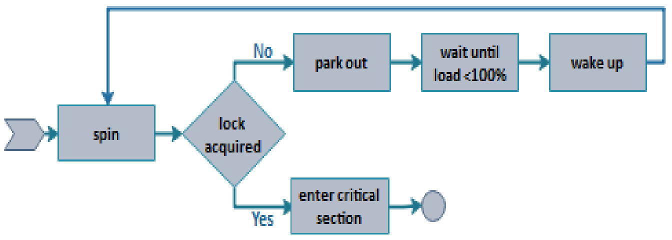
We motivate the need for intelligent learning of sleeping and spinning and provide background on reinforcement learning as applicable to hybrid primitives. Detailed descriptions are out of the scope of this paper, and the interested reader can refer to [10-13] for hybrid methods and to [16-18] for reinforcement learning.

A. Hybrid Synchronization Primitives

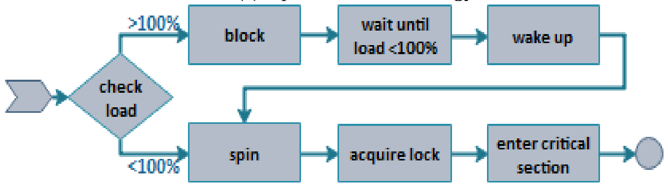
The blocking method extends queue-based spinning protocols in two ways. In the case of the spin-then-block method [10, 11, 19, 20], once a thread enters the system, it either spins or blocks depending on the level contention on the lock. In the other case, a thread may not block right away but may spin for a while and then park itself out, which is the spin-then-park [2, 12-15] strategy. The issue both of the methods are addressing is whether spinning or blocking a better choice at a particular point of time.

Later, researchers suggested [14] to improve it further and move scheduler decisions off the critical path by decoupling contention management from load control. A separate control daemon thread periodically estimates the system load. A set of randomly selected threads is parked out or woken up in response to load change. A more recent work [15] extends this technique for NUMA architectures by maintaining a load metric per socket. In both of the works, system load is the driving factor when deciding about blocking and waking up threads that is done in bulks. Fig. 1(a) and Fig. 1(b) illustrate these approaches. The latter work also addresses memory footprint challenges. But this is not the focus of this work.

Duration of sleep is important. When parked out threads wake up too early, they burn CPU cycles, whereas waking up later than expected lengthens the critical path (a thread still sleeps even though the lock is free). In Fig. 2(a) the lock is released at time t′. A contending thread that is currently sleeping wakes up at t′ and spins until t′. Should it predict lock release time more accurately, it could have slept a bit more so that to wake up right before the lock is released, which would minimize the unnecessary burning of CPU time. In Fig. 2(b) a thread sleeps such that it wakes up much later the lock is released. The lock holder frees it at time t′, but the thread continues to sleep until t″ which lengthens the critical path. This time the issue is different while the reason is the same. To avoid these issues, threads should be able to judge the consequences of their actions. If sleeping for the smallest possible amount of time yields unnecessary sleeping, then a thread should never sleep, no matter what other conditions are, in which case spinning is the only choice. Also, because the number of possible sleep durations is large, threads should be encouraged to transfer learning from one state to other similar states.

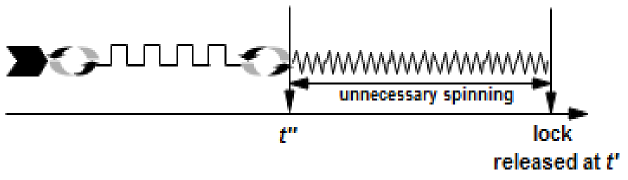


(a) Spin-then-Park Strategy.

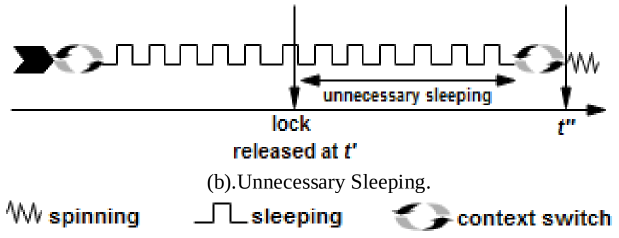


(b) Spin-then-Block Strategy.

Fig 1. Spin-then-Park and Spin-then-Block Strategies.



(a).Unnecessary Spinning.



(b).Unnecessary Sleeping.

Fig 2. Spinning and Sleeping Unnecessarily.

According to previous [14] and recent research [15] estimating system load in blocking decision eliminates scheduler interaction from the blocking phase and improves latency of the lock handoff phase. Threads sleep for a duration proportional to the overload metric value. However, there are other factors that should be taken into account when making sleeping decisions, such as, length of the queue of nodes created by contending threads. If a thread decides to take a sleep, then it should also take into account length of the queue. Each successor thread will hold the lock for the time it takes to execute the critical section. This means that for the same overload factor a thread should sleep for different durations depending on the length of the queue created by successor threads. Otherwise, it may result in unnecessary spinning or sleeping or both.

That is, the following conclusions can be made:

- When making sleeping decisions, both system load and number of successor contending threads should be considered. A thread should have both a global and a local view of its environment.
- To decide about spinning or sleeping, threads should be able to evaluate their actions.

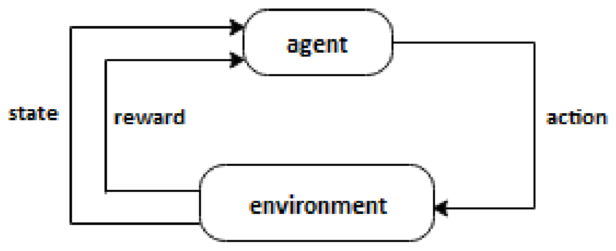
- As the system changes its behavior, a thread should be able to 1) adapt accordingly, and 2) use its experience to act optimally in the future.

B. *Reinforcement Learning and its Applicability to Hybrid Synchronization Methods*

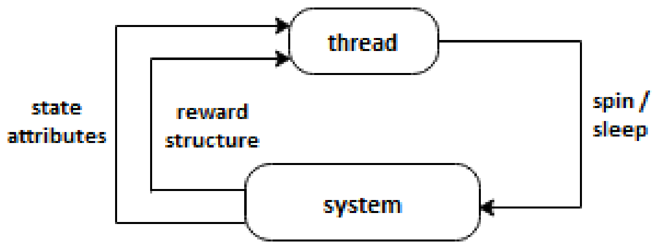
Once a thread enters the system, it has to choose one of several options to contend for the lock: spin, spin for some time, and then sleep or take a sleep. A thread prefers one of these options depending on some factors, such as the load of the system. Instead of just heuristically choosing one of the available actions, a thread may also retrieve feedback from it to determine how good was the action it took at this particular state of the environment. The thread may then remember this and use it to act more intelligently and efficiently in the future. By behaving in this manner, the thread collects a set of state-action pairs, and whenever it enters any state, it chooses the action that is the best one at this state. The feedback must correctly reflect the goal that the thread is trying to achieve. Thus, a thread can be treated as a RL agent whose aim is to learn to behave optimally in an uncertain environment by interacting with it.

Situated between supervised and unsupervised learning, RL is an area of machine learning that deals with sequential decision making problems in which the feedback is limited [18]. Basic concepts of RL are agents, environments, actions, states, rewards, and policies. An agent represents the learning decision maker. The environment is where the agent learns by taking actions, which is the set of all possible moves an agent can make. Whenever the agent takes action, the environment responds to it by placing the agent in a certain state which is an instantaneous situation where the agent finds itself i.e., the environment's input is agent's current state and action in that state and the output is the new state and the corresponding reward. The reward, which is an immediate scalar signal, is the feedback the agent receives as a consequence of its action and helps to measure how good the action at that state was. It essentially evaluates the agent's action in the current state. The policy is a function that maps states to actions and effectively is a strategy that the agent exploits to decide about the next action in the current state. The goal of the agent is to learn the best policy i.e., the policy that maximizes the long-term reward. Fig. 3(a) represents the agent-environment interaction.

Fig. 3(b) shows how a thread's interaction with the system fits within this framework. The thread (the agent) decides to spin or sleep. Note that now it does not block but takes a timed sleep since one of the purposes is to remove lock handoff from the critical path. Consequently, the thread receives a reward (for example, the reward can be modeled as the cost of lock acquisition in terms of CPU cycles). As a result, the thread finds itself in a new state where it can take a different action (for example, sleep for a different duration). Eventually, the thread collects a set of state-action pairs which it then can utilize.



(a).A RL Agent Interacting with its Environment.



(b).A Thread as a RL Agent Interacting with its System.

Fig 3. RL Framework and Thread as a RL Agent.

RL problems are formalized using Markov Decision Processes (MDP). MDP is a tuple <S, A, T, R, γ> with the Markov property, that is, the current state provides sufficient statistics about the future, and thus, all the past information can be discarded. S and A is a finite state and action spaces correspondingly.

T is a transition function, which is the probability distribution over the state space for each states ϵ S, and action a ϵ A. R is an expected reward for taking action in a state and γ is a discount factor. The discount factor guarantees, from one hand, that the algorithm converges, and on the other hand, tells the agent how important are immediate rewards compared to future rewards. The closer it is to 0, the less important future rewards are compared to immediate rewards, whereas values closer to 1 make future rewards count as much as immediate rewards. Transition function and reward together completely define the model of the environment. Fig. 4 shows an example of a non-deterministic MDP state diagram that models an environment with two states S1 and S2, and in both states, the agent has two actions A1 and A2. If the agent, for example, in state S2, takes action A1, it transitions to state S1 and receives a reward of 9 with a probability of 0.3 and finds itself in the same state and receives a reward of 4 with a probability of 0.7. It is important to note that the next state is not determined just by the agent's action in the current state but also depends on the behavior of the environment. For example, in the context of hybrid synchronization methods, the next state (we suppose, at least scheduler load can be one of the state attributes) partly depends on the action of the thread in the current state and partly on the scheduling decisions which is not under the thread's control.

When the model of the environment is known, the agent can derive the optimal policy by using the transition and reward functions. However, in the absence of these functions (which is the case with hybrid methods), the agent needs to interact with the environment and observe its responses, in which case the algorithm is referred to as model-free. In this case, the agent derives the optimal policy without using neither transition function nor the reward function. A popular model-free algorithm used for estimating optimal policy is Q-Learning [17, 18] (more in the next section), which associates a value with each state-action pair and derives optimal policy from these values. Typical issues that the agent needs to address are the following.

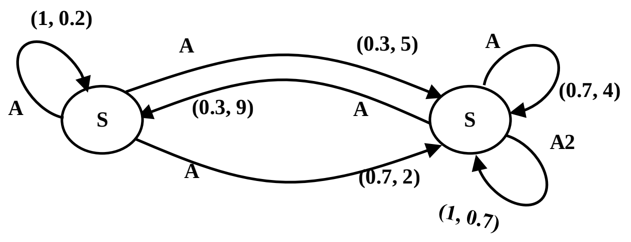


Fig 4. A Non-Deterministic MDP State Diagram with Two States and Two Actions.

Temporal credit assignment. The agent tries to collect the largest amount of reward in the long run. For each received reward, it needs to determine whether it leads to the desired outcome. An action that yields high immediate reward may lead the agent to an undesirable state; in other cases, taking an action that yields no reward may seem undesirable first but may be critical to driving the system to the state with the highest reward. For example, some threads may take sleep for a specific duration and get a high reward for this action because it wastes fewer CPU cycles. However, if too many threads do so when the scheduler is very busy, then the scheduler can quickly become a bottleneck. Therefore, to act accurately, the agent needs careful planning.

Exploration vs Exploitation. To achieve its goal, the agent needs to interact with its environment to gather more information that may lead to an optimal policy, and at the same time, it needs to exploit at best information it has found so far. An excessive investigation might lead to the best policy, but it causes long learning periods, whereas too little investigation may have the agent to accept a suboptimal policy early on.

Generalization. The state space can be exponentially large, and the agent might need to visit a huge number of possible states to try actions and evaluate rewards at those states. Besides, the agent may not have a chance to visit the same state twice over its lifetime. In such cases, the agent has to generalize of experience learned in previous states to new states.

III. RL-BASED HYBRID SYNCHRONIZATION METHODS

Both spin-then-block and spin-then-park methods can be formulated as a RL problem. As stated above, both methods aim to decide whether spinning or sleeping is a more efficient choice at a particular state of the system. If a thread now acts as a decision maker, then it can take any of the available actions. Therefore, it will be agnostic about which hybrid primitive it follows. It is simply trying to find out best (from a reward perspective) actions at certain states. For example, if a thread spins for some time and quickly acquires the lock, then it remembers this choice at states. In a different state, it may notice that spinning does not result in high reward and will prefer a timed sleep.

A. Formulation of Hybrid Primitives as a RL Problem

Now the reward structure needs to be defined, as well as state and action spaces that threads can use to decide about their optimal behavior. Here, also described the techniques can be used to overcome the challenges mentioned in the previous section are also.

Action. A thread can either spin or take a timed sleep but never a combination of both with spinning being first. Initial spinning of the spin-then-park strategy is a waste of CPU cycles now, since if a thread would be scheduled out until it gets the lock, then it should not have spun but rather have taken a timed sleep when it entered the system. To amortize the cost of scheduling out and waking up threads can park out and wake up in batches as previous [14] and recent [15] research suggests.

Reward. The goal of a thread is to acquire the lock in the cheapest and fastest way. If a thread manages to acquire the lock solely by spinning, then it receives a positive reward, otherwise, a negative reward, in which case it would be scheduled out by the scheduler. Next time the thread visits the same or a similar state, it should take a timed sleep and consequently receives a positive reward equal to the duration of the sleep if, as a result, the thread did not sleep unnecessarily (still slept while the lock was free). Eventually, it may sleep for even more amount of time to minimize spinning at that state. Otherwise, the thread receives a negative reward of the same magnitude (to discourage it from sleeping for this duration at this state again because it will add up to the critical path which has to be avoided). Hence, the more a thread sleeps, the more reward it receives, given that the sleep does not result in sleeping unnecessarily. To determine whether it has slept more than necessary, the thread can check whether it has at least one failed spin after waking up. If upon waking up, the thread grabs the lock as a result of the first spin that means the lock was free by the time it woke up, in which case the thread is punished. Acquiring the lock solely by spinning (which should always be possible in case of undersubscribed situation and relatively fewer number of successor threads) should yield the highest reward to encourage the thread to prefer spinning in the first place. Reward structure is shown in Fig. 5.

State. As recent research indicates [15] system load plays an important role in deciding about sleeping: a thread should sleep only in oversubscribed cases and should never do so in undersubscribed situations. The duration of the sleep should be proportional to the overload factor. However, as explained in the previous section, even in undersubscribed cases, a thread may prefer sleeping if the latter is cheaper than spinning (which could be because of a huge number of threads contending for the lock). Hence, state attributes are system load and the number of successor threads contending for the lock.

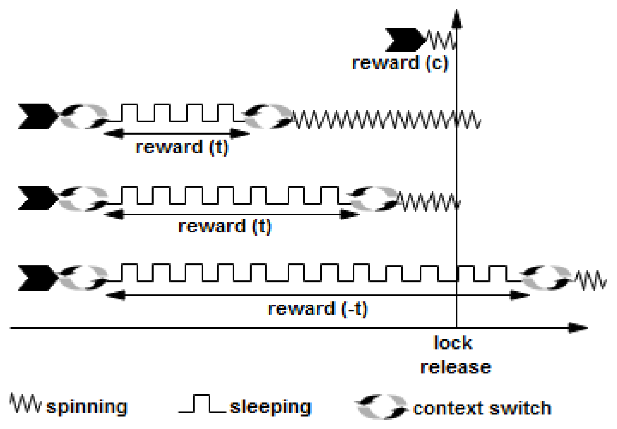


Fig 5. Reward Structure for a Thread as an Agent.

B. Solving Challenges of the Agent

Estimating actions. The temporal credit assignment problem can be addressed by one of the well-known algorithms in RL Q-learning [17, 18], where the agent learns Q-values associated with each state-action pair. One of the key properties of Q-learning is that to derive the optimal policy π*, the agent needs to determine the best action as defined by Q-value. Q-value of a state-action pair (s, a) under policy π is the reward for taking action a in state s plus the sum of discounted future rewards if the policy π is followed thereafter. The agent learns all Q-values for a particular state. Therefore, if at each state, the agent chooses the action that has the largest Q-value, then it effectively follows the optimal policy.

As threads spin or sleep, they continuously update estimates of Q-values based on the rewards they receive. When a thread executes action a_c in state s_c it receives a reward r, transitions to a new state s_n, and chooses an action a_n. The Q-value associated with taking action a_c in state s_c can then be updated by an error using the SARSA [18] update rule:

$$Q(s_c,a_c) \leftarrow Q(s_c,a_c) + \alpha\left[r + \gamma Q(s_n,a_n) - Q(s_c,a_c)\right] \qquad (1)$$

Recall that γ is a discount factor and determines the importance of future rewards. The learning rate α (or step size) determines how quickly the agent learns. Setting it to 0 means Q-values are never updated, that is, the agent does not learn at all, whereas higher values of it facilitate faster learning periods. In the context of hybrid methods, presumably very high values of for example, γ = 0.95, should work quite well. This is because a thread is highly agnostic about the effect of its action on the future state of the system. But the value for should be experimentally tuned for good performance. The SARSA rule is guaranteed to converge to the optimal policy, assuming that each state can be visited by the agent infinitely often.

Addressing exploration-exploitation tradeoff. If a thread never chooses certain actions in a given state, it would not be able to learn the associated Q-values. Even if the optimal policy has been already learnt, the dynamics of the system can make the current policy obsolete. Furthermore, even though threads are not willing to spend much time on learning, they have to try different actions in a given state to evaluate corresponding rewards. Therefore, threads must continuously explore their environment while at the same time, utilize the best policy they have found so far.

To balance exploration with exploitation, one can implement a widely used ε-greedy [18] action selection technique. The agent (the thread) takes the actions that are optimal most of the time but to try more actions and potentially find ones with a higher reward, introduces randomness (ε factor). Threads randomly take actions with probability ε which intuitively should be set to small values to guarantee that they continue trying different actions in each state, while the majority of the time utilizing the best policy they have derived. It is also important to note that after a while, it is possible to gradually decrease the ε probability so that exploitation prevails, and the model converges to an optimal policy.

Efficient generalization and quantization. Model-free RL techniques assume that Q-values can be stored in a look-up table with one entry for each. However, when the state space is large, the issue is not only excessive storage requirements for storing Q-values but also time to accurately maintain these values. This problem is known as the curse of dimensionality and can be addressed in two ways.

One way consists of discretizing the state space into a smaller number of cells. In this case, all states within each cell are aggregated and linked to a single Q-value. However, if the discretization of the state space is coarse-grained, then some states can be hidden, preventing the agent from learning the optimal policy. On the other hand, a fine-grained discretization may result in too many cells, and the agent may not be able to generalize, and the amount of training data will increase.

CMAC [21] is a computationally efficient generalization and resolution technique with extremely fast learning capability and the special architecture, which makes it effective from an implementation perspective. It takes an arbitrary number of state variables and lays axis-parallel rectangles over them, known as tilings (as shown in Fig. 6). Overlapping partitions are called tiles. The number of the tiles and the widths of the tilings are generally set at design time.

The tilings are offset from each other by the same amount and maintain the weights of each of its tiles. The center of each tile determines which state values activate which tiles. The method computes a value of any given point as a sum of the weights of the tiles, one per tiling. That is, to calculate a Q-value for a given state, all values from tiles sharing the center are summed up. Analogously, an update to a value will heavily affect nearby points and not as heavily farther points. The function approximation is trained by adjusting the weights of each of the involved tiles. In this way, CMAC achieves the quantization of continuous state space into tiles while retaining the capability to generalize by means of several overlapping tilings. Other function approximation methods, such as radial-basis, instance, and case-based approximators [22], are not suitable candidates in the context of hybrid synchronization methods.

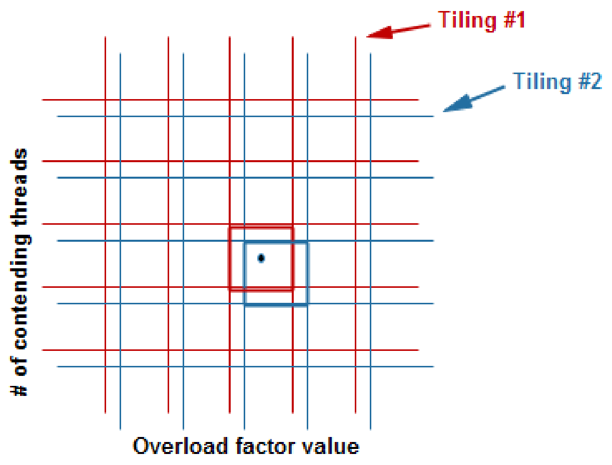


Fig 6. CMAC using Two Overlapping Tiles for Efficient Resolution. Two Tilings are shown to simplify the Figure.

IV. Discussions and Future Work

Recently, it has been suggested [23] to formulate the spin-then-block method as a reinforcement learning problem. However, it still leaves much room for improvement, which was the focus of this work. First, the reward structure has been refined. Secondly, it is shown that the same idea could be applied to the spin-then-park strategy. Thirdly, it is suggested to take system overload factor as one of the state attributes (rather than the only number of currently running and waiting threads on CPU), and that action can be a decision of a batch of threads. Finally, the techniques and algorithms that are suited well in the context of hybrid methods for overcoming issues faced by the agent are described (the thread). Effectively, the technique presented in [14] and [15] is wrapped into a RL framework, but this idea has been arrived at independently.

Although the approach presented here promises competitive results, there are certain challenges that should be considered. Threads lifetime can be very short and adding policy related updates can lengthen it. One way to overcome this issue is to run updates not for every action of every thread but every few actions or every few time units. Another solution to this problem is to run additional helper threads to monitor, track and optimize threads behavior.

The second issue is related to the number of policies. When many lock instances are involved, the number of policies to be maintained can be huge. Two threads contending for different locks may exploit the same policy if the length of the critical section protected by these locks is the same. That is, the method can significantly reduce the number of policies to be maintained if policies can be clustered on the length of the critical section. Threads then will maintain a fewer number of policies, one per cluster. Future experiments will reveal more details on this.

Currently, no experiments have been set up to evaluate the suggested approach. Nevertheless, the theoretical argumentations presented here allow having a base to claim that the technique, if implemented, will allow application and system programmers to avoid the burden of balancing spinning and blocking. The method will monitor itself and optimize and adapt to the dynamics of the system.

The primary goal of the near future work is to test the presented method and compare it with the state-of-the-art implementations of hybrid primitives, such as [15]. Also, extensive experiments will reveal the limitations of the suggested technique and cases where it may not be applicable.

Another part of the future work is to apply this idea for the case of NUMA machines, where threads are encouraged to spin locally rather than remotely. For example, the state can be modeled as the load level of the interconnect module. Threads can be given larger rewards if they spin locally and smaller rewards if spinning is remote (remote spinning cannot be eliminated completely, otherwise threads, ultimately, will spin only locally and will not make progress). In this way, threads can learn that local spinning is preferred whenever they are about to start contending for the lock.

V. Conclusion

Designing a hybrid synchronization primitive that performs well in both under- and oversubscribed scenarios is challenging. In this work, a RL based approach for implementing hybrid synchronization methods (namely, the spin-then-block and spin-then-park strategies) has been presented. It is suggested to make use of principles of machine learning, which a more generic approach. It may release the application developers and system programmers from choosing the appropriate contention management strategy and will optimize and adapt itself to the system as it changes its behavior.

Acknowledgment

This work has been carried out at the Center of Data Analytics and Research at ADA University.

References

[1] Facebook. A persistent key-value store for fast storage environments, http://rocksdb.org, 2012.

[2] X. Leroy, The open group base specifications, http://pubs.opengroup.org/onlinepubs/9699919799, no. 7, 2016.

[3] Torvalds L. The linux kernel archives, https://www. kernel.org, 2017.

[4] I. Calciu, D. Dice, Y. Lev, V. Luchangco, VJ. Marathe, N. Shavit, "NUMA-aware reader-writer locks," Proceedings of Eighteenth ACM Symposium on Principles and Practice of Parallel Programming, pp. 157–166, 2013.

[5] M. Chabbi, J. Mellor-Crummey, "Contention-conscious, locality-preserving locks,", Proceedings of Twenty-First ACM Symposium on Principles and Practice of Parallel Programming, vol. 22, pp. 1–14, 2016.

[6] D. Dice, VJ. Marathe, N. Shavit, "Flat-combining NUMA Locks," Proceedings of Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 65–74, 2011.

[7] D. Dice, VJ. Marathe, N. Shavit, "Lock cohorting: a general technique for designing NUMA locks," Proceedings of Seventeenth ACM Symposium on Principles and Practice of Parallel Programming, pp. 247–256, 2012.

[8] JP. Lozi, F. David, G. Thomas, J. Lawall, G. Muller, "Fast and portable locking for multicore architectures," ACM Transactions on Computer Systems, no. 33, vol. 4, pp. 1–62, 2016.

[9] V. Luchangco, D. Nussbaum, N. Shavit, "A hierarchical CLH queue lock," Proceedings of Twelveth International Conference on Parallel Processing, pp. 801–810, 2006.

[10] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, A. Vainshtein "Optimal strategies for spinning and blocking," Journal of Parallel and Distributed Computing, n. 21, vol. 2, pp. 246-254, 1994.

[11] H. Franke, R. Russell, MK. Fuss, "Futexes and furwocks: fast userlevel locking in linux," Proceedings 2002 Ottawa Linux Summit, pp. 479-495, 2002.

[12] I. Molnar, Linux rwsem, http://www.makelinux.net/ ldd3/chp-5-sect-3, 2006

[13] I. Molnar, D. Bueso, "Generic mutex subsystem, https://www.kernel.org/doc/Documentation/locking/mutex-design.txt,", 2016.

[14] FR. Johnson, R. Stoica, A. Ailamaki, TC. Mowry, "Decoupling Contention Management from Scheduling,". Proceedings of Fiftenth ACM International Conference on Architectural for Programming Languages and Operating Systems, pp. 117-128, 2010.

[15] S. Kashyap, C. Min, T. Kim, "Scalable NUMA-aware blocking synchronization primitives," Processdings of the USENIX Conference on Usenix Annual Technical Conference, pp. 603-615, 2017.

[16] D. Bertsekas, Neuro Dynamic Programming: Athena Scientific, 1996.

[17] T. Mitchell, Machine Learning: McGraw-Hill, Boston, 1997.

[18] R. Sutton, A. Barto, Reinforcement Learning: MIT Press, 1998.

[19] J. Mauro, R. McDougall: Solaris Internals: Core Kernel Components: Sun Microsystems Press, 2001.

[20] JK. Ousterhout, "Scheduling techniques for concurrent systems,". Proceedings of Third International Conference on Distributed Computing Systems, pp. 22-30, 1982.

[21] R. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding,", Proceedings of Eigth International Conference on Neural Information Processing Systems, pp. 1038-1044, 1996.

[22] JC. Santamaria, RS. Sutton, A. Ram, "Experiments with reinforcement learning in problems with continuous state and action spaces,". Adaptive Behavior, n. 6, vol. 2, pp. 163–217, 1997.

[23] F. Ganjaliyev, "Spin-then-sleep: A machine learning alternative to queue-based spin-then-block strategy," International Journal of Advanced Computer Science and Applications, n. 10, vol. 3, pp.605-609, 2019.