Spin-Then-Sleep: A Machine Learning Alternative to Queue-based Spin-then-Block Strategy

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

***Abstract*—One of the issues with spinlock protocols is excessive spinning which results in a waste of CPU cycles. Some protocols use the hybrid, spin-then-block approach to avoid this problem. In this case, the contending thread may prefer relinquishing the CPU instead of spinning, and resumes execution once notified. This paper presents a machine learning framework for intelligent sleeping and spinning as an alternative to the spin-then-block strategy. This framework can be used to address one of the challenges faced by this strategy: the delay in the critical path. The work suggests a reinforcement learning based approach for queue-based locks that aims at having threads learn to spin or sleep. The challenges of the suggested technique and future work are also discussed.**

Keywords—Spinlock; spin-then-block; reinforcement learning; queue-based lock; intelligent sleeping

I. Introduction

Spinlocks have been widely used in multicore systems as a mechanism to guarantee concurrent access of threads to a critical section of code. A thread will poll (spin on) a variable in a loop to grab the lock, to enter such a shared piece of code. Once the lock is free, the thread will flip it and can enter the critical section. After it has finished executing the critical region, it flips the flag back to its original value so that other threads can acquire the lock as well.

Researchers have developed different types of spinlock protocols. Test-and-test-and-set with exponential backoff (TTSE) is the simplest among them [1]. Here, all threads spin on a globally shared lock flag by issuing a read operation on it until the lock is found free. At this point, a thread issues the test-and-set atomic instruction to acquire the lock. A random delay is inserted between consecutive spins, to reduce the simultaneous thread attack upon lock release. TTSE protocol is recommended for low and medium contention levels, as it scales poorly when contention for the lock is high.

Ticket locks [2] maintain global counters to provide concurrent access of threads to a critical section. The lock is composed of two variables: a ticket and a grant variable. Whenever a thread wants to acquire the lock, it atomically increments grant variable value and spins unless the two are equal. Once these variables are equal, the thread can enter the critical section. When the thread exits the critical section, it advances the value of the grant variable. Ticket locks guarantee First-In-First-Out (FIFO) order but suffer from the same issue of “thundering herd” that TTSE protocol does.

Queue-based locks [3, 4, 5] spread contention on the lock by maintaining a list of linked nodes created by contending threads. Threads do not spin on a single lock variable, but each thread spins on a flag of its successor [3] or the flag of its own [4], thereby spreading contention among different memory locations in the system. Once the lock holder exits critical section, it updates either its flag (when predecessor spins on it) or the predecessor’s flag (when the predecessor spins on its flag). Though vulnerable to preemption [6], queue-based locks are an elegant solution for high contention, and they guarantee FIFO order.

Spinlocks are an attractive synchronization solution when the critical section is short. However, when contention for the lock is high, spinning can be inefficient either, since concurrent threads may cause unnecessary CPU utilization. To avoid burning CPU cycles, the spin-then-block approach is used: a thread does not spin but relinquishes the CPU, and upon lock release, the holder wakes up the waiting thread which in turn grabs the lock. From the other hand, this adds up to the critical path of the application because every unlock phase requires waking up the waiting thread. A better option would be not to block and sleep until notified but to go into a timed sleep so that to wake up just in time – right before the lock release. Thus, this would achieve two important goals at the same time: first, avoid unnecessary CPU burn and second, remove lock handoff delay. We call it *spin-then-sleep* strategy.

The questions this work addresses are the following: Once a node created by a contending thread joins a queue, should the thread spin or should it sleep? Also, if it decides to take a sleep, then how much it should sleep? For the first question, the thread has to estimate which either of the two ways will utilize fewer CPU cycles. As to the second question, the thread has to be able to predict when the lock will be released.

Key idea: The suggestion is to treat the thread as an agent whose goal is to automatically learn the cheapest and fastest way to acquire lock via interaction with the system.

The rest of this paper is structured as follows: Section 2 reviews related work. A short background is provided in Section 3. The suggested approach is presented in Section 4. Finally, the challenges, limitations and future work are discussed in Section 5.

II. Related Work

This paper presents an approach that can serve as an alternative to the spin-then-block strategy. The key feature is to feed adaptivity into spinning and sleeping. So, the closest works to the one presented here are adaptive spinlocks [7, 8, 9], that have been of particular interest to researchers as well. These works aim at making spinlocks self-aware. That is the algorithm monitors and tunes itself accordingly. Thus, in [7] a reactive algorithm is developed which utilizes three protocols: TTSE, combining tree [10] and MCS lock [4] which is a queue-based lock. The algorithm switches between protocols depending on the contention level on the lock. For example, when the TTSE protocol fails to get the lock after some number of times, it switches to the MCS lock. In the opposite direction, the algorithm makes a switch when the queue is found to be empty for a number of successful fetch-and-op requests.

Another work [8] develops a backoff protocol that does not require experimentally tuned parameters. Here, the finding is that backoff delay depends strongly on the delay outside of critical section (DoCS) which is defined as the time between when the lock holder releases the lock and the first attempt to reacquire the lock. A heuristic, $base_l$, is found that depends on DoCS and which has the following form:

$$\text{base}_l = \frac{a \cdot \text{DoCS} + b}{\text{DoCS}^2} \# \quad (1)$$

The DoCS variable is computed via overhead that is defined as follows:

$$\text{overhead} = \frac{\text{latency of remote memory reference}}{\text{latency of L1 cache reference}} \# \quad (2)$$

The algorithm needs only this variable. Function $base_l$ is computed once for each lock, and the algorithm adjusts backoff delay from this value depending on the load level that is divided into two phases: load rising phase and load dropping. Whenever a spinning thread observes a rise or drop in the load, it adjusts its delay derived from the variable $base_l$.

Authors of [9] have developed a spinlock library Smartlocks that uses reinforcement learning method of machine learning to achieve a user-defined goal which can be related to performance, power, problem-specific criteria or some combination of these. The application must be connected to a specific framework that measures the performance characteristics of it. Performance related data that arrive from this interface serve as a reward signal to the machine learning engine of the Smartlocks that run in separate helper threads. The library currently supports TTSE, Ticket Locks, MCS and a few other and maintains three main components: The Protocol Selector, the Wait Strategy Selector, and the Lock Acquisition Scheduler. Protocol Selector is responsible for switching between protocols when a predefined threshold of contention level is reached. The Wait Strategy Selector defines what action threads must take when they fail to get the lock and is not implemented since each protocol has a fixed waiting strategy. The function of the Lock Acquisition Scheduler component is to generate policies for lock acquisition and to switch between them. The policy is not updated at every lock acquisition request but every few attempts which are not related to application lock acquisitions.

III. Background

This section gives brief information on the spin-then-block strategy. We also motivate the need for intelligent learning of sleep duration, as well as when to spin and when to sleep and provide a short background on reinforcement learning too.

A. Anatomy of Spin-Then-Block Strategy

Once a thread links its node to a queue of nodes created by contending threads for acquiring the lock, it has two options: spin or release CPU and resume when notified. If the contention for the lock is low, the thread would prefer spinning, since it will provide faster lock acquisition and avoid scheduler interaction. In case the contention for the lock is high, the thread may prefer giving up the CPU by suspending itself which involves a context switch. The thread, then, will wait until the lock holder explicitly wakes it up upon lock release. The notification will be followed by another context switch, to restore the state of the thread to what it was before the suspension. This behavior is known as a spin-then-block method. Solaris mutex [11] is an example of it. This mutex spins at low and medium contention and switches to blocking when contention rises. Spin-then-block strategy suffers from one major drawback: notification and subsequent wakeup of the waiting thread lengthen the critical path. If the thread could approximate timestamp of lock release, then it could have gone into timed sleep so that to wake up right before the lock is freed which would eliminate lock handoff delay, thereby reducing the length of the critical path. The third option is to spin for a while and then park itself out which is known as spin-then-park strategy. In this work, we don't consider this.

B. Motivation

An important factor here is duration of the sleep. Assume, a thread T_1 which holds the lock is executing the critical section. Suppose, it has acquired the lock at time t_1' and will release it at time t_1. A thread T_2 adds a node the queue and sleeps such that it wakes up at $t_2 < t_1$. Another thread T_3, then, enters the system, links its node to the queue and sleeps as well such that it will wake up at $t_3 > t_1$. Once thread T_2 wakes up at time t_2 , it will spin from t_2 to t_1. If, in comparison to $t_1 - t_1'$ the difference $t_1 - t_2$ is huge, then the sleep duration was too small which will cause unnecessary spinning. Thread T_2 could have slept instead, should it predict lock release time more accurately.

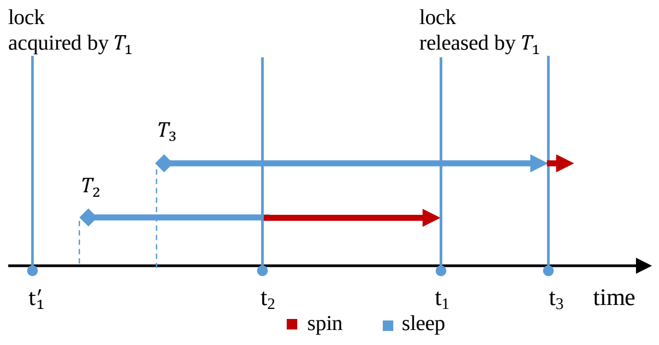
Additionally, thread T_3 will not be able to grab the lock once it is free, since by the time lock is released it will not have its sleep finished (if a thread that has gone into a timed sleep, there is no way to wake it up). Thread T_3 should have slept for shorter amount of time. In other cases, a thread should not sleep at all if sleeping for the smallest amount of time always yields sleeping more than necessary, no matter how many threads contend for the lock and how loaded the system is. İn this case pure spinning should the preferred choice. Fig. 1 illustrates these scenarios. Hence, it is crucial to be able to decide whether sleeping at all is a good choice or not and if it is then to sleep for such a period of time that will minimize spinning by maximizing sleep duration without sleeping unnecessarily (still sleeping even though the lock is free).

C. *Reinforcement Learning Paradigm*

Reinforcement Learning (RL) is a class of supervised learning algorithms in machine learning [12]. The goal of RL is to have an agent learn how to behave in an uncertain environment by interacting with it. A scalar reward signal guides the learning process and the agent has to learn to maximize it.

RL is formalized using the Markov Decision Process (MDP). MDP is defined as a tuple $< S,\ A,\ T,\ R >$, where S is a set of states, A is a set of actions, T is a transition probability function, and R is a scalar reward. A state is collection of characteristics that represent every state that the agent can be in. The transition function a is probability distribution over the state space for each state $s \in S$ and action $a \in A$. Reward function is an expected reward for performing an action in a state. Transition function together with reward function defines the model of the environment.

RL is a *model-free* technique, i.e. it assumes that the agent does not possess any information about the environment. Thus, the agent must interact with the environment to collect the reward. At each step, the agent senses the current state, chooses an action and transitions to the next state followed by receiving a reward for choosing this action at this state. The goal of the agent is to learn an optimal policy that maps states to actions and maximizes its cumulative reward over the long-term. The agent tries to learn the optimal policy without learning transition and reward functions. Fig. 2(a) depicts agent-environment interaction.



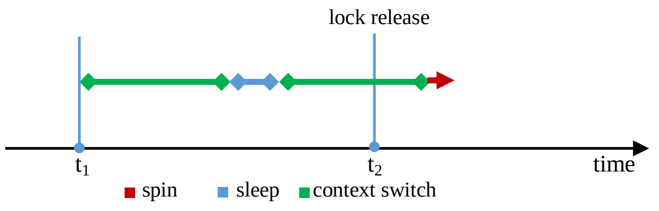
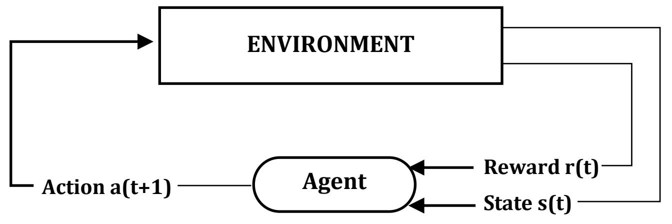


Fig. 1. (a). Sleeping and Spinning Redundantly; Thread T_2 could have Continued Sleeping from t_2 to t_1 Rather than Spin; Thread T_3 should not have Slept from t_1 to t_3; (b). Sleeping for the Shortest Amount of Time Always Yields Unnecessary Sleeping Since the Lock is Passed by.



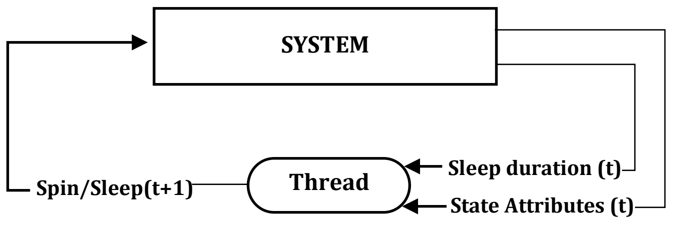


Fig. 2. A Reinforcement Learning Agent Interacting with the Environment; (b) Thread as an RL-Agent.

Fig. 2(b) shows how spin-then-sleep strategy maps to the RL framework. The thread, which represents the agent, takes actions, such as spin or sleep. As a result, the thread receives a reward signal. The reward can be designed in different ways. For example, if any sleep that does not yield unnecessary waiting is enough, then the reward can take only three values: 0 for pure spinning, 1 for a sleep that does not result in unnecessary sleeping and -1, otherwise. The thread then transitions to a different state where it takes the same or different actions. In this way, thread learns the best action at each state. The next section discusses the state, and the reward structure is in more detail.

IV. RL-Based Spin-Then-Sleep Strategy

This section explains how the spin-then-sleep strategy can be formulated as an RL problem. It describes what serves as a reward, action, and state.

Reward. The reward has to lead to the goal. A thread that linked its node to a queue has two choices to proceed: spin only or sleep followed by spinning. The latter should be preferred if it does not yield redundant sleeping because it will be cheaper. Otherwise, pure spinning is preferred. From the other hand, to eliminate spinning completely, the thread may sleep for a sufficiently large period of time. In such a case, upon wakeup, the thread will grab the lock right away because it is free. However, the lock could have been freed long ago. The length of the critical path will be delayed dramatically then. Thus, upon acquiring the lock, the thread must know whether its sleep resulted in unnecessary sleeping or not. The thread can find it out by requiring *at least one spin to fail and the subsequent spin to succeed*. At least one failed spin will guarantee that by the time the thread requests the lock, the holder has not yet released it yet. So, the reward is defined as follows: given that a sleep followed by spinning does not result in redundant sleeping, the more a thread sleeps, the more reward it receives. On the contrary, if a sleep for some duration followed by spinning does result in redundant sleeping, then it receives a negative reward. Pure spinning gets a reward of 0. Fig. 3 depicts these cases.

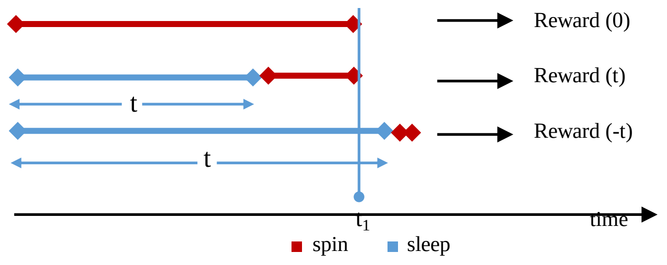


Fig. 3. Rewards for Sleeping and Spinning. Lock is Released at Time t_1.

```
acquire_lock (duration, …) {
  if (duration = 0) then spin;
  else {sleep(duration); spin;}
}
```

Fig. 4. Pseudo Code for Action Acquire_Lock.

Action. Spinlock protocols usually maintain three methods: a method for lock acquisition, lock release, and execution of critical section. Typically, it is the method for lock acquisition, say spin(), where thread continuously spins until it gets the lock. This method of the protocol and routine for sleeping (which includes sleep and wakeup), say sleep(), can be united into a single action of the thread as an agent, say acquire_lock(). That is, the action acquire_lock acts as a function of a single parameter – sleep duration. If this duration is zero, then no sleeping is involved, and the thread only spins. Otherwise, it sleeps for the specified duration and spins the rest of the time. Fig. 4 shows a pseudo code for this. Additionally, the thread-agent will have other two actions (for execution of critical section and lock release) that are no different from the two methods of the thread. Action acquire_lock will always be followed by the action for executing critical section which in turn is followed by the action for releasing the lock.

State. It is assumed that the system is running on Linux. In order to derive state attributes, one needs to determine what affects the time it takes the thread to resume when it intends to take a sleep. Whenever a thread is about to do so, the scheduler is invoked. Scheduler activity results in scheduling latency and dispatch latency. The former is the time it takes to make scheduling decisions, i.e. time to insert a thread into scheduler runqueue (queue of threads that are ready to run on CPU but cannot because CPU is busy) or pick up one from the runqueue to run on CPU. Starting from the 2.6.23 kernel, Linux implements the Completely Fair Scheduler (CFS) [13]. CFS spends $O(\log N)$ time for insert and delete operations, where N is the number of threads in the runqueue, and constant time for a search operation. It achieves that by making use of the red-black tree to hold tasks sorted by their weights and always picking up the leftmost node of the tree to run on the CPU. Thus, scheduler latency which is essentially a function of number threads in the runqueue (perhaps of priority classes as well), contributes to time it takes the thread to be rescheduled on CPU.

Dispatch latency is the time it takes to complete a context switch which is the time to store the state of the thread going into sleep and restore the state of another thread to run. After the first context switch is completed, the current thread now sleeps. Sleep duration should take into consideration the number of contending threads for the lock. The more threads contend for the lock, the more a thread should sleep to eliminate spinning as much as possible. Once sleep duration is over, there is no guarantee that it will get access to the CPU immediately (in an overloaded system). It depends on how loaded (busy) the system is. The load of the system can be expressed as a function of scheduler runqueue and number of threads executing on CPU, for example, as a ratio of average number of threads running on CPU to the average number of threads in the scheduler runqueue per unit of time.

Therefore, the number of threads in the scheduler runqueue, number of threads currently running on CPU and number of threads contending for the lock can serve as candidates for state attributes. At this point, the spin-then-sleep strategy can be regarded as an RL problem.

V. Discussions And Conclusion

Modeling of the spin-then-sleep strategy as a reinforcement learning problem promises competitive results. However, certain challenges and limitations are encountered as well.

State space, as well as action space, is continuous. Therefore, the learning process may be inefficient both from performance and storage point of view. Besides, since the state space is large, the thread may never have a chance to visit the same state more than once. Hence, the thread will not be able to try actions at that particular state. In such a case, a generalization technique such as CMAC [14] can be utilized. One can use it to generalize the learned experience from previous states to new states.

Another challenge is related to the exploration-exploitation tradeoff. From one side, threads need to try different actions to see their results (rewards), and from the other hand, threads are not willing to spend much time on learning, since they have to progress the application for which performance is crucial. To balance the exploration-exploitation tradeoff, one can use soft-max policy. To improve it further, one can trigger computations (reward calculation, policy update) not after *every action of every thread* but every few actions, like in [9] or every few time units.

Reward evaluation is easy to do, and policy update can be embedded into threads lock release phase which, intuitively, should require much fewer CPU cycles than lock handoff. Another option is to have additional threads to maintain it. However, if each lock would maintain a separate policy, then space requirements can be dramatic. Locks can be clustered on some property. An appropriate candidate for it can be the *length of critical section* protected by the lock. Locks clustered to a particular group will maintain a separate policy. It, thus, will reduce the number of total policies, even though additional contention points may arise as multiple threads may attempt to update the same policy at the same time. Future experiments will reveal more details on this.

Though the presented approach is generally quite promising, there exist situations in which case it cannot be applicable. First, it assumes that context switch time is constant which is not always the case. The direct cost of context switch that includes pipeline flush and Translation Lookaside Buffer (TLB) reload will be different for different threads. Moreover, context switch also has associated indirect cost. When a thread wakes up and resumes execution, it may not find the data it needs in the CPU cache, and thus a cache miss will occur. This will affect the time it takes the thread to resume. For different memory access models, this cost will vary. Also, the reward structure is entirely agnostic of the load of the scheduler. It targets at minimizing the cost associated with lock acquisition and may do so even at the expense of deteriorating scheduler performance. In an extremely overloaded system, mostly sleeping will be preferred but too many context switches can make the scheduler very busy.

This work has explored one of the challenges faced by the spin-then-block method related to critical path delay at the lock handoff phase. As a solution, a more generic, a machine learning based approach is suggested to have threads learn when to sleep or spin. The technique models lock acquisition and release as a reinforcement learning problem. It can also be used to release the software designer from hardcoding cases that decide sleeping or spinning. As of now, no experimental setup has been done to test this design. Future studies will concentrate on running experiments to improve it, for example, by refining the reward structure. Certain developments can be made to reduce the action space as well. All this is a part of future work.

Acknowledgment

This work has been carried out in Center for Data Analytics Research at ADA University.

References

[1] T. E. Anderson, “The Performance of spin lock alternatives for shared-memory multiprocessors,” IEEE Transactions on Parallel and Distributed Systems, vol. 1, pp. 6–16, 1990.

[2] D. P. Reed, R. K. Kanodia, “Synchronization with event counts and sequencers,” Communications of the ACM, vol. 22, pp. 115-123, 1979.

[3] J. M. Mellor-Crummey, M.L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” ACM Transactions on Computer Systems, vol. 9, pp. 21-65, 1991.

[4] P. S. Magnusson, A. Landin, E. Hagersten, “Quelocks on cache coherent multiprocessors,” Proceedings of Eights International Parallel Processing Symposium, pp. 165-171, 1994.

[5] A. Kägi, D. Burger, J. R. Goodman, “Efficient synchronization: let them all eat QOLB,” Proceedings of the Twenty-Fourth Annual International Symposium on Computer Architecture, pp. 170-180, 1997.

[6] B. He, W. N. Scherer, M. L. Scott, “Preemption adaptivity in time-published queue-based spin locks,” Proceedings of the Twelfth International Conference on High Performance Computing, pp. 7-8, 2005.

[7] B. H. Lim, A. Agarwal, “Reactive synchronization algorithms for multiprocessors,” ACM SIGOPS Operating System Review, vol. 28, pp. 25-35, 1994.

[8] P. H. Ha, M. Papatriantafilou, P. Tsigas, “Reactive spin-locks: a self-tuning approach,” Proceedings of the Eighth International Symposium on Parallel Architectures, Algorithms and Networks, pp. 33-39, 2005.

[9] J. Eastep, D. Wingate, M. D. Santambrogio, A. Agarwal, “Smartlocks: self-aware synchronization through lock acquisition scheduling,” Proceedings of the Seventh International Conference on Autonomic Computing, pp. 215-224, 2010.

[10] J. R. Goodman, M. K. Vernon, P. J. Woest, “Efficient synchronization primitives for large-scale cache-coherent multiprocessors,” Proceedings of the third international conference on Architectural support for programming languages and operating systems, pp. 64-75, 1989.

[11] J. Mauro, R. McDougall, “Solaris internals: core kernel components,” Sun Microsystems Press, 2001.

[12] R. Sutton, “Reinforcement Learning: An Introduction,” MIT Press, 2017.

[13] C. S. Wong, I. Tan, and R. D. Kumari, F. Wey, “Towards achieving fairness in the linux scheduler,” ACM SIGOPS Operating Systems Review, vol. 42, pp. 34–43, 2008.

[14] R. Sutton, “Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding,” MIT Press, 1996.