Reinforcement Learning for Scheduling Threads on a Multi-Core Processor

Saurabh Shrivastava

Introduction

With the advent of multi-core processors, the complexity of scheduling threads has gone up considerably. Most schedulers look only at the priority of threads that are ready to run to make a scheduling decision. Because multi-core processors have shared resources e.g. the L2 cache, the behavior of a thread running on one core can affect the performance of thread running on other cores e.g. two threads which hog the L2 cache if scheduled together on different cores can have worse performance than if they were co-scheduled with some other threads which did not hog the cache [8]. The situation is more complicated because the same thread can change its behavior over time e.g. it could be memory bound for some time and then become computationally bound later on. It usually also is the case that the threads communicate with each other to perform a task (this pattern also changes over time) and it makes sense to co-schedule these threads on different cores so that they can share data through the L2 cache.

There is a need for a scheduler which can learn about which threads can be co-scheduled, which should not be co-scheduled, which threads need to be grouped together to run on the same core, keeping in mind that the same thread can behave differently at different times. A scheduler with a fixed algorithm will not be able to achieve good performance in all scenarios.

Reinforcement Learning (RL) can be used here because we have a way of giving feedback to the algorithm to say if it is doing a good job or not by monitoring progress by looking at the core utilization for each thread e.g. % scheduled (more the better), % ready (the less the better), % blocked (the less the better) and/or processor performance counters e.g. # of instructions retired in an interval (the more the better), # of cache misses (the less the better) , # of synchronization instructions failed (the less the better). These goals cannot be achieved all at the same time e.g. # of synchronization failures can be brought down by running all threads on the same core but this would lead to decrease in the aggregate instructions execution rate. This problem calls for auto-tuning which can be achieved by RL.

Related work

In [4], performance counters and address tracing facility (not available on all processors) was used to find out cache sharing patterns, threads were clustered based on these patterns and then threads in the same cluster were run on the same core. In [5], an analytical model of cache, memory hierarchy and CPU was developed which was then used along with hardware performance counters to make determinations of when to move threads. In any of the above schemes RL was not used nor did they model inter-process communications. In [1], which comes closest to our approach, a benefit function, the parameters of which were the hardware performance counters and other ones learnt using RL, was computed. This function was run on each thread to find which thread move would bring the most benefit to the whole system.

The parameters of the RL scheduling algorithm

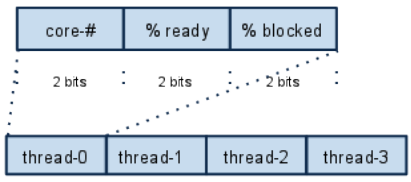
To make RL work, generally we need to define the state S of the system, the value function V for each state, the actions A which can be taken, the rules of transitions T between states, the model of the environment P_{sa} and the reward function R which indicates the reward received from the environment after each transition. Since it is difficult to model a processor and operating system dynamics and there is the need to learn online, we resort to Q learning where Q (state-action) values are learnt directly online without learning P_{sa} with the caveat that learning will take longer.

In our experiment, the state S of the system is determined by the core utilization, readiness, “blocking”ness and the core assignment of each thread all of which is available from the CPU profiler. For more precise modeling, we could also include the “behavior” of each thread i.e. the interaction pattern with other threads. The “behavior” could be inferred from the scheduler statistics by tracking the use of synchronization primitives but as this would require changing the OS kernel, we didn’t use the “behavior” feature.

The reward R is the fraction of time each thread was scheduled vs not-scheduled i.e. (scheduled - ready - blocked time) / (scheduled + ready + blocked time), indicating that we would like the scheduler to learn that we prefer running threads over blocked threads. We could also use the performance counters on the processor to generate the reward, but for this experiment we didn’t choose to.

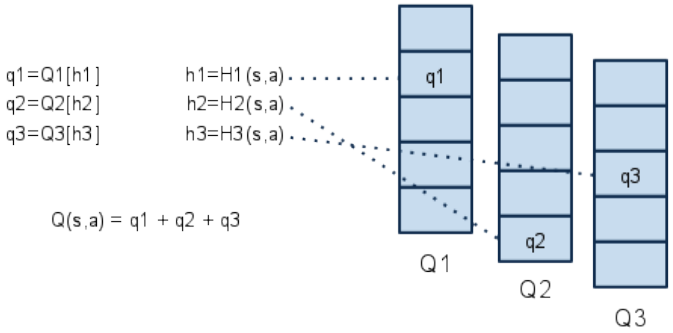
When the RL algorithm is delivered the reward for its last action, it is given an opportunity to perform further action. In our case the action is limited to moving a thread from one core to another or do nothing at all. The RL scheduling algorithm behaves like a meta-scheduler by pinning the threads to specific cores by using an OS API to set the “coremask” for each thread. The OS scheduler then makes scheduling decisions respecting this affinity. Ideally we would like to create “thread groups” and let the OS scheduler schedule groups on any core it wants, but in our OS we don’t have this flexibility.

Representing Q-Values, State-Action



s: state of the environment

a: action (move thread to a core)



CMAC representation of Q values

The state of each thread is represented as <core-#, %ready, %blocked> where core-# is the core to which the thread is assigned indicated by two bits, %ready and %blocked is the quartile of time the thread was ready to run or was blocked on other threads encoded by two bits each. The action is represented as

two bits of thread-# and two bits of the core-# to which this thread should be assigned. The Q state-action is thus 28 bits in size consisting of 24 bits of environment state and 4 bits of action.

Keeping a table of 2^{28} Q values will take a lot of space and will not generalize, so CMAC (Cerebellar Model Articulation Controller) [6] hashing is used to reduce space requirements and at the same time also enable generalization (unexplored values can be approximated) by hashing similar states to the same bucket. The CMAC scheme utilizes multiple hash tables with different hash functions for different table and allows Q values for a yet unknown state-action be approximated by the Q values of several nearby state-actions. Points which are nearby affect values of other nearby points by hashing into the same bucket whereas far away points which hash to different buckets have no effect.

In our implementation we use five hash tables of size 2^{14} where each hash functions extracts different set of 14 bits of the 28 bit state-action to come up with a bucket. In the figure, Q(s,a) is hashed to locations h1, h2, h3 in the CMAC tables Q1, Q2, Q3 respectively and so Q(s,a) = q1+q2+q3. If we have (s_1,a_1) which is close to (s,a), it could be that it hashes to locations h1, h2, h4 and so $Q(s_1,a_1)$ = q1+q2+q4, whereas if we had (s_2,a_2) which was far away from (s,a) then it would hash to entirely different locations and not share any values with (s,a).

Q-Learning

We use SARSA ($state_t$, $action_t$, $reward_{t+1}$, $state_{t+1}$, $action_{t+1}$) updates to the Q-Learning algorithm which is a “on-policy learning algorithm” where the Q value of the previous state is updated based on the reward and the action actually taken (see `RLQSarsa` below). We use the parameter alpha (= 0.10) which determines the learning rate and gamma (= 0.95) the discount factor. Actions are chosen using the e-greedy method with parameter epsilon (= 0.05) to exploit learning 95% of the time but also do exploration by randomly choosing actions 5% of the time (see `RLGetAction` below). The algorithm `RLQ` (see below) is invoked every one second to get the new state, reward, perform learning and take action.

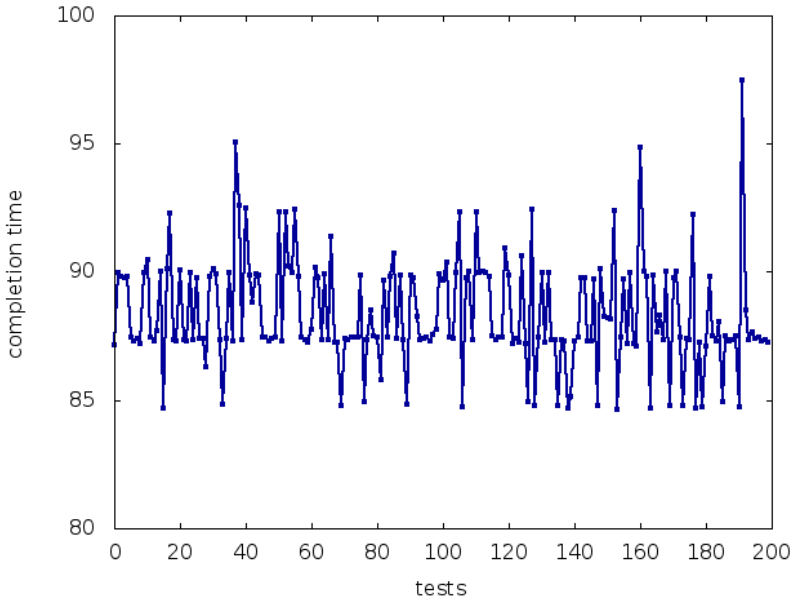
```
void RLGetStateAndReward (...)
{
   newState = // from CPU utilization for previous interval
   // reward scheduled time over un-scheduled time
   reward = (usSched-usReady-usBlocked)/
            (usSched+usReady+usBlocked)
}

void RLGetAction (newState) // get the next e-greedy action
{
   if (rand() < epsilon)
      // take a legal random action
   else
      // lookup Q tables for the newState
      // and choose action with max Q value
}

void RLQSarsa (...) // perform SARSA update
{
   h1 = H1(s,a); // (s,a) are the old state-action values
   h11 = H1(s1,a1); // (s1,a1) are the new state-action values
   Q1[h1] = (1-alpha)*Q1[h1] + alpha*(reward + gamma*Q1[h11]);
   Q2[h2] ...
   ...
}
```

```
void RLQ (tCpuUsage* pCpuUsage) // perform Q learning
{
   RLGetStateAndReward (pCpuUsage, &newState, &reward);
   newAction = RLGetAction (&newState);
   RLPerformAction (&newAction); // move a thread if needed
   RLQSarsa (&prevState, &prevAction, reward,
             &newState, &newAction);
   prevState = newState;
   prevAction = newAction;
}
```

Results



The RL scheduling algorithm was used on a repeatable test running IP routing protocols on a 8-core processor. Four busiest threads were identified and four cores were reserved for them, the rest of the threads were free to run on the remaining cores. We limited our experiment to only four threads to reduce the state space.

Even though the state space is huge, it usually is the case that the algorithm walks around in some small portion of the state space i.e. there are no totally random jumps from one portion of the state space to other (even when we do random exploration, it is to a nearby state). This can be seen by dumping the CMAC tables, most of the values are unchanged from initialization time and the values which are changed are usually seen in clusters. In the our experiment, the RL algorithm ran every one second and it could get only ~80 samples per test, so the Q tables were saved and restored across consecutive tests. 200 tests were performed over a period of 35 hours and ~16K learning opportunities presented themselves.

It can be seen from the above figure that the downward spikes increase over time indicating that the algorithm made better scheduling decisions as time progressed. It is known that SARSA takes a long time to converge because the environment is not modeled and so it can also be seen that the RL scheduler is still exploring because sometimes it takes actions which lead to worse completion times.

Conclusion

The main hurdle for using Q Learning is the large state space which needs to be learnt online: with just three features per thread the state-action space became 28 bits. It is known that Q learning is slow, but if the threads are long lived e.g. for several hundreds days which is the case with IP routers, then there is enough opportunity to learn and using Q learning makes sense.

Here we only looked at CPU utilization for generating rewards, in the future, we could also look at processor performance counters e.g. greater rewards for greater DRAM bandwidth used. If it is known that in some multi-core environment scalability is limited due to threads blocking on each other, we could also model degree of interactions between the threads e.g. for four threads, by six combinations of interactions with two bits per interaction.

The scheduling algorithm as presented includes the core-# as part of the state and so can handle heterogeneous cores for which regular OS schedulers face difficulty. This RL scheduler algorithm can also be used to place VMs (Virtual Machines) across servers in a data-center where we can treat VMs as threads and cores as servers.

References

[1] Operating System Scheduling On Heterogeneous Core Systems by Alexandra Fedorova, David Vengerov, Daniel Doucette
[2] Adaptive Utility-Based Scheduling in Resource-Constrained Systems by David Vengerov
[3] Workstation Capacity Tuning using Reinforcement Learning by Aharon Bar-Hillel, Amir Di-Nur, Liat Ein-Dor, Ran Gilad-Bachrach, Yossi Ittach
[4] Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors by David Tam, Reza Azimi, Michael Stumm
[5] PAM: A Novel Performance/Power Aware Meta-scheduler for Multi-core Systems by Mohammad Banikazemi, Dan Poff, Bulent Abali
[6] Self-Optimizing Memory Controllers: A Reinforcement Learning Approach by Engin Ipek, Onur Mutlu, Jose F. Martınez, Rich Caruana
[7] Reinforcement Learning: An Introduction by Richard S. Sutton, Andrew G. Barto
[8] Cache-fair thread scheduling for multicore processors by Ra Fedorova and Margo Seltzer and Michael D. Smith