740 - Assignment 2 Report (Akhil Polamarasetty)

Overview

In this assignment I have implemented a scaled down version of the PIFO queue datastructure for flow scheduling defined in “Programmable Packet Scheduling at Line Rate” by Sivaraman et.al. The key aspects of the proposal are as follows:

- Scheduling algorithms have two major decisions: what order packets should be scheduled and when they should be scheduled. And the authors note that these decisions can be made at the enqueuing stage of a scheduler making PIFO or Push-In-First-Out queues a natural choice.
- PIFOs enable network operators to express different types of scheduling algorithms that follow one principle: order of packets doesn’t change once they are in the buffer. The authors further show that hierarchical algorithms that do violate above principle can also be expressed through a tree of PIFOs.
- Implementing PIFO to support line-rate scheduling is also a challenge and building a PIFO to store all packets would be difficult in hardware as the number of packets is in the order of 100K making the enqueue operation expensive. The authors overcome this by assuming that packets within a flow always have descending priority so PIFO can just be used on the flows instead reducing the number of entries & thereby comparisons to 10k.

Implementation

In this assignment I have implemented a PIFO data structure that supports enqueuing into an arbitrary location and dequeues only from the head. Below I have briefly described some of the implementation details:

- I have used a linked-list based architecture where the enqueue operation uses a basic pair-wise comparison approach to insert a new element into the queue. To improve on the enqueue operation one can replace the linked-list based implementation to use priority_queue containers of C++. But I have chosen to implement the linked-list architecture as I think it’s more closer, in performance, to the hardware implementation (using flip-flops) of the paper.
- Furthermore, similar to the authors I have also implemented a map of FIFO queues called the ‘RankStore’ that is used to store the individual packets of each flow. Each queue is indexed by the ‘flowId’ which is just a value belonging to the range of (0, number of flows).
- Using the PIFO and RankStore datastructures I have implemented the LSTF or least-slack-time-first algorithm. To stay inline with the paper’s assumption on the order of packets within the flow, I am always assigning increasing slack time for the later packets of a given flow. But the flows themselves can have the base slacks i.e. slack of the first packet set to a value in the range of (base_slack_time, base_slack_time + slack_time_range).
- The rank for LSTF is computed using the slack and arrival times in the packet. Because we have a single switch in this experiment the arrival time is also filled in by the host. To compute rank we simply add slack and arrival time, this ensures that at any point the packets with lowest slack time will be dequeued first. I also compute the wait time in the slack which can be used to obtain remaining slack time or updating slack time of the packet. This is done by simply removing the current time from the priority/rank.
- So in this experimental implementation of LSTF we have 4 controllables: numFlows, minimum slack time, slack_time_range and number of packets within a flow.

Conclusion & Experiments

#Flows	Pkts per flow	Min slack time (millisecs)	Slack range (millisecs)	Deadlines met (%)	Avg wait time (millisecs)
32	100	1	10	29-33	6.016
32	100	10	20	89-93	9.575
64	100	10	20	48-55	9-10
64	100	50	100	100	80-100
128	100	10	20	28-35	15-20
128	100	50	100	98-100	65-70
256	100	10	20	12-14	29-33
256	100	50	100	89-95	50-55
512	100	10	20	3-6	50-55
512	100	50	100	65-75	50
1024	100	50	100	30-35	60

- In all the experiments the number of hosts is equal to the number of flows and each host is simulated using a thread. The hosts and switch share the packet queue or the rankStore and it is managed using a mutex.
- The above table shows the list of values for which the algorithm was tested. As we can see, unlike the paper, my implementation is not able to meet all the deadlines for 1000s of flows even with a slack time of milliseconds. And there are possibly two major reasons for these results:
    - I am using mutex to synchronize access to the rankStore and this adds a lot of latency due to the contention which increases with the number of hosts. This is reflected in the performance above, where the % of deadlines met for the same configuration of slack times reduces with an increase in number of flows i.e. hosts.
    - Another major reason for milliseconds delay or wait time could be that the program is running entirely in user space and in software stack. Optimizing the PIFO datastructure to use priority queues or more optimized search can save on the cost but I still may not achieve the line-rate performance of the paper.
- For the assignment I have only implemented the LSTF version but unlike the original proposal I am not just using a single PIFO queue. Instead as mentioned earlier the rankStore and PIFO implementation is essentially similar to the multilevel approach suggested in the paper for minimum rate guarantees.