Jack Farley

12/5/21

CS 5114

Mini-Project Report

Goal: In their paper, "Infinite CacheFlow in Software-Defined Networks," Katta et. al. take on the problem of limited rule capacity for software defined networking in modern hardware switches. To solve this problem, they design a system called CacheFlow, which acts as a single switch, caching popular rules in a hardware switch and storing the rest in software switches. For my project, I implemented a basic version of the CacheFlow system and assessed its effectiveness on a simple workload. The paper describes three possible algorithms for caching OpenFlow rules — the dependent set algorithm, the cover set algorithm, and the mixed set algorithm — each of which I implemented in this system. I also created a network framework for my CacheFlow switches, and ran a basic test to assess the correctness and effectiveness of each of these algorithms.

Approach: For this project, I decided to write my implementation of the CacheFlow switches in python. Each switch stores a list of all rules that have been installed and also a hardware and software switch where packets will eventually be routed once they enter the cache switch. The switch allows rules to be added and removed and packets to be sent in, and it updates the cache of rules on the hardware switch every time a rule is installed and every ten packets that are sent in. Depending on the algorithm specified when the cache switch is created, it uses one of three functions to update the rules installed on the hardware switch, each of which gets the weights assigned to each rule, constructs a dependency graph for all of the rules, and then runs the corresponding caching algorithm. The cache switch keeps track of how many packets have been sent through the switch and how many of those have found the necessary rule in the hardware switch (a cache hit).

Rather than incorporating these switches into another networking framework like mininet, I decided to build out a basic networking framework myself to run these switches in. The goal of this decision was to be able to focus more on the caching algorithms described in the paper and to avoid spending too much time integrating the switches into another framework. Since the focus of this project was on the caching algorithms, I was not too concerned about building switches that exactly resembled how a CacheFlow switch would look in production.

Overall, I think this decision worked out pretty well. The networking framework I designed allows one to create various switches and hosts, define links between those nodes to create a custom topology, and send custom packets into the network at any host. This system ended up being quite sufficient to test the effectiveness of the caching algorithms.

Evaluation: To evaluate my work in this project, I constructed a basic network using this networking framework, and sent a small workload of packets through this network to test the correctness and effectiveness of my cache switches. I ran this test for each of the three rule-caching algorithms. The networking framework stores the number of packets that have been sent into the network and the number of packets that have successfully reached their destination, so to test correctness, I knew that my switches were implemented correctly when no packets were dropped and each of them successfully reached their destination. To evaluate the effectiveness and correctness of my rule-caching algorithms, I checked the cache hit and miss statistics at the end of each of these tests to make sure that a significant amount of the packets moving through a switch resulted in cache hits.

Results: As for results, the main metric that I was interested in was the hit rate of the caching algorithms. I set up the test and workload so that each of the algorithms would at least partly face some version of the situation for which it is supposed to be ideal. And after running the tests, the results were pretty much as expected. The dependent set algorithm had a hit rate of 47.7%, which was pretty good for this test, while the cover set algorithm had a hit rate of 40%. The cover set algorithm's main advantage is the ability to cache high-weight rules without having to cache their entire dependency chains, and it did see a benefit from this, but that was not enormous since the dependency chains in the workload were not huge. The dependent set algorithm's main advantage, on the other hand, is being able to optimize the ratio of the weight to the number of rules cached at each step, but it is not able to cover large dependency chains. For this particular workload, it seems that optimizing that ratio ended up being more beneficial, but that would not always be the case. Finally, the mixed set algorithm saw a hit rate of 52%, which, since it combines that dependent set and cover set algorithms to achieve a strategy that acts as the best of both worlds, is also as expected.

Another interesting result is that all of the algorithms perform better on larger workloads since they are able to develop a better understanding of the relative popularity of each flow. When packets are initially being sent into a cache switch, it does not know which flows are going

to be more popular and thus which rules it should cache, but over time it is able to learn this information and improve its performance.

Attribution: For this project, I only used one outside source, which was the paper "Infinite CacheFlow in Software-Defined Networks" by Katta et. al. It was a super interesting read and ended up making for a very cool project.