Network Algorithmics: Machine Learning for Packet Classification HyperCuts algorithm with P4

Victor Rios
University of California, Los Angeles

Jing Xu
University of California, Los Angeles

Chuhua Sun
University of California, Los Angeles

Bowen Wang
University of California, Los Angeles

Abstract

Programming Protocol-independent Packet Processors (P4) is a programming language for controlling packet-forwarding planes in networking devices. HyperCuts [1] is a packet classification algorithm that is based on a decision tree structure. This project aims to implement HyperCuts in P4, which demonstrates the advantage of using reconfigurable switch chips.[1]

1 Introduction

Nowadays, there is an increasing demand to use reconfigurable switch chips instead of fixed-function switch chips. Fixed-function switch chips are limited in that they can't add new forwarding and monitoring functionality and can't move resources between functions. Reconfigurable switch chips are designed to solve these issues, and P4 is the programming language to program them. With P4 and the related compiler, it is possible to achieve many different designs, such as implementing machine learning algorithms. This project implements HyperCuts, a packet classification algorithm in P4, meaning that the switch can classify packets following the efficient HyperCuts algorithm. We will provide more detailed background about packet classification, HyperCuts and P4 in section 2, and introduce our implementation in section 3 with the experiment results followed in section 4. Future work will be described in section 5 and section 6 is our conclusion for the project.

2 Background

2.1 Packet Classification

Packet classification is the process of determining the rule with the highest priority for each arriving packet at routers. The rule consists of values based on one type of match from the exact match, the prefix match, and the range match. Among various algorithms for Packet Classification available, the most representative ones include linear searching, precomputed rules, RFC or Hierarchical Cuts, and HyperCuts.

2.2 HyperCuts

In this section, we briefly introduce a classification algorithm called HyperCuts. HyperCuts is based on a decision tree structure and could be considered a further advanced version of HiCuts, a famous classification algorithm. In HiCuts, each node of the decision tree represents a hyperplane. On the other hand, HyperCuts enhances it to be a K-dimensional hypercube. HyperCuts could provide an order of magnitude improvement over other classification algorithms by utilizing the extra degree of freedom given by the hypercube and original heuristics to acquire optimal hypercubes. With the aspect of space usage, HyperCuts is way more efficient than HiCuts since it uses two to ten times less memory. For speed, the worst-case search time of HyperCuts could achieve five times less than that of HiCuts. Not only for HiCuts, but HyperCuts also perform better than many other existing algorithms such as EGT-PC in consideration of both space and speed. HyperCuts has another outstanding advantage in that it is flexible as it can construct a pipeline and generate per packet-arrival-time classification result with the support of fast updates. Moreover, HyperCuts is based on range match, it is able to utilize more fields, and leaves contain all the matched rules, which share the same set of fields that represent intervals. While Hypercuts focuses on finding theoretical solutions, the importance of formatting the rules is significant.

2.3 P4

In this section, we will introduce P4, a high-level programming language for software-defined networks. P4 is the first representation of a programmable layer that allows developers to express the things they want to do with the network in a highly flexible way. It allows them to fully arbitrarily define how packets traversing programmable data plane blocks will

[1] Out project repo: https://github.com/aannmmoo/HyperCuts-P4

be processed. It drives a shift from hardware to software and from software to services regarding functionality. Developers could also run the P4 program along with the P4 runtime to partially define the interface for the control plane and the data plane to communicate based on the metadata converted by the P4 compiler from the forwarding behavior described by P4. With the help of a specially designed chip, P4 reversed the bottom-up paradigm to a top-down approach for network devices. The programmer defines the network feature set in the P4 program. Then the compiled code and the configuration are injected into the network device. The main purpose of P4 is to better describe the behavior of the data plane of any system or appliance that modifies, forwards, or inspects network traffic. It is an open-source community with a well-structured, high-quality, merit-based governance model.

3 Implementation

HyperCuts is comprised of two portions. The first generates the tree while the second search the tree during run-time. The first portion was done using an existing implementation that is publicly available. In general, this portion of HyperCuts can be done using an existing implementation due to the fact that the tree is generated outside of the data plane and done so before run-time. The second portion of HyperCuts, the search algorithm, was the only component that needed to be implemented in P4. The code responsible for packet classification is located within the ingress module of P4 for the v1Model architecture. This seemed appropriate given that packet classification tends to occur at the ingress, but it is also possible to run the HyperCuts or some variant of it at the egress. On another note, the navigation of non-leaf nodes could have been handled solely with standard coding methods within the control block, but we opted to use a combination of code and the tables offered by P4. However, one thing to consider as a consequence is that this uses up table resources for the non-leaf nodes, which may not leave enough for the leaf nodes. We are unsure if this is a legitimate issue in practice, but it is a potential issue to consider if the tree becomes large enough.

3.1 Tree structure in P4

The original tree structure generated by HyperCuts can be translated into a P4 program by creating a table for every node in the tree: both leaf and non-leaf. In our implementation, the search key used in every non-leaf node is comprised of the typical 5-tuple used in packet classification (IP src Addr, IP dst Addr, protocol, TCP src Port, TCP dst Port), and this key is searched using a range match. Note that this key contains redundant information as the only fields required are those where the cuts occur. The five fields were kept so that the template code could be easily understood, but the unnecessary fields should be removed in practical application in order to save resources. There are only four actions available to every internal node: 1) setting the index for the next internal node to be visited 2) identifying that a leaf has been reached and noting which leaf it was 3) identifying that a null leaf node has been encountered (this third action was provided since the tree generator we used generates null leaf nodes that contain no rules, and those do not require a table) 4) a default action such that if no ranges are matched than the packet is dropped (we assume that the packet classification acts as a white list). In order for the recursive search to work, every non-leaf node at the same level of the tree must have a unique identifier so that the search knows which table to apply next. The search key to every entry is a five-tuple with a range for every field. These ranges effectively create a 5-dimensional space that represents the reduced range that is covered by the child node.

The search key used in every leaf node is also comprised of the typical 5-tuple used in packet classification and also uses a range match, but unlike non-leaf nodes, every field is essential for the key as the search has come to an end and rules apply filters to every field. The actions available to a leaf node are the actions corresponding to every rule in the reduced rule set as well as a default rule that drops the packet if no rules were matched. The search key for every entry in a leaf table is precisely the ranges that correspond to each rule in the reduced rule set. Note that every leaf node must have a unique identifier so that the program knows which table to apply once the recursive search has ended.

In addition to these tables, metadata is used to help facilitate the search since common iterative functionality such as loops and recursion are not natively supported by P4. This metadata includes a continuing bit that states whether the search should continue or not, a variable that holds the index for the next non-leaf node to visit, and a variable that holds the index for the leaf that was chosen.

3.2 Search in P4

The search algorithm begins after the packet is initially parsed and packet headers are validated. The first step is to apply the root table, and because this table is considered a non-leaf node, it will either set the index for the next non-leaf node to visit or change the continue bit to 0, stopping the search, as well as potentially setting the index for the leaf that was chosen. A series of conditional switch statements follow the root table's application, and each switch statement represents a level on the tree. If the continue bit is still set, then the variable holding the non-leaf node index is used in conjunction with the next sequential switch statement to locate the appropriate table that should be applied. This process repeats until either a leaf is found or the packet falls out of any valid range, at which point any remaining non-leaf switch statements are skipped over. At the end of the search, there is a final switch statement that attempts to apply a table corresponding to the leaf index that was chosen, if no leaf was chosen, then the packet is

dropped. The application of a leaf table will produce the desired behavior for the rule that is matched, and if no rule is matched, then the packet will be dropped. At this point, packet classification has concluded, and the packet can continue to be forwarded.

4 Experiment

In this section, we will first introduce our experiment environment and then analyze the performance.

4.1 Setup

The experiment runs in a virtual machine followed the instruction in the P4 tutorial GitHub repo[2]. The host machine is a MacBook Pro of intel core i7 with a total of two cores and 16GB memory. The virtual machine consists of a P4 behavioral model, P4 compiler, and Mininet environment. The behavioral model is a simulator for the P4 software switch, and the compiler supports compiling a P4 program to a JSON file that the behavioral model can interpret. The Mininet is an emulator that can create a virtual network for evaluation.

In the virtual environment we set up in mininet, there are two hosts $h1$ and $h2$, and a switch $s1$. We will run a send python script on $h1$ that will send a bunch of packets with different source IP, destination IP, source port, destination port, and either TCP protocol or UDP protocol. The ip address will be randomized in the range of 0-15, and port will be generated in the range of 1-4. We will continually receive packets for the receiver side and verify the packet is tagged correctly through the router (we use the tos field in the IP packet to indicate the matched rule). The switch $s1$ will act as the control logic set in the P4 file. Right now, we use the same rule sets as in the HyperCuts paper, except if we receive a packet that matches rule 10 (default rule for UDP packet), we will drop the packet. This is for testing the drop packet functionality of our switch. When the switch finds the matching rule with the highest priority, it will change the tos field in the IP packet to be the rule number so that the receiver can know which rule the packet matches to and verify the implementation.

4.2 Performance

The HyperCuts implementation is compared with the linear scan method. The linear scan method means that in the P4 code, we only have a single root table, and it is expected to search linearly in the table to find the first rule that matches the income packet. However, due to the small rule set size, the difference between the two methods is subtle. They both need approximately 2 seconds to receive a packet. We think on the scale of 10 rules, sending the packets still dominates the whole process time. Theoretically, as a rule, set size scales up, the time to find the matching rule will take more time. In this case, the HyperCuts implementation is expected to have better performance than the linear scan version. However, currently we do not know whether P4 table resources will be exhausted when rules scale up, and this could be a potential issue in practice as mentioned in secion 3.

5 Future Work

Due to the time limit, we only provide an implementation for the example in the original Hypercuts paper in P4. We've found a full-featured Hypercuts implementation on Github[3], and it will output the tree structure given a set of rules. The rules are generated from the class bench project that is originally proposed by WashU team[4]. Each rule will specify the source IP range, destination IP range, source port range, destination port range, and the protocol. Then, the pre-compute algorithm of Hypercuts will generate a tree structure.

In the tree structure, a non-leaf node will contain information about how it will be cut, what range it will follow, and what its children node are. An example looks like this:

```
0; ([3, 1], [8, 4]); [1483495558, 2954081590, 883698240, 4285638383,
0, 65536, 20, 55445, 6, 18]; [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
39 40 41 42 43 44 45 46 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71 72 74 75 76 77 78 79 80 81 82 83 84 85 86
87 88 89 90 91 92 93 94 96 97 98 99 ]; [1 2 3 4 5 6 9 10 17 18 20 21
22 25 26 27 28 29 30 32 ]; NO
```

Figure 1: Example of non-leaf node 0

Where the first term is a node id, the second term ([3, 1], [8, 4],) means that the third dimension could be cut into eight times while the first dimension four times, the third one [1483495558, 2954081590, 883698240, 4285638383, 0, 65536, 20, 55445, 6, 18] represents the range of source IP, destination IP, source port, destination port and protocol, the fourth one illustrated rule id that is contained in the current node and the next one shows the id of all child nodes.

When the number of rules in a node is less than a certain threshold, the node will not further divide. That is, a leaf node will not have any children nodes.

```
2; None; [1483495558, 2954081590, 883698240, 1734183276,
0, 65536, 6949, 13878, 6, 18]; []; []; LEAF
```

Figure 2: Example of leaf node 2

Given such a generated tree structure, the future work would be to generalize the project automatically, that is, write a program that will transform any Hypercuts tree structure to corresponding P4 code and construct an automated workflow.

[2] https://github.com/p4lang/tutorials

[3] https://github.com/LystarmyNZT/neurocuts-master

[4] https://www.arl.wustl.edu/classbench/

6 Conclusion

In this project, we implement HyperCuts with sample rule sets in P4, and this demonstrates the flexibility of reconfigurable switch chips.

References

[1] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224, 2003.