The CLASHoFIRES: Who's Got Your Back?

Jamal Hadi Salim, Lucas Bates
Mojatatu Networks
Ottawa, Ont., Canada

hadi@mojatatu.com, lucasb@mojatatu.com

Abstract

This paper takes a performance perspective look at three classifiers that are part of the Linux Traffic Control (TC) Classifier-Action(CA) subsystem architecture. Two of the classifiers, namely *(e)bpf* and *flower* were recently integrated into the kernel. A blackbox performance comparison is made between the two new classifiers and an existing classifier known as *u32*.

Keywords

Linux, tc, classifiers, filters, actions, qdisc, packet processing, Software Defined Networking, iproute2, kernel, tc, u32, flower, ebpf, bpf, bpf-jit.

Introduction

The *tc* subsystem[1] provides powerful policy definable packet-processing capabilities in the Linux kernel.
The term *tc* will interchange-ably be used in the document to refer to both the kernel subsystem as well as the popular *tc* utility (used to configure the tc kernel subsystem).
A CA(Classifier-Action) subset of the *tc* subsystem is attached to a qdisc[1]. In this paper we focus only on a small subset of the Linux CA *tc* subsystem, the "C" part(packet classifiers). The qdiscs essentially are holders of classifiers which in turn hold filters at the two per-port hooks. It should be noted that while those are the only two anchors currently in use, it is feasible to attach the CA subsystem via a qdisc on many other hooks within the network stack; as an example of a recent addition refer to [2].

To provide context, we repeat some of the applicable content described in [1].
There are two guiding principles for the classifier architecture in *tc*:

1. *It is <u>not possible</u> to have a universal classifier* because underlying technologies are constantly changing.
2. *Sometimes we need more than one classifier type*, each with different capabilities, to properly match a given policy signature.

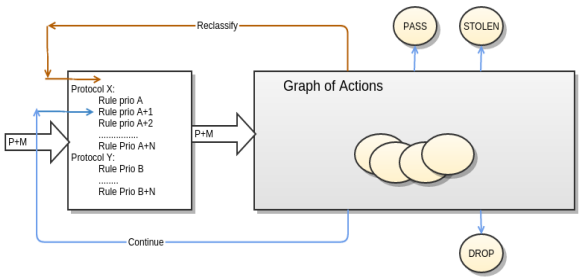

Figure 1: filter-action flow control

The CA design choice has fostered innovation[2] which has provided the opportunity to introduce two new classifiers, which we talk about in this paper, namely: (e)bpf[3] and flower[4].

Figure 1 illustrates the typical layout of how tc CA works. An incoming packet (alongside metadata) is examined using filter rules which are priority ordered. Policy control decides on the packet processing flow. Policy filter rules could be composed of the same type (classification algorithm) or they could be diverse and each filter rule could use a different classification algorithm. The choice of a classification algorithm could be a matter of taste or policy intent[3]. We are not going to go into the details of the packet pipeline control of a policy graph; for details, the reader is referred to [1].

In this paper we set out to do performance analysis of the two new classifiers in comparison to the u32 classifier.

What started as a simple trip to benchmark 3 different *tc* classifiers for a netdev paper became a journey documented in this paper. It was non-trivial to zone in on a few tests that would be considered fair. We spent 4-6 weeks analyzing different network subsystems where *tc* applies and in the process performed thousands of tests.
Investing all that effort led us to a path of defining more specific tests and refining them to meet our end goals. We

1 The *root qdisc* is anchored at the egress point of a port whereas the *ingress qdisc* is anchored on the ingress side. Note: on the egress side a CA graph can be attached at different qdisc hierarchies and not just the root qdisc.

2 by letting a thousand flowers bloom i.e. not allowing monopolies of classification algorithms

3 e.g. to first match using a classifier that looks at some header and then classify further using another rule that uses a classification algorithm specialized in string searches

justify our choices of tests in Section “Defining The Tests” and define the refinement of the tests and collection of results in Section “Preparing For The Clash Of The Classifiers”.

Meeting The Players

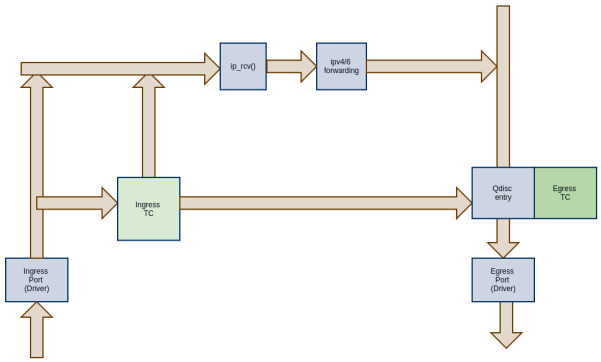

Illustration 2: Basic Landscape

In this section we provide context for the landscape where the classifiers run.

Illustration 2 provides a nutshell summary of an IP packet processing path in relation to the CA subsystem.
An incoming IP packet on a linux netdev/port that needs to be forwarded goes through the following paths:

1. Optionally subjected to the ingress qdisc and therefore optionally the CA subsystem within ingress qdisc.
2. Hit the input of IP processing entry point *ip_rcv()*.
3. Hit the IP forwarding code where selection of the next hop and netdev port is selected.
4. Optionally subjected to the egress qdisc of the selected egress netdev port and therefore optionally the egress CA subsystem.
5. Finally sent out on the egress port selected

An outgoing packet (from the host, top right hand corner arrow of illustration 2) goes through a similar process:

1. Selection of nexthop and egress netdev port (not shown in illustration 2)
2. Optionally subjected to the egress qdisc of the selected netdev port and therefore optionally the CA subsystem.

As observed we have a choice to use either the ingress CA or egress CA for the purpose of our testing.

The *bpf* Classifier

Linux extends the classic Berkeley Packet Filter(bpf)[4] in two ways: by using sockets as attachment hooks instead of ports/netdevs and by providing additional decision branching[4].

4 Original bpf definition had a binary choice; either the matched packet is to be allowed or dropped.

Illustration 3 shows a simple linux bpf program's internals.

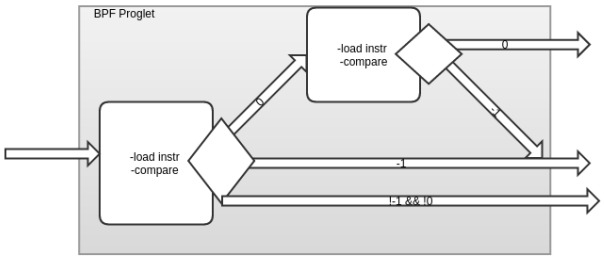

Illustration 3: Classic BPF

Within the kernel, bpf uses a register based VM (as opposed to a stack based one such as found in the Java interpreter) which makes it easy to map to local CPU instruction sets: therefore a just-in-time (jit) bpf variant exists in the kernel for many supported CPU architectures. A bpf program can fetch data from the packet(via load instructions), store data and constants in its registers, perform operations on packet data and compare(via compare instructions) the results against constants or other loaded packet data before issuing a verdict.

A user application needing to install a filter in the kernel typically would assemble instructions and compile them in user space before pushing the resulting bytecode into the kernel. Although it is trivial to write C code to generate bytecode, in our tests we used a utility called pcapc[8].

While standard Linux socket filters encapsulate a monolithic bpf program, the tc bpf classifier allows combining multiple bpf programs to achieve a policy as shown in illustration 4.

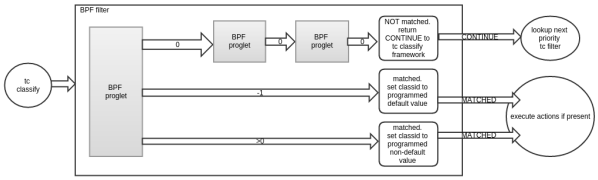

Illustration 4: Classic BPF use in tc

Illustration 4 also shows the tc bpf classifier with the three possible verdicts a bpf program emits. For the sake of brevity, we refer the reader to the netdev11 paper on the tc bpf classifier [9].

There are a few features that we did not include in our testing for lack of time: Starting kernel 4.1, the tc bpf classifier uses ebpf[10] which provides much more powerful packet processing capabilities. None of the tests documented exercised those features. We also did not use bpf to craft tc actions or use the DA(Direct Action) mode where the lookup and resulting action could be all crafted with bpf bytecode.

The *flower* Classifier

The flower classifier[3] started life trying to be a 14 tuple openflow classifier. During its patch submission phase, David Miller requested that it be reconsidered to instead utilize the kernel flow cache. It was then rewritten in its current format which classifies based on linux kernel flow cache fields.

The flow cache is built when a packet bubbles up or down the network stack. Each network sublayer updates the flow cache and may re-use the cached information from the previous sublayer.

Flower supports the following tuples which are collected in the flow cache:

source MAC, destination MAC, ipv4 or v6 source and destination IP addresses, and source and destination transport port number. Additionally, the netdev/port a forwarded packet arrived on is a valid classification tuple (eg is useful at the egress CA classification in case of forwarding paths). Essentially, these 8 tuples are used for lookups in flower.

There are other tuples collected in the flow cache such as GRE keys, MPLS, vlanids and TIPC which flower ignores at the time of publication but potentially will be available in the future.

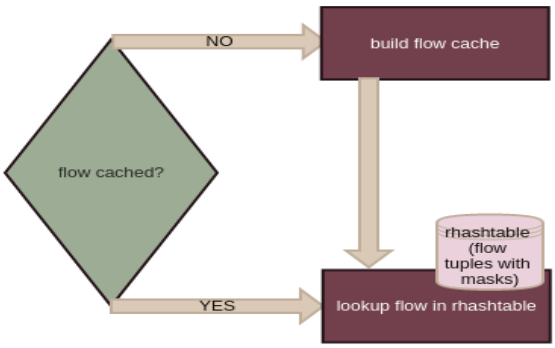

Illustration 5: The Flower Classifier

A user programs policy into the kernel using the tc utility by specifying the flow cache tuples of choice. The filter rules are stored in the kernel in a hash table and used in the packet path for lookups.

As illustration 5 demonstrates, when a packet arrives at the specific CA (in/egress) subsystem, flower checks if the packet already has the flow cache populated. If the cache does not exist yet, flower then creates it by invoking all the relevant subsystems to fill in their corresponding flow cache fields. If, however, the cache already exists then flower uses the packet's flow cache fields as keys to lookup the (policy populated) hash table.

Upon a match, the resulting bound action graph is then exercised.

The *u32* classifier

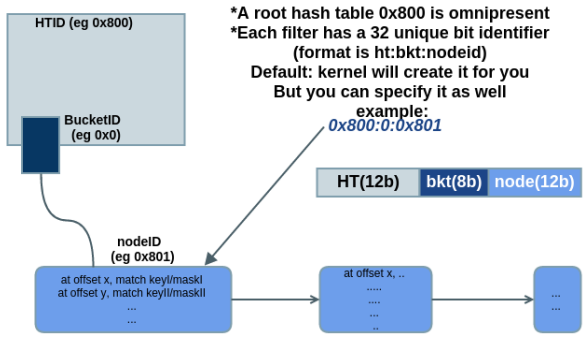

Illustration 6: Basic u32 classifier

The ugly (or Universal) 32bit key Packet Classifier has been around since the introduction of traffic control into Linux.

U32 uses 32-bit key/mask chunks on arbitrary packet offsets for filter matching. The filter nodes each constitute one or more of these 32bit key/mask/offset constructs which are used for matching. Nodes hang off hash table buckets.

Nodes have 32 bit handles that uniquely identify them as illustrated in figure 6 describing their location. The 32bit handles are split into 12bit hash table id, 8bit bucket id and 12bit node id. This means the system can have a maximum of 4096 hash tables, each with 256 buckets and with each bucket holding a maximum of 4096 nodes.

Nodes can link to next level hash tables, other nodes and buckets. A very efficient protocol parse tree can be crafted using these described semantics as we will demonstrate later.

The default u32 classifier setup is shown in Illustration 6. A default hash table (hash table id 0x800) with a single bucket (bucket id 0) is created; user entries are populated in order of priority. Essentially this becomes a priority ordered linked list of filters. An incoming packet will be parsed and matched in the filter priority list. The first match wins (meaning there could be other low priority filters which partially or fully match the packet) and the resulting bound action graph is then exercised.

In more complex setups (as we describe later) the packet headers can be incrementally parsed and the packet walked through the mesh of the constructed hash tables to eventually come to a leaf node which holds a bound action graph.

Defining The Tests

We define the classifiers and their associated algorithms as the System Under Test(SUT).

The starting assumption was that all classifiers should be able to handle:

- Per flow filter rules as opposed to a grouping of flows (e.g. via hashing algorithms). We define an ip flow as the classical 5 tuple specification (source ip address, destination ip address, ip protocol, source transport port and destination transport port)
- Account for every packet and byte at a per-flow level filter (using counters) as test validation.

It is also required that we be fair to all 3 classifier algorithms chosen and pick tests that did not favor one algorithm over another.

Picking The Metrics

We picked several metrics to compare the different classifiers. We list them here to illustrate our thought process.

- Datapath Throughput performance
- Datapath Latency
- Usability
  - operator friendliness
  - programmatic interfaces and/or scriptability
- Control path throughput and latency

We spent a lot of time on a discovery journey to define the constraints but we were only able to test the data path throughput performance in time for the conference.

We will no more than an educated opinion on the usability metric; and we hope in future work to cover the remaining outstanding metrics.

Reducing Test Variables

Given that the classifiers are surrounded by a lot of kernel code, the results could be influenced by a lot of other kernel and hardware variables; therefore, we needed as best as we could to isolate the SUT such that our results are not distorted by distracting overhead. To focus on the SUT, therefore, required reducing as many variables as possible.

Our initial instinct for throughput and latency tests was to connect two physical machines back to back: a sender machine which generates traffic to a receiver machine where the SUT resided. The sender machine would use pktgen[6] to send packets to the SUT machine. The configuration of the SUT machine would decide what packet path(per illustration 2) to take to get to the SUT. The packet processing would then complete when packets get forwarded back to the sender box where our results would be captured.

As it turned out, preliminary tests with this approach had so many variables that it affected the results and analysis; we spent time staring at profiles and decided against pursuing such a setup. The following were identified as possible hazards:

– System multi-processing contention and locks

– Driver code paths (both ingress + egress)

– Slow system code paths

– Intermediate handoff queues (backlog, egress qdisc etc)

Pktgen Ingress Mode

The first thing we elected to do was to run pktgen in ingress mode [7]. This meant we did not need an external sender machine and could therefore reduce variable cost of the ingress driver(driver interrupts, code path etc). This mode also allowed us to generate packets on a single stream on one cpu; which helped us achieve our goal to not have contention across multiple CPU threads being accounted for.

Using The Dummy Netdev

The second thing we chose to help in reducing variables was the use of the Linux dummy driver. The dummy driver acts as a black hole for any packets sent to it. It counts packets and their associated bytes then drops them on the floor. By using the dummy driver we do not have to worry about sending packets externally and the associated overhead of the driver (interrupts, locks, long code paths etc).

SUT Machine Parameters

As noted above, at this point in our progression, we had achieved the pleasure of running all our tests on a single machine. We chose to use the Intel NUC[11] for its size[5]. The NUC has the following parameters:

- Quad core i7-5557u @3.10Ghz
- 16G RAM (1600Mhz) Dual memory channels

Kernel choice:

- net-next 4.4.1-rc1 with two patches, namely:
  - bug fix for flower classifier[12]
  - pktgen egress enhancement to not bypass the egress qdisc[6]

Picking The Battle Scene

At this point in our journey we had identified the different scenarios for testing. The reincarnation of Illustration 2 is shown in Illustration 7. Given our interests are to test classifiers in a fair way for all three classification algorithms, we chose to experiment by dropping and accounting for packets at different code paths illustrated by the blackholes in Illustration 7.

5 So we could bring it to the conference and show live testing and results (which we did).

6 Jamal created this patch. John Fastabend independently came up with a different patch after a discussion on netdev. John has promised to merge the two and submit upstream.

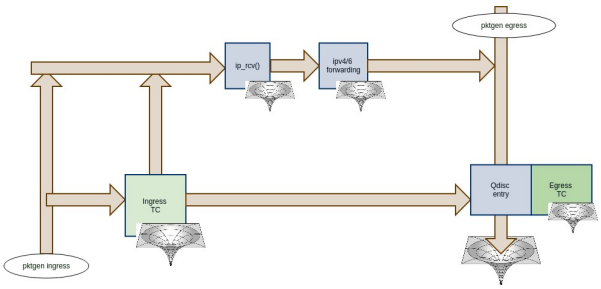

Illustration 7: Many Roads To Take

There are two possible sources of packets; one at the *pktgen ingress*(emulating packets coming from outside the machine) and another at the *pktgen egress* (emulating packets coming from host side withing the machine)

Dropping At *ip_rcv()*

On the ingress path we start with intentionally setting the wrong destination MAC address via pktgen; this means the network stack will not recognize that packet as belonging to the system. The result is that *ip_rcv()* will drop the packets it receives.

With that setup in place went through the following scenarios:

1. Optionally add the ingress qdisc in the packet test path. When the qdisc is skipped we refer to the test in the document as “no qdisc”.
2. When ingress qdisc is present, optionally add a CA subsystem with choice of classifier under test.
    - When the qdisc is used but the CA subsystem is skipped, we refer to the test as “qdisc only”.
    - When a CA subsystem is added, we install a single policy match with an action which then accepts and counts the packet. We repeat this exercise for all classifiers under test.

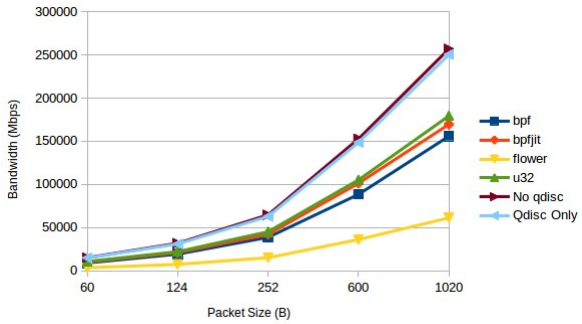

Illustration 8: Mbps Throughput Of Packets Dropped At ip_rcv()

Pktgen was made to send, using a single core, 30 seconds worth of bursts as fast as possible; all tests are repeated 4 times. Pktgen reports the averaged achieved throughput which we record and graph.

Illustration 8 shows the results. The difference in performance when an ingress qdisc (“qdisc only”) was installed vs when none existed (“no qdisc”) was less than 1%. As can be observed, at packet size of 1020 bytes both tests showed pktgen throughput of about 250Gbps. This result was impressive for a single processor performance. But more importantly it demonstrates that the presence of an ingress qdisc did not add overhead that would gravely affect our results collection.

For classifiers the results demonstrate that the differences between u32, bpfjit and bpf were also very small (although again consistently reproducible) at 173Gbps, 166Gbps, and 151Gbps. On the other hand flower did not fare as well capping at about 63Gbps. Should be noted the numbers for both u32 and bpfjit were north of 20Mpps.

These tests were unfair to flower. The flower classifier thrives on flow cache being populated. Such a case would happen on host-sourced packets. In our test, the flower classifier had to rebuild the cache for every single packet. Flower's usage of rhashtable was a clear bottleneck that was visible. However, that may be expected with the 64 bit compares used and better results may be achieved if we did 2x32-bit compares. So very likely there is room to improve the rhashtable comparator.

It was also pointed to us that the pcapc tool generated non-optimal bpf bytecode for both bpf and bpfjit. We were hoping to get results with more optimal bpf bytecode but could not get it done before paper submission deadline. We hope to publish our results and update the paper when we get the test done.

In all our tests in the rest of this paper, we verified that the bpfjit extension always outperformed bpf without jit. For this reason we stopped testing any further bpf without jit.

Dropping at Ingress CA

Our next sets of experiments involved adding a drop policy at the ingress. It was felt this would shorten the code path and make it easier to account for.

Illustration 9 shows the results. To our surprise, only flower showed consistent improvement over dropping at ip_rcv(). This maybe explained as due to the effect of memory pressure. We did not have time to investigate further.

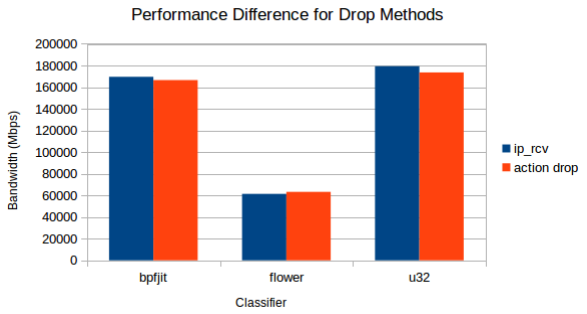

Illustration 9: Throughput comparison of dropping at ip_rcv vs tc action

The difference between dropping at the two locations was so small that given a choice between the two test setups, we choose to drop at tc ingress due to the convenience of the testing and the simplicity of result collection.

Ingress To Egress Path

At this point in our progression we set out to test sourcing packets at ingress but allowing them to proceed to the egress.

Our initial goal was to forward packets to an ipv4 route-selected nexthop via the dummy device and experiment with dropping packets at the different hooks on the egress. We were very surprised at the performance degradation of forwarding. We ended spending a lot of cycles chasing this ghost. We stared at a lot of profiles which seemed to indicate the fib_trie lookup was the bottleneck even with a single route.

To analyze further, we removed the ingress qdisc and filters from the test path and introduced a blackhole route. The blackhole route drops the packets right after ip lookup (reducing the overhead of the second leg which traverses code towards and including the dummy end drops). As the illustration 10 shows, we see a pktgen throughput of 25Gbps vs 21Gbps for blackhole drop vs routing to egress which points to the fact the processing leg after route lookup is not a large contributor to these results I.e the theory is that the bottleneck lies in IP forwarding.

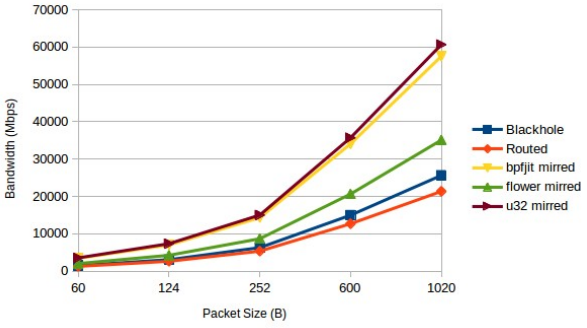

Illustration 10: Ingress To Egress Path

To get a different forwarding view, we experimented with the test classifiers using the mirred action to redirect to the dummy device(bypassing IP forwarding). Illustration 10 shows that this gave us in most cases 3 times the performance. While it is not totally fair to compare the two (forwarding does a few more things), we are still puzzled by these numbers. It is possible there were some icache effect due to the long code path. We did not have time to investigate further; chasing this ghost meant taking time away from our real goal of testing the classifiers. We report these results for anyone interested in pursuing them further. The experiments are very simple to reproduce.

So at this point in our investigation, it was clear to us we do not want to proceed with testing by sourcing at ingress and proceeding all the way to the egress due to the forwarding overhead polluting the results. And for that reason we stopped running that specific path's tests.

Pktgen Source At Egress And Dropping At CA

Our next step was to test sourcing on egress (emulating the host stack sourcing these packets).

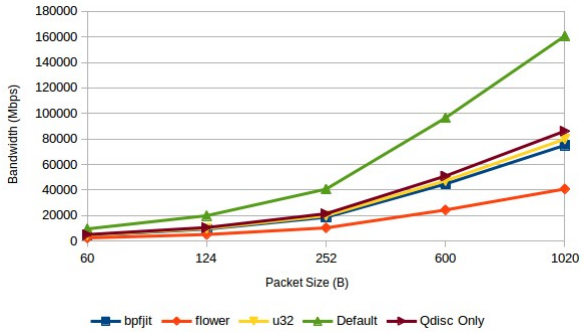

Illustration 11: Egress Transmit and TC Drop

Illustration 11 shows the results at 1020B. In the “Default” case we see the egress qdisc being bypassed and the dummy device dropping packets. The throughput performance was 160Gbps, a lot slower than the ingress side but nevertheless still formidable for a single CPU.

In order to add classifiers, we needed to add an egress qdisc. And things got interesting when we did that: as illustrated with graph “qdisc only” we see a throughput drop of almost 50%. Profiles show the egress qdisc lock being the culprit. The performance discrepancy surprised us given we are only running a traffic stream on a single CPU and no NIC hardware overhead existed. This dispells the popular myth that the egress cost is related to driver DMA overhead.

U32 and bpfjit showed a tiny drop on performance in the comparison against adding an egress qdisc (both in the 80Gbps range) indicating the main overhead comes from the qdisc. Flower showed degraded performance for the same reason explained for the ingress side (I.e related for need to rebuild flow cache).

At this point we decided that it was a bad idea to run our tests sourced at the egress.

We decided to stop experimenting with the different hooks and just settle on the ingress tc drop tests.

Jitter Effect On Collected Results

On all our test, as mentioned earlier, we always made 4 runs of each test lasting 30 seconds. We would then take the average of the 4 runs and compare it against the minimum and maximum results. Illustration 12 shows a summary across different packet sizes.

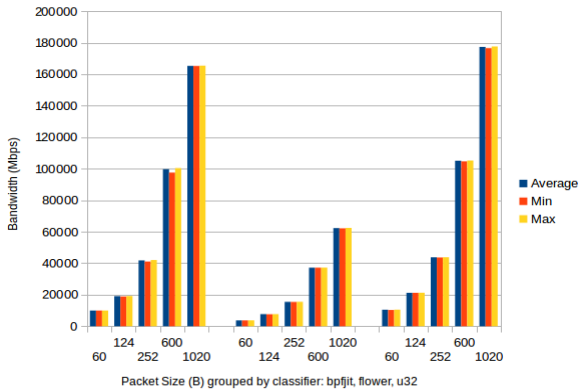

Illustration 12: Ingress Drop Action With Min/Avg/Max runs

We observe that the jitter was so small it felt inconsequential. So while we continued to collect all the results; going forward on this paper we are going only to show the averages.

Packet Size Effect On Collected Results

Another observation we made is that the effect of packet size was not very large. I.e our packets/second results were not very much affected by packet size.

Illustration 13 shows the results for earlier experiments.

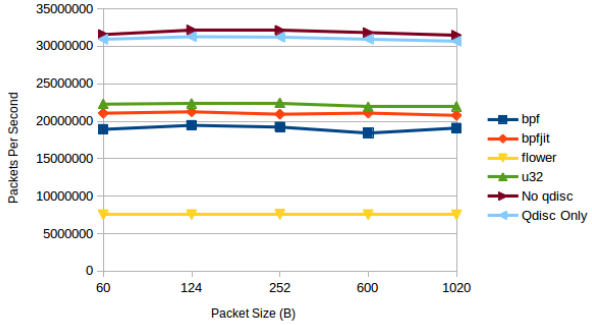

Illustration 13: PPS Drop At ip_rcv() with different packet sizes

How Many Classification Tuples?

So far all our reported tests have been running with a single flow and single tuple to match on. This is not very realistic real world scenario. We therefore tested where each of the classifiers looks up the five tuples described earlier. Illustration 14 shows the results.

The performance differences between the two scenarios were not huge.

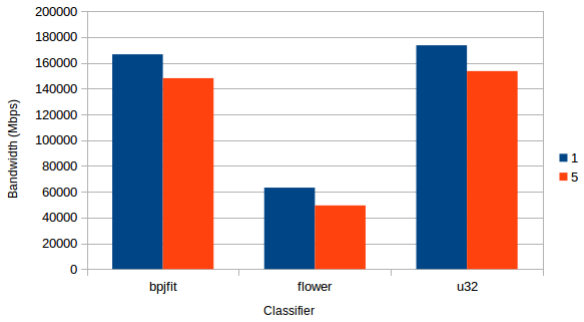

Illustration 14: varying number of tuples from 1 to 5

So from this point on, all our experiments will run with 5 tuples.

Preparing For The Clash Of The Classifiers

At this point in our investigation, we had decided on the our parametrization as constituting the following:

- All test running on a single core.
- Ingress tc qdisc, per-flow 5 tuple match classifier filter rule and drop.
- ignore bpf: BPFjit was always better
- Focus on average of 4 runs each 30 seconds ignoring max and min values in result illustration
- Use a single Packet size of 1020B

The Final Confrontation

Now that we had our SUT, tests and metrics well defined we set to run our core tests.

Given time constraints we settled on doing only throughput tests. We selected to vary the number of configured filters and looking for the best vs worst case scenarios.

For best case scenario tests, we arranged the rules such that the packet lookup finds a target matching filter on first lookup. By varying the number of filters we would expose any issues with possibly lookup dependencies in classification algorithm.

For worst case scenario, we arranged the rules such that a target filter match is found last. Essentially we forced the

lookup to be a linked list walk with the target match found last.

In both cases we repeated tests with incrementally adding filters ranging from 1 to 1000.

Best Case Scenario

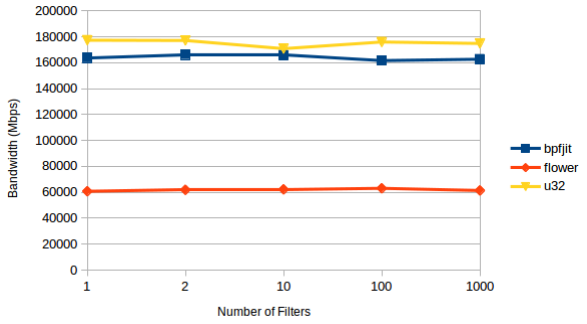

Illustration 15: Best Case Lookup vs number of rules

As can be observed for all the 3 classifier types varying the number of filters did not have any impact. U32 performance was around 175Gbps; bpfjit around 165Gbps and flower around 65Gbps.

Worst Case Scenario

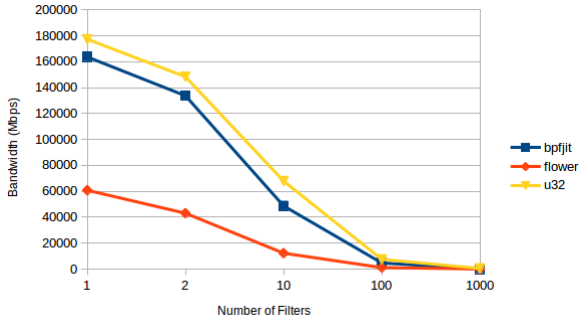

Illustration 16: Worst Case Lookup vs number of filters

As observed there is a very sharp drop in performance as the number of rules goes beyond 100.

To put it into perspective:

All the classifiers performances went down by a magnitude each time the number went up by a magnitude. In the worst case, at 1000 rules, u32 outperformed the other two being able to process about 463Mbps while bpfjit was able to do 73Mbps and flower did 88Mbps.

We believe that given in each case the lookups were linear with increasing number of filters these deteriorating results were to be expected. We did not have time to investigate why for example u32 fared so much better than the other two. We are going to take a second look at this in the future.

Scripting u32 to improve performance

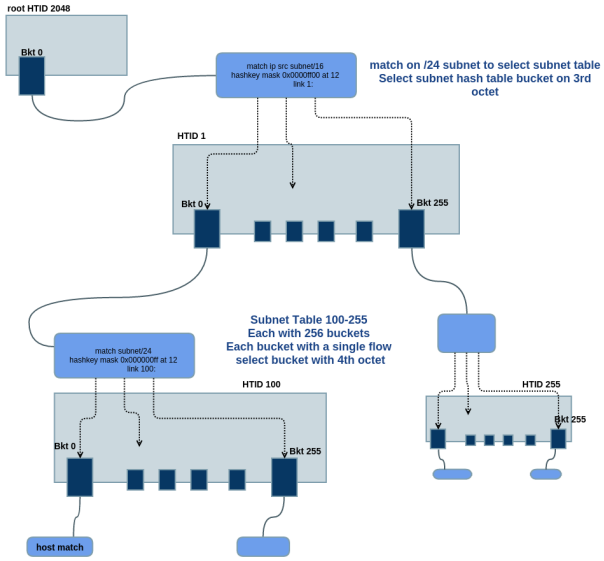

Illustration 17: Scripting u32 for multi-trie lookup

As mentioned earlier, the u32 classifier allows for scripting to dictate how lookups occur. In this case we arranged the classification based on our knowledge of the traffic patterns.

We made the first lookup hash on the expected subnet /24 and then based on the 3rd octet of the source IP address selected a bucket on hash table 1.

Each bucket on hash table 1 was linked to a secondary hash table (for a total of 256 buckets). For each secondary table bucket we would look at the 4th octet of the source IP address and select bucket.

On each secondary table bucket we had a single match inserted for a total of 64K matches.

Any of the 64K entries could be found in 3 lookups.

We then generated traffic that had 64K different flows.

Our results are shown in Illustration 18. The results were very consistent in the range of 115Gbps at 1020B packet size. It was clear from experimentation that we could have added over 128K flows and still achieved the same performance numbers - but we did not have time to pursue such an experiment further.

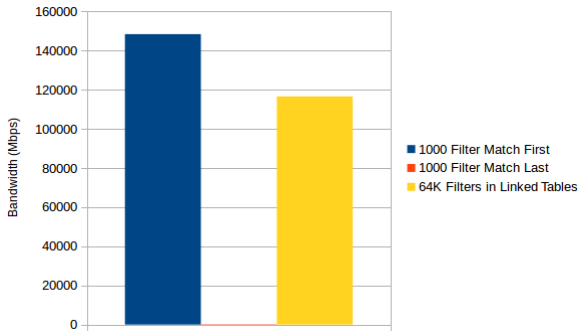

Illustration 18: Multi-trie results

If you take away the fact that we were aware of how the traffic patterns looked like and therefore optimized for the best case scenario, these are impressive numbers considering we run on a single core.

It should be noted with the ebpf extensions, the bpf classifier can be taught likewise to behave this way programmatically (not by scripting, rather coding and compiling).

Usability

We did not perform a formal analysis of the usability of the different classifiers, so what we are positing is merely an opinion.

U32 and bpf do not fare well from a human usability point of view; bpf is more human friendly[7] than u32 whereas flower was the best of all 3.

From a code programming flexibility and usability point of view, bpf is the winner.

U32 can be scripted, as we demonstrated, to provide powerful custom lookups. It is arguably the best operator friendly classifier.

Conclusion

We started with intent to work on performance analysis of 3 tc classifiers: bpf, flower and u32. Instead the majority of our time was spent on a journey of discovery on how best to perform the analysis in a fair and non-intrusive way.

We argue that our most important contribution is the documentation of the journey we took. We hope it inspires other netdevers, when doing performance testing, to watch closely on details such as metrics, assumptions made and isolation of the SUT from other subsystem noise contribution. The wise saying "numbers speak loud" applies[8] with the caveat lector that: Seville Oranges and Ottawa strawberries are fruits but different; a fair comparison requires understanding of taste-bud metrics as opposed to the falsehood of striving to claim the oranges as better strawberries[9]. We hope the reader is left with at least the view that we tried hard to achieve that goal when comparing the 3 classifiers.

Overall, given the constraints we faced we conclude u32 was the most performant classifier.

The bpf classifier was impressive – and as described could be tweaked to give better numbers. The flower classifier would perform much better with host-stack sourced packets. We did not have time to pursue either angle of validation and we leave this to future work.

Future Work

There are several opportune activities that the community could undertake to get us to the next level.

The bpf classifier performance testing with ebpf helpers is of definite interest to the netdev community. Of additional interest is to see if integration of actions in bpf classifiers provides even more improved performance.

Testing the Flower classifier on an egress path with many flows is something that we would like to pursue. We believe Flower will shine in such a setup.

It is our opinion that both u32 and flower will hands down beat bpf in the control to datapath update if the rules were to be generated and updated on the fly due to the fact that the bpf program will have to be generated before being installed. We hope to prove (or disprove) this point in the future.

Acknowledgements

The preparation of these instructions and the LaTeX and LibreOffice files was facilitated by borrowing from similar documents used for ISEA2015 proceedings.

7 If you ignored the fact that you need a bpf bytecode compilation (which maybe harder to debug).

8 Posting of netperf results and claiming victory

9 Yes, that is a pretty lame way of saying "orange-apple"; but coolest way of mentioning the last two cities where netdev took place in a relevant sentence on performance;->

References

1. Jamal Hadi Salim, “Linux Traffic Control-Action Subsystem Architecture”, Proceedings of Netdev 0.1, Feb 2015
2. Daniel Borkman, “classact qdisc patches”, netdev mailing List, Jan 2016. kernel commit 1f211a1b929c804100e138c5d3d656992cfd5622
3. Jiří Pírko, “Implementing Open vSwitch datapath using TC”, Proceedings of Netdev 0.1, Feb 2015
4. Steven McCanne, Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", Dec 1992
5. Daniel Borkman, “BPF classifier”, net/sched/cls_bpf.c
6. netc/core/pktgen.c
7. Alexei Starovoitov, kernel commit: commit 62f64aed622b6055b5fc447e3e421c9351563fc8
8. pcapc https://github.com/pfactum/pcapc.git
9. "On getting tc classifier fully programmable with cls_bpf" (Daniel Borkmann) @netdev11
10. “BPF In-kernel Virtual Machine” Alexei Starovoitov @netdev01
11. Intel NUC http://www.intel.com/content/www/us/en/nuc/products-overview.html
12. J Hadi Salim, kernel commit: 66530bdf85eb1d72a0c399665e09a2c2298501c6

Author Biography

Jamal Hadi Salim has been dabbling on Linux and open source since the early 90s. He has contributed many things both in the Linux kernel and user-space with a focus in the networking subsystem. Occasionally he has been known to stray and write non-networking related code and on rare occasions, such as this, documentation.

Lucas Bates discovered Linux through a friend in the late 90s. He once paid for a Linux distro CD – money he considers well spent to this day. Lately he's been immersing himself deeper into tc and iproute2.