Picking Low Hanging Fruit from the FIB Tree

Alexander Duyck

Red Hat, Inc
alexander.h.duyck@redhat.com

Abstract

As network rates continue to increase, the amount of time to process packets decreases. This puts pressure on a number of areas in the kernel, but one area where this particularly stands out is the IPv4 forwarding information base look-up. This paper will go over a number of changes recently made to the FIB trie processing code and data structures to improve both the performance and reliability of processing the IPv4 addresses. In addition, it will propose some possible approaches under consideration in order to improve performance. This will further enable processing at higher packet rates.

Keywords

IP address look-up, performance, trie, longest prefix match

Introduction

The IPv4 forwarding information base (FIB) as implemented in the Linux kernel is based on a level and path compress trie or LPC-trie. The structure is a dynamic variant of a static level compressed trie, or LC-trie and is based off of work by Nilsson and Tikkanen [1]. The advantage to this structure is that the performance approaches that of a static trie. At the same time, it reduces the waste by compressing the trie to reduce the number of empty nodes.

Traditionally, tries are described as having an O(N) complexity, where N is defined as the length of the key [2]. However when used for IPv4 address look-up within the Linux kernel this can approach $O(N^2)$ as prefix matching requires back-tracing through the trie searching for the longest prefix match. The LPC-trie helps to reduce this by aggregating multiple bits of the key into each node, but this reduction is only linear and does not resolve the underlying issue with the prefix match algorithm.

In this work, the look-up algorithm was modified to improve the performance. The idea being to reduce the complexity as the original algorithm took a number of unnecessary steps. This resulted in time wasted checking keys that could not be the prefix of the look-up value.

Making tnode and leaf Function Similarly

The first step in improving the performance of the look-up algorithm was to make use of unused space within the leaf structure to bring the leaf and tnode members closer to each other in form and function. Specifically, by adding variables representing the position in the key and the number of bits represented by a given node, it becomes possible to incorporate both the validation of the prefix and determining the location of the next child into a single set of instructions.

```
struct tnode {
        t_key key;
        unsigned char bits;
        unsigned char pos;
        struct tnode __rcu *parent;
        struct rcu_head rcu;
        union {
                struct {
                        unsigned int full_children;
                        unsigned int empty_children;
                        struct tnode __rcu *child[0];
                };
                struct hlist_head list;
        };
};
```

Figure 1. Structure of tnode after modification

In the 3.19 and earlier kernels, the tnode and leaf only shared a parent and key value. The least significant bit of the parent value was used to identify if the object was a leaf or a tnode. This resulted in several issues. It becomes difficult to make assumptions about the layout of the next object when the layout is defined based on the contents of the object itself. The code experienced significant delays due to these dependencies. To resolve this, both the leaf and the tnode structures were merged into one shared structure with the following layout as shown in Figure 1. After the modification, the two structures were merged. A tnode with a bits value of 0 is a leaf containing a hash list of leaf info data, else the object represents a tnode containing up to 2 ^ bits children.

An advantage to this shared structure is that as a leaf is limited to a single child only index 0 will be a valid index for a leaf. This makes it possible to use the index value generated by comparing the look-up value to the key contained within the node to determine if a prefix mismatch has occurred, and this is applicable to both a tnode or a leaf.

Processing the Prefix Mismatch

The next step in improving the performance was to address the cases in which a prefix mismatch had occurred. A prefix mismatch has occurred any time the bits in the key that are a part of the prefix do not match the given search value. So for example the prefix 192.168.1.0/24 would match 192.168.1.1, but it will not match 192.168.2.1 as the first 24 bits of the two values do not match.

In the 3.19 kernel, this was accomplished by performing an xor of the search value and the current node's key. Then the most significant bit of the difference between the two was found using the __fls function. Finally, the resultant value was then used to generate a mask so that the mask could be used to check for any bits less significant or equal to the mismatch in the node key.

```
pref_mismatch = mask_pfx(cn->key ^ key, cn->pos);
if (pref_mismatch) {
    /* fls(x) = __fls(x) + 1 */
    int mp = KEYLENGTH - __fls(pref_mismatch) - 1;

    if (tkey_extract_bits(cn->key, mp, cn->pos - mp) != 0)
            goto backtrace;

    if (current_prefix_length >= cn->pos)
            current_prefix_length = mp;
}
```

Figure 2. Previous logic for determining prefix mismatch

To reduce the overhead of this call it is first necessary to address the fact that the prefix mismatch depends on finding set bits. To avoid false mismatches we need to clear the unused bits of the tnode key. This is easily done when a tnode is allocated by performing two shifts on the key based on the pos and bits values. This step is not necessary for a leaf as such an action was already being performed as a part of the leaf allocation process. Once the key has been prepared the process of determining if a prefix mismatch has occurred is as simple as the one line of code as seen in figure 3. This code works by generating a mask based on the least significant bit in the node's key. The result of a value OR'ed with the negative version of itself will always provide a mask that starts at the least significant set bit of the value. After that, it is a matter of performing an XOR between the node key and the look-up value to generate a delta and then preforming an AND of the delta and mask to test for any differences that would be in the prefix of the node key.

```
if ((key ^ n->key) & (n->key | -n->key))
    goto backtrace;
```

Figure 3. New logic for determining prefix mismatch

Strip Bits from the Index Instead of the Key

In previous kernels, prefix matching was performed by stripping bits from the child index via a “chopped_off” count. This count tracked the number of bits that had been stripped from the index value. This value was then fed into a “current_prefix_length” value which was used to mask the key during a search into nodes that could be a longest prefix match. This code was simplified to consist of “cindex &= cindex – 1” as this will always strip the least significant bit from any value. The chopped_off and current_prefix_length variables were dropped from the longest prefix matching loop as they provided no actual value once the update of cindex had been simplified.

Avoid Diving into Shallow Suffixes

The look-up time for a longest prefix match can approach $O(N^2)$ due to the search for the variations on a given look-up value with the least significant bits stripped. However, in the case of many IPv4, routes they will share a common prefix length of 24 or 16 [3]. This means that any search after stripping any bit beyond the first 8 or 16 will simply result in a failed look-up, or a look-up that will need to back-trace to the default route.

A tracking value named slen has been added to the tnode structure to avoid unnecessary look-ups. This value tracks the length of the longest suffix or host identifier contained within the given tnode. By tracking this value, it is then possible to avoid back-tracing into nodes which do not contain a longest prefix match. As a result, look-up times are reduced from $O(N^2)$ to something approaching O(N).

Performance Evaluation

In order to evaluate the performance gains from these changes a simple network was setup in which a VM running on an Intel Core i7-4930K CPU running at 3.4Ghz. The VM was assigned an ixgbe network interface and allocated a set of dummy interfaces. The IP addresses of the ixgbe and dummy interfaces that received traffic were configured so that they would exist in the deepest part of the trie and have a significant number of bits set. This created an environment that would allow for the maximum amount of time spent in the prefix matching state. The resulting trie had a maximum depth of seven for the local trie and six for the main trie. The source and destination addresses at the maximum depth within both tries.

Three different tests were conducted. The first was a simple routing test which received packets from the ixgbe port and routed them to one of the dummy ports. This allowed for testing of two look-ups both of which failed on the local trie, and then succeeded on the main trie. The second test was a receive test in which the traffic from the ixgbe port was received and then dropped locally. This tested two look-ups on the local trie with one success, and one look-up in the main trie with one success. The final test was a transmit test in which packets were transmitted to the dummy port from the stack and dropped there. This consisted of one look-up in the local trie which failed, and one look-up in the main trie with one success.

From the results in figure 4, we can see that the changes have significantly reduced the look-up time as measured in nanoseconds. Based on the data, we can derive the look-up and back-trace time for both the 3.19-rc kernel and 3.20-rc kernel. The times would appear to match the expected models in which the time for look-up is in the 10s of nanoseconds, while back-trace is in the 100s of nanoseconds for the 3.19-rc kernel. For the 3.20-rc kernel back-trace is only in the 10s of nanoseconds. This is consistent with the transition from $O(N^2)$ to O(N) as predicted.

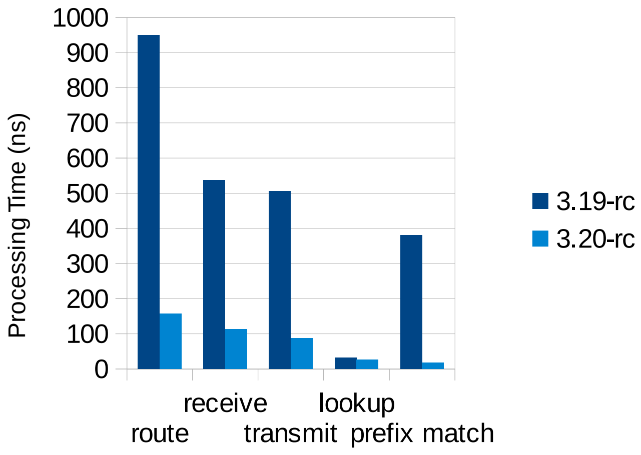

Figure 4: Results of performance testing for initial changes

With the changes made so far, we have been able to reduce the complexity for prefix matching from $O(N^2)$ to O(N). While this helps to improve the performance, there is still a significant amount of time spent on IPv4 look-up. In the case of the test configuration, the look-up time is still over 100ns per frame for routing and receive workloads, and more than 70ns for transmit workloads.

Removing Leaf Info

One of the biggest reasons for fib_table_looup consuming so much time is the amount of memory that must be accessed. In the main look-up loop, as many as two cache-lines are accessed per tnode, one per leaf, one per leaf info, and one per FIB alias. One approach that can help to reduce total overhead is to remove unnecessary objects from the look-up path.

The leaf info object provides a mechanism for grouping multiple FIB aliases that share a prefix length into a single hash list. However, such a grouping is not common as often there is only one alias per prefix length. There are no functions that truly consider the leaf info when consuming FIB aliases. It happens that the FIB alias structure has a byte of free space that could be used to store a prefix length value. In my test environment, it was found that removing these objects and directly linking leaves to the fib_alias hash resulted in a savings of up to 5ns per look-up.

Wrap Pointers in Key Vector

For every pointer in the trie, with the exception of FIB aliases, there is a key value pair that is associated with it. This key is normally used to determine which of the pointers to use, or if the next pointer should be used at all. By wrapping all of the pointers in a key vector, it becomes possible to treat the entire trie as one uniform structure, including the root.

One change that was tested was to introduce the key vector as seen in figure 5, and then encapsulate the trie root, all tnodes, and the leaves so that all elements had a key vector. An advantage to this is that it then becomes possible to simplify insertion, deletion, and search as we start with a key node instead of starting with NULL. As a result, all tnodes will have a parent that is ultimately represented by the trie root node. This allows for an improvement of as much as 5ns per longest prefix match. This is because it is possible to start prefix matching at the root node, in which case we know there is no further work to do without having to deference any additional pointers.

```
struct key_vector {
        union {
                struct hlist_head leaf;
                struct key_vector __rcu *tnode;
        };
        t_key key;
        unsigned char pos;
        unsigned char bits;
        unsigned char slen;
};
```

Figure 5: Key Vector Structure

The root node, tnodes, and leaves all must follow a specific pattern so that they are recognizable by the algorithm when being searched. In order to facilitate this, there are several specific values that apply to each of the node types:

- If pos + bits <= 32 and bits > 0, then the node is a tnode
- If pos == 0 and bits ==0, then the node is a leaf
- If pos == 32 and bits == 0, then the node is a root node

These properties lead to several interesting behaviors. Since the root node has a position of 32, any key that is compared against it will always be a prefix match. This result in moving onto the child contained within node 0. Any leaf will always perform a full prefix match during look-up, and could be considered as a tnode with only one child. The leaf will point to the FIB alias hash list as its only child.

Up-level the Key Vector

The final opportunity for improvement is to up-level the key vector from the individual nodes, into the level above it. Specifically, each node in the trie already contained a key vector, and a key vector is 16B on 64b or 12B on 32b systems, while a pointer is only 8B or 4B respectively. If we were to modify the key vectors so that instead of containing an array of pointers, they instead contained a pointer to an array of key vectors, the result would be that the key and key data for a given trie node or leaf would actually be contained in the trie node one level above it. This should result in as much as a 50% reduction in the number of cache-lines that need to be accessed in a look-up. It will also reduce the overall memory used, as a leaf would require 32B on a 64b system, and a key vector in the array would only require 16B due to coalescing the RCU structures.

In implementing this, there were several issues discovered. The key, pos, and bits field must be RCU protected values to avoid possibly accessing an out-of-bounds element within a child of the key vector array. This results in the current implementation experiencing an O(N^2) insertion and deletion time. This is because the parent of any new leaf or tnode must be replaced if one of the protected values is changing. We believe this can be reduced back to O(N) complexity and may be resolved in the near future by simply replacing any resized region as a single unit instead of as individual parts.

In testing, several issues were encountered. Several performance issues were found that needed to be addressed in the original changes submitted to the kernel. It was necessary to replace a shift of the index with a comparison of a shifted mask instead. This allowed for a reduction in the length of dependency chains within the main look-up code. Once this was done, a reduction of up to 10ns per look-up could be observed for the routing and receive scenarios.

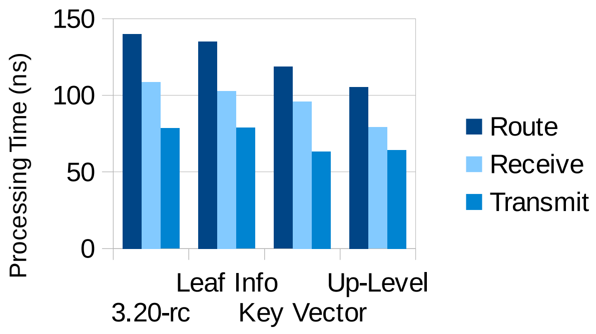

Figure 6: Results of incremental changes

Conclusions

As can be seen in figure 6, the initial round of changes accepted into the 3.20-rc kernel still have a considerable amount of room left for improvement. With the three changes described earlier, it is possible to reduce worst-case look-up times by as much as an additional 25%.

Further work is still needed as fib_table_lookup still consumes a considerable amount of time compared to other parts of the IPv4 network stack within the Linux kernel. To improve beyond the current limits, a complete redesign of the forwarding information base may be necessary as the lower limits for the FIB table look-up are approaching the L2 cache latency of the system. As a result further code optimization may yield little to no gain.

Acknowledgments

We would like to thank Jamal Hadi Salim and the selection committee for Netdev0.1 for inviting us to present and submit this paper.

References

1. Stefan Nilsson, Matti Tikkanen, “Implementing a dynamic compressed trie”, *Proceedings WAE'98*, http://www.nada.kth.se/~snilsson/publications/Dynamic-trie-compression-implementation/text.pdf
2. Jun-Ichi Aoe, Katsushi Morimoto, Takashi Sato, “An Efficient Implementation of Trie Structures”, *Software-Practice and Experience, Vol 22* (1992): 695-721
3. Jing Fu, OlofHagsand, Gunnar Karlsson “Improving and Analyzing LC-Trie Performance for IP-Address Lookup”, *Journal of Networks, Vol 2, No 3* (2007): 18-27

Author's Biography

Alexander Duyck has worked as a Senior Software Engineer at Red Hat, Inc since October of 2014 . There, he works on improving the performance of the networking subsystem within the Linux kernel. Prior to joining Red Hat he had worked for seven years at Intel as a Senior Software Engineer. There he helped to maintain and create the Linux wired Ethernet drivers igb, ixgbe, and fm10k.