CS 216 Project

IP Lookups with Hints

Trevor Hackett
Department of Computer Science, UCLA
trhackett@ucla.edu

Siva Keseva Reddy K
Department of Computer Science, UCLA
sivakesava@cs.ucla.edu

Yuren Shen
Department of Computer Science, UCLA
yurens@cs.ucla.edu

Vishrant Vasavada
Department of Computer Science, UCLA
vasavada@cs.ucla.edu

ABSTRACT

Internet address lookup used to be a challenging problem ten to fifteen years back. However, wire speed solutions for lookup are now readily available. Even then, we believe that these solutions don't address redundancy involved in lookups when an IP packet is passed from one router to the next. Even if the previous router has already matched prefix up to length L, the next router will still start its search for the matching prefix from scratch. We imagine that routers can be smarter and achieve faster forwarding times by ignoring those first L bits. As a packet moves from source to destination, that L, matched prefix length increases, (after a certain point). Our idea is thus to pass a length hint to next router indicating that the previous router has already found matching prefix of length L and so it should start matching bits from (L+1)th position.

We evaluate our hypothesis by simulating and computing IP lookup using a 1-bit trie with and without length hints. The results are beginning to show dilvulge ways that routers can be smarter about prefix matching using hash tables. Initially, it seemed that adding another structure to the lookup process was making lookup take longer, but a more thorough experimentation process shows that it is possible to improve lookup time with a hash table on the prefix length (L as described above). Using this process, routers can improve forwarding time even more than they currently are, keeping up with ever-increasing wire-speeds.

1 INTRODUCTION AND MOTIVATION

When a packet arrives at a router, it matches its Internet Protocol (IP) address with entries of prefixes in its forwarding table to decide where the packet should go to next. There are many solutions to this problem, and many of them operate at extremely high speeds. But, none of them utilize the opportunity to pass information about the lookup done at one router to improve the speed at the next hop. We find that in many cases, we can utilize prefix matching information from previous routers and pass it as a hint to the next hop to reduce redundancy.

For example, if we want to send a packet from the network of cs.ucla.edu to cs.ucsb.edu, we would first match the IP prefix of ucsb.edu for say 16 bits somewhere in the middle (probably somewhere near ucsb), we call it router A. The currently adopted algorithm would require the packet to match with the forwarding table from scratch in later hops than router A. But we would assume that in later hops of router A, they almost definitely have the prefix entry of ucsb.edu. Thus, in our approach, we would like to pass the information that we have previously matched with ucsb.edu in router A, and start from the 17th bit to try to match with cs.ucsb.edu which is a more specific address included in the address of ucsb.edu.

We have applied several principles of network algorithmics from [2] in our approach. Firstly, we have practiced principle 12, adding state and utilizing it to improve speed in our approach. We added information about history of prefixes matched for that packet when passing it around. The goal is to eliminate the cost of starting over the prefix matching from the first bit, instead searching based on previous matches. We also have applied principle 9, passing hints in module interfaces to our algorithm. Our approach records the previous longest prefix match, and we pass that hint to the next hop. In this way, we also achieved principle 1, avoiding obvious waste in common situations, because it removes redundancy in the process.

2 IMPLEMENTATION

We implement [1] our idea in C++.

2.1 Data structures

Each router keeps a unibit trie to store and organize IP prefixes. Each node in unibit trie consists of a left pointer pointing to the left child location with an address bit 0, a right pointer pointing to the right child location with an address bit 1, and a prefix from prefix database if its corresponds to the concatenated bit entries of all the nodes in a trie from the root node to this node.

For example, consider a sample database as shown in Table 1:

Label	Prefix
P1	10*
P2	111*
P3	11001*
P4	1*
P5	0*
P6	1000*
P7	100000*

Table 1: Prefix Database

The unibit trie constructed from this sample database is as shown in figure 1. P1, P2, ..., Pn are the prefixes in the database above and A1, A2, ..., Ak are the pointer labels. Note that this trie supports two basic operations - *insert* and *search*.

We maintain another data structure, *length table*, which is a size 32 static array of hashmaps. Each hashmap stores a mapping of prefix bits to the node in the trie corresponding to that prefix. Because

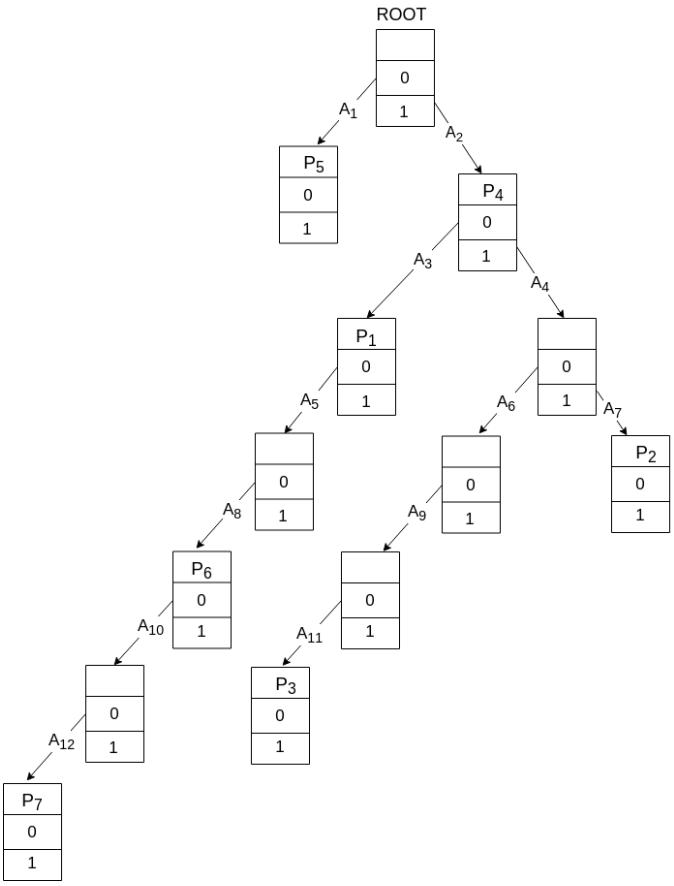

Figure 1: Unibit Trie

all prefixes of length I are found at level I of the trie, each hashmap corresponds to a level in the trie. For index I, if there's no node at the level I, we simply store nothing. This hashmap is implemented using *unordered_map* of C++ STL library instead of using *map*. This is because the average lookup time for unordered_map is **O(1)** as they are implemented using hash tables while for map it will be **O(log n)** as they are implemented using a binary search tree. Order is not a concern of ours.

The length table for our example above looks as in Table 2:

Index	Hashmap
0	empty
1	0: A1, 1: A2
2	10: A3, 11: A4
3	100: A5, 110: A6, 111:A7
4	1000: A8, 1100: A9
5	10000: A10, 11000: A11
6	100000: A12
7...31	empty

Table 2: Length Table

2.2 Algorithm

2.2.1 Insertion. Insertion into the trie requires three pieces of information: *prefix*, *mask* and *next hop*. The algorithm first converts the prefix passed as a string (e.g. "150.160.170.180") to an integer. It

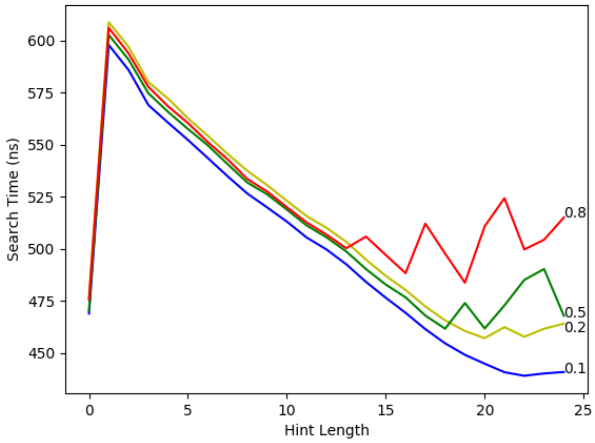

Figure 2: Results of timed searches across prefix length and across load factor

then uses bitwise shift operation to extract bits one by one starting from the most significant bit and traverse down the trie (i.e. go left or right depending on the value of the bit) adding new nodes as necessary. Each node would carry *prefix data*, i.e. prefix and next hop information, and a pointer to left and right child nodes. The mask decides how many bits to use from the passed prefix. While traversing down, it also maintains a string of concatenated bits from root to the current position. Whenever a new node is created, this concatenated bit string and a node are added into hashmap held in length table at the index for the corresponding position in the trie.

2.2.2 Search. Search takes full destination IP address and a hint. An IP packet would carry both these pieces of information from which router can easily extract and perform longest prefix match search. Our procedure differs from normal prefix matching in a trie in that before we even enter the trie, we search for the hint-length prefix in the hashmap corresponding to all prefixes of length hint first. Searching in the trie then starts at that node, rather than the root. This search is then performed in the usual way - last matched prefix is remembered and trie is traversed down until a NULL is encountered. If there is a match with a longer prefix, last matched prefix is updated. Finally, once NULL is encountered, last matched prefix is returned.

3 RESULTS

This unibit trie with hint passing via hashmaps was tested using two IPv4 databases. One database has next hop addresses associated with each IP address and was used to build the lookup table (including the trie and the hashmap). The other had IP addresses alone and was used to perform search to get back the next hop address. A C++ timer was used to get the time to build the trie and hash tables as well as the time to perform the search. The tests were run un a linux-based machine with the Intel Xeon E5620 Quad Core Processor that has 12M cache and 2.40GHz clock rate.

Given almost 600,000 prefixes to add to the lookup trie, the insertion process took on around 1.5 seconds. Once the trie was built,

lookup was done on 500 IP addresses. Search of a single IP address was timed for 1,000,000 repetitions, each of course resulting in the same next hop address. This was done to average out any noise in the C++ timer–it claimed nanosecond resolution but seemed to be getting minorly different results between searches due to precision errors. The total search time was then averaged across searches to get the time to get the next hop address given that IP address. This timer process was done for each length hint from 0 to 24 on each of the 500 IP addresses looked up. The average search time for each hint length was plotted in Figure 2.

This experiment was repeated on varying maximum load factors for the hashmaps of prefixes. Each hashmap had the same maximum load factor, which is defined as the number of pairs in the hashmap divided by the number of buckets. The lower the value for the load factor (between 0 and 1), the less collisions expected for the hashmap since inserted items will be distributed more evenly out across the wider array of buckets. There is a tradeoff here between the memory used by the hashmap (a wider table means more memory) and the number of collisions (the narrower a table, the more collisions).

It may be useful to understand the building time for the trie when looking at subsequent search times. We can see that when the max load factor is really low, the building time is longer. This is because each time the actual load factor (which increases with each element inserted) exceeds the maximum load factor, a new, wider table is created and each element is re-inserted. This is going to cost insertion overhead. At the same time, there is a different cost associated with inserting into a hashmap with a large maximum load factor. The more collisions there are, the longer it takes to search through the collided elements (since more elements will collide to the same bucket). The build time for the trie and hashmap given various max load factors is given in Table 3. Not all of these max load factors were used in the actual search experiment since they gave no more interesting results than the ones in Figure 2.

4 EVALUATION

Maximum Load Factor	Trie and Hash Table Building Time (in seconds)
Base (no hash table)	0.712
0.1	1.533
0.2	1.464
0.3	1.469
0.4	1.512
0.5	1.517
0.6	1.525
0.7	1.528
0.8	1.530

Table 3: Average build time of the Trie and Hash Table given different maximum load factors

Looking first at Table 3, we can look at how the load factor for the hash tables affects building overhead. The first thing we see is that adding in a hash table incurs a large building penalty. The building time without the hash tables (Base) is less than half the building time of a system using the hash tables (all others).

Looking a bit deeper, the relationship between maximum load factor and build time is not a simple one. As we said in the Results section, the frequency of increasing the size of the hash table (triggered by the actual load factor exceeding the maximum) as well as the collision penalty both affect the building time. Based on Table 3, there seems to be some ideal load factor that will minimize the building time. Our results point to 0.2 as the ideal but repeating the build with finer granularity (0.75, 0.25) could result in even faster build times. A complicating factor in this is that minimizing the build time will affect the search time and in not always intuitive ways. As we can see from Figure 2, the load factor that yields the best search results is 0.1, whose build time is not minimal.

The build time may not be easy to minimize, but Figure 2 seems to imply that getting stable and minimal search times is more predictable. The first thing to note is that the hashmaps are not searched for a hint length of 0. This explains why the search time is so small for hint length of 0, regardless of load factor. Without searching in the hashmap, looking up the address in the prefix table from root to longest match takes about 475 nanoseconds. For hint length of 1, the hash tables spike up to about 600 nanoseconds with a maximum load factor of 0.1 being the fastest.

From hint length of 1 up to hint length of 24, a maximum load factor of 0.1 steadily decreases in search time. This is because the search skips each level of the trie that the hint gives it. So for a hint length of 6, it gets the hash table containing the mapping of prefixes length 6 to the node in the trie corresponding to that prefix. Then, it can skip the top 5 levels of the trie by jumping straight to the node it got from the hash table and start the search for the rest of the prefix from there.

The monotonically decreasing search time for hint length given a maximum load factor of 0.1 indicates that storing the precomputing various length prefixes in a hash table indeed saves search time. This is our most compelling result. As the hint length increases, so do the number of levels in the trie we can skip, ultimately resulting in fewer memory accesses to find the corresponding next hop address.

The problem then is that for other maximum load factors, the search time is not monotonically decreasing and has various strange spikes. The larger the load factor, the earlier this spiking occurs and the more dramatic they are. Because of the black box nature of the g++ compiler's hash function, it is hard to predict where collisions will occur, but we should expect to see a uniform distribution of hashes across different prefixes in the hashmaps. Because of this, there must be some not easily seen pattern in the input IP addresses that are being used to build the table.

Looking at the actual input, we can see that all inputs range from 1.0.*.0 to 223.225.*.0. But, rather than utilizing all 256 possible values for the third field, each 16-bit prefix has only ten corresponding values for the third field. Also, all prefix have a 0 for the last field.

The input addresses cycle through most of the first 16 bit potential prefixes (1.0.*.0 to 223.255.*.0). The most reasonable explanation for the spiking pattern is that there is not an even distribution from 0 to 255 for the 10 values of the third field, causing a slight input pattern when building the table. This pattern is exacerbating collisions in the longer length prefix hash tables (starting at length 15). For example, if 1.0.*.0, 132.156.*.0, and 255.1.*.0 all had the integer 15 as their third value, the hash function might accidentally hash all of these to the same bucket. The problem may be small for just three addresses but will scale horribly out of control if hundreds of thousands of IP addresses have a similar pattern.

This is bad news, but it can be fixed with a small enough maximum load factor. As the line for 0.1 shows, as long as there are enough buckets, the collisions will be kept small and search time minimal. This result is promising since it shows that the process of passing hints to expedite the lookup in the unibit trie works. The further into the trie we jump, the less time search takes.

5 CONCLUSION

If it is the case the lookup tables update infrequently and new IP address, next hop mappings are rarely added, then insertion overhead is acceptable. The router can take the extra second to build a lookup table that includes 32 hashmaps and it will likely see improved search times. But if the network is very dynamic and mappings are being added and removed regularly, this could be expensive. There should be a feasible way to be lazy and only recompute the lookup table when there is some time. For example, if it kept a smaller dynamic lookup table where it temporarily puts new mappings that come in, then it can buffer new or changing mappings there and insert them into the lookup table when there is time. This is similar to the way a cache works in memory. Or, if a mapping changes or is removed, there could possibly be a way to flag that IP address in the lookup table so that if lookup is done on it, the router knows that it has changed. Then, when recomputing the lookup table itself, the router can scrap the changed addresses. This parallels the way some garbage collectors flag dirty memory to be cleaned up in the future.

Repeating the entire experiment with finer granularity will point to a maximum load factor that will result in an overall minimal time, but if that means that the build time is too great, it still may not work for the desired system. There could be another problem if memory is tight, since a smaller load factor means a more memory-demanding hashmap. Choosing the ideal value is going to depend on the network in which the router will be placed. Routers in very dynamic and changing networks will shy away from extremely bad build and update times but routers that donfit plan on changing often probably wonfit worry. It is up to operators to calibrate the system to match their requirements.

REFERENCES

[1] https://github.com/SivaKesava1/1bit-Trie

[2] George Varghese. 2004. Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.