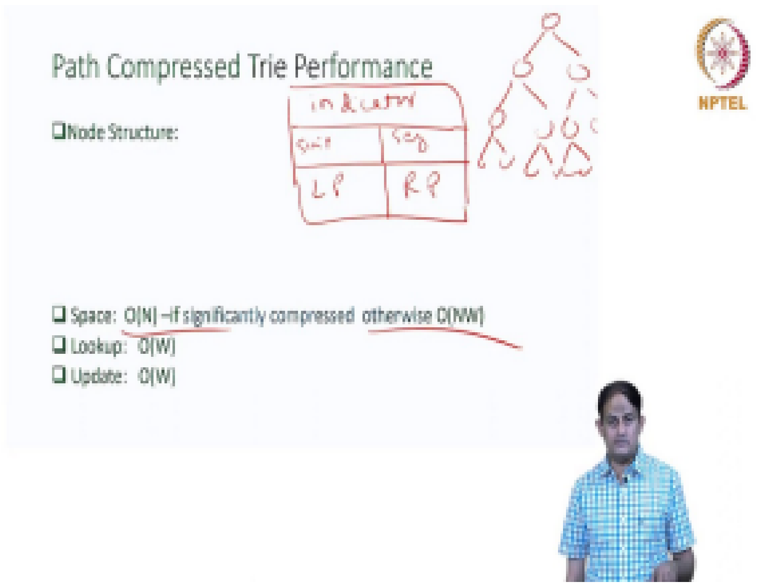
Advanced Computer Networks

Professor Neminath Hubballi
Department of Computer Science and Engineering
Indian Institute of Technology Indore

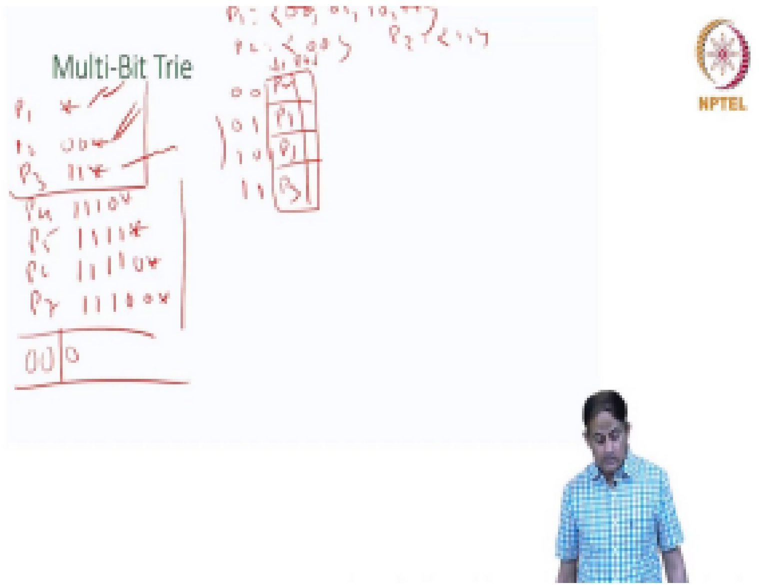
Lecture 06
IP Table Lookup: Trie based Data Structures

(Refer Slide Time: 00:19)



Path Compressed Trie Performance

- Node Structure:
- Space: O(N) – if significantly compressed otherwise O(NW)
- Lookup: O(W)
- Update: O(W)



Multi-Bit Trie

Now, the question that we ask is, is it the only optimization that you can bring? Are there any other optimizations that are possible in this structure? The answer is Yes. And that brings us to the discussion of what we call the multi-bit trie, where the notion of matching more than one bit is taken and then streamlined and I am trying to alter the data structure a little bit.

And in each iteration, when I do the reference, I am going to match a fixed number of the bits. Remember, in the previous case, when I was trying to match, the multiple bits are matched only when there is the skip value and the segment and I got some numbers precisely greater than 0.

Other than that, only one bit is actually getting matched. So, for example, here and here, you are actually doing only a one-bit match. So, I am going to extend by having a structure with a fixed number of bits matched at every iteration. So, let us motivate how this is actually done and what exactly the data structure looks like, and what is the performance of that with an example.

So, let us try to take this set of prefixes, as usual, P_1 has got the default route *, P_2 is let us say 00*, P_3 is 11*, P_4 is 1110*, P_5 is 1111*, and P_6 is 11110* and P_7 is 11100*. So, these are the set of prefixes that I got, basically seven of them.

And now, I want to construct an optimized structure where more than one bit are matched; let us say I am going to match two bits at a time, so in every iteration, I am going to match at least two bits. When I traverse from point X to Y, I am going to match the two bits, if I decide to go with the three length, then at every iteration and every label is going to indicate the 3-bit comparison.

So, now how do I actually do this? let us try to construct the data structure, multi-bit trie structure, and let us say I am going to match two bits in every iteration. So, I am going to have a table that looks something like this: it has got four entries, so 00 is one possibility, 01 is another possibility, 10 is another possibility, and 11 is another possibility; I am going to write in this one what prefix is actually going to be matched for these values.

So, for example, if I get an IP address, which is starting with, let us say 000, what I am going to do this is the destination IP address, there is something else beyond this, I am going to take the first two bits and go to this table and ask what is the given these two bits, what is the best prefix match that I can find from among these seven of them 00.

So, there is one which is P_2, just saying 00* there are two of them, P_1 is also match that is * is anyway going to match P_2 is going to match because it is starting with 00* and P_3 is not match because it are is starting with 11*. So, that is not it is you have got these two possibilities, either

P_2 or P_1, and the given the how the routing is done, the best match and the longest prefix that matches that is going to take the priority over the other things.

So, in a nutshell, what I am trying to say is P_1 is actually a superset of prefix P_2, whatever P_1 includes it includes 0 it includes 1, it includes 00 it includes everything is matched. P_2 is actually a subset of that. So, whatever is the best is P_2 is the best then I am going to take it on that.

Now, P1 is not having a prefix of length two, P_2 and P_3 have got a length of two. So, I can so every time something starts with 00, I can actually go and directly say that P_2 is the best match. Every time, the destination IP address starts with 11, I might say that P_3 is the best match on a similar note and now anything which is starting with 01, anything that is starting 10, we do not know.

So, that will actually be the best match that I can find for these two is the P1. So, I write P1 here, and then P1 here. So, one way to think of this is I am going to first consider all the prefixes of length of two length 2 whatever is there. And then if something is not having length two, then I am going to expand that.

So, for example, P_1, is going to include the 00 case, 01 case, 10 case, and 11, all of these are actually matched. But among them, P_2 is actually matching 00, I am going to strike out to 00 from P_1's case, and P_3 is matching 11, and I am going to strike out 11 from this set. And then for the remaining cases, I am going to write P_1 as the prefix.

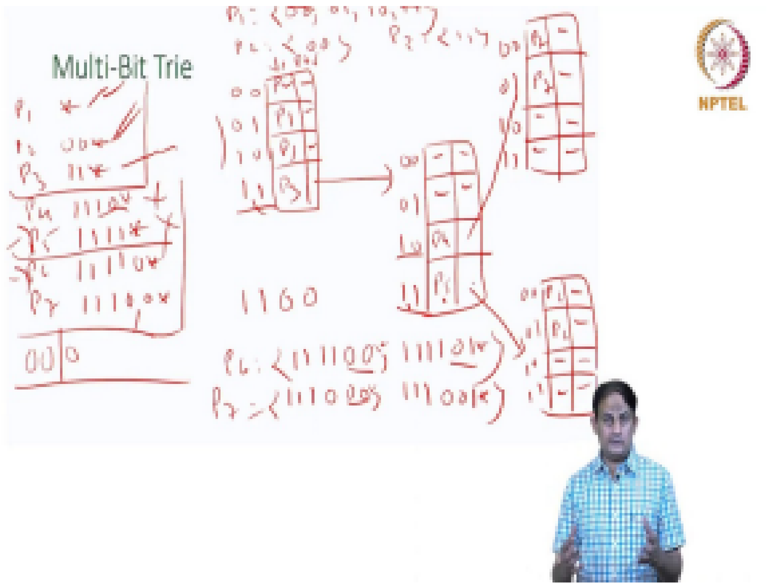
So, this is the prefix, this column indicates the prefix, the best prefix for those two bits that is matched and the next column in this case in this table, indicates the pointer. So, remember, there are some more prefixes which are longer than length of two, we need to accommodate all of them.

So, the way to do it is, if any of the prefixes which are of length more than two bits here, that we are going to, I am going to have it in the second level structure, one level structure, 2 bits case is done.

So, if your match is only two bits, then at the first level I am going to complete, and if the other prefixes are of length greater than two, then I need to accommodate at the second level. So, in order to do that, what we will do is, we are going to find out if any of the prefixes which are upto length two are a superset of the other prefixes which are of a length greater than two.

In this case, 11* is actually a superset of P_4, which is 1110*, which is also a superset of P_5, which is 1111* and similarly P_6 and P_7. So, in effect, P_3 is actually superset of P_4, P_5, P_6 and P_7, it includes everything that P_4 includes it matches everything that P_5 includes, and so forth.

(Refer Slide Time: 08:03)



So, in order to do that, what I am going to do is I am going to have a pointer from this location to the next level. And again, I am going to match two bits in this case. So, one for the 00 case, and second for the 01 and 10, and 11, these are the four possibilities, now look for the those prefixes which are of the length four or less than that, that I am going to accommodate here in this case P_4 and P_5.

Since P_3 is a prefix of P_4 and P_5, two bits are matched here. And the next case is the next two bits, 11 is matched, I need to find out 10 11 followed 10 is where? here it is the here I am going to write that this is the case of P_4 and 11 followed by 11 two 1s are matched here and the second two 1s is going to match here and this is the case of P_5. So P_5 two 1s followed by two 1s are taken care here and here.

And now for the 11 followed by 00, 11 followed by a 01, what is the match? 11 followed by 00 it does not exist even the P_6 and P_7 are not the cases. So, for these two cases, we do not have that. what is the best match I can have if the destination IP address is 11 followed by a 00 and P_4 is not the right match, P_5 is also not the right match, what is the best match that you can find out is the P_3 which is already there in the first level trie so in order to do that, I am going to mark these as blank ones.

So, there is no additional prefixes that are matched for using these set of the values. Whatever you found out in the first case itself is the best match that you have found out. Now, length for up to length four are taken care.

P_2 is not the superset of any of the prefixes here, I am going to mark this as blank this as blank, this as blank and only P_3 is a prefix of P_4 and P_5 that we note here. And similarly, these two also will not have any expansions.

Now, the prefixes up to length four are taken care. And now there are prefixes which are greater than length four so, which in this case P_6 and P_7 which are of length six, I am going to write what is if P_4 is a prefix of something if P_5 is prefix of something that I need to extend further.

So, P_6 says 1111, which is we are right now here and then followed by a 0*. So, in order to do that, I am going to have a pointer here and then construct another line and write all four possibilities 00, 01,10 and 11. The pointer is pointing to this table. And what it says is, it is a length five.

So, again, you do the P_6 if we expand P_6 will include what all 11110 and I want a length of multiples of two, so this includes, the six bits can be 0, or it can be 111101*. So, these 2 cases, so either the last bit 2 bit can be 00, or 01. These two cases, P_6 includes so I am going to write P_6 here and P_6 here.

And this is the last, we have, I am going to mark all of these pointers as blanks, and this is blank, this is blank, because 11 followed by 11 followed with 10 is the best match that you can find out is the P_5 in this example case, so that they can care. And similarly, if P_4 is a prefix of something else, so then again, we need to accommodate that as well.

So, in this case as P_4 is 1110, triple 1 followed by 0 is P_7. So, I am going to have a table here. And then the again the four cases 00 01 10 and 11, and this is again, you do the expansion of P_7. So, those 2 cases P_7 would involve triple 1 followed by 0, and another 0, and then a * : triple 1 followed by double 0, and then 1 and then a *.

So, two 1s in the first level, two 1s in the second level, and then 10 in the second level, and then two 0s, this is going to be with P_7, and then 01, this is going to be also matched with the P_7 these all pointers and the other in this table are blanks.

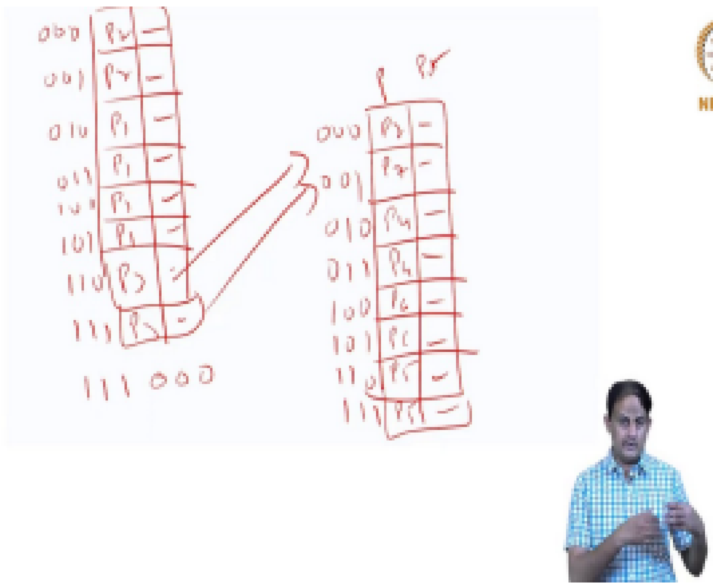
So, this is how I construct the multi-bit trie. So, in effect, what I am trying to say is, you decide how many number of the bits I am going to match in one comparison. If the processor that exist in the router is able to do multi bit comparisons in one go, you actually decrease the height of the trie.

Let us say I am going to do three bit comparison, I am going to do two bit, I am going to do four bits, eight bits, whatever it is possible, you do all of them thereby you considerably reduce the height of trie.

So, you can think up with the extension of the previous case, all that we did is formulae in every instead of doing a variable number of the bit comparisons at every node in the previous case, all that I am saying is at every level node I take the fixed number of the bits and do a search in this table and you can even hash it and then in one iteration you are going to get what is the corresponding node and then you traverse this trie and wherever you find a blank pointer that is the best match that you can find for that particular destination IP address.

So, for two bit multi bit trie this is the structure so I can also very well do it with the three bits as well. So, let us try to do that for the same set of the sequences how the three bit trie would look like.

(Refer Slide Time: 14:53)



So, if you do with three bit trie then what is the case? You are going to have 8 number of possibilities and at every iteration I'm going to do a 3 bit comparison 000, 001, 010 and 011, and the next one 100 and 101 110 and then finally 111 so, 8 possibilities are there. Again, the table would have 2 parts, first part for the prefix and second one for the pointer. And then out of this, you actually try to match what is the best possibility that you get.

So, in the previous example, if you find a 000, that is best matched with P_2, I am going to write a P_2 here. If it is 00, followed by a 1, then also in this case, the P_2 is the best match, I am going to write here P_2. And if it is 0 followed by 1, then in among these setup the prefixes, what is the best match * is the best match you point, so I am going to write P_1 here, and if it is 0 followed by 11, then also * is the best match that you can find.

So, then I am going to write P_1 here, and 1 followed by double 0, none of these prefix matched, then the best match that you can find is the default route that is the P_1, now write P_1 here 101 will also have best match that you can find is the P_1 that I am going to write here.

So, double 1 followed by * is there that is P_3 and that can be 0. So, I am going to write P_3, here. 110 is P_3. And 11 followed by 1 can also be matched with P_3. I am going to write P_3 here. So, all these cases will have pointers as well. But only P_3 is actually prefix of something else that we understood. 00 is none of the prefixes which are greater than length 3, or having the subset of this, P_2, and even so I am going to omit that. So, these 2 cases are there.

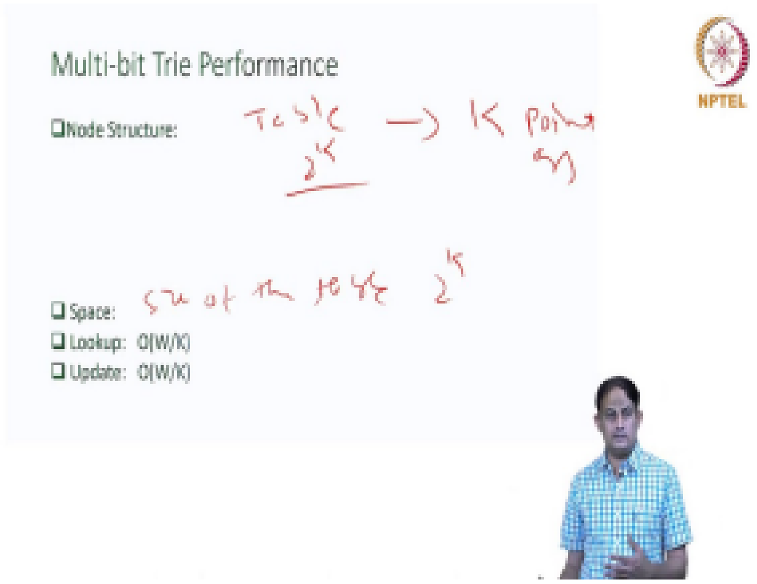
So, I am going to now have a second-level structure, so remember, what is the longest prefix length we got P_5. So, if I do 3-bit comparison at a time, then the 6 is the next level. So, in two levels, I am going to be able to finish the entire comparison, the second level will have options from 000, 001, 010, 011, 100, 101, 110, and then 111.

So, this is the prefix first column and then the pointer. IP address is 111000. So, for triple 1 followed by triple 0, P_7 is the best match that you can find out among the set of prefixes, and for triple 1 followed by double 01 again, the best match that you can find is P_7. Similarly, this is going to be P_4, this is going to be P_4, this is going to be P_6, this is going to be P_6 and this is going to be P_5 and this is going to be P_5.

So, the longest prefix that we got is of length 5. So, we are at level 2 and 3 bits at a time. So, 6 bit cases are taken care and all of these actually pointers are having the values. Let us say decide to match key bits at a time and I got to 32 bits possibility 32 divided by k bits which is 4 then the you need to construct this structure for up to length 8, and then you will be able to find out the match.

So, the larger the length of the match in one go, if I decide to do 16 bits and then in only 2 levels, I will be able to do it again but that comes at the expense of the table size growing. So, at each level we will have 2^{16} entries so that is not desirable. So, there is a trade off how many number of the bit comparison that you can do and what would be the height of the total structure that you get.

(Refer Slide Time: 19:52)



Multi-bit Trie Performance

- Node Structure:
- Space:
- Lookup: O(W/K)
- Update: O(W/K)

So, with that background in mind, let us try to understand what is the performance of this multi bit trie? What advantage that it brings? So, the first thing is what is the node structure. So, I have got a table in the first place and each table has got k number of the pointers and 2^k number of the entries. So, at every level, you will have got where k is the length of the match that you are going to do in one iteration, in one go I am going to do the k bit matching.

So, accordingly, if you use a linked list, then you can create a structure that has got 2^k number of the entries and then those many number of the pointers, that is one possibility or you can use the kind of the HashMap or the algorithms where in order of one you can find out which actually prefixes 3, in this case, k bits are matched, and then accordingly, you can take the pointer. So, that is the alteration that is done.

And in terms of the space, how much space you require, 2^k number of entries in one table, and then those many number of the pointers. So, the size of the table so, which is 2^k, each node has got the 2^k number of the entries, which is going to accommodate all the possibilities in that structure. And the lookup operation is now speed up by k times, if w is the length, 32 bit IP address are there.

In each iteration, I am going to match 2 bits or 3 bits, 4 bits, whatever it is, and w the total number of the memory references, the number of the pointers that will travel is reduced by k times because every time you take one pointer, you actually match k number of the bits and the next time you go you again match the k number of the bits.

So, w/k is the total number of the lookup operations that you do. And again, as usual, the update operations will also take w/k number of times, in the worst case, you will traverse the entire w/k number of tables, that is, the height of the trie, and then you are going to add a new entry in the table. So, that is the performance of this multi-bit trie.

So, let us summarize what the optimization that we discussed first is, we took the prefixes and then constructed the binary trie, which is actually doing one-bit comparison at a time. And then in the worst case, we said that for IP version 4 address, you require 32 number of the memory references and the pointer references.

So, that is going to be a little slow, we try to optimize that we brought up another variant of the binary trie, which is path compressed trie, which is doing some kind of variable number of the bit comparisons at every node, depending upon whether the intermediate nodes are compressed or not that we indicated by adding a couple of variables inside the node structure, which is the segment length and the skip value.

And then we went one step further and then said that instead of doing the variable number of the bit comparisons at every iteration, I am going to do a fixed number of the bit comparisons at every level. So, be it 2 bits, 3 bits or 4 bits, then we said no we came up with this data structure, which is the multi bit trie.

So, we argued that if your processor is able to support these k bit comparisons in one go, then the lookup speed is going to be k times faster than the binary tries baseline performance, although it is not exactly k times possible but on average you will get the k times, why it is not exact because in multi-bit comparisons you also need to read that in one when we visit in a particular node, you need to find out what is, you need to read 3 bits of the input and then do 3 bit comparison which may not be exactly as much fast as the one bit reading and one-bit comparison.

But on average you get k times the honest improvement over the binary trie. So, with this we will stop at this point of time. In the next class, we will see what the other possibilities that exist? Are there any other optimization that I can do for the lookup operation?