Chapter 13
Fast Packet Forwarding

There is more to life than increasing its speed.
—Mahatma Gandhi\(^1\)

In this chapter I talk about various tricks for speeding up packet forwarding. First I explain schemes that add extra headers. Then I talk about two families of address lookup algorithms—Trie and binary search—together with some variants.

13.1 Using an Additional Header

In this section I describe schemes devised to allow routers to make a forwarding decision based on a small header with small addresses rather than IP addresses, on the assumption that it would be prohibitively expensive to build a really fast router that made forwarding decisions based on the regular IP header. The disadvantage of these schemes is that they require adding more information onto the header. IP would probably be a better protocol with this information, but schemes such as the ones described in Sections 13.3 and 13.4 make these schemes less necessary, at least for the purpose of fast packet forwarding.

All the schemes are conceptually similar. One scheme, known as \textit{threaded indices}, was invented by Chandranmenon and Varghese.\(^2\) Another is specific to IP and ATM. One team that developed this sort of approach was Newman, Minshall, and Huston.\(^3\) Another was Parulkar, Schmidt, and Turner.\(^4\) Within the IETF, this sort of scheme, known first as \textit{tag switching}, is now called MPLS (multiprotocol label switching). Originally, these

\(^1\) Quote 29 from Byrne, \textit{1911 Best Things}, 11.

347
schemes were designed to make it possible to build fast routers, but then, using techniques such as those described in Sections 13.3 and 13.4, people built routers fast enough on native IP packets. So now MPLS is thought to be mostly a technique for classifying the type of packet for quality of service or for assigning routes for traffic engineering (the ability to control what the routes are rather than have routing simply compute whatever it wants).

These schemes involve something that looks like having an ATM header on your IP packet. But with ATM, you must set up a route first before you can send data. With these techniques, suppose that router R2 receives a packet from R1 for destination D. Without the extra header, or with a header with a label of 0, R2 makes a slow path forwarding decision based on the IP header. R2 may, if it wants, tell R1 what to put into the packet to save R2 time in the future. There are two strategies for this.

- Tell your neighbors your desired label as part of the routing protocol. For example, in a distance vector protocol, announce not only your distance to a destination but also the label you’d like someone forwarding to you to use for that destination.
- Do it after receiving data that needs to go through the slow path. As a result of this, send a message to the node that sent you the packet telling it a label to use in the future.

The second scheme has more latency for the first packet because it must go through the slow path until the downstream router can be notified of the label, but it allows a smaller label space because only labels for currently active flows through the router need to be assigned.

In the IP/ATM schemes, the label is an ATM VCI. In the threaded indexing scheme, the label is whatever would be convenient for R2. After the labels are set up, the packet forwarding looks just like ATM, but with large packets. But packets can still be routed even if labels aren’t set up at all, or set up only on parts of the path, because the labels are an optimization.

The protocols being developed by the MPLS working group are currently subject to change. There are no RFCs, and only a lot of Internet drafts. But the group seems to be converging on basically reinventing ATM. There would be a backbone that routes based on “labels” rather than IP headers. The ingress router would assign the packet to a “forwarding equivalence class” (in other words, a VC) and label it appropriately. Within the backbone, routing would be based totally on the label. You look at the input port and label, and you find it in a table that tells you the proper output port and label.

I think this is a fine way to do things. The main thing I dislike about ATM is the small cell size. MPLS takes all the rest of the ideas of ATM but uses IP-sized packets.

### 13.2 Address Prefix Matching

Address matching is the most critical part of making a high-performance router because a router’s performance is based on how quickly it can forward a packet. To forward a
packet, a router must extract the destination address from the packet and find the matching forwarding database entry.

In layer 2 (bridge) forwarding and in CLNP and DECnet level 1 routing, routing is based on exact matches. For example, in CLNP, the ID field is the field on which level 1 routers make their routing decisions. A CLNP level 1 router must have an entry in the forwarding database exactly matching the ID portion of the destination address in the packet. In IP, finding the proper entry in the ARP cache involves essentially the same problem. There are hashing algorithms to find exact matches.

In contrast, in IP, IPX, and routing on the area field of CLNP, there might be several forwarding database entries that match a particular destination address, and it is the most specific or longest address that must be found.

For example, in IP, each entry in the forwarding database corresponds to a 32-bit address and a 32-bit mask. The destination address field in a packet contains a 32-bit value. The router must find the forwarding database entry that has the mask with the most 1's that matches that destination address. All the following examples match destination address 11001111 01011100 00000000 10001111:

1. Value: 11001111 01011100 00000000 10001111
   Mask: 11111111 11111111 11111111 11111111

2. Value: 11001111 01011100 00000000 00000000
   Mask: 11111111 11111111 00000000 00000000

3. Value: 11001111 01011100 00000000 00000000
   Mask: 11111111 11111111 11000000 00000000

In the preceding examples, it is the first item that should match the destination address because the mask is the most specific. However, as IP was originally specified, there can be ambiguities if noncontiguous subnet masks are used. For example, suppose the following two entries were in the forwarding database:

1. Value: 11001111 01011100 00000000 00000000
   Mask: 11111111 11111111 11111110 00000000

2. Value: 11001111 01011100 00000000 00001111
   Mask: 11111111 11111111 00000000 01111111

Both masks have the same number of 1's, and both match the destination address. No standard ever specified which entry, in this case, should be considered a match. If one router chose the first entry and another chose the second, a routing loop could result.

As discussed in Chapter 6, noncontiguous subnet masks are not particularly useful. They are extremely confusing, and they make it difficult to build efficient routers. If subnet masks are contiguous (that is, if they have the property that no 1 bit appears to the right of any 0 bit), then the longest-matching-address-prefix algorithms (described in Sections 13.3 and 13.4) can be used to efficiently find the forwarding database entry for a particular destination.
Because nobody believes in noncontiguous masks, IPv4, IPv6, IPX, and CLNP all route based on the longest matching address prefix. If noncontiguous subnet masks are allowed in IP, it is possible in the general case that no efficient algorithm exists. Trying every mask in the forwarding database might be the best algorithm.

In address-prefix-matching algorithms, the forwarding database consists of a dictionary of address prefixes. The problem is to find the longest initial substring of the destination address that is included in the forwarding database.

### 13.3 Longest Prefix Match with Trie

The Trie algorithm is described by Knuth. The addresses in the forwarding database are put into a tree data structure. Each vertex represents a prefix. The root consists of the zero-length prefix (which matches any address). There are two pointers from each vertex: a “1” and a “0.” If no string in the dictionary contains the vertex’s string plus the extra bit value, that pointer either is missing or points to a vertex indicating failure.

Each vertex also has a flag associated with it, which indicates whether that string, terminated there, is in the dictionary. In Figure 13.1 the flag is indicated with an asterisk.

![Trie structure for longest prefix match](image)

**Dictionary:**

\[
*, 00^*, 001^*, 0001^*, 11^*, 101^*, 0101^*, 111^*, 10100^*
\]

Figure 13.1 Trie structure for longest prefix match

---

Assume that we have the destination address 10101101. We'd use the Trie structure in Figure 13.1 by starting at the top and traversing the path indicated by the destination address, remembering the last time we saw a flag (an asterisk). Starting at the top, the zero-length prefix has a flag. The first bit of 10101101 is 1, so we go to the right and get to the node with the name "1." No asterisk, so {} is still the longest prefix matched so far. The second bit of 10101101 is 0, so we go to the left, to the node 10, and still no asterisk. The third bit is 1, so we go to the right to node 101*. Aha! An asterisk. So 101 is the longest prefix matched so far. We continue. The next bit is 0, and we get to node 1010. The next bit is 1. But there's no pointer marked 1 out of node 1010, so at this point we know that the longest prefix matching 10101101 is 101.

Maintaining the data structure is not difficult. To add an entry, say 1011, follow the pointers to where 1011 would be in the tree. If no pointers exist for that string, they should be added. If the vertex for the string already exists—say, in adding the prefix 01—the vertex merely needs to be marked as being in the dictionary, that is, an asterisk must be added. To delete an entry that has children, the vertex associated with the string is simply unmarked as a legal termination point (the asterisk is removed). To delete an entry that has no children, the vertex and the pointer pointing to it are deleted, and the parent vertex is examined. If it has other children or is marked as a legal termination point, it is left alone. Otherwise (it is not a legal termination point and there are no remaining children), that vertex, too, is deleted, and so on up the tree until a vertex that has other children or is marked as a termination point is found.

This algorithm requires lookup time proportional to the average length of the strings in the dictionary, regardless of how many entries the dictionary contains.

In the case of CLNP, prefix lengths are specified in terms of numbers of nibbles (4-bit chunks), so instead of two pointers from each node, there might be 16. And the search would be faster because 4 bits at a time would be searched.

The following sections describe improvements to Trie.

### 13.3.1 Collapsing a Long Nonbranching Path

If there is a path in the Trie structure that does not branch and has no asterisk, then you can collapse it so that the entire substring can be found in one probe. For example, in Figure 13.2, you can remove the nodes 01 and 010 by noting that in order to take the "1" branch out of node 0, you must match the three subsequent bits 101.
13.3.2 Trading Memory for Search Time

Even though IP prefixes can be any number of bits, Srinivasan and Varghese showed that it is not necessary to search a bit at a time. The idea is simple. Suppose you decide that you want to search 4 bits at a time. You simply expand a prefix that isn’t of length 0, 4, 8, 12, 16, 20, 24, 28, or 32 into several prefixes (8 for prefixes 1 more than a multiple of 4, 4 for prefixes 2 more than a multiple of 4, and 2 for prefixes 3 more than a multiple of 4). So, for example, the prefix 10 would expand into 1000, 1001, 1010, and 1011. Any address that matched the prefix 10 would have to match one of \{1000, 1001, 1010, or 1011\}.

Of course there are subtleties. For example, suppose you have prefixes 1, 10, 100, and 1000. When you expand the prefix “1,” you add prefixes 1000, 1001, 1010, and so on. When you expand “10,” you get 1000, 1001, and so on. When you expand 100 you get 1000, 1001, and so on. And you already have prefix 1000, which happens, conveniently, to be 4 bits long. The authors call this prefix capture. What should you do? Keep all the prefixes? If not, which ones do you discard—the ones padded from smaller prefixes or the ones from larger prefixes? (See homework problem 2.)

Also, the scheme does not require always searching the same number of bits, although at each node you search a fixed number of bits. So, for example, you could at the top level search 4 bits, but then under the node 1101 you could search the next 6 bits and under the node 1110 you could search 2 bits.

---

If you decided to do this scheme in a straightforward way but use prefix lengths that were, say, multiples of 16 bits, then the 64,000 multiplier for memory would get annoying. Degermark et al.\(^7\) published a clever scheme for data compression for expanded prefixes. It was called Lulea (the university of the inventors of the scheme). It is very difficult to understand and I include it here because the paper is often cited. But it’s not necessary to understand this in order to understand the rest of this chapter, so you could skip to Section 13.3.3.

Assume that you want to expand prefixes to lengths that are multiples of 16 bits. You will wind up with \(2^{16}\) child nodes. If you really had all those prefixes, there’s no way to save memory. But suppose you really had only a small fraction of the \(2^{16}\) combinations that were “real” prefixes. Most of the nodes would then be the result of expanded smaller prefixes, and you’d be wasting a lot of memory. The Lulea scheme allows you to do data compression in this case.

Imagine that you are searching the next 16 bits after some string \(X\) (see Figure 13.3).

![Figure 13.3 Lulea scheme looking at the 16 bits following X](image)

We’re looking at the portion of the address beyond \(X\). For clarity, I’ll simply call the prefix \(X01\) “01” and ignore the fact that the prefix actually starts with the string \(X\). So starting with \(X\) we have a set of prefixes, and we’re going to expand them to 16 bits. The prefix 01 would wind up getting expanded into \(2^{14}\) child nodes, one for every possible combination of the bottom 14 bits (all the child nodes between 0100 0000 0000 0000 and 0111 1111 1111 1111). But it saves space to keep track only of the endpoints of ranges rather than listing all the \(2^{14}\) nodes.

The straightforward way of having \(2^{16}\) child nodes is to have an array \(2^{16}\) wide. To look at the node for the prefix 1101 0100 1000 0001, you’d index into the array by 1101 0100 1000 0001. Each item in the array would have to be fairly large to carry other information you’d like to keep, such as a pointer to the next array representing the children of that node and the number of bits to be searched after that node.

But Lulea wants to keep really minimal storage. Let’s call the array that has one element for each of the \(2^{16}\) child nodes the child-array. Assume for now that there’s only a

single bit at each location in the child-array (see Figure 13.4). The rule is that if that child node is the result of a 0-padded prefix or exact match, then that bit is set. Also, if it's the entry immediately following a 1-padded prefix it's set. Otherwise, it's zero. Note that in some cases the entry immediately following a 1-padded prefix might already be set because it is a 1-padded prefix. That would occur, for example, with prefixes 0* and 1* because the value immediately to the right of the end of the 0* range is the left side of the 1* range.

All the entries with a 0 in the child-array have the same information as the closest entry (on the left) with a 1-bit. There's another array, which I'll call the info-array, that contains the real information you want (such as the pointer to the array representing the next generation for the Trie structure, or the forwarding information), but with a single element representing all the nodes in a range (see Figure 13.5). The challenge is to know the proper index into that array. For example, assume that the prefixes are 0, 1, and 001. Then the child-array elements at 0000 0000 0000 0000, 1000 0000 0000 0000, and 0010 0000 0000 0000 (representing the small end of a range) will be set. And the child-array element 0100 0000 0000 0000 (which is 1 more than 0011 1111 1111 1111) will be set because it represents the end of the range of addresses covered by 1111. There is no bit set for the end of 1* (because it falls off the right end). And there is no bit set for the end of 0* (because it's already set in this case because of the left end of 1*).
Referring to Figure 13.5, the info-array has only four elements. All the addresses in range A and C match prefix 0. The addresses in range B match 001. And the addresses in range D match 1. The addresses in range A need to access the first element in info-array. The addresses in range B need to access the second element in info-array, and so on.

The straightforward method for knowing which entry in info-array to access is to count the number of 1 bits to your left. But that would be slow. So Lulea takes extra space in the child-array—say, every 64 bits—to store the number of 1 bits to the left. You need to access that number and count the number of 1’s to the left of you in your own 64-bit chunk. The total will be your index into info-array.

### 13.3.3 Binary Search on Prefix Lengths

This idea was invented by Waldvogel, Varghese, Turner, and Plattner.\(^8\) The idea is to do a binary search on prefix lengths. Suppose you don’t bother with the prefix expansion in the preceding section. This means that there are 32 possible prefix lengths in your dictionary. First, you do a hash on the first 16 bits of the destination address to see whether it matches any of the prefixes of length 16. If it does, you look for a longer prefix, so (by binary search rules) you search for a prefix of length 24. Not there? Now check against 20, and so forth.

Well, this doesn’t quite work. Consider the prefix 0011 1011 0000 1111 101. That’s a length of 19. Suppose we have an IP address that matches that prefix. When we check

against the prefixes of length 16, we must make sure that we know we must search for longer prefixes. If we did the most straightforward thing and if the prefix 0011 1011 0000 1111 didn’t happen to be in the dictionary, we’d think we had failed! So, we must be a little more clever.

What needs to happen is that for all prefixes larger than 16, we must put a marker into the prefixes of length 16. The marker indicates there is a longer prefix that begins with the found 16-bit prefix. In this case, we’d put 0011 1011 0000 1111&, where “&” indicates that a longer prefix exists. And just in case the destination address matches those 16 bits but not the actual prefix (say, if the address were 0011 1011 0000 1111 0101 1100 0000 1110), we also must keep the longest prefix matched so far. So for each prefix in the table of prefixes of length 16, we keep

- The 16-bit value
- The “&” flag that indicates whether there’s actually a longer prefix
- The size of the longest prefix so far (if these 16 bits match)

For example, if 0011 1011 is in the database but there is no longer prefix until the 19-bit prefix 0011 1011 0000 1111 101, then the entry in the prefixes of length 16 would contain

- 0011 1011 0000 1111 (the 16-bit prefix)
- & (indicating a longer prefix exists)
- The value 8, indicating that if searching for a longer prefix fails, the longest match will be the first 8 bits of 0011 1011 0000 1111, or 0011 1011

For example, if the prefix 0011 1011 0000 1111 exists and a longer one also exists starting with the same 16 bits, then there would be an entry in the table of prefixes of length 16 that would contain

- 0011 1011 0000 1111 (the 16-bit prefix)
- & (indicating a longer prefix exists)
- The value 16, indicating that if searching for a longer prefix fails, the longest match will be 0011 1011 0000 1111

I think that’s really cool! There are many other subtleties, such as making a good hash at each stage of the search. If the hash isn’t good, it might take several probes at each stage to find the matching prefix.

An approach to finding a good hash is to calculate a custom-made hash for each prefix length, given the prefixes that happen to be in the database. For example, there might be a hash algorithm that can take a seed value. You try various seed values until you find one that never has more than, say, three items that collide. Then you triple the size of your fetch so that you retrieve all three values and never require more than one probe.

Another subtlety is that you don’t need to put in all shorter prefixes for a prefix. You need only put in prefixes in the spots that binary search would search. For example, in the
case of a 19-bit prefix, you put in a marker for that prefix only at length 16. Next, you’d probe 24 (and fail, so you must search for a shorter prefix). Then you’d probe 20 (and fail, so you must search for a shorter prefix). Then you’d probe 18 (and find another marker, so you must search for a longer prefix). Then you’d find it at 19. If the prefix were of length, say, 30, you’d need markers at 16, 24, and 28. (See homework problem 5.)

13.3.4 Exploiting Parallelism with Special Hardware

This approach to speeding up prefix matching is mine, from work done around 1997. It does not require additional memory, as do the schemes in Sections 13.3.2 and 13.3.3. However, it requires a special-purpose lookup engine consisting of $k$ registers. Each stage of the lookup compares a portion of the destination address against all $k$ registers simultaneously, with the “winner” being the register with the longest match. The winning register points to a data structure that will hold the bit patterns that the registers will compare at the next stage.

We must look at all the prefixes in the dictionary to prepare the data structure for use by the lookup engine. This process does not need to be fast because it only prepares for fast lookup and is not invoked when an actual packet is forwarded. The second step involves the actual lookup procedure.

13.3.4.1 Preparing the Data Structure for the Lookup Engine

The special-purpose hardware has $k$ registers. For concreteness, let’s use the value $k = 13$. After the hardware is built, $k$ is fixed. The larger $k$ is, the faster the lookup will be and the more expensive the special-purpose hardware.

The goal is to divide all the prefixes in the dictionary into $k$ hash buckets. Instead of matching to a fixed number of bits in the prefix, we match to the maximal number of bits possible with the particular prefixes in the forwarding database. Start by naming three of the registers 0, 1, and *. Divide the items in the dictionary into those three hash buckets. The ones that start with 0 go into the bucket called “0.” The ones that start with 1 go into the bucket called “1.” The zero-length prefix (if it exists in the dictionary) goes into the bucket called “*”.

Next, get rid of any buckets that have no prefixes assigned. For example, if none of the prefixes in the dictionary starts with 1, get rid of the bucket called 1.

Next, expand the name of the prefix to be as long as possible. If all the prefixes in a particular bucket start with the same string, expand the name of that bucket to be that common prefix. For example, if all the prefixes in the bucket labeled 0 actually start with 0010 1, rename that bucket 0010 1 (see Figure 13.6). (Note that the spaces are there only for readability; 0010 1 is actually 00101.)
If fewer than \( k \) hash buckets are being used at this point, find the hash bucket with the largest number of prefixes and divide that. For example, if the bucket named 0010 1 has the most prefixes, divide it into three hash buckets: 0010 10, 0010 11, and 0010 1* (see Figure 13.7). Take the prefixes formerly assigned to the bucket 0010 1 and put them into the appropriate bucket 0010 10, 0010 11, or 0010 1* depending on whether the next bit in the prefix is 0 or 1 or the prefix ends after 0010 1.

Then repeat the steps of getting rid of empty buckets, expanding the names of buckets, and then, if fewer than \( k \) buckets are still in use, divide the bucket that has the most entries.

To divide a bucket in this way requires two additional buckets. But suppose there's only one extra bucket, and every bucket named \( X \) contains prefixes \( X0, X1, \) and \( X* \). \( X \) can be divided by using only buckets \( X0 \) and \( X* \). The prefixes starting with \( X0 \) will benefit from an extra bit searched during that iteration, but the prefixes starting with \( X1 \) can be searched only through the string \( X \). At the next iteration (if \( X* \) wins), there will need to be a bucket for \( * \), and all the other bucket names will begin with "1."

The data structure for the first search lookup consists of the values to be loaded into the \( k \) registers. The data structure has \( k \) entries that contain

- The number of bits in the name of the hash bucket
- The name of the hash bucket (the set of bits to compare against the destination address)
- \( \text{LNGST} \), which equals the number of bits in the longest matching prefix so far
A pointer to the data structure for the next stage of lookup if this register wins, where "wins" means it is the longest match to the destination address.

We keep LNGST in case the lookup ultimately fails. For example, if the database contains prefixes 01 and 0111 1111 1111 111 (along with lots of others, of course) and if the destination address is 0111 1111 1111 1101 1010, it is possible for the first iteration to search up to 0111. At that point LNGST would return 2 bits, because 01 is a prefix in the dictionary and we assume that 011 and 0111 are not in the dictionary. At the next stage, a further 6 bits might be matched, with a register containing 1111 11. Still, LNGST would return 2 because there are still no matching prefixes longer than 01. At the next iteration all registers might fail, in which case LNGST will equal 2, and the answer is the prefix consisting of the first 2 bits of the destination address.

The data structure for the next search lookup is prepared exactly like the previous one, but it looks at the value of the prefixes starting where the preceding stage left off. For example, if the register with value 0010 110 wins and the destination address is 0010 1101 1110 10..., then the lookup will be based starting from the eighth bit in the destination address. At the next stage, even though all the prefixes in the dictionary that have not yet been eliminated as matches for the destination start with 0010 1, the names of the buckets will omit the 0010 110 and will be named based on the bits following 0010 110.

This procedure winds up examining a maximal number of bits rather than being constrained to look at a fixed number of bits at each lookup stage. In addition, it allows \( k \) comparisons in parallel, where each of the comparisons looks at the appropriate number of bits.

The worst-case performance of this algorithm occurs when each time a bucket with \( n \) prefixes is split, it results in two buckets, each with one prefix, and the remaining bucket with \( n - 2 \) prefixes. However, that bucket will have a name of length at least \( k/2 \). With straightforward Trie with \( k \) comparisons, you could look at \( \log_2 k \) bits. So this allows exponentially greater numbers of bits to be compared at each iteration, even in the pathological worst case: the prefixes look like *, 0, 1*, 10, 11*, 110, 111*, 1110, 11110, ..., where in each case there is only a single prefix in any of the buckets that end in a 0 (or *).

If you are troubled by the pathological case, we can modify the program to improve the worst-case performance. If, for example, the majority of the prefixes wind up in the bucket 11110 and we want to go deeper into the address on that lookup for the vast majority of prefixes that begin 1111 0, then all the prefixes in *, 0, 1*, 10, 11*, 110, 111*, 1110, 1111* can be compressed into the single bucket *, which will match all destination addresses that start with anything other than 1111 0. Then 1111 0 can be divided again to make use of the remaining \( k - 1 \) registers. It means that for prefixes that start with anything other than 1111 0, that stage of the lookup will make no progress because the address lookup will not search past where it searched in the preceding stage. However, because there are so few prefixes this is not a problem. For example, if there are fewer than \( k \) prefixes in the buckets that got combined into *, the next search is guaranteed to find the answer for those prefixes.
13.3.4.2 Doing a Lookup

The first stage of an address lookup (see Figure 13.8) consists of loading the \( k \) registers from the initial location, which contains \( k \) items. Each item consists of the number of bits to compare against, the value of the bits to compare against, and a pointer to the data structure to be loaded if this is the winning register.

![Figure 13.8 Doing a lookup](image)

The destination address is compared against all registers in parallel. The winning register is the one that contains the largest number of bits that match the destination address. The output is a pointer to the data to be loaded into the registers for the next iteration of the lookup, or an indication that this is the end of the search. If the search is not over, the registers are loaded from the indicated location, and the next step of the search starts after the preceding search leaves off. For example, if the preceding step started at bit 11 and the winning register matched 8 bits, the next stage would start comparing at bit 19 in the destination address.

If the search is over—meaning that there is no pointer to a next iteration—the answer is indicated by LNGST, which gives the number of bits in the destination address that form the longest matching prefix.

13.3.4.3 Optimizations

Ideally, the data structure would be on the chip so that the data would not need to be loaded across a potentially small bandwidth pipe to the \( k \) registers. If the information must be transmitted at each stage of the lookup, ironically the search time can increase with larger \( k \) because the time to transfer the data onto the chip is proportional to \( k \), assuming a serial interface for transferring the data. If the transfer time is sufficiently large, that will dominate the search time rather than the number of probes. So it is a good idea to minimize the amount of information that must be transferred at each stage of the lookup in the case that the memory that holds the database is off the chip. If the search chip contains storage for the data structure, these optimizations are not necessary.
If the register loader can be intelligent, it can load the needed number of bits for each register. Otherwise, the bit string must be padded to be a constant length. If you want to avoid loading the chip with the pointer to data to be loaded for the next stage of the search, you can have the output of the comparison be the register number, with some off-chip location using that output as an index into a table of pointers.

We can save money on the registers, and on data transfer time, by not building each register with the ability to store or compare a full-length address into the prefix. For example, we could limit the number of bits to be searched each time to one-fourth the size of an address. This might result in an address lookup step in which the number of prefixes has not been reduced, but the number of bits matched in the address after that iteration will be one-fourth the size of the address, and that is still very good compared with any of the existing Trie-based schemes.

## 13.4 Binary Search

Ordinary binary search when you’re doing longest prefix matching does not quite work. However, with a few enhancements, a binary search scheme can work. This scheme was originally suggested by Butler Lampson. The idea is to think of a prefix 01* as a range of values from 010000000000000000000000 to 0111111111111111111111. If all the prefixes are padded—once with 0’s and once with 1’s—and then sorted, a binary search will determine into which range the destination address belongs. There are subtleties. The prefix 10 padded with 0’s looks just like the prefix 1000 padded with 0’s, and the prefix 1101 padded with 1’s looks just like the prefix 110 padded with 1’s.

Start by padding each of the prefixes in the dictionary to a length equal to 1 bit greater than the length of an address. For each padded prefix, note the size the prefix was before it was padded. The reason for this padding is to handle the case of a prefix that is equal to the size of an address, so that it, too, will appear as two padded prefixes and there does not need to be a special case to handle it.

### 13.4.1 Sort the Prefixes

Next, sort the padded prefixes. Note that both of the padded prefixes 101 and 1010 result in a padded prefix equal to 1010 0000.... When you’re sorting padded prefixes that end in 0, sort prefixes in which the unpadded prefix is a smaller number of bits as being smaller. (Note: For clarity I write the pad in a smaller type size and also put spaces after every 4 bits when the number is longer than five digits, again for readability.) For example, prefix 1010 0000 would be considered smaller than 1010 0000. With padded prefixes that end in 1, do the opposite, so that prefix 1011 1111 would be considered larger than 1011 1111. Do this even though numerically, of course, 1010 0000 is equal to 1010 0000.
13.4.2 Add Prefix Length to 1-padded Prefixes

The padded prefixes are equivalent to the beginning address (the 0-padded address) and ending address (the 1-padded address) of address ranges. The idea is to find the address range in which the destination address resides. We must indicate, when the destination address fits to the right of a 1-padded prefix, how many bits of the prefix should be matched. For example, Figure 13.9 shows the ranges if the prefixes in the database are {}, 1, 10, 100, 101, 1110.

\[
\begin{array}{c}
0000-11111 \\
\hline
10000-11111 \\
\hline
10000-10111 & 11100-11101 \\
\hline
10000-10011 & 10100-10111
\end{array}
\]

Figure 13.9 Address ranges for binary search

This can be thought of as nested parentheses, as shown in Figure 13.10.

\[
0000 \ 10000 \ 10000 \ 10000 \ 10011 \ 10100 \ 10111 \ 10111 \ 11100 \ 11101 \ 11111 \ 11111
\]

( ( ( ( ) ) ) ( ) )

Figure 13.10 Address ranges as nested parentheses

If the destination address we search for fits to the right of a left parenthesis, the prefix represented by the left parenthesis is the longest match. If, however, it fits to the right of a right parenthesis, we must find the matching left parenthesis to indicate the longest match. Consider the destination address 1100. That would fit in the place shown in Figure 13.11.

\[
0000 \ 10000 \ 10000 \ 10000 \ 10011 \ 10100 \ 10111 \ 10111 \ 11100 \ 11101 \ 11111 \ 11111
\]

( ( ( ( ) ) ) ( ) )

1100

Figure 13.11 Fitting 1100 into the ranges

This means that 1100 is outside the range 10000 to 10111. But what would it actually match? The way to discover the matching left parenthesis is to work backward, incrementing for each right parenthesis and decrementing for each left parenthesis until
the count reaches \(-1\). That prefix is the one that matches for destinations that fit after the right parenthesis. So, for example, in the preceding example, we'd count as shown in Figure 13.12.

\[
\begin{array}{cccccccccccc}
00000 & 10000 & 10000 & 10000 & 10011 & 10100 & 10111 & 11100 & 11101 & 11111 & 11111 \\
( & ( & ( & ( & ) & ( & ) & ) & ) & \\
-1 & 0 & 1 & 2 & 1 & 2 & 1 & \\
1 & 1100
\end{array}
\]

Figure 13.12 Fitting 1100 into the ranges.

Because the prefix 1 padded with 0's is the place where we reach \(-1\), that is the prefix that matches any address falling between 10111 and 11100. So the prefix length for 10111 is listed as 1 bit, meaning that addresses falling in the range just to the right of it match the 1-bit prefix “1.”

### 13.4.3 Get Rid of Duplicate Padded Prefixes

Now get rid of duplicate padded prefixes, such as 10000, 10000, and 10000. When the padded prefixes are numerically the same, take the rightmost padded prefix and delete the ones immediately to the left. For example, keep 10000 and delete 10000 and 10000. This ensures that something that fits after 10000 matches the longest prefix of 100 rather than 10 or 1. Note that 10011 will be included in the sorted prefixes, so the destination address cannot fit immediately after 10000 unless it matches 100 because if it starts with 101 it will be to the right of 10011. Similarly, with 1-padded prefixes we keep the rightmost and delete the ones to the left. That is because it is impossible to be greater than 10111 without also being greater than 10111. We keep the duplicates around in order to make the parenthesis counting work out. However, when the parenthesis counting is completed, the duplicates can be removed so that there are fewer padded addresses to search.

### 13.4.4 K-ary Search

The idea of doing \(k\)-ary search in software was introduced by Lampson, Srinivasan, and Varghese.\(^9\) The idea is to notice that memory accesses are much slower than extra instructions (for example, 50 instructions for one memory access). So if the memory bus allows, say, loading six child pointers simultaneously, then each stage of the branch can

be a 6-ary tree rather than a binary tree. In CPU it is still necessary to do a binary search among the six selected values, but that time is dwarfed by the savings in memory accesses.

Independently, I developed k-ary search designed for hardware. It is easy to build a special-purpose k-ary search engine similar in principle to the scheme in Section 13.3.4. This was inspired by Ravi Sethi’s comment that the worst-case performance for the scheme in Section 13.3.4 occurs when the prefixes get split unequally. With k-ary search, you can ensure equal splits each time. But you are searching twice as many prefixes and taking twice the space as the parallel Trie scheme in Section 13.3.4.

Arrange the condensed padded prefixes for k-ary search by picking the padded prefixes at locations $ml/k, 2ml/k, \ldots, (k-1)ml/k$ and putting them into a data structure for loading into the registers for the first iteration. Also keep a data structure of $k+1$ entries, each with a pointer to a data structure for the next iteration. The $j$th entry indicates the prefixes to be compared if the $j$th register is the largest value still smaller than the destination address (that is, the destination address fits between the value in the $j$th register and the value in the $j+1$st register). The 0th entry indicates the prefixes to be compared if none of the registers is smaller than the destination address (that is, the destination address lies within the first chunk of $1/k$ prefixes).

The first iteration will narrow down the potential prefix matches to $1/k$ of the original. For each of the $k+1$ ranges where the destination can be, after $k$ comparisons, another data structure is prepared with the prefixes at $1/k, 2/k, 3/k \ldots$ of the prefixes in that range. This is continued until there are $k+1$ or fewer prefixes remaining, in which case the next iteration will find the answer.

### 13.4.5 Doing a Lookup

Now assume that our program has been written and it looks very similar to the parallel Trie algorithm of Section 13.3.4. The first stage of an address lookup consists of loading the $k$ registers from the initial location, which contains addresses to compare against for the first iteration.

The destination address is compared against all registers in parallel. The winning register is the largest one that is smaller than the destination address. The output is the register number of the winning register, or 0 if the destination lies to the left of the smallest register. The output is used as an index into pointers for the data to be loaded for the next iteration. The final iteration has narrowed the search to the exact prefix, so the final table is a table of matching prefixes, indexed by register numbers.
Homework

1. Continue collapsing Figure 13.2 as much as possible, as recommended in Section 13.3.1.

2. Assume you're doing prefix expansion to multiples of 4 bits as specified in Section 13.3.2. Which prefixes should you keep when prefix capture occurs? For example, assume packets matching 10* should go to neighbor N1, and packets for 100* to neighbor N2, and packets for 1000* to neighbor N3. 1000 will result from expanding 10, and from expanding 100, as well as from the prefix 1000. Which forwarding information should be stored for 1000?

3. Update the structure in Figure 13.1 assuming the following changes to the dictionary. Add prefix 1000. Add prefix 1. Delete prefix 101. Then delete prefix 10100.

4. Consider the IP prefix-matching algorithm of Section 13.3.2. What is the worst-case memory expansion required for searching 4 bits at a time?

5. Consider the IP prefix-matching algorithm of Section 13.3.3. What prefix length would require adding the most marker prefixes at smaller lengths? What prefix length would require adding the fewest marker prefixes?

6. Consider the IP prefix-matching algorithm of Section 13.3.3. What is the worst-case additional memory required?

7. Modify the IP prefix-matching algorithm of Section 13.3.3 by checking only for prefix lengths that actually exist. The basic idea is to include a bitmask with each prefix. This bitmask indicates all longer lengths of prefixes in the database that include that matching prefix as the initial string.