CHAPTER

Prefix-match lookups 11

You can look it up.
—Traditional

Consider a flight database in London that lists flights to a thousand US cities. One alternative would be to keep a record specifying the path to each of the 1000 cities. Suppose, however, that most flights to America hub through Boston, except flights to California, which hub through Los Angeles. This observation can be exploited to reduce the flight database from a thousand entries to two prefix entries (USA* $\rightarrow$ Boston; USA.CA.* $\rightarrow$ LA).

A problem with this reduction is that a destination city like USA.CA.Fresno will now match both the USA* and USA.CA.* prefixes; the database must return the longest match (USA.CA.*). Thus prefixes have been used to compress a large database but at the cost of a more complex *longest-matching-prefix* lookup.

As described in Chapter 2 the Internet uses the same idea. In the year 2022 core routers stored only around 900,000 prefixes instead of potentially billions of entries for each possible Internet address. For example, to a core router, all the computers within a university, such as UCLA (University of California, Los Angeles), will probably be reachable by the same next hop. If all the computers within UCLA are given the same initial set of bits (the network number or prefix), then the router can store *one* entry for UCLA instead of thousands of entries for each computer in UCLA.

The world has changed significantly since the first edition as follows.

- *Prefix Growth:* Since the first edition, the core routing table has grown from around 150,000 in 2004 to around 900,000 prefixes in 2022.
- *IP v6:* When the first edition was written, 128-bit IPv6 was being talked about but did not penetrate significantly because of the prevalence of Network Address Translation (NAT) boxes. However, the popularity of mobile and Internet of Things (IoT) devices have led to rapid deployment of IPv6, with IPv6 availability of Google users at over 30% (Google, 2022).
- *DRAM versus SRAM:* Many router hardware lookups find it cheaper to use a small amount of on-chip memory and large low-latency external DRAM instead of SRAM.
- *Programmable Chips with TCAM:* Chips, such as Intel's Tofino-3 (Intel Corporation, 2022), have emerged that have a fairly large amount of TCAM and are programmable. While IP lookups can be done using TCAM, new data structures can use TCAM to scale to larger databases. Further, they can be programmed to do different IP lookup schemes using a new higher level language for router programming called P4 (P4 Open Source Programming Language, 2022).
- *Software Defined Networks (SDN):* The SDN movement allows the control plane (and with programmable chips, even the data plane) to be changed by a centralized controller. Each router simply

Network Algorithmics. https://doi.org/10.1016/B978-0-12-809927-8.00018-X

allows a set of match-action rules that can be used to implement bridge lookups, MPLS lookups, IP lookups, or other forms of lookup by simply changing the definition of a match.

- *Multicore Processors:* Processors have become faster and multicore. Thus software implementations of IP lookups at Gigabit speeds are feasible today.
- *Network Function Virtualization (NFV):* There is a growing trend among mobile carriers to replace complex middleboxes (that performed network functions such as longest matching prefix, firewalls, parental controls etc.) with flexible software realizations, a trend called Network Function Virtualization (NFV, 2022).

Despite these changes, the underlying algorithmic ideas have remained except for small variations that we will point out including fast software implementations such as DXR (2022) and algorithms optimized for large amounts of DRAM and small on-chip SRAM such as SAIL (Yang et al., 2014).

The entire chapter is organized as follows. Section 11.1 provides an introduction to prefix lookups. Section 11.2 describes attempts to finesse the need for IP lookups. Section 11.3 presents non-algorithmic techniques for lookup based on caching and parallel hardware. Section 11.4 describes the simplest technique based on unibit tries.

The chapter then transitions to describe seven more sophisticated schemes: multibit tries (Section 11.5), level-compressed tries (Section 11.6), Lulea-compressed tries (Section 11.7), Tree bitmap (Section 11.8), binary search on prefix ranges (Section 11.9) (with a modern manifestation called DXR), binary search on prefix lengths (Section 11.11), and linear search on prefix lengths (Section 11.12).

The chapter ends with Section 11.13 on memory allocation issues, Section 11.14 on fixed function lookup chips, and Section 11.15 on programmable chips, and the P4 language to program them. The techniques described in this chapter (and the corresponding principles) are summarized in Table 11.1.

Quick reference guide

The most important lookup algorithms in our opinion for an implementor today are as follows. At speeds up to 100 Gbps in hardware or software using DRAM technology, the simplest and most effective scheme is based on binary search on prefix ranges (DXR) (Section 11.9) and is unencumbered by patents. At faster speeds, especially using more expensive SRAM technology, the most effective algorithm described in this chapter is Tree bitmap (Section 11.8). On the other hand, a simple scheme using small on-chip SRAM and external DRAM is SAIL (Section 11.12). Finally, Section 11.15 describes the P4 language, and potential IP lookup implementations in programmable router chips like Tofino-3 (Intel Corporation, 2022) that leverage both CAM and RAM.

11.1 Introduction to prefix lookups

This section introduces prefix notation, explains why prefix lookup is used, and describes the main metrics used to evaluate prefix lookup schemes.

11.1.1 Prefix notation

Internet prefixes are defined using bits and not alphanumerical characters, of up to 32 bits in length. To confuse matters, however, IP prefixes are often written in dot-decimal notation. For example, at the

Table 11.1 Principles involved in the various prefix-lookup schemes described in this chapter.

Number	Principle	Lookup technique
P2a, **P10**	Precompute indices	Tag switching
P2a, **P10** **P4a**	Pass indices computed at run time Exploit ATM switch hardware	IP switching
P11	Cache whole IP addresses	Lookup caches
P5	Hardware parallel lookup	CAMs
P4b	Expand prefixes to gain speed	Controlled expansion
P13	Strides as a degree of freedom	Variable-stride tries
P4b **P12**, **P2a**	Compress to gain speed Precomputed count of bits set	Lulea tries
P15 **P12** **P2a**	Use efficient search Add marker state Precompute marker watch	Binary search on prefix lengths
P2a	Precompute range to prefix matching	Binary search on prefixes

time of writing UCSD has a 16-bit prefix 132.239. Each of the decimal digits between dots represents a byte. Since in binary 132 is 10000100 and 239 is 11101111, the UCSD prefix in binary can also be written as 1000010011101111*, where the wildcard character * is used to denote that the remaining bits do not matter. UCSD hosts have 32-bit IP addresses beginning with these 16 bits.

Because prefixes can be variable length, a second common way to denote a prefix is by slash notation of the form A/L. In this case A denotes a 32-bit IP address in dot-decimal notation and L denotes the length of the prefix. Thus the UCSD prefix can also be denoted as 132.239.0.0/16, where the length 16 indicates that only the first 16 bits (i.e., 132.239) are relevant. A third common way to describe prefixes is to use a mask in place of an explicit prefix length. Thus the UCSD prefix can also be described as 128.239.0.0 with a mask of 255.255.0.0. Since 255.255.0.0 has 1's in the first 16 bits, this implicitly indicates a length of 16 bits.[1]

Of these three ways to denote a prefix (binary with a wildcard at the end, slash notation, and mask notation), the last two are more compact for writing down large prefixes. However, for pedagogical reasons, it is much easier to use small prefixes as examples and to write them in binary. Thus in this chapter we will use 01110* to denote a prefix that matches all 32-bit IP addresses that start with 01110. The reader should easily be able to convert this notation to the slash or mask notation used by vendors. Also, note that most prefixes are at least 8 bits in length; however, to keep our examples simple, this chapter uses smaller prefixes.

[1] The mask notation is actually more general because it allows noncontiguous masks where the 1's are not necessarily consecutive starting from the left. Such definitions of networks actually do exist. However, they are becoming increasingly uncommon and are nonexistent in core router prefix tables. Thus we will ignore this possibility in this chapter.

11.1.2 Why variable-length prefixes?

Before we consider how to deal with the complexity of variable-length-prefix matching, it is worth understanding why Internet prefixes are variable length. Given a telephone number such as 858-549-3816, it is a trivial matter to extract the first three digits (i.e., 858) as the area code. If fixed-length prefixes are easier to implement, what is the advantage of variable-length prefixes?

The *general* answer to this question is that variable-length prefixes make more efficient use of the address space. This is because areas with a large number of endpoints can be assigned shorter prefixes, while areas with a few endpoints can be assigned longer prefixes.

The *specific* answer comes from the history of Internet addressing. The Internet began with a simple hierarchy in which 32-bit addresses were divided into a network address and a host number; routers only stored entries for networks. For flexible address allocation, the network address came in variable sizes: Class A (8 bits), Class B (16 bits), and Class C (24 bits). To cope with the exhaustion of Class B addresses, the Classless Internet Domain Routing (CIDR) scheme (Rekhter and Li, 1996) assigns new organizations multiple contiguous Class C addresses that can be aggregated by a common prefix. This reduces core router table size.

Today, the potential depletion of the address space has led Internet registries to be very conservative in the assignment of IP addresses. A small organization may be given only a small portion of a Class C address, perhaps a /30, which allows only four IP addresses within the organization. Many organizations are coping with these sparse assignments by sharing a few IP addresses among multiple computers, using schemes such as network address translation, or NAT.

Thus, CIDR and NAT have helped the Internet handle exponential growth with a finite 32-bit address space. While adoption of IP with a 128-bit address (IPv6) is rapidly increasing (Google, 2022), the effectiveness of NAT in the short run and the complexity of rolling out a new protocol have made 32-bit IP addresses still dominant at the time of writing.

The bottom line is that the decision to deploy CIDR helped save the Internet, but it has introduced the complexity of longest-matching prefix lookup.

11.1.3 Lookup model

Recall the router model of Chapter 2. A packet arrives on an input link. Each packet carries a 32-bit Internet (IP) address.[2]

The processor consults a forwarding table to determine the output link for the packet. The forwarding table contains a set of *prefixes* with their corresponding output links. The packet is matched to the longest prefix that matches the destination address in the packet, and the packet is forwarded to the corresponding output link. The task of determining the output link, called *address lookup*, is the subject of this chapter, which surveys lookup algorithms and shows that lookup can be implemented at gigabit and terabit speeds.

Before searching for IP lookup solutions, it is important to be familiar with some basic observations about traffic distributions, memory trends, and database sizes, which are shown in Table 11.2. These in turn will motivate the requirements for a lookup scheme.

[2] While most users deal with domain names, recall again that these names are translated to IP addresses by a directory service called DNS before packets are sent.

Table 11.2 Some current data about the lookup problem and the corresponding implications for lookup solutions.

Observation	Inference
1. 250,000 concurrent flows in backbone	Caching works poorly in backbone routers
2. 50% are TCP acks	Wire speed lookup needed for 40-byte packets
3. Lookup dominated by memory accesses	Lookup speed measured by number of memory accesses
4. Prefix lengths from 8 to 32	Naive schemes take 24 memory accesses
5. 1 million prefixes today and multicast and host routes	With growth, require 500,000–1 million prefixes
6. Unstable BGP, multicast	Updates in milliseconds to seconds
7. Higher speeds need SRAM	Worth minimizing memory
8. IPv6, multicast delays	Both 32-bit and 128-bit lookups crucial today

First, a study of backbone traffic (Thompson et al., 1997) as far back as 1997 showed around 250,000 concurrent flows of short duration, using a fairly conservative measurement of flows. Measurement data shows that this number is only increasing to easily over a million concurrent TCP flows. This large number of flows means caching solutions do not work well.

Second, the same study (Thompson et al., 1997) showed that roughly half the packets received by a router are minimum-size TCP acknowledgments. Thus it is possible for a router to receive a stream of minimum-size packets. Hence, being able to prefix lookups in the time to forward a minimum-size packet can finesse the need for an input link queue, which simplifies system design. A second reason is simple marketing: Many vendors claim wire speed forwarding, and these claims can be tested. Assuming wire speed forwarding, forwarding a 40-byte packet should take no more than 32 nanoseconds at 10 Gbps (OC-192 speeds), and 8 nanoseconds at 40 Gbps (OC-768).

Clearly, the most crucial metric for a lookup scheme is lookup speed. The third observation states that because the cost of computation today is dominated by memory accesses, the simplest measure of lookup speed is the worst-case number of memory accesses. The fourth observation shows that backbone databases have all prefix lengths from 8 to 32, and so naive schemes will require 24 memory accesses in the worst case to try all possible prefix lengths.

The fifth observation states that while current databases are around 900,000 prefixes, the possible use of host routes (full 32-bit addresses) and multicast routes means that future backbone routers will have prefix databases of over 1 million prefixes.

The sixth observation refers to the speed of updates to the lookup data structure, for example, to add or delete a prefix. Unstable routing-protocol implementations can lead to requirements for updates on the order of milliseconds. Note that whether seconds or milliseconds, this is several orders of magnitude below the lookup requirements, allowing implementations the luxury of precomputing (**P2a**) information in data structures to speed up lookup, at the cost of longer update times.

The seventh observation comes from Chapter 2. While standard (DRAM) memory is cheap, DRAM access times are currently around 20–40 nanoseconds, and so higher-speed memory (e.g., off- or on-chip SRAM, 1–5 nanoseconds) may be needed at higher speeds. While DRAM memory is essentially unlimited, SRAM and on-chip memory are limited by expense or unavailability. Thus a third metric is memory usage, where memory can be expensive fast memory (cache in software, SRAM in hardware) as well as cheaper, slow memory (e.g., DRAM, SDRAM).

Note that a lookup scheme that does not do incremental updates will require two copies of the lookup database so that search can proceed in one copy while lookups proceed on the other copy. Thus it may be worth doing incremental updates simply to reduce high-speed memory by a factor of 2!

The eighth observation concerns prefix lengths. IPv6 requires 128-bit prefixes. Multicast lookups require 64-bit lookups because the full group address and a source address can be concatenated to make a 64-bit prefix. However, the full deployment of both IPv6 and multicast is still proceeding. Thus at the time of writing, it is important to be able to support both 32-bit and 128-bit IP lookups.

In summary, the interesting metrics, in order of importance, are lookup speed, memory, and update time. As a concrete example, a good on-chip design using 16 Mbits of on-chip memory may support any set of 1,000,000 prefixes, do a lookup in 8 nanoseconds to provide wire speed forwarding at terabit speeds, and allow prefix updates in 1 millisecond.

The following notations is used consistently in reporting the theoretical performance of IP lookup algorithms. N denotes the number of prefixes (e.g., 900,000 for large databases in 2022), and W denotes the length of an address (e.g., 32 for IPv4).

Finally, two additional observations can be exploited to optimize the expected case.

O1: Almost all prefixes are 24 bits or less, with the majority being 24-bit prefixes and the next largest spike being at 16 bits. Some vendors use this to show worst-case lookup times only for 24-bit prefixes; however, the future may lead to databases with a large number of host routes (32-bit addresses) and integration of ARP caches.

O2: It is fairly rare to have prefixes that are prefixes of other prefixes, such as the prefixes 00* and 0001*. In fact, the maximum number of prefixes of a given prefix in current databases is seven.

While the ideal is a scheme that meets worst-case lookup time requirements, it is desirable to have schemes that also utilize these observations to improve average storage performance.

11.2 Finessing lookups

The first instinct for a systems person is not to solve complex problems (like longest matching prefix) but to *eliminate* the problem.

Observe that in virtual circuit networks such as ATM, when a source wishes to send data to a destination, a call, analogous to a telephone call, is set up. The call number virtual circuit index [VCI] at each router is a moderate-size integer that is easy to look up. However, this comes at the cost of a round-trip delay for call setup before data can be sent.

In terms of our principles, ATM has a previous hop switch pass an index (**P10**, pass hints in protocol headers) into a next hop switch. The index is precomputed (**P2a**) just before data is sent by the previous hop switch (**P3c**, shifting computation in space). The same abstract idea can be used in datagram

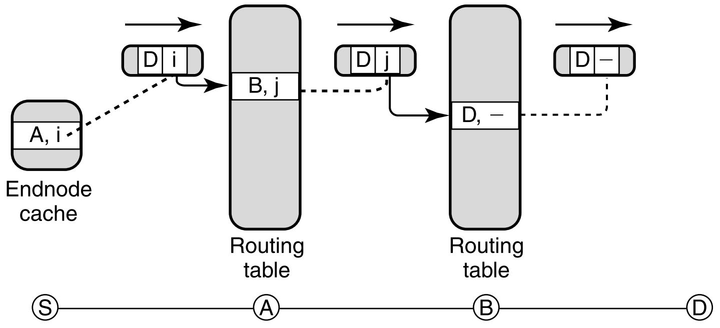


FIGURE 11.1

Replacing the need for a destination lookup in a datagram router by having each router pass an index into the next router's forwarding table.

networks such as the Internet to finesse the need for prefix lookups. We now describe two instantiations of this abstract idea: tag switching (Section 11.2.1) and flow switching (Section 11.2.2).

11.2.1 Threaded indices and tag switching

In threaded indices (Chandranmenon and Varghese, 1996), each router passes an index into the next router's forwarding table, thereby avoiding prefix lookups. The indexes are precomputed *by the routing protocol* whenever the topology changes. Thus in Fig. 11.1 source S sends a packet to destination D to the first router A as usual; however, the packet header also contains an index i into A's forwarding table. A's entry for D says that the next hop is router B and that B stores its forwarding entry for D at index j. Thus A sends the packet on to B, but first it writes j (Fig. 11.1) as the packet index. This process is repeated, with each router in the path using the index in the packet to look up its forwarding table.

The two main differences between threaded indices and VCIs are as follows. First, threaded indexes are per *destination* and not per active *source–destination pair* as in virtual circuit networks such as ATM. Second, and more importantly, threaded indexes are precomputed *by the routing protocol whenever the topology changes*. As a simple example, consider Fig. 11.2, which shows a sample router topology where the routers run the Bellman–Ford protocol to find their distances to destinations.

In Bellman–Ford (used, for example, in the intradomain protocol Routing Information Protocol [RIP] (Perlman, 1992)), a router R calculates its shortest path to D by taking the minimum of the cost to D through each neighbor. The cost through a neighbor such as A is A's cost to D (i.e., 5) plus the cost from R to A (i.e., 3). In Fig. 11.2 the best-cost path from R to D is through router B, with cost 7. R can compute this because each neighbor of R (e.g., A, B) passes its current cost to D to R, as shown in the figure. To compute indices as well, we modify the basic protocol so that *each neighbor reports its index for a destination in addition to its cost to the destination*. Thus in Fig. 11.2 A passes i and B passes j; thus when R chooses B, it also uses B's index j in its routing table entry for D. In summary, each router uses the index of the minimal-cost neighbor for each destination as the threaded index for that destination.

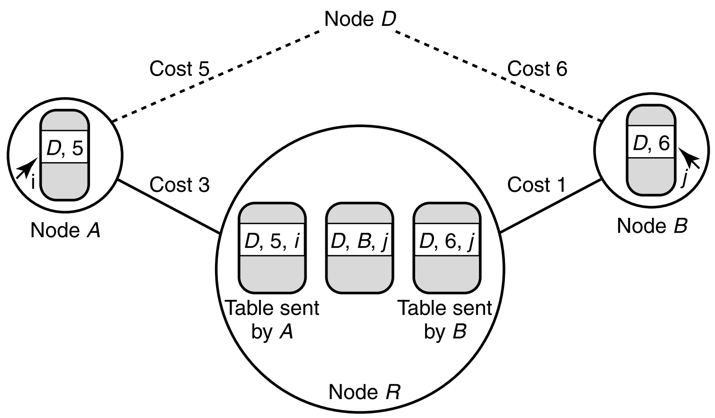


FIGURE 11.2

Setting up the threaded indexes or tags by modifying Bellman–Ford routing.

Cisco later introduced tag switching (Meiners et al., 2008), which is similar in concept to threaded indices, except tag switching also allows a router to pass a stack of tags (indices) for multiple routers downstream. Both schemes, however, do not deal well with hierarchies. Consider a packet that arrives from the backbone to the first router in the exit domain. The exit domain is the last autonomously managed network the packet traverses—say, the enterprise network in which the destination of the packet resides.

The only way to avoid a lookup at the first router, R, in the exit domain is to have some earlier router outside the exit domain pass an index (for the destination subnet) to R. But this is impossible because the prior backbone routers should have only one aggregated routing entry for the entire destination domain and can thus pass only one index for all subnets in that domain. The only solution is either to add extra entries to routers outside a domain (infeasible) or to require ordinary IP lookup at domain entry points (the chosen solution). Today tag switching is flourishing in a more general form called *multiprotocol label switching* (MPLS) (Meiners et al., 2008). However, neither tag switching nor MPLS completely avoids the need for ordinary IP lookups.

11.2.2 Flow switching

A second proposal to finesse lookups was called *flow switching* (Newman et al., 1997; Parulkar et al., 1995). Flow switching also relies on a previous hop router to pass an index into the next hop router. Unlike tag switching, however, these indexes are computed on demand when data arrives, and they are then cached.

Flow switching starts with routers that contain an internal ATM switch and (potentially slow) processors capable of doing IP forwarding and routing. Two such routers R1 and R2 are shown in Fig. 11.3. When R2 first sends an IP packet to destination D that arrives on the left input port of R1, the input port sends the packet to its central processor. This is the slow path. The processor does the IP lookup and switches the packet internally to output link L. So far nothing is out of the ordinary.

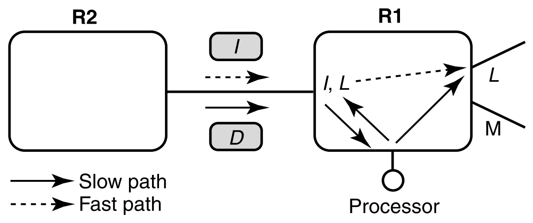


FIGURE 11.3

In IP switching, if R1 wishes to switch packets sent to D that are destined for output link L, R1 picks an idle virtual circuit I, places the mapping I, L in its input port, and then sends I back to R2. If R2 now sends packets to D labeled with VCI I, the packet will get switched directly to the output link without going through the processor.

Life gets more exciting if R1 decides to "switch" packets going to D. R1 may decide to do so if, for instance, there is a lot of traffic going to D. In that case R1 first picks an idle virtual circuit identifier I, places the mapping $I \rightarrow L$ in its input port hardware, and then sends I back to R2. If R2 now sends packets to D labeled with VCI I to the input port of R1, the input port looks up the mapping from I to L and switches the packet directly to the output link L without going through the processor.

Of course, R2 can repeat this switching process with the preceding router in the path, and so on. Eventually, IP forwarding can be completely dispensed with in the switched portion of a sequence of flow-switching routers.

Despite its elegance, flow switching seems likely to work poorly in the backbone. This is because backbone flows are short lived and exhibit poor locality. A contrarian opinion is presented in Molinero-Fernandez and McKeown (2002) where the authors argue for the resurrection of flow switching based on TCP connections. They claim that the current use of circuit-switched optical switches to link core routers, the underutilization of backbone links running at 10% of capacity, and increasing optical bandwidths all favor the simplicity of circuit switching at higher speeds.

Both IP and tag switching are techniques to finesse the need for IP lookups by passing information in protocol headers. Like ATM, both schemes rely on passing indices (**P10**). However, tag switching precomputes the index (**P2a**) at an earlier time scale (topology change time) than ATM (just before data transfer). On the other hand, in IP switching the indices are computed on demand (**P2c**, lazy evaluation) after the data begins to flow. However, neither tag nor IP switching completely avoids prefix lookups, and each adds a complex protocol. We now look afresh at the supposed complexity of IP lookups.

11.2.3 Status of tag switching, flow switching, and multiprotocol label switching

While tag switching and IP switching were originally introduced to speed up lookups, IP switching has died away. However, tag switching in the more general form of multi-protocol-label switchings (MPLS) (IETF MPLS Charter, 1997) has reinvented itself as a mechanism for providing flow differentiation to provide quality of service. Just as a VCI provides a simple label to quickly distinguish a flow, a label allows a router to easily isolate a flow for special service. In effect, MPLS uses labels to finesse the need for packet classification (Chapter 12), a much harder problem than prefix lookups. Thus although prefix matching is still required, MPLS is also de rigueur for a core router today.

Briefly, the required MPLS fast path forwarding is as follows. A packet with an MPLS header is identified, a 20-bit label is extracted, and the label is looked up in a table that maps the label to a forwarding rule. The forwarding rule specifies a next hop and also specifies the operations to be performed on the current set of labels in the MPLS packet. These operations can include removing labels ("popping the label stack") or adding labels ("pushing on to the label stack").

Router MPLS implementations have to impose some limits on this general process to guarantee wire speed forwarding. Possible limits include requiring that the label space be dense, supporting a smaller number of labels than 2^{20} (this allows a smaller amount of lookup memory while avoiding a hash table), and limiting the number of label-stacking operations that can be performed on a single packet.

11.3 Non-algorithmic techniques for prefix matching

In this section we consider two other systems techniques for prefix lookups that do not rely on algorithmic methods: caching and ternary CAMs. Caching relies on locality in address references, while CAMs rely on hardware parallelism.

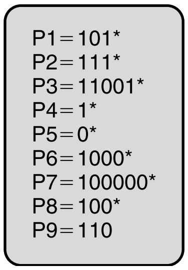
11.3.1 Caching

Lookups can be sped up by using a cache (**P11a**) that maps 32-bit addresses to next hops. However, cache hit ratios in the backbone are poor (Newman et al., 1997) because of the lack of locality exhibited by flows in the backbone. The use of a large cache still requires the use of an exact-match algorithm for lookup. Some researchers have advocated a clever modification of a CPU cache lookup algorithm for this purpose (Chiueh and Pradhan, 1999). In summary, caching can help, but it does not avoid the need for fast prefix lookups.

11.3.2 Ternary content-addressable memories

Ternary content-addressable memories (CAMs) that allow "don't care" bits provide parallel search in one memory access. Today's CAMs can search and update in one memory cycle (e.g., 10 nanoseconds) and handle any combination of 100,000 prefixes. They can even be cascaded to form larger databases. CAMs, however, have the following issues.

- *Density Scaling:* One bit in a TCAM requires 10–12 transistors, while an SRAM requires 4–6 transistors. Thus TCAMs will also be less dense than SRAMs or take more area. Board area is a critical issue for many routers.
- *Power Scaling:* TCAMs take more power because of the parallel compare. CAM vendors are, however, chipping away at this issue by finding ways to turn off parts of the CAM to reduce power. Power is a key issue in large core routers.
- *Match Arbitration:* The match logic in a CAM requires all matching rules to arbitrate so that the highest match wins. Older-generation CAMs took around 10 nanoseconds for an operation, but this is no longer as much of an issue for modern TCAMs.
- *Extra Chips:* Given that many routers, such as the Cisco GSR and the Juniper M160, already have a dedicated Application Specific Integrated Circuit (ASIC) (or network processor) doing packet



P1=101*
P2=111*
P3=11001*
P4=1*
P5=0*
P6=1000*
P7=100000*
P8=100*
P9=110

FIGURE 11.4

Sample prefix database used for the rest of this chapter. Note that the next hops corresponding to each prefix have been omitted for clarity.

forwarding, it is tempting to integrate the classification algorithm with the lookup without adding CAM interfaces and CAM chips. Note that CAMs typically require a bridge ASIC in addition to the basic CAM chip and sometimes require multiple CAM chips.

- *Programmable Chips with built-in TCAM:* By contrast to the problem of extra chips cited by the last bullet (that is caused by separate CAM and packet forwarding chips), a game change in recent years has been the emergence of programmable chips capable of performing forwarding with TCAM built in. For example, Intel's Tofino-3 (Intel Corporation, 2022) contains 384 TCAM blocks of size 44×512 each, enough to support a large enterprise or data center but not enough for the backbone without some algorithmic tricks.

In summary, CAM technology is rapidly improving and is supplanting algorithmic methods in smaller routers. However, for larger core routers that may wish to have databases of a million routes in the future, it may be better to have solutions (as we describe in this chapter) that scale with standard memory technologies such as SRAM. SRAM is likely always to be cheaper, faster, and denser than CAMs. While it is clearly too early to predict the outcome of this war between algorithmic and TCAM methods, even semiconductor manufacturers have hedged their bets and provide *both* algorithmic and CAM-based solutions.

11.4 Unibit tries

It is helpful to start a survey of algorithmic techniques (**P15**) for prefix lookup with the simplest technique: a *unibit trie*. Consider the sample prefix database of Fig. 11.4. This database will be used to illustrate many of the algorithmic solutions in this chapter. It contains nine prefixes, called P1 to P9, with the bit strings shown in the figure.

In practice there is a next hop associated with each prefix omitted from the figure. To avoid clutter, prefix names are used to denote the next hops. Thus in the figure, an address D that starts with 1 followed by a string of 31 zeroes will match P4, P6, P7, and P8. The longest match is P7.

Fig. 11.5 shows a unibit trie for the sample database of Fig. 11.4. A unibit trie is based on the simple algorithmic technique (**P15**) of divide and conquer based on the bits in the destination address, starting

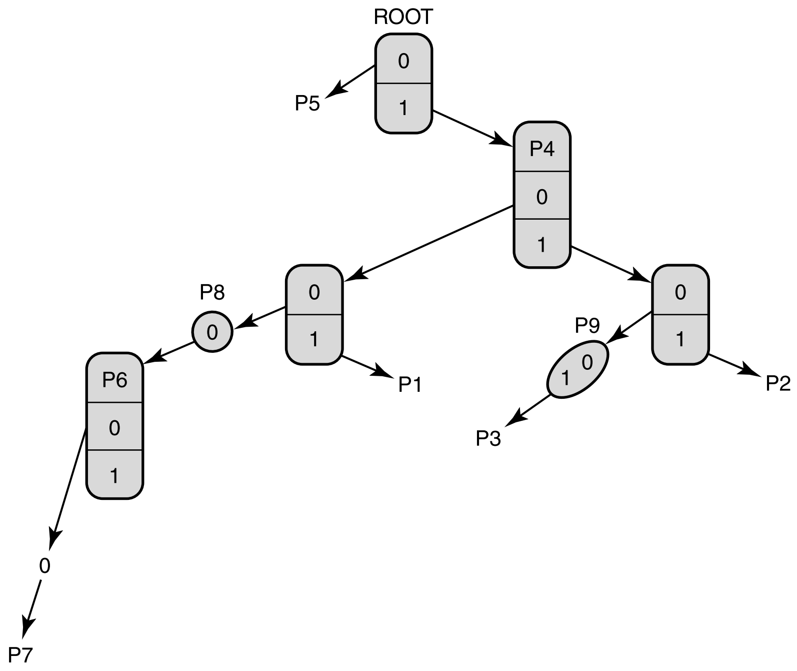


FIGURE 11.5

The one-bit trie for the sample database of Fig. 11.4.

with the most significant. A unibit trie is a tree in which each node is an array containing a 0-pointer and a 1-pointer. At the root all prefixes that start with 0 are stored in the subtrie pointed to by the 0-pointer and all prefixes that start with a 1 are stored in the subtrie pointed to by the 1-pointer.

Each subtrie is then constructed recursively in a similar fashion using the remaining bits of the prefixes allocated to the subtrie. For example, in Fig. 11.5 notice that P1 = 101 is stored in a path traced by following a 1-pointer at the root, a 0-pointer at the right child of the root, and a 1-pointer at the next node in the path.

There are two other fine points to note. In some cases, a prefix may be a substring of another prefix. For example, P4 = 1* is a substring of P2 = 111*. In that case, the smaller string, P4, is stored inside a trie node on the path to the longer string. For example, P4 is stored at the right child to the root; note that the path to this right child is the string 1, which is the same as P4.

Finally, in the case of a prefix such as P3 = 11001, after we follow the first three bits, we might naively expect to find a string of nodes corresponding to the last two bits. However, since no other prefixes share more than the first 3 bits with P3, these nodes would only contain one pointer apiece. Such a string of trie nodes with only one pointer each is called a one-way branch.

Clearly one-way branches can greatly increase wasted storage by using whole nodes (containing at least two pointers) when only a single bit suffices. (The exercises will help you quantify the amount of wasted storage.) A simple technique to remove this obvious waste (**P1**) is to compress the one-way branches.

In Fig. 11.5 this is done by using a text string (i.e. "01") to represent the pointers that would have been followed in the one-way branch. Thus in Fig. 11.5 two trie nodes (containing two pointers apiece) in the path to P3 have been replaced by a single text string of 2 bits. Clearly, no information has been lost by this transformation. (As an exercise, determine if there is another path in the trie that can similarly be compressed.)

To search for the longest matching prefix of a destination address D, the bits of D are used to trace a path through the trie. The path starts with the root and continues until search fails by ending at an empty pointer or at a text string that does not completely match. While following the path, the algorithm keeps track of the last prefix encountered at a node in the path. When search fails, this is the longest matching prefix that is returned.

For example, if D begins with 1110, the algorithm starts by following the 1-pointer at the root to arrive at the node containing P4. The algorithm remembers P4 and uses the next bit of D (a 1) to follow the 1-pointer to the next node. At this node, the algorithm follows the 1-pointer to arrive at P2. When the algorithm arrives at P2, it overwrites the previously stored value (P4) by the newer prefix found (P2). At this point search terminates because P2 has no outgoing pointers.

On the other hand, consider doing a search for a destination D' whose first 5 bits are 11000. Once again, the first 1 bit is used to reach the node containing P4. P4 is remembered as the last prefix encountered, and the 1 pointer is followed to reach the rightmost node at height 2.

The algorithm now follows the third bit in D' (a 0) to the text string node containing "01." Thus we remember P9 as the last prefix encountered. The fourth bit of D' is a 0, which matches the first bit of "01." However, the fifth bit of D' is a 0 (and not a 1 as in the second bit of "01"). Thus the search terminates with P9 as the longest matching prefix.

The literature on tries (Knuth, 1973) does not use *text strings* to compress one-way branches as in Fig. 11.5. Instead, the classical scheme, called a *Patricia trie*, uses a *skip count*. This count records the number of bits in the corresponding text string, not the bits themselves. For example, the text string node "01" in our example would be replaced with the skip count "2" in a Patricia trie.

This works fine as long as the Patricia trie is used for *exact* matches, which is what they were used for originally. When search reaches a skip count node, it skips the appropriate number of bits and follows the pointer of the skip count node to continue the search. Since bits that are skipped are not compared for a match, Patricia requires that a complete comparison between the searched key and the entry found by Patricia be done at the end of the search.

Unfortunately, this works very badly with prefix matching, an application that Patricia tries were *not* designed to handle in the first place. For example, in searching for D', whose first 5 bits are 11000 in the Patricia equivalent of Fig. 11.5, search would skip the last two bits and get to P3. At this point the comparison will find that P3 does not match D'.

When this happens, a search in a Patricia trie has to backtrack and go back up the trie searching for a possible shorter match. In this example, it may appear that search could have remembered P4. But if P4 was also encountered on a path that contains skip count nodes, the algorithm cannot even be sure of P4. Thus it must backtrack to check if P4 is correct.

Unfortunately, the BSD implementation of IP forwarding (Wright and Stevens, 1995) decided to use Patricia tries as a basis for best matching prefix. Thus the BSD implementation used skip counts; the implementation also stored prefixes by padding them with zeroes. Prefixes were also stored at the leaves of the trie, instead of within nodes, as shown in Fig. 11.5. The result is that prefix matching can, in the worst case, result in backtracking up the trie for a worst case of 64 memory accesses (32 down the tree and 32 up).

Given the simple alternative of using text strings to avoid backtracking, doing skip counts is a bad idea. In essence, this is because the skip count transformation does *not* preserve information, while the text string transformation does. However, because of the enormous influence of BSD, a number of vendors and even other algorithms (e.g., Ref. Nilsson and Karlsson, 1998) have used skip counts in their implementations.

11.5 Multibit tries

Most large memories use DRAM. DRAM has a large latency (say 30 nanoseconds) when compared to register access times (2 nanoseconds). Since a unibit trie may have to make 32 accesses for a 32-bit prefix, the worst-case search time of a unibit trie is at least 32 * 30 = 0.96 microseconds. This clearly motivates *multibit* trie search. The number of bits one can search at a time is a degree of freedom algorithmic techniques (**P13**) we can exploit.

To search a trie in *strides* of say 4 bits, the main problem is dealing with prefixes like 10101* (length 5), whose lengths are not a multiple of the chosen stride length, 4. If we search 4 bits at a time, how can we ensure that we do not miss prefixes like 10101*? *Controlled prefix expansion* solves this problem by transforming an existing prefix database into a new database with *fewer prefix lengths* but with potentially *more prefixes*. By eliminating all lengths that are not multiples of the chosen stride length, expansion allows faster multibit trie search, at the cost of increased database size.

For example, removing odd prefix lengths reduces the number of prefix lengths from 32 to 16 and would allow trie search 2 bits at a time. To remove a prefix like 101* of length 3, observe that 101* represents addresses that begin with 101, which in turn represents addresses that begin with 1010* or 1011*. Thus 101* (of length 3) can be replaced by two prefixes of length 4 (1010* and 1011*), both of which inherit the next hop forwarding entries of 101*.

However, the expanded prefixes may collide with an existing prefix at the new length. In that case, the expanded prefix is removed. The existing prefix is given priority because it was originally of longer length.

In essence, expansion trades memory for time (**P4b**). The same idea can be used to remove any chosen set of lengths except length 32. Since trie search speed depends linearly on the number of lengths, expansion reduces search time.

Consider the sample prefix database shown in Fig. 11.4, which has nine prefixes, P1 to P9. The same database is repeated on the *left* of Fig. 11.6. The database on the *right* of Fig. 11.6 is an equivalent database, constructed by expanding the original database to contain prefixes of lengths 3 and 6 only. Notice that of the four expansions of P6 = 1000* to 6 bits, one collides with P7 = 100000* and is thus removed.

11.5.1 Fixed-stride tries

Fig. 11.7 shows a trie for the same database as Fig. 11.6, using expanded tries with a fixed stride length of 3. Thus each trie node uses 3 bits. The replicated entries within trie nodes in Fig. 11.7 correspond

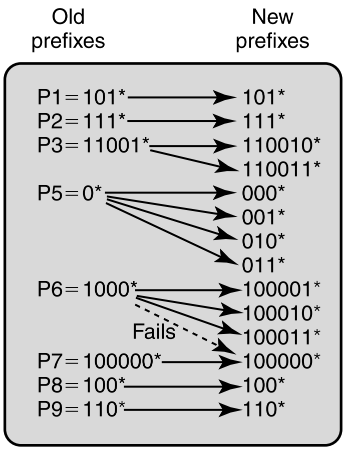


FIGURE 11.6

Controlled expansion of the original prefix database shown on the left (which has five prefix lengths, 1, 3, 4, 5, and 6) to an expanded database (which has only 2 prefix lengths, 3 and 6).

exactly to the expanded prefixes on the right of Fig. 11.6. For example, P6 in Fig. 11.6 has three expansions (100001, 100010, 100011).

These three expanded prefixes are pointed to by the 100 pointer in the root node of Fig. 11.7 (because all three expanded prefixes start with 100) and are stored in the 001, 010, and 011 entries of the right child of the root node. Notice also that the entry 100 in the root node has a stored prefix P8 (besides the pointer pointing to P6's expansions), because P8 = 100* is itself an expanded prefix.

Thus each trie node element is a record containing *two* entries: a stored prefix and a pointer. Trie search proceeds 3 bits at a time. Each time a pointer is followed, the algorithm remembers the stored prefix (if any). When search terminates at an empty pointer, the last stored prefix in the path is returned.

For example, if address D begins with 1110, search for D starts at the 111 entry at the root node, which has no outgoing pointer but a stored prefix (P2). Thus search for D terminates with P2. A search for an address that starts with 100000 follows the 100 pointer in the root (and remembers P8). This leads to the node on the lower right, where the 000 entry has no outgoing pointer but a stored prefix (P7). The search terminates with result P7. Both the pointer and stored prefix can be retrieved in one memory access using wide memories (**P5b**).

A special case of fixed-stride tries, described in Gupta et al. (1998), uses fixed strides of 24, 4, and 4. The authors observe that DRAMs with more than 2^{24} locations are becoming available, making even 24-bit strides feasible.

11.5.2 Variable-stride tries

In Fig. 11.7 the leftmost leaf node needs to store the expansions of P3 = 11001*, while the rightmost leaf node needs to store P6 (1000*) and P7 (100000*). Thus because of P7 the rightmost leaf node needs to examine 3 bits. However, there is no reason for the leftmost leaf node to examine more than

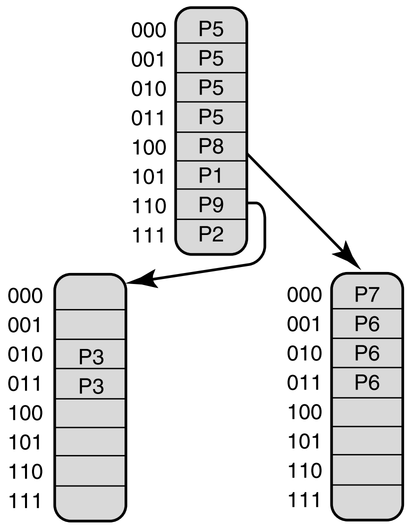


FIGURE 11.7

Expanded trie (which has two strides of 3 bits each) corresponding to the prefix database of Fig. 11.6.

2 bits because P3 contains only 5 bits, and the root stride is 3 bits. There is an extra degree of freedom that can be optimized (**P13**).

In a *variable-stride* trie, the number of bits examined by each trie node can vary, even for nodes at the same level. To do so, the stride of a trie node is encoded with the pointer to the node. Fig. 11.7 can be transformed into a variable-stride trie (Fig. 11.8) by replacing the leftmost node with a four-element array and encoding length 2 with the pointer to the leftmost node. The stride encoding costs 5 bits. However, the variable stride trie of Fig. 11.8 has four fewer array entries than the trie of Fig. 11.7.

Our example motivates the problem of picking strides to minimize the total amount of trie memory. Since expansion trades memory for time, why not minimize the memory needed by optimizing a degree of freedom (**P13**), the strides used at each node? To pick the variable strides, the designer first specifies the worst-case number of memory accesses. For example, with 40-byte packets at 1-Gbps and 80-nanosecond DRAM, we have a time budget of 320 nanoseconds, which allows only four memory accesses. This constrains the maximum number of nodes in any search path (four in our example).

Given this fixed height, the strides can be chosen *to minimize storage.* This can be done using dynamic programming (Srinivasan and Varghese, 1999) in a few seconds, even for large databases of 150,000 prefixes. A degree of freedom (the strides) is optimized to minimize the memory used for a given worst-case tree height.

A trie is said to be optimal for height h and a database D if the trie has the smallest storage among all variable-stride tries for database D, whose height is no more than h. It is easy to prove (see exercises) that the trie of Fig. 11.8 is optimal for the database on the left of Fig. 11.6 and height 2.

The general problem of picking an optimal stride trie can be solved recursively (Fig. 11.9). Assume the tree height must be h. The algorithm first picks a root with stride s. The $y = 2^s$ possible pointers in

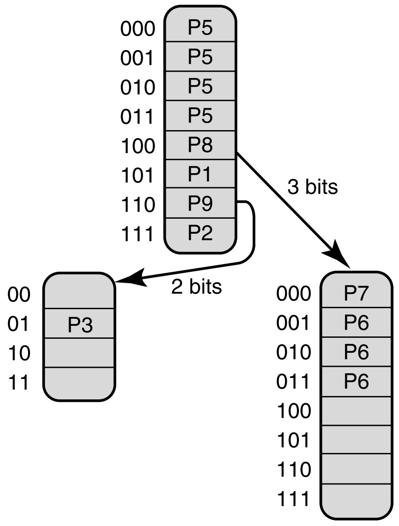


FIGURE 11.8

Transforming the fixed-stride trie of Fig. 11.7 into a variable-stride trie by encoding the stride of each trie node along with a pointer to the node. Notice that the leftmost leaf node now only contains four locations (instead of eight), thus reducing the number of locations from 24 to 20.

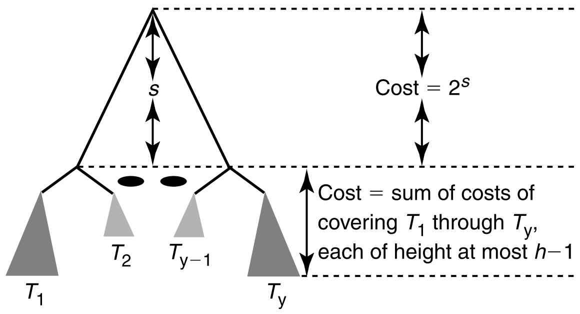

s
Cost = 2^s
Cost = sum of costs of covering T1 through Ty, each of height at most h−1
T1
T2
Ty−1
Ty

FIGURE 11.9

Picking an optimum variable-stride trie via dynamic programming.

the root can lead to y nonempty subtries $T_1, \dots T_y$. If the s-bit pointer p_i leads to subtrie T_i, then all prefixes in the original database D that start with p_i must be stored in T_i. Call this set of prefixes D_i.

Suppose we could recursively find the optimal T_i for height $h-1$ and database D_i. Having used up one memory access at the root node, there are only $h-1$ memory accesses left to navigate each subtrie T_i. Let C_i denote the storage cost required, counted in array locations, for the optimal T_i. Then for a fixed root stride s, the cost of the resulting optimal trie $C(s)$ is 2^s (cost of root node in array locations) plus $\sum_{i=1}^{y} C_i$. Thus the optimal value of the initial stride is the value of s, where $1 \leq s \leq 32$, that minimizes $C(s)$.

A naive use of recursion leads to repeated subproblems. To avoid repeated subproblems, the algorithm first constructs an auxiliary 1-bit trie. Notice that any subtrie T_i in Fig. 11.9 must be a subtrie N of the 1-bit trie. Then the algorithm uses dynamic programming to construct the optimal cost and trie strides for each subtrie N in the original 1-bit trie for all values of height from 1 to h, building bottom-up from the smallest-height subtries to the largest-height subtries. The final result is the optimal strides for the root (of the 1-bit subtrie) with height h. Details are described in Srinivasan and Varghese (1999).

The final complexity of the algorithm is easily seen to be $O(N * W^2 * h)$, where N is the number of original prefixes in the original database, W is the width of the destination address, and h is the desired worst-case height. This is because there are $N * W$ subtries in the 1-bit trie, each of which must be solved for heights that range from 1 to h, and each solution requires a minimization across at most W possible choices for the initial stride s. Note that the complexity is linear in N (the largest number, around 150,000 at the time of writing) and h (which should be small, at most 8), but quadratic in the address width (currently 32). In practice, the quadratic dependence on address width is not a major factor.

For example, Srinivasan and Varghese (1999) show that using a height of 4, the optimized MAE-East database required 423 KB of storage, compared to 2003 KB for the unoptimized version. The unoptimized version uses the “natural” stride lengths 8, 8, 8, 8. The dynamic program took 1.6 seconds to run on a 300-MHz Pentium Pro. The dynamic program is even simpler for fixed-stride tries and takes only 1 milliseconds to run. However, the use of fixed strides requires 737 KB instead of 423 KB.

Clearly, 1.6 seconds are much too long to let the dynamic program be run for every update and still allow millisecond updates (Labovitz et al., 1997). However, backbone instabilities are caused by pathologies in which the same set of prefixes S are repeatedly inserted and deleted by a router that is temporarily swamped (Labovitz et al., 1997). Since we had to allocate memory for the full set, including S, anyway, the fact that the trie is suboptimal in its use of memory when S is deleted is irrelevant. On the other hand, the rate at which new prefixes get added or deleted by managers seems more likely to be on the order of days. Thus a dynamic program that takes several seconds to run every day seems reasonable and will not unduly affect worst-case insertion and deletion times while still allowing reasonably optimal tries.

11.5.3 Incremental update

Simple insertion and deletion algorithms exist for multibit tries. Consider the addition of a prefix P. The algorithm first simulates search on the string of bits in the new prefix P up to and including the last complete stride in prefix P. Search will terminate either by ending with the last (possibly incomplete) stride or by reaching a nil pointer. Thus for adding P10 = 1100* to the database of Fig. 11.7, search follows the 110-pointer and terminates at the leftmost leaf trie node X.

For the purposes of insertion and deletion, for each node X in the multibit trie, the algorithm maintains a corresponding 1-bit trie, with the prefixes stored in X. This auxiliary structure need not be in fast memory. Also, for each node array element, the algorithm stores the length of its present best match. After determining that P10 must be added to node X, the algorithm expands P10 to the stride of X. Any array element to which P10 expands (which is currently labeled with a prefix of a length smaller than P10) must be overwritten with P10.

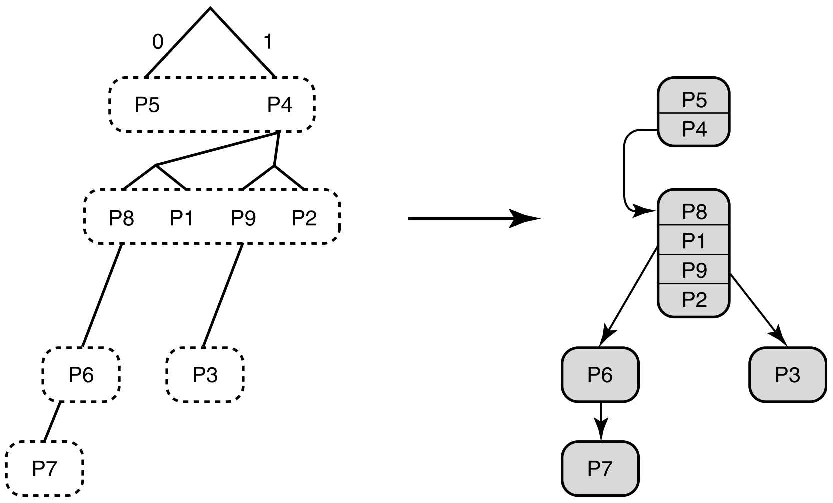


FIGURE 11.10

The level-compressed (LC) trie scheme decomposes the 1-bit trie recursively into full subtries of the largest size possible (*left*). The children in each full subtrie (shown by the dotted boxes) are then placed in a trie node to form a variable-stride trie that is specific to the database chosen.

Thus in adding P10 = 1100*, the algorithm must add the expansions of 0* into node X. In particular, the 000 and 001 entries in node X must be updated to be P10.

If the search ends before reaching the last stride in the prefix, the algorithm creates new trie nodes. For example, if the prefix P11 = 1100111* is added, search fails at node X when a nil pointer is found at the 011 entry. The algorithm then creates a new pointer at this location that is made to point to a new trie node that contains P11. P11 is then expanded in this new node.

Deletion is similar to insertion. The complexity of insertion and deletion is the time to perform a search ($O(W)$) plus the time to completely reconstruct a trie node ($O(S)$, where S is the maximum size of a trie node). For example, using 8-bit trie nodes, the latter cost will require scanning roughly $2^8 = 256$ trie node entries. Thus to allow for fast updates, it is crucial to also limit the size of any trie node in the dynamic program described earlier.

11.6 Level-compressed (LC) tries

An LC trie (Nilsson and Karlsson, 1998) is a variable-stride trie in which every trie node contains no empty entries. An LC-trie is built by first finding the largest-root stride that allows no empty entries and then recursively repeating this procedure on the child subtries. An example of this procedure is shown in Fig. 11.10, starting with a 1-bit trie on the left and resulting in an LC trie on the right. Notice that P4 and P5 form the largest possible full-root subtrie—if the root stride is 2, then the first two array entries will be empty. The motivation, of course, is to avoid empty array elements, to minimize storage.

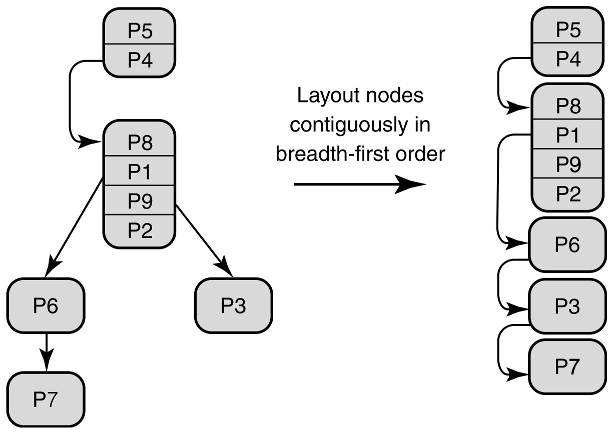


FIGURE 11.11

Array representation of LC tries.

However, general variable-stride tries are more tunable, allowing memory to be traded for speed. For example, the LC trie representation using a 1997 snapshot of MAE-East has a trie height of 7 and needs 700 KB of memory. By comparison, an optimal variable-stride trie (Srinivasan and Varghese, 1999) has a trie height of 4 using 400 KB. Recall also that the optimal variable-stride calculates the best trie for a given target height and thus would indeed produce the LC trie *if* the LC trie were optimal for its height.

In its final form, the variable-stride LC trie nodes are laid out in breadth-first order (first the root, then all the trie nodes at the second level from left to right, then third-level nodes, etc.), as shown on the right of Fig. 11.11. Each pointer becomes an array offset. The array layout and the requirement for full subtries make updates slow in the worst case. For example, deleting P5 in Fig. 11.10 causes a change in the subtrie decomposition. Worse, it causes almost every element in the array representation of Fig. 11.11 to be moved upward.

11.7 Lulea-compressed tries

Though LC tries and variable-stride tries attempt to compress multibit tries by varying the stride at each node, both schemes have problems. While the use of full arrays allows LC tries not to waste any memory because of empty array locations, it also increases the height of the trie, which cannot then be tuned. On the other hand, variable-stride tries can be tuned to have short height, at the cost of wasted memory because of empty array locations in trie nodes. The Lulea approach (Degermark et al., 1997), which we now describe, is a multibit-trie scheme that uses fixed-stride trie nodes of large stride but uses *bitmap* compression to reduce storage considerably.

We know that a string with repetitions (e.g., AAAABBAAACCCCC) can be compressed using a bitmap denoting repetition points (i.e., 10001010010000) together with a compressed sequence (i.e.,

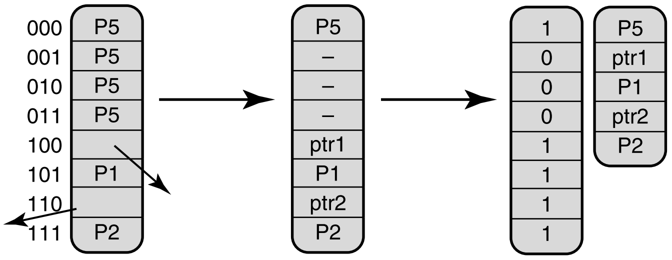


FIGURE 11.12

Compressing the root node of Fig. 11.7 (after leaf pushing) using the Lulea bitmap compression scheme.

ABAC). Similarly, the root node of Fig. 11.7 contains a repeated sequence (P5, P5, P5, P5) caused by expansion.

The Lulea scheme (Degermark et al., 1997) avoids this obvious waste by compressing repeated information using a bitmap and a compressed sequence without paying a high penalty in search time. For example, this scheme used only 160 KB of memory to store the MAE-East database. This allows the entire database to fit into expensive SRAM or on-chip memory. It does, however, pay a high price in insertion times.

Some expanded trie entries (e.g., the 110 entry at the root of Fig. 11.7) have two values, a pointer and a prefix. To make compression easier, the algorithm starts by making each entry have exactly one value by pushing prefix information down to the trie leaves. Since the leaves do not have a pointer, we have only next-hop information at leaves and only pointers at nonleaf nodes. This process is called *leaf pushing*.

For example, to avoid the extra stored prefix in the 110 entry of the root node of Fig. 11.7, the P9 stored prefix is pushed to all the entries in the leftmost trie node, with the exception of the 010 and 011 entries (both of which continue to contain P3). Similarly, the P8 stored prefix in the 100 root node entry is pushed down to the 100, 101, 110, and 111 entries of the rightmost trie node. Once this is done, each node entry contains either a stored prefix or a pointer but not both.

The Lulea scheme starts with a conceptual leaf-pushed expanded trie and replaces consecutive identical elements with a single value. A node bitmap (with 0's corresponding to removed positions) is used to allow fast indexing on the compressed nodes.

Consider the root node in Fig. 11.7. After leaf pushing, the root has the sequence P5, P5, P5, P5, ptr1, P1, ptr2, P2 (ptr1 is a pointer to the trie node containing P6 and P7, and ptr2 is a pointer to the node containing P3). After replacing consecutive values with the first value, we get P5, -, -, -, ptr1, P1, ptr2, P2, as shown in the middle frame of Fig. 11.12. The rightmost frame shows the final result, with a bitmap indicating removed positions (10001111) and a compressed list (P5, ptr1, P1, ptr2, P2).

If there are N original prefixes and pointers within an original (unexpanded) trie node, the number of entries within the compressed node can be shown never to be more than $2N + 1$. Intuitively, this is because N prefixes partition the address space into at most $2N + 1$ disjoint subranges and each subrange requires at most one compressed node entry.

Search uses the number of bits specified by the stride to index into the current trie node, starting with the root and continuing until a null pointer is encountered. However, while following pointers, an uncompressed index must be mapped to an index into the compressed node. This mapping is accomplished by counting bits within the node bitmap.

Consider the data structure on the right of Fig. 11.12 and a search for an address that starts with 100111. If we were dealing with just the uncompressed node on the left of Fig. 11.12, we could use 100 to index into the fifth array element to get ptr1. However, we must now obtain the same information from the compressed-node representation on the right of Fig. 11.12.

Instead, we use the first three bits (100) to index into the root-node bitmap. Since this is the second bit set (the algorithm needs to count the bits set before a given bit), the algorithm indexes into the second element of the compressed node. This produces a pointer ptr1 to the rightmost trie node. Next, imagine the rightmost leaf node of Fig. 11.7 (after leaf pushing) also compressed in the same way. The node contains the sequence P7, P6, P6, P6, P8, P8, P8, P8. Thus the corresponding bitmap is 11001000, and the compressed sequence is P7, P6, P8.

Thus in the rightmost leaf node, the algorithm uses the next 3 bits (111) of the destination address to index into bit 8. Since this bit is a 0, the search terminates: There is no pointer to follow in the equivalent uncompressed node. However, to retrieve the best matching prefix (if any) at this node, the algorithm must find any prefix stored before this entry.

This would be trivial with expansion because the value P8 would have been expanded into the 111 entry, but since the expanded sequence of P8 values has been replaced by a single P8 value in the compressed version, the algorithm has to work harder. Thus the Lulea algorithm *counts* the number of bits set before position 8 (which happens to be 3) and then indexes into the third element of the compressed sequence. This gives the correct result P8.

The Lulea paper (Degermark et al., 1997) describes a trie that uses fixed strides of 16, 8, and 8. But how can the algorithm efficiently count the bits set in a large bitmap, say of 64K bits in size, that a 16-bit stride needs to use? Before you read on, try to answer this question using principles **P12** (adding state for speed) and **P2a** (precomputation).

To speed up counting set bits, the algorithm accompanies each bitmap with a summary array that contains a cumulative count (*precomputed*) of the number of set bits associated with fixed-size chunks of the bitmap. Using 64-bit chunks, the summary array takes negligible storage. Counting the bits set up to position i now takes two steps. First, access the summary array at position j, where j is the chunk containing bit i. Then access chunk j and count the bits in chunk j up to position i. The sum of the two values gives the count.

While the Lulea paper uses 64-bit chunks, the example in Fig. 11.13 uses 8-bit chunks. The large bitmap is shown from left to right, starting with 10001001, as the second array from the top. Each 8-bit chunk has a summary count that is shown as an array above the bitmap. The summary count for chunk i counts the cumulative bits in the previous chunks of the bitmap (not including chunk i).

Thus the first chunk has count 0, the second has count 3 (because 10001001 has three bits set), and the third has count 5 (because 10000001 has two bits set, which added to the previous chunk's value of 3 gives a cumulative count of 5).

Consider searching for the bits set up to position X in Fig. 11.13, where X can be written as J011. Clearly, X belongs to chunk J. The algorithm first looks up the summary count array to retrieve $numSet[J]$. This yields the number of bits set up to but not including chunk J. The algorithm then retrieves chunk J itself (10011000) and counts the number of bits set until the third position of chunk J. Since the first three bits of chunk J are 100, this yields the value 1. Finally, the desired overall bit count is $numSet[J] + 1$.

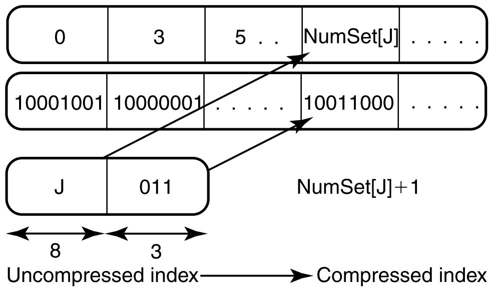


FIGURE 11.13

To allow fast counting of the bits set even in large bitmaps (e.g., 64 Kbits), the bitmap is divided into chunks and a summary count of the bits set before each chunk precomputed.

Notice that the choice of the chunk size is a trade-off between memory size and speed. Making a chunk equal to the size of the bitmap will make counting very slow. On the other hand, making a chunk equal to a bit will require more storage than the original trie node! Choosing a 64-bit chunk size makes the summary array size only 1/64 the size of the original node, but this requires counting the bits set within a 64-bit chunk. Counting can easily be done using special instructions in software and via combinational logic in hardware.

Thus search of a node requires first indexing into the summary table, then indexing into the corresponding bitmap chunk to compute the offset into the compressed node, and finally retrieving the element from the compressed node. This can take three memory references per node, which can be quite slow.

The final Lulea scheme also compresses entries based on their next-hop values (entries with the same next-hop values can be considered the same even though they match different prefixes). Overall the Lulea scheme has very compact storage. Using an early (1997) snapshot of the MAE-East database of around 40,000 entries, the Lulea paper (Degermark et al., 1997) reports compressing the entire database to around 160 KB, which is roughly 32-bits per prefix.

This is a very small number, given that one expects to use at least one 20-bit pointer per prefix in the database. The compact storage is a great advantage because it allows the prefix database to potentially fit into limited on-chip SRAM, a crucial factor in allowing prefix lookups to scale to OC-768 speeds.

Despite compact storage, the Lulea scheme has two disadvantages. First, counting bits requires at least one extra memory reference per node. Second, leaf pushing makes worst-case insertion times large. A prefix added to a root node can cause information to be pushed to thousands of leaves. The full tree bitmap scheme, which we study next, overcomes these problems by abandoning leaf pushing and using *two* bitmaps per node.

11.8 Tree bitmap

The tree bitmap (Eatherton et al., 2004) scheme starts with the goal of achieving the same storage and speed as the Lulea scheme, but it adds the goal of fast insertions. While we have argued that fast

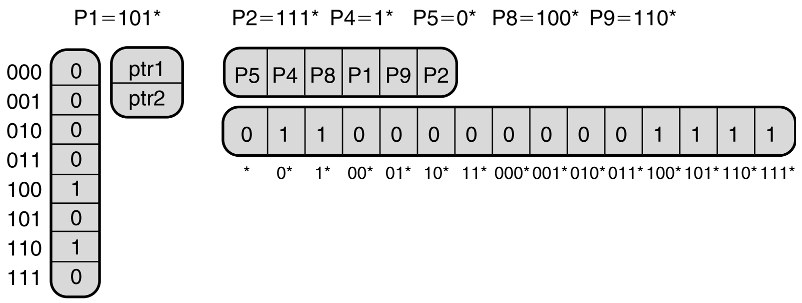


FIGURE 11.14

The tree bitmap scheme allows the compression of Lulea without sacrificing fast insertions by using *two* bitmaps per node. The first bitmap describes valid versus null pointers, and the second describes internally stored prefixes.

insertions are not as important as fast lookups, they clearly are desirable. Also, if the only way to handle an insertion or deletion is to rebuild the Lulea-compressed trie, then a router must keep two copies of its routing database, one that is being built and one that is being used for lookups. This can potentially double the storage cost from 32 bits per prefix to 64 bits per prefix. This in turn can halve the number of prefixes that can be supported by a chip that places the entire database in on-chip SRAM.

To obtain fast insertions and hence avoid the need for two copies of the database, the first problem in Lulea that must be handled is the use of leaf pushing. When a prefix of a small length is inserted, leaf pushing can result in pushing down the prefix to a large number of leaves, making insertion slow.

11.8.1 Tree bitmap ideas

Thus the first and main idea in the tree bitmap scheme is that there are *two* bitmaps per trie node, one for all the internally stored prefixes and one for the external pointers. Fig. 11.14 shows the tree bitmap version of the root node in Fig. 11.12.

Recall that in Fig. 11.12, the prefixes P8 = 100* and P9 = 110* in the original database are missing from the picture on the left side because they have been pushed down to the leaves to accommodate the two pointers ($ptr1$, which points to nodes containing longer prefixes such as P6 = 1000*, and $ptr2$, which points to nodes containing longer prefixes such as P3 = 11001*). This results in the basic Lulea trie node, in which each element contains either a pointer or a prefix but not both. This allows the use of a *single* bitmap to compress a Lulea node, as shown on the extreme right of Fig. 11.12.

By contrast, the same trie node in Fig. 11.14 is split into *two* compressed arrays, each with its own bitmap. The first array, shown vertically, is a *pointer array*, which contains a bitmap denoting the (two) positions where nonnull pointers exist and a compressed array containing the nonnull pointers, $ptr1$ and $ptr2$.

The second array, shown horizontally, is the *internal prefix array*, which contains a list of all the prefixes within the first 3 bits. The bitmap used for this array is very different from the Lulea encoding and has one bit set for every possible prefix stored within this node. Possible prefixes are listed lexicographically, starting from $*$, followed by $0*$ and $1*$, and then on to the length-2 prefixes (00*, 01*, 10*, 11*), and finally the length-3 prefixes. Bits are set when the corresponding prefixes occur within the trie node.

Thus in Fig. 11.14, the prefixes P8 and P9, which were leaf pushed in Fig. 11.12, have been resurrected and now correspond to bits 12 and 14 in the internal prefix bitmap. In general, for an r-bit trie node, there are $2^{r+1} - 1$ possible prefixes of lengths r or less, which requires the use of a $(2^{r+1} - 1)$ bitmap. The scheme gets its name because the internal prefix bitmap represents a trie in a linearized format: Each row of the trie is captured top-down from left to right.

The second idea in the tree bitmap scheme is to keep the trie nodes as small as possible to reduce the required memory access size for a given stride. Thus a trie node is of fixed size and contains only a pointer bitmap, an internal prefix bitmap, and child pointers. But what about the next-hop information associated with any stored prefixes?

The trick is to store the next hops associated with the internal prefixes stored within each trie node in a separate array associated with this trie node. Putting next-hop pointers in a separate result array potentially requires two memory accesses per trie node (one for the trie node and one to fetch the result node for stored prefixes).

However, a simple lazy evaluation strategy (**P2b**) is *not* to access the result nodes until search terminates. Upon termination, the algorithm makes a final access to the correct result node. This is the result node that corresponds to the last trie node encountered in the path that contained a valid prefix. This adds only a single memory reference at the end, in addition to the one memory reference required per trie node.

The third idea is to use only *one* memory access per node, unlike Lulea, which uses at least two memory accesses. Lulea needs two memory accesses per node because it uses large strides of 8 or 16 bits. This increases the bitmap size so much that the only feasible way to count bits is to use an additional chunk array that must be accessed separately. The tree bitmap scheme gets around this by simply using smaller-stride nodes, say, of 4 bits. This makes the bitmaps small enough that the entire node can be accessed by a single wide access (**P4a**, exploit locality). Combinatorial logic (Chapter 2) can be used to count the bits.

11.8.2 Tree bitmap search algorithm

The search algorithm starts with the root node and uses the first r bits of the destination address (corresponding to the stride of the root node, 3 in our example) to index into the pointer bitmap at the root node at position P. If there is a 1 in this position, there is a valid child pointer. The algorithm counts the number of 1's to the left of this 1 (including this 1) and denotes this count by I. Since the pointer to the start position of the child pointer block (say, y) is known, as is the size of each trie node (say, S), the pointer to the child node can be calculated as $y + (I * S)$.

Before moving on to the child, the algorithm must also check the internal bitmap to see if there are one or more stored prefixes corresponding to the path through the multibit node to position P. For example, suppose P is 101 and a 3-bit stride is used at the root node bitmap, as in Fig. 11.14. The algorithm first checks to see whether there is a stored internal prefix 101*. Since 101* corresponds to the 13th bit position in the internal prefix bitmap, the algorithm can check if there is a 1 in that position (there is one in the example). If there was no 1 in this position, the algorithm would back up to check whether there is an internal prefix corresponding to 10*. Finally, if there is not a 10* prefix, the algorithm checks for the prefix 1*.

This search algorithm appears to require a number of iterations, proportional to the logarithm of the internal bitmap length. However, for bitmaps of up to 512 bits or so in hardware, this is just a matter of simple combinational logic. Intuitively, such logic performs all iterations in parallel and uses a priority encoder to return the longest matching stored prefix.

Once it knows there is a matching stored prefix within a trie node, the algorithm does not immediately retrieve the corresponding next-hop information from the result node associated with the trie node. Instead, the algorithm moves to the child node while remembering the stored-prefix position and the corresponding parent trie node. The intent is to remember the last trie node T in the search path that contained a stored prefix, and the corresponding prefix position.

Search terminates when it encounters a trie node with a 0 set in the corresponding position of the extending bitmap. At this point, the algorithm makes a final access to the result array corresponding to T to read off the next-hop information. Further tricks to reduce memory access width are described in Eatherton's MS thesis (Eatherton, 1995), which includes a number of other useful ideas.

Intuitively, insertions in a tree bitmap are very similar to insertions in a simple multibit trie without leaf pushing. A prefix insertion may cause a trie node to be changed completely; a new copy of the node is created and linked in atomically to the existing trie. Compression results in Eatherton et al. (2004) show that the tree bitmap has all the features of the Lulea scheme, in terms of compression and speed, along with fast insertions. The tree bitmap also has the ability to be tuned for hardware implementations ranging from the use of RAMBUS-like memories to on-chip SRAM.

11.8.3 PopTrie: an alternate bitmap algorithm

Many years after the initial Lulea and Tree bitmap algorithms, a new bitmap algorithm called Poptrie (Asai and Ohara, 2015) was proposed. The algorithm gets its name from the population count instruction (called *popcount*) that can count the number of bits set (in say a 64-bit machine word) on a modern CPU. The setting is for a software implementation, perhaps for Network Function Virtualization. The motivation is that since the Tree bitmap of stored prefixes is not a simple linear bitmap it cannot be calculated easily in software. The Poptrie paper (Asai and Ohara, 2015) describes some experiments to suggest that Tree bitmap is fairly slow in software, and suggests an alternative.

The alternative to Tree bitmap is again to use an aggressive form of leaf-pushing so insertion is again slow compared to Tree bitmap. In Poptrie, every leaf stores an associated prefix (corresponding to its most specific ancestor prefix). Thus every multibit trie element either is a null pointer with an associated leaf-pushed prefix, or contains a pointer to a descendant node. Thus each node has a basic bitmap where the 1's represent valid pointers to descendants and the 0's represent stored prefixes. Thus each trie node has the basic bitmap and two base pointers: the first base pointer (base0) points to the start of the stored prefix array, and the second (base1) to the start of the descendant pointer array.

Poptrie breaks an IP address into 6-bit chunks (a uniform stride of 6 bits) so each basic bitmap is 64 bits. During search, when indexing into the bitmap using the current chunk of the address, if the bit is 1, then there is a descendant node. In that case, search continues by counting the number of 1's and using the count to index into the compressed array of pointers whose start is base1. If the indexed bit is a 0, search terminates with a stored prefix. Search now ends by counting the number of 0's and indexing into the base0 array to retrieve the next hop associated with the longest match. Because Poptrie uses 64 bit bitmaps, the counting can be done efficiently in software by a popcount instruction in 1 cycle.

The basic scheme as described takes a tremendous amount of storage because of the very aggressive leaf pushing. But many of the consecutive entries have the same stored prefix. As in our description of

Lulea, one can do a form of run-length encoding using a second bitmap, called the LeafVector. Thus if there is a run of consecutive stored prefixes say P4, P4, P4, P5. ., in the base0 array this is replaced by one copy of P4 and a LeafVector bitmap 1001 ...

In summary, all three schemes use different but closely related semantics for bitmaps. Lulea uses a single bitmap and large strides, but uses a summary bitmap to recover speed. Tree bitmap does not do leaf pushing, but uses two bitmaps, one for compressing pointers and one Tree bitmap that represents the stored prefixes. Poptrie does aggressive leaf pushing, and uses two bitmaps: one that distinguishes pointers from prefixes, and one that compresses consecutive stored prefixes.

While the Poptrie paper shows better performance than Tree bitmap in software, there are two concerns. First, it is not clear that Lulea cannot simulate the forwarding performance of Poptrie with smaller (say 6-bit) strides and using the popcount instruction to count bits. Second, as we will see below, the current fastest method in software uses a different strategy, based on binary search on prefix ranges.

11.9 Binary search on ranges

So far, all our schemes (unibit tries, expanded tries, LC tries, Lulea tries, tree bitmaps) have been trie variants. Are there other algorithmic paradigms (**P15**) to the longest-matching-prefix problem? Now, exact matching is a special case of prefix matching. Both binary search and hashing (Cormen et al., 1990) are well-known techniques for exact matching. Thus we should consider generalizing these standard exact-matching techniques to handle prefix matching. In this section we examine an adaptation of binary search; in the next section we look at an adaptation of hashing.

In *binary search on ranges* (Lampson et al., 1998), each prefix is represented as a range, using the start and end of the range. Thus the range endpoints for N prefixes partition the space of addresses into $2N + 1$ disjoint intervals. The algorithm (Lampson et al., 1998) uses binary search to find the interval in which a destination address lies. Since each interval corresponds to a unique prefix match, the algorithm *precomputes* this mapping and stores it with range endpoints. Thus prefix matching takes $\log_2(2N)$ memory accesses.

Consider a tiny routing table with only two prefixes, P4 = 1* and P1 = 101*. This is a small subset of the database used in Fig. 11.6. Fig. 11.15 shows how the binary search data structure is built as a table (left) and as a binary tree (right).

The starting point for this scheme is to consider a prefix as a range of addresses. To keep things simple, imagine that addresses are 4 bits instead of 32 bits. Thus P4 = 1* is the range 1000 to 1111, and P1 = 101* is the range 1010 to 1011. Next, after adding in the range for the entire address space (0000 to 1111), the endpoints of all ranges are sorted into a binary search table, as shown on the left of Fig. 11.15.

In Fig. 11.15, the range endpoints are drawn vertically on the left. The figure also shows the ranges covered by each of the prefixes. Next, *two* next-hop entries are associated with each endpoint. The leftmost entry, called the $>$ entry, is the next hop corresponding to addresses that are *strictly greater* than the endpoint but strictly less than the next range endpoint in sorted order. The rightmost entry, called the $=$ entry, corresponds to addresses that are *exactly equal* to the endpoint.

For example, it should be clear from the ranges covered by the prefixes that any addresses greater than or equal to 0000 but strictly less than 1000 do not match any prefix. Hence the entries

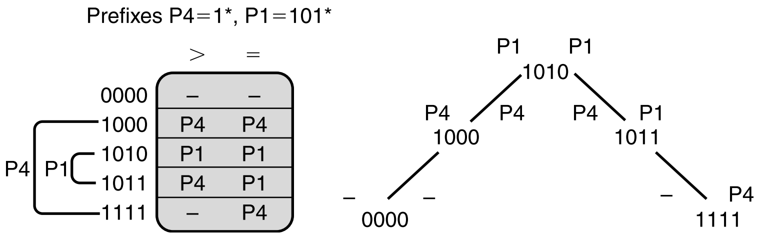


FIGURE 11.15

Binary search on values of a tiny subset of the sample database, consisting of only prefixes P4 = 1* and P1 = 101*.

corresponding to 0000 are –, to denote no next hop.[3] Similarly, any address greater than or equal to 1000 but strictly less than 1010 must match prefix P4 = 1*.

The only subtle case, which illustrates the need for two separate entries for > and =, is the entry for 1011. If an address is strictly greater than 1011 but strictly less than the next entry, 1111, then the best match is P4. Thus the > pointer is $P4$. On the other hand, if an address is exactly equal to 1011, its best match is P1. Thus the = pointer is P1.

The entire data structure can be built as a binary search table, where each table entry has three items, consisting of an endpoint, a > next-hop pointer, and a = next-hop pointer. The table has at most $2N$ entries, because each of N prefixes can insert two endpoints. Thus after the next-hop values are precomputed, the table can be searched in $\log_2 2N$ time using binary search on the endpoint values. Alternatively, the table can be drawn as a binary tree, as shown on the right in Fig. 11.15. Each tree node contains the endpoint value and the same two next-hop entries.

The description so far shows that binary search on values can find the longest prefix match after $\log_2 2N$ time. However, the time can be reduced using binary trees of higher radix, such as B-trees. While such trees require wider memory accesses, this is an attractive trade-off for DRAM-based memories, which allow fast access to consecutive memory locations (**P4a**).

Computational geometry (Preparata and Shamos, 1985) offers a data structure called a *range tree* for finding the narrowest range. Range trees offer fast insertion times as well as fast $O(\log_2 N)$ search times. However, there seems to be no easy way to increase the radix of range trees to obtain $O(\log_M N)$ search times for $M > 2$.

As described, this data structure can easily be built in linear time using a stack and an additional trie. It is not hard to see that even with a balanced binary tree (see exercises), adding a short prefix can change the > pointers of a large number of prefixes in the table. A trick to allow fast insertions and deletions in logarithmic time is described in Warkhede et al. (2001).

Naively done, binary search on prefix values is somewhat slow when compared to multibit tries. However, unlike the other trie schemes, all of which are subject to patents, binary search is free of such restrictions. Thus at least a few vendors have implemented this scheme into hardware. In hardware, the

[3] In a core router, no prefix match implies that the message should be dropped; in a router within a domain no prefix match is often sent to the so-called default route.

use of a wide memory access (to reduce the base of the logarithm) and pipelining (to allow one lookup per memory access) can make this scheme sufficiently fast.

The original paper (Lampson et al., 1998) written in 1998 suggests two optimizations to improve speed even in software. First, it suggests improving "the worst-case number of memory accesses of the basic binary search scheme with a precomputed table of best-matching prefixes for the first bits ... if there are prefixes of longer length with that prefix the array element stores a pointer to a binary search table/tree that contains all such prefixes" (Lampson et al., 1998). The paper shows that this simple trick of using an array as a front-end reduces the maximum number of prefixes in each partitioned table from over 38000 to 336, reducing the worst case binary table size to 336, which makes binary search faster (10 memory accesses versus $\log_2 N + 1$ where N is the size of the original table). Second, it suggests using larger radixes instead of binary trees and exploiting the cache line size of a Pentium processor to make such k-way searh efficient in software.

Best of all, a highly optimized form of the original binary search on prefix ranges paper with initial table lookup (Lampson et al., 1998) called DXR (Zec et al., 2012) first suggested in 2021 currently appears to offer the fastest software implementations of over 2.5 Billion IP lookups per second on a commodity CPU (AMD R7-1700) with 8 cores in 2022. This makes it a good building block for Network Function Virtualization devices. We now describe the new optimizations in DXR.

11.10 Binary search on ranges with Initial Lookup Table

The core idea in DXR (Zec et al., 2012) is still binary search on ranges with an initial table. Thus rather than have 1 binary search, if the root has N pointers there can be pointers to N different binary search tables. Once again, each prefix is converted to a range (start and end address).

DXR, however, goes beyond the original binary search on ranges idea with an initial table (Lampson et al., 1998) using multithreading and careful compression to fit into the cache hierarchy of a modern CPU in clever ways. First, they use a *multithreaded* implementation so the apparent cost of binary search can be hidden by increasing the number of hardware threads. Second, they use several compression ideas to reduce memory to make it more likely for the database to fit into the L2/L3 caches. Neighboring address ranges that resolve to the same next hop are merged. Next, the end address can be derived from the start of the next interval. Thus each entry only needs the start address (2 bytes, because the first 16 or more bits have already been resolved by direct lookup), and the next hop (1 or 2 bytes to index a next hop table). A further twofold compression can be achieved for chunks which reference only 8-bit next hop indices and correspond to prefix lengths up to 24 bits. For these, the range start can be compressed to 8 bits or less, which is a big savings. Finally, they experiment with initial table sizes beyond 2^{16} and find better numbers by using anywhere from 16 to 21 bits.

Using a 16 bit initial table, for instance, is compact and takes "less than 2 bytes per prefix, and exceeds 100 million lookups per second (Mlps) on a single commodity CPU core in synthetic tests with uniformly random queries" (DXR, 2022). Using 21 bits, allows "200 Mlps per CPU core at the cost of increased memory footprint, and deliver aggregate throughputs exceeding two billion lookups per second" (DXR, 2022) using 8 cores. A version of DXR merged in FreeBSD uses a two-stage trie with a 16-4 split before proceeding with binary search.

Note that partitioning (into multiple prefix search tables) also reduces the cost of updates because a prefix addition or the deletion of a prefix P does not affect all the partitioned tables, but affects only the tables that are pointed to by root entries that match P. This translates into a huge savings since a full build is much more costly than an incremental update: The cost of a full build for a 16-bit initial table is reported to be 70 msec and goes up to 300 msec for a 20-bit initial stride (Zec and Mikuc, 2017), whereas that of an incremental update is reported to be only 10s of microseconds on average for random tests (Zec et al., 2012).

There seems to be two fundamental reasons why DXR does so well. First, the bigger reason is that careful compression allows it to fit into the cache hierarchy; as the table size grows, the cache size of modern processors should also grow. Second, the seemingly long time taken for binary search seems to be hidden by thread parallelism. A recent updated Zec and Mikuc (2017) suggests that as CPUs scale to more hardware threads (say 36 threads) this could easily double the throughput of DXR.

Despite its success for IPv4, it is unlikely that DXR will work well as is for IPv6 because of 128-bit keys and a more sparse address space. However, it is worth pointing out that the original paper Lampson et al. (1998) suggests a multicolumn binary search for binary search of long identifiers (see the Exercise in this chapter on Multicolumn search). Multicolumn search with multithreading (and the compression ideas used in DXR) may be the basis for a fast software IPv6 implementation but more work is needed.

The code for DXR is available online and has been integrated into FreeBSD which allows it to be used as part of a Network Function Virtualization device based on FreeBSD.

11.11 Binary search on prefix lengths

In this section we adapt another classical exact-match scheme, hashing, to longest prefix matching. Binary search on prefix lengths finds the longest match using $\log_2 W$ hashes, where W is the maximum prefix length. This can provide a very scalable solution for 128-bit IPv6 addresses. For 128-bit prefixes, this algorithm takes only seven memory accesses, as opposed to 16 memory accesses using a multibit trie with 8-bit strides. To do so, the algorithm first segregates prefixes by length into separate hash tables. More precisely, it uses an array L of hash tables such that $L[i]$ is a pointer to a hash table containing all prefixes of length i.

Assume the same tiny routing table, with only two prefixes, P4 = 1* and P1 = 101*, of lengths 1 and 3, respectively, that was used in Fig. 11.15. Recall that this is a small subset of Fig. 11.6. The array of hash tables is shown horizontally in the top frame (A) of Fig. 11.16. The length-1 hash table storing P4 is shown vertically on the left and is pointed to by position 1 in the array; the length-3 hash table storing P1 is shown on the right and is pointed to by position 3 in the array; the length-2 hash table is empty because there are no prefixes of length 2.

Naively, a search for address D would start with the greatest-length hash table l (i.e., 3), would extract the first l bits of D into D_l, and then search the length-l hash table for D_l. If search succeeds, the best match has been found; if not, the algorithm considers the next smaller length (i.e., 2). The algorithm moves in decreasing order among the set of possible prefix lengths until it either finds a match or runs out of lengths.

The naive scheme effectively does *linear* search among the distinct prefix lengths. The analogy suggests a better algorithm: binary search (**P15**). However, unlike binary search on prefix *ranges*, this is binary search on prefix *lengths*. The difference is major. With 32 lengths, binary search on lengths takes five hashes in the worst case; with 32,000 prefixes, binary search on prefix ranges takes 16 accesses.

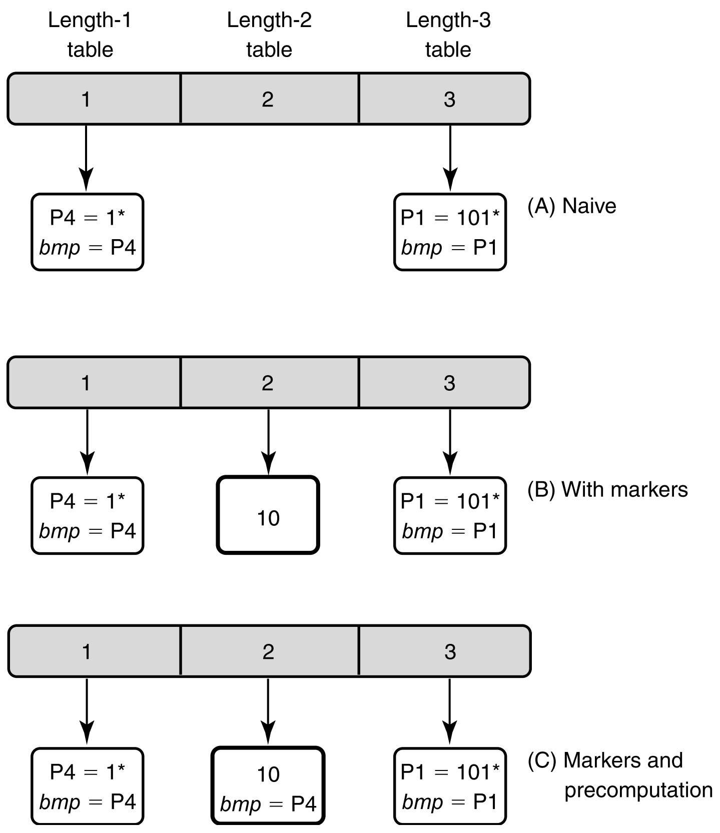


FIGURE 11.16

From naive linear search on the possible prefix lengths to binary search.

Binary search must start at the *median* prefix length, and each hash must divide the possible prefix lengths in half. A hash search gives only two values: *found* and *not found*. If a match is found at length m, then lengths strictly greater than m must be searched for a longer match. Correspondingly, if no match is found, search must continue among prefixes of lengths strictly less than m.

For example, in Fig. 11.16, part (A), suppose search begins at the median length-2 hash table for an address that starts with 101. Clearly, the hash search does not find a match. But there is a longer match in the length-3 table. Since only a match makes search move to the right half, an "artificial match," or *marker*, must be introduced to force the search to the right half when there is a potentially longer match.

Thus part (B) introduces a bolded marker entry 10, corresponding to the first two bits of prefix P1 = 101, in the length-2 table. In essence, state has been added for speed (**P12**). The markers allow probe failures in the median to rule out all lengths greater than the median.

Search for an address D that starts with 101 works correctly. Search for 10 in the length-2 table (in part (B) of Fig. 11.16) results in a match; search proceeds to the length-3 table, finds a match with

$P1$, and terminates. In general, a marker for a prefix P must be placed at all lengths that binary search will visit in a search for P. This adds only a logarithmic number of markers. For a prefix of length 32, markers are needed only at lengths 16, 24, 28, and 30.

Unfortunately, the algorithm is still incorrect. While markers lead to potentially longer prefixes, they can also cause search to follow false leads. Consider a search for an address D' whose first three bits are 100 in part (B) of Fig. 11.16. Since the median table contains 10, search in the middle hash table results in a match. This forces the algorithm to search in the third hash table for 100 and to fail. But the correct best matching prefix is at the first hash table – i.e., P4 = 1*. Markers can cause the search to go off on a wild goose chase! On the other hand, a backtracking search of the left half would result in linear time.

To ensure logarithmic time, each marker node M contains a variable $M.bmp$, where $M.bmp$ is the longest prefix that matches string M. This is precomputed when M is inserted into its hash table. When the algorithm follows marker M and searches for prefixes of lengths greater than M, and if the algorithm fails to find such a longer prefix, then the answer is $M.bmp$. In essence, the best matching prefix of every marker is precomputed (**P2a**). This avoids searching all lengths less than the median when a match is obtained with a marker.

The final version of the database containing prefixes P4 and P1 is shown in part (C) of Fig. 11.16. A *bmp* field has been added to the 10 marker that points to the best matching prefix of the string 10 (i.e., P4 = 1*). Thus when the algorithm searches for 100 and finds a match in the median length-2 table, it *remembers* the value of the corresponding *bmp* entry P4 before it searches the length-3 table. When the search fails (in the length-3 table), the algorithm returns the *bmp* field of the last marker encountered (i.e., P4).

A trivial algorithm for building the simple binary search data structure from scratch is as follows. First determine the distinct prefix lengths; this determines the sequence of lengths to search. Then add each prefix P in turn to the hash table corresponding to $length(P)$. For each prefix, also add a marker to all hash tables corresponding to lengths $L < length(P)$ that binary search will visit (if one does not already exist). For each such marker M, use an auxiliary 1-bit trie to determine the best matching prefix of M. Further refinements are described in Waldvogel et al. (1997).

While the search algorithm takes five hash table lookups in the worst case for IPv4, we note that in the expected case most lookups should take two memory accesses. This is because the expected case observation *O1* shows that most prefixes are either 16 or 24 bits (at least today). Thus doing binary search at 16 and then 24 will suffice for most prefixes.

The use of hashing makes binary search on prefix lengths somewhat difficult to implement in hardware. However, its scalability to large prefix lengths, such as IPv6 addresses, is notable.

11.12 Linear search on prefix lengths with hardware assist

The last section shows we can make binary search on prefix lengths work by adding markers and other mechanisms. Unfortunately, this adds complexity and makes insertion slow. The idea in this section is to revisit naive linear search on prefix lengths (that we discarded in the last section), but add some hardware assistance to make it practical.

Assume the setting is hardware with on-chip memory (with say 1 nsec access time) of a few Mbytes and a massive amount of slower DRAM (with say 50 nsec access time). How can we do IP Lookups with a small number (say 1 or 2) of sequential DRAM accesses? We can easily also afford up to to 32 on-chip memory accesses because the on-chip accesses are dwarfed by a single DRAM access. We will describe two such schemes, both based on using bitmaps to represent the stored prefixes at each length. Since these bitmaps are compact, they can be stored in on-chip memory; a corresponding hash table for each length is stored in off-chip DRAM using a scheme such as d-left (Broder and Mitzenmacher, 2001) (described in Section 10.3.3). For example, if there are just two prefixes $P4 = 1*$ and $P1 = 100*$. The length-1 bitmap would be 10 (first bit set corresponding to $P4$, the length-2 bitmap would be 0000, and the length-3 bit map would be 00010000 (fourth bit is set corresponding to $P1$).

The skeleton idea, then, is to search on-chip either sequentially or in parallel to find the longest matching *length*, say L. Then a hash table access is made to an off-chip DRAM hash table that stores all L bit prefixes using the first L bits of the address as a key to retrieve the next hop. Clearly, a naive use of indexed bitmaps does not work well because at prefix lengths such as 32 the corresponding bitmap is of size 2^{32} which is too large to fit into on-chip memory. We will describe two schemes that show how to deal with this memory explosion caused by larger length bitmaps.

11.12.1 Using Bloom Filters to compress prefix bitmaps

Observe that the bitmaps, especially at larger lengths will be very sparse. Thus each bitmap can be compressed by what is called a Bloom Filter (Bloom, 1970). In a Bloom Filter any entry x is hashed a small constant number of times (often 3 times) with independent hash functions (say $h_1(x) = i, h_2(x) = j, h_3(x) = k$, and bits i, j, k are set in the bitmap. Thus if there are N_L prefixes at length L, the theory shows that the N_L prefixes require cN_L bits, where c is a small constant.

While we will study Bloom filters later in this book when we survey techniques for measurement and security, for now it is worth noting that Bloom filters have no false negatives, but can have false positives. It may happen that when doing a lookup for entry y that is not present, the corresponding indices $h_1(y), h_2(y), h_3(y)$ are all set, leading search to incorrectly conclude that y exists in that Bloom Filter. However, if c is sufficiently large, the probability of a false positive is low.

A paper by Dharmapurikar et al. (2003) suggests using 32 Bloom Filters to compress all the stored prefixes stored at each length on-chip, followed by off-chip lookups. If i is the longest length Bloom Filter that matches, an off-chip access is made to off-chip Hash Table i. However, the Bloom filter may rarely have a false positive, in which case the chip will try the next smallest Bloom filter length that matched, and so on. While in the worst case there can be several DRAM accesseses (such pathological IP addresses can be cached), the expected number of DRAM accesses is close to 1. Nevertheless, this lack of deterministic performance leads to the next proposal called SAIL.

11.12.2 SAIL: Uncompressed Bitmaps up to a pivot level

SAIL (Yang et al., 2014) starts by observing that up to some length (which they refer to as a *pivot* length) one can easily store all the bitmaps without further compression. In particular, they suggest that storing all bitmaps up to 24 requires only $\Sigma_{i=1}^{24} 2^i = 32$ Mbits. Thus their implementation pivots at length 24, but this could be changed. While 4 Mbytes is large, it is feasible in on-chip SRAM. Better still, the amount of required on-chip memory remains constant even as the IPv4 database increases arbitrarily. They also observe that prefixes of length greater than 24 bits are rare: hence they use prefix expansion to store all such prefixes in 256 element multibit trie nodes indexed by their 24-bit prefix in off-chip DRAM. Note that we need 256 element trie nodes for these rare cases because the expansion of a 24-bit prefix results in up to 256 32-bit entries.

Because SAIL is also designed to work in software, SAIL starts lookup at length 24. SAIL first determines whether the longest match is exactly 24 bits, less than 24 bits, or possibly greater than 24 bits. If the match is 24 bits, SAIL terminates by a lookup into the 24-bit next hop table in DRAM. If the match is less than 24 bits, search continues sequentially from 23 to 1, trying each bitmap on-chip until the longest match length is found at say W; this is followed by a lookup to the W length hash table in DRAM. Finally, if the match is greater than 24 (this should occur rarely), SAIL uses the last 8 bits of the IP address to index into the corresponding multibit trie node in DRAM corresponding to the first 24-bits of the address.

The key mechanism then is to efficiently implement a predicate that determines whether the longest match is equal to 24, < 24, or possibly greater than 24. This is trickier than it may seem. Consider the conceptual unibit trie level corresponding to the pivot length 24. Let the first 24 bit prefix of the address being looked up be P.

There are four cases to consider. If there is no trie node corresponding to the path P, then the longest match must be less than 24. Second, if there is a trie node but it is a pure pointer, then search must continue at levels greater than 24. But Case 2 can fail as a match is not guaranteed to be found at longer lengths (as in binary search on prefix lengths). The third case is that the corresponding trie node is a stored prefix but not a pointer; but in this case search terminates with a 24-bit match). The fourth case is that the corresponding trie node is both a stored prefix and a pointer.

For example, consider a smaller version of our earlier database with $P4 = 1*$, $P2 = 101*$, $P3 = 11001*$, $P7 = 1000000*$ and a new prefix $P10 = 100*$. If we assume the pivot level is 3, Case 1 occurs say when the first bits of the address are 000, and Case 2 occurs when the first 3 bits are 110 (a pure pointer pointing to $P3$). Further, Case 3 occurs when the first 3 bits being matched are 101 (they match $P2$ and no longer prefix. Finally, Case 4 occurs when the first 3 bits being matched are 100 (they match $P10$ and also potentially the longer match $P7$.

A single bitmap at the pivot level cannot distinguish these four cases because a single indexed bit has only 2 possible values. Thus SAIL uses an idea called *pivot pushing* to reduce four cases to three cases. For example, case 4 (which should be rare) can be eliminated by pushing the stored prefix to length 25 by expansion; thus, the prefix can be stored in an off-chip multibit trie node. That still leaves 3 cases. Case 1 is distinguished by a corresponding 0 in the 24-bit on-chip bitmap. However, Case 2 and Case 3 both correspond to a 1 in the bitmap. They are distinguished by one more lookup to the corresponding next hop array of Length 24 bit prefixes. If there is a match (Case 3), search terminates with a 24-bit match.

If there is no match in the 24-bit next hop array and there is a 1 in the corresponding bit in the Length 24 bitmap, we are in Case 2 (pure pointer) and search must proceed to an off-chip multibit trie node. But as in binary search on prefix lengths this can cause us to hare off on a wild goose chase when the best match is actually at lengths less than 24. SAIL fixes this problem as usual using precomputation. The longest match corresponding to the pointer is pushed to Level 25 in case no other longer match works. Again just as in Case 4, such pushing should be done rarely, because prefixes greater than 24 are rare.

The overall performance of SAIL in a hardware model requires 2 DRAM accesses in the (rare) worst case for a match that is longer than 24-bits. This is because it requires a DRAM access into the

24-bit next hop table (to distinguish Case 2 and Case 3) followed by a lookup into the multibit trie node (for Case 2).

While SAIL was possibly designed in this way to make the software implementation fast, we have already seen that DXR is likely to outperform SAIL on modern CPUs because of its smaller cache footrpint. By contrast, SAIL requires 4 Mbytes of cache. Thus the SAIL ideas seem more relevant for hardware settings with on-chip memory and cheap and large off-chip DRAM. If that is indeed the setting, then SAIL can be simplified as follows.

First, we can segregate the DRAM into 2 separate parallel banks that can be looked up in parallel. The first DRAM bank can be used to store next hops for prefixes of all lengths strictly less than or equal to 24, while a second bank can contain all the multibit trie nodes for prefixes of lengths greater than 24. If these two lookups can be done in parallel in one DRAM access time there is no need for pivot pushing. The bit map corresponding to Length 24 is treated like the other bit maps of length < 24, with bits set only for valid prefixes of that length (and not for pointers to greater length prefixes as in vanilla SAIL). Instead, a hardware thread searches through all the bitmaps on-chip of lengths less than or equal to 24 looking for the longest match length; a second parallel thread looks up the multibit trie node (if it exists) corresponding to the full 32 bits, using the first 24 bits as a key. It is easy to combine all the results since Thread 2 results (if any) are more specific than Thread 1 results.

Finally, all the next hop tables can be indexed using say d-left hashing described in the chapter on exact matching (more specifically in Section 10.3.3). The resulting hardware requires less DRAM (at most the size of the prefix database instead of the indexed arrays used in the SAIL paper (Yang et al., 2014). It is also easier and faster to add and delete prefixes because the pivot pushing has been eliminated. This simplification was probably not considered in the SAIL paper (Yang et al., 2014) because of the need to do a software implementation. However, given the existence of DXR, a simplified SAIL seems most useful in a pure hardware setting with on-chip SRAM and off-chip DRAM.

11.13 Memory allocation in compressed schemes

With the exception of binary search and fixed-stride multibit tries, many of the schemes described in this chapter need to allocate memory in different sizes. Thus if a compressed trie node grows from two to three memory words, the insertion algorithm must deallocate the old node of size 2 and allocate a new node of size 3. Memory allocation in operating systems is a somewhat heuristic affair, using algorithms, such as best-fit and worst-fit, that do not guarantee worst-case properties.

In fact, all standard memory allocators can have a worst-case fragmentation ratio that is very bad. It is possible for allocates and deallocates to conspire to break up memory into a patchwork of holes and small allocated blocks. Specifically, if Max is the size of the largest memory allocation request, the worst-case scenario occurs when all holes are of size $Max - 1$ and all allocated blocks are of size 1. This can occur by allocating all of the memory using requests of size 1, followed by the appropriate deallocations. The net result is that only $\frac{1}{Max}$ of memory is guaranteed to be used, because all future requests may be of size Max.

The allocator's use of memory translates directly into the maximum number of prefixes that a lookup chip can support. Suppose that—ignoring the allocator—one can show that 20 MB of on-chip memory can be used to support 640,000 prefixes in the worst case. If one takes the allocator into account and $Max = 32$, the chip can guarantee supporting only 20,000 prefixes!

Matters are not helped by the fact that CAM vendors at the time of writing were advertising a worst-case number of 100,000 prefixes, with 10-nanoseconds search times and microsecond update times. Thus, given that algorithmic solutions to prefix lookup often compress data structures to fit into SRAM, algorithmic solutions *must* also design memory allocators that are fast and that minimally fragment memory.

There is an old result (Robson, 1974) that says that *no allocator* that does not compact memory can have a utilization ratio better than $\frac{1}{\log_2 Max}$. For example, this is 20% for $Max = 32$. Since this is unacceptable, *algorithmic solutions involving compressed data structures must use compaction*. Compaction means moving allocated blocks around to increase the size of holes.

Compaction is hardly ever used in operating systems for the following reason. If you move a piece of memory M, you must correct all pointers that point to M. Fortunately, most lookup structures are trees, in which any node is pointed to by at most one other node. By maintaining a *parent* pointer for every tree node, nodes that point to a tree node M can be suitably corrected when M is relocated. Fortunately, the parent pointer is needed only for updates and not for search. Thus the parent pointers can be stored in an off-chip copy of the database used for updates in the route processor, without consuming precious on-chip SRAM.

Even after this problem is solved, one needs a simple algorithm that decides when to compact a piece of memory. The existing literature on compaction is in the context of garbage collection (e.g., Refs. Wilson, 1992; Lai and Baker, 1996) and tends to use *global* compactors that scan through all of the memory in one compaction cycle. To bound insertion times, one needs some form of *local* compactor that compacts only a small amount of memory around the region affected by an update.

11.13.1 Frame-based compaction

To show how simple local compaction schemes can be, we first describe an extremely simple scheme that does minimal compaction and yet achieves 50% worst-case memory utilization. We then extend this to improve utilization to closer to 100%.

In *frame merging*, assume that all M words of memory are divided into $\frac{M}{Max}$ frames of size Max. Frame merging seeks to keep the memory utilization to at least 50%. To do so, all nonempty frames should be at least 50% full. Frame merging maintains the following simple invariant: All *but one* unfilled frame is at least 50% full. If so, and if $\frac{M}{Max}$ is much larger than 1, this will yield a guaranteed utilization of almost 50%.

Allocate and deallocate requests are handled (Sikka and Varghese, 2000) with the help of tags added to each word that help identify free memory and allocated blocks. The only additional restriction is that all holes be contained *within* a frame; holes are not allowed to span frames.

Call a frame *flawed* if it is nonempty but is less than 50% utilized. To maintain the invariant, frame merging has one additional pointer to keep track of the current flawed frame, if any. Now, an allocate could cause a previously empty frame to become flawed if the allocation is less than $\frac{Max}{2}$.

Similarly, a deallocate could cause a frame that was filled more than 50% to become less than 50% full. For example, consider a frame that contains two allocated blocks of size 1 and size $Max - 1$ and hence has a utilization of 100%. The utilization could reduce to $\frac{1}{Max}$ if the block of $Max - 1$ is deallocated. This could cause two frames to become flawed, which would violate the invariant.

A simple trick to maintain the invariant is as follows. Assume there is already a flawed frame F and that a new flawed frame, F', appears on the scene. The invariant is maintained by merging the contents

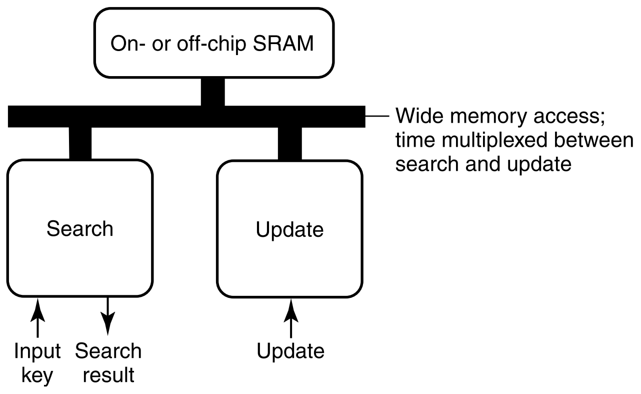

On- or off-chip SRAM
Wide memory access; time multiplexed between search and update
Search
Update
Input key
Search result
Update

FIGURE 11.17

Model of a lookup chip that does a search in hardware using a common SRAM that could be on or off chip. In some cases, the external memory is cheap low-latency DRAM

of F and F' into F. This is clearly possible because both frames F and F' were less than half full. Note that the only compaction done is *local* and is limited to the two flawed frames, F and F'. Such local compaction leads to fast update times.

The worst-case utilization of frame merging can be improved by increasing the frame size to $kMax$ and by changing the definition of a flawed frame to be one whose utilization is less than $\frac{k}{k+1}$. The scheme described earlier is a special case with $k = 1$. Increasing k improves the utilization, at the cost of increased compaction. More complex allocators with even better performance are described in Sikka and Varghese (2000).

11.14 Fixed Function Lookup-chip models

Classically for Terabit speeds, lookup schemes have been implemented on chips rather than on network processors. We start in Fig. 11.17 by describing a model of a lookup chip that does search and update, and then describe an alternate model for programmable processors such as Intel's (aquired from Barefoot) Tofino-3 (Intel Corporation, 2022). The lookup chip (Fig. 11.17) has a Search and an Update process, both of which access a common memory that is either on or off chip (or both). The Update process allows incremental updates and (potentially) does a memory allocation/deallocation and a small amount of local compaction for every update.

The actual updates can be done either completely on chip, partially in software, or completely in software. If a semiconductor company wishes to sell a lookup chip using a complex update algorithm (e.g., for compressed schemes), it may be wiser also to provide an update algorithm in hardware. If the lookup chip is part of a forwarding engine, however, it may be simpler to relegate the update process completely to a separate CPU on the line card.

The external memory could be SRAM as in Fig. 11.17 but is more likely to be cheaper DRAM today. Each access to external memory can be fairly wide if needed, even up to 1000 bits. This is quite feasible today using a wide bus. The search and update logic can easily process 1000 bits in parallel in one memory cycle time. Recall that wide word accesses can help, for example, in the tree bitmap and binary search on values schemes, to reduce search times.

Search and Update use time multiplexing to share access to the common memory that stores the lookup database. Thus the Search process is allowed S consecutive accesses to memory, and then the Update process is allowed K accesses to memory. If S is 20 and K is 2, this allows Update to steal a few cycles from Search while slowing down Search throughput by only a small fraction. Note that this increases the latency of Search by K memory accesses in the worst case; however, since the Search process is likely to be pipelined, this can be considered a small additional pipeline delay.

The chip has pins to receive inputs for Search (e.g., keys) and Update (e.g., update type, key, result) and can return search outputs (e.g., result). The model can be instantiated for various types of lookups, including IP lookups (e.g., 32-bit IP addresses as keys and next hops as results), bridge lookups (48-bit MAC addresses as keys and output ports as results), and classification (e.g., packet headers as keys and matching rules as results).

Each addition or deletion of a key can result in a call to deallocate a block and to allocate a different-size block. Each allocate request can be in any range from 1 to Max memory words. There is a total of M words that can be allocated. The actual memory can be either off chip, on chip, or both. Clearly, even off-chip solutions will cache the first levels of any lookup tree on chip. On-chip memory is attractive because of its speed and cost. Unfortunately, on-chip memory is limited by current processes which makes it difficult to support databases of 1 million prefixes without external memory, whether DRAM or SRAM.

Internally, the chip is typically heavily pipelined. The lookup data structure is partitioned into several pieces, each of which is concurrently worked on by separate stages of logic. Thus the internal SRAM will likely be broken into several smaller SRAMs that can be accessed independently by each pipeline stage.

There is a problem (Sikka and Varghese, 2000) with statically partitioning SRAM between pipeline stages, because memory needs for each stage can vary as prefixes are inserted and deleted. One possible solution is to break the single SRAM into a fairly large number of smaller SRAMs that can be dynamically allocated to the stages via a partial crossbar switch. However, designing such a crossbar at very high speeds is not easy although such an approach was used in the Procket router (Chung et al., 2004).

All the schemes described in this chapter can be pipelined because they are all fundamentally based on trees. All nodes at the same depth in a tree can be allocated to the same pipeline stage.

11.15 Programmable Lookup Chips and P4

Many routers use fixed function lookup chip models similar to those in the last section, where a limited amount of programmability can be done in firmware. For example, for the wide area Arista 7500R3/7800R3 and 7280R3 routers that scale to more than 2.5 million routes, their documentation (Arista Corporation, 2010) states that "internally FlexRoute uses an algorithmic approach to performing lookups".

By contrast, in recent years, chips, such as Intel's Tofino-3 (Intel Corporation, 2022), have emerged that have a fairly large amount of TCAM and are programmable using higher level languages such as

P4. We will explore P4 and the use of CAMs briefly in this section. For example, Intel's Aurora 710, based on Intel Tofino 3.2T switching silicon, claims (Intel Corporation, 2022) to allow data centers to increase the IP routing table size to 1.2M. Further, the TCAM is distributed among a set of physical stages that can be programmed using P4.

The Tofino-3 is an example of what is called the Reconfigurable Match Table (RMT) (Bosshart et al., 2013) approach to programmable network processors. Briefly, the RMT approach is a generalization of the fixed function chip described in the last section which is internally pipelined, and where each stage has access to on-chip SRAM and is devoted to a single function. By contrast, in the RMT architecture the chip internally has a large number (say 32) of stages that are *anonymous* (not devoted to any function) and *programmable* (they can be programmed to perform basic functions on packet headers). Further, each stage has both RAM and CAM. In fact, the CAM in say the Tofino-3 is so plentiful that it can support a large number of routes without any further algorithmic approaches.

As a packet flows through the RMT chip, each packet header is streamed through the stages with successive packet headers following in lockstep to keep the pipeline full. While the SRAM and the CAM pages are divided among the physical stages, a single logical stage of processing (e.g., a level of processing in a tree) can get more memory by being allocated more physical stages.

Each physical stage can be programmed not just in firmware by internal experts but by *network operators* in the field using a higher language called P4 (though our experience is that P4 programming is also somewhat esoteric). This field-programmability is similar to the programmability offered by Field Programmable Gate Arrays (FPGAs). Unlike FPGAs, chips like Tofino-3 are much faster. They gain speed, however, by offering limited programmabilty, using the P4 language that we now explore.

11.15.1 The P4 language

P4 (P4 Open Source Programming Language) is a high-level language for "programming protocol-independent packet processors". P4's versatility and flexibility change the way network functions are implemented at high speeds. P4 offers:

1. *Reconfigurability*: Even after the P4 program is deployed to a reconfigurable hardware router, an operator can modify the programs in the field as desired, unlike traditional fixed function ASICs.
2. *Protocol independence*: The P4 language can specify how the processor deals with packet headers, regardless of what protocols are used. In particular, while P4 can support IP headers it is not limited to IP. It thus generalizes the SDN approach by allowing custom headers and not just programmable routes.
3. *Target independence:* The same P4 program can be deployed to a variety of hardware, such as CPUs, FPGAs, and programmable processors such as Tofino-3.

P4 needs to allow reconfigurability and yet allow Terabit implementations that are as fast as the ones implemented using custom ASICs in the past. Traditional FPGAs are fully reconfigurable but at least an order of magnitude slower than custom ASICs. P4 resolves this quandary by allowing *limited reconfigurability* that suffices for network implementations.

P4 does this by first *allowing new headers* to be defined using a programmable parser. If, for example, an organization wants to add a new security tag (say STAG) between the Ethernet and IPv4 headers, P4 allows the operator to define the STAG length and fields, and place it between the Ethernet and IPv4 headers. Second, P4 defines a way to *process new headers* where each header is processed by a Table, which can be indexed either by exact matching or with wildcarded bits (thus abstracting across exact match, longest matching prefix and ACL lookup). In a running example, the STAG field could be processed by an exact match table that drops certain values of the STAG that are in the table.

Finally, P4 requires specifying *a control flow graph that chains together the processing of tables* to describe the overall processing of a packet based on its header fields. Thus in the above example, the Control Flow Graph would specify that the header is processed by the STAG table after Ethernet, after which the header is sent to the IPv4 lookup table.

In summary, a P4 program has three major parts: a header specification for parsing, a specification of the tables used to process headers, and a control flow graph that describes the "main" program which specifies how control flows between tables.

What makes P4 attractive is, as we said earlier, the RMT (Bosshart et al., 2013) architecture that provides a set of generic match-action stages arranged in a linear pipeline. The wires between stages are also short unlike traditional FPGA interconnects. Each stage is furnished with a sufficient amount of CAM and RAM and multiple parallel processors. The programmer or compiler can then assign multiple physical pipeline stages (or even a fraction of a stage) to logical tables defined in the control flow graph.

Note that P4 only offers limited reconfurability. Some examples of things that cannot be programmed by P4 include queueing disciplines (the Tofino-3, does, however offer a palette of queue disciplines), stateful processing (e.g., NAT) and content processing (e.g., Intrusion detection, detecting content signatures). A great deal of recent research has appeared to tackle some of those issues. For example, packet transactions (Sivaraman et al., 2016b) attempts to allow programmable stateful processing and schemes like PIFO (Sivaraman et al., 2016a) attempt to allow programmable queue disciplines. We will consider these schemes in the chapter on packet scheduling.

Despite these limitations, P4 appears to be popular. Some important use cases are as follows. First, simply reallocating resoures can be useful. In a core router there is more need for forwarding tables and less for ACLs, while the reverse is true for an edge router. Today's vendors make distinct hardware products for each; instead a single P4 router can simply be programmed to provide specific routers for various market segments. Second, adding new headers is often very time consuming. For example, VXLAN (Mahalingam et al., 2020) took years to be added to custom ASICs; but this could be done in a few hours on a P4 router. Finally, there are new applications such as measurement and security that could be programmed in enterprise-specific ways. We will consider some of these in the chapters on measurement and security.

11.15.2 IP Lookups in the P4 Model

At first glance, chips like the Tofino-3 that use the RMT model trivialize IP lookup because they have plenty of CAM. However, the amount of CAM in these chips do not scale to wide area databases. Could we do better by combining algorithmic methods with TCAM?

For instance, MashUp (Rios and Varghese, 2022) is a a "mash up" of algorithmic and hardware techniques that uses a tree of TCAM and SRAM blocks. While a tree of CAMs has been used for reducing power consumption or update costs (e.g., CoolCAM (Zane et al., 2003) and TreeCAM (Vamanan and Vijaykumar, 2011)), MashUp focuses instead on reducing TCAM bits. It can also easily be implemented in modern reconfigurable pipeline chips such as Tofino-3 (Intel Corporation). As a consequence, MashUp can extend the reach of chips like Tofino-3 to the backbone IPv4 and IPv6 databases, or to much larger data centers than is possible using today's solution of a single logical TCAM.

Mashup is based on a technique called “tiling trees”. In other words, Mashup takes into account the internal grain sizes of TCAMs when building a lookup tree. Note that TCAMs come in units, with a grain size of W (width) by D (depth) like memory pages. The Tofino-3, for example, has $W = 44$ and $D = 512$. Each unit TCAM or block in Tofino-3 can do a longest match on up to 512 prefixes of up to 44 bits (any of which can be the wilcard *) in a single clock cycle. For example, to fit the current IPv6 database size of 150,000 prefixes of up to 64 bits, the straightforward approach is to stitch together 2 TCAM blocks horizontally (64<88) and 300 of these pairs vertically ($150,00/512$ is approximately 300).

By contrast, MashUp builds a tree of TCAM blocks starting with a root which can fit (in other words is “tiled”) into a TCAM block. This breaks up the bit-by-bit branching tree (trie) of prefixes into subtrees which we can recursively tile. To pick the tree level at which to define the root TCAM, observe that the tree has pointers, which represent “overhead” not present in a single logical CAM.

The first idea is to reduce pointer overhead by cutting the trie into subtries at heights (which they call *lean lengths*) where the number of downstream pointers are small. For example, the paper (Rios and Varghese, 2022) shows that for a public wide area IPv6 database, the pointer overhead is only 0.41% of total database size at height 20 but rises to 6.84% at height 32.

The second idea is to reduce the wasted space of tiled subtrees by packing up to D subtrees in a single TCAM block. Because the remaining prefix bits may repeat across different subtrees, this requires adding a disambiguating tag of $\log_2 D$ bits. Despite this additional cost, packing ensures that any wasted space (which can be as large as a unit TCAM) in the last block is amortized over D subtrees. In other words, Mashup packs subtrees into a single TCAM block to reduce internal fragmentation.

The third idea is to do a “currency exchange” where “nearly full” subtrees are replaced by SRAM pages in a process the authors call “RAM hybridization”. Since SRAM cannot do variable length matching, this must be remedied by “expanding” variable length prefixes to the maximum length prefix in the subtree. Hybridization allows currency “arbitrage” because SRAM pages are cheaper and more plentiful than TCAM blocks (3 to 1 in Tofino-3).

The Mashup paper shows results for IPv4 database sizes of 900k prefixes using all three techniques: lean lengths, tag aggregation and RAM hybridization. A four stage tree with strides of 16-4-4-8 reduces TCAM bits by $7\times$ compared to the straightforward solution of using a single logical CAM, at the cost of around 1000 SRAM pages, which is much less than the RAM required for an all RAM solution. More details can be found in the Mashup paper (Rios and Varghese, 2022).

11.16 Conclusions

It is important to gain some perspective after the large number of isolated lookup variants described in this chapter. Thus we conclude with a summary of the state of the art in lookups, and a survey of the common principles used in their design.

State of the Art in Lookups: Lookup schemes are coming under severe pressure in core routers as both table sizes (up to 1 million prefixes) and speed (several Terabits of aggregate throughput) ratchet upwards. MPLS, once thought to be a way to finesse lookups, is now mostly used to avoid packet classification for traffic engineering purposes. CAMs are nibbling away at even the core router space with chips like Tofino-3, but still do not scale to wide area database sizes (though techniques like Mashup Rios and Varghese (2022) can remedy that). Thus many core router vendors such as Arista (Arista Corporation, 2010) still use and design algorithmic schemes for lookups based on RAM.

For hardware schemes without CAM at Terabit speeds in the core, Tree Bitmap still seems appropriate even after so many years. However, there may be patents that restrict the use of Tree bitmap. This may make other bitmap schemes such as Poptrie (Asai and Ohara, 2015) attractive. Tree bitmap was used in Cisco's CRS-1 Router. On the other hand, ideas based on simplified SAIL (Yang et al., 2014) (as described earlier) may be appropriate for hardware settings that use cheap and plentiful off-chip DRAM.

For data centers and enterprise networks, there seems to be adequate CAM in chips like the Tofino-3, especially when leveraged (as in schemes like Mashup (Rios and Varghese, 2022)) to reduce power and TCAM bits.

For software implementations, the DXR scheme (Zec et al., 2012) seems to be the best approach today especially in the context of Network Function Virtualization (NFV). The underlying binary-search-on-ranges scheme allows efficient multithreading whose throughput scales with the number of cores and fits into the L1 cache. It is also unencumbered by patents.

Finally, binary search on prefix lengths is attractive because of its scaling properties to large address lengths. Unfortunately, its use of hashing makes it hard to guarantee lookup times. It is, however, used by a few vendors in software implementations. It may be a contender in the future as IPv6 becomes more dominant.

The bottom line is that algorithmic solutions together with pipelining can scale with link speeds as long as SRAM speeds scale to match packet arrival times. All the schemes studied in this chapter can be pipelined to provide one lookup per memory access time. The choice between CAMs and algorithmic schemes will continue to be hard to quantify and will probably be made on an ad hoc basis for each product.

Fundamentally, if compressed trie schemes can use less than 32 bits per prefix, compressed tries can use fewer transistors and less power than CAMs. This is because in a CAM the lookup logic is distributed in each of N memory *cells*, whereas in an algorithmic solution the lookup logic, albeit more complicated, is distributed among a small, constant number of *stages*. A careful VLSI scaling analysis of these two approaches would be very useful.

Underlying Principles: Although this is a chapter about lookups and thinking about lookups requires paying attention to current market trends, it is important not to forget that this is a book about underlying principles. It is plausible that routers in the misty future may use all-optical switches and all-optical processing, even for lookups. In that case, the specific algorithms described in this chapter may be discarded; but perhaps the underlying design principles will remain.

All of the schemes described in this chapter start with the algorithmic principle of divide and conquer (divide by bits in the address, address ranges, or prefix lengths) but gain efficiency by other principles. First, most schemes use precomputation, which trades slower insert/delete times for fast search times. The schemes also exploit hardware features such as wide memories, leverage fast and slow memories, trade memory for time, and optimize the degrees of freedom in a given design. Table 11.1 summarizes some of the schemes and the principles used in them. Many of them also use what we could call *information-preserving transformations*. For example, replacing a sequence of one-way branches with a text string, or representing a prefix as a start and end of range.

Finally, this chapter cannot hope to do justice to all the interesting IP lookup schemes that have been published in the academic and patent literature. You can look it up.

11.17 Exercises

1. **Caching Prefixes:** Suppose we have the prefixes 10*, 100*, and 1001*. Hugh Hopeful would like to cache prefixes instead of entire 32-bit addresses. Hugh's scheme keeps a set of prefixes in the cache (fast memory), in addition to the complete set of prefixes in slow memory. Hugh's scheme first does a best-matching-prefix search in the cache; if a matching prefix is found, the next hop of the prefix is used. If no matching prefix is found, a best-matching-prefix search is done for the entire database and the resulting prefix cached. Periodically, prefixes that have not been matched for a while are flushed from the cache. Alyssa P. Hacker quickly gives Hugh a counterexample to show him that his scheme is flawed and that caching prefixes are tricky (if not impossible). Can you?
2. **Encoding Prefixes in a Constant Length:** We said in the text that encoding prefixes like 10*, 100*, and 1000* in a fixed length could not be done by padding prefixes with zeroes. It clearly can be done by padding with zeroes and adding an encoding of the prefix length. We want to study a more efficient method.
   - How many possible prefixes on 32 bits can there be?
   - Show how to encode all such prefixes using a fixed length of 33 bits. Make sure that 10*, 100*, and 1000* encode to different values.
   - Can you use this fixed-length encoding of prefixes to have the multiple hash tables used in Section 11.11 be packed into a single hash table? Why might this help to decrease the chances of hash collisions for a given memory size?
3. **Quantifying the Benefits of Compressing One-Way Branches:**
   - For a unibit trie that does not compress one-way branches, show that the maximum number of trie nodes can be $O(N \cdot W)$, where N is the number of prefixes and W is the maximum prefix length. (*Hint*: Generate a trie that uses $\log_2 N$ levels to generate N nodes, and then hang a long string of $N - W$ nodes from each of the N nodes.)
   - Show that a unibit trie with text strings to compress one-way branches can have at most $2N$ trie nodes and $2N$ text strings.
   - Extend your analysis to multibit trie nodes with a fixed stride. How would you implement text string compression in such tries?
4. **Controlled Prefix Expansion:** Code up an efficient algorithm that expands a set of prefixes to any target set of lengths $L_1, \ldots, L_k$. Check your algorithm using the sample database of Fig. 11.6. What is the complexity of your algorithm?
5. **Optimal Variable-Stride Trie:** Prove that the varied-stride trie of Fig. 11.8 is optimal for a trie height of 2. Use the recursive formulation shown in the text.
6. **Reducing Memory References in Lulea:** The naive approach to counting bits shown in Fig. 11.13 should take three memory references (to access *numSet*, to read the appropriate chunk of the bitmap, and to access the compressed trie node for the actual information.) Show how to use **P4a** to combine the first two accesses into a single access.
7. **Next Node versus Leaf Pushing in Lulea:** Before we applied Lulea compression, we first leaf pushed the expanded trie of Fig. 11.7. The motivation was to make every entry either a pointer or a prefix but not both. Suppose we have a special prefix entry at the top of every trie node; if any

entry in a trie node has pointer p and prefix P, we push P to the top of the node pointed to by p. Thus we would push the prefix P8 in the 100 entry of the root of Fig. 11.7 to the top of the rightmost trie node.

- We cited leaf pushing as one of the reasons for slow insertion times in the Lulea scheme. Does next-node pushing allow incremental insertion for the Lulea scheme?
- How would you modify trie search to take into account the fact that prefixes can be stored at the top of (potentially large) trie nodes? How would this increase the search time (in memory accesses) of the Lulea scheme?

8. **CAM Node Compression:** Instead of using the Lulea scheme for compression, we could just store all the prefixes within a trie node without expansion. If we use small trie nodes (3- or 4-bit strides), a chip can potentially read all the entries in a node and internally do a comparison to find the best-matching prefix within the node. Describe the details of such a scheme.
9. **Tree Bitmap Algorithm:** The tree bitmap algorithm described in the text requires rooting through the internal prefix bitmap to decide if there was a matching prefix at a trie node N before moving on. This requires a greater access width (to access the internal prefix bitmaps) and more time. Consider adding state to the next node in the search path (**P12**) and one more final memory access to avoid this overhead.
10. **Multicolumn Binary Search:** In Chapter 4 we saw how to efficiently use binary search when the identifiers were wide. Explain how to combine this idea with that of binary search on prefixes explained in this chapter in order to do IPv6 lookups (up to 128-bit prefixes). How does this scheme compare with the other schemes in terms of lookup performance for IPv6?
11. **Binary Search with Fast Incremental Updates:** (This is difficult.) Find a way to remove all the problems of updates to binary search. The key problem is that if a large prefix range R contains lots of disjoint prefix ranges $R_1, \dots R_k$, then the spaces between the ranges R_k must be precomputed to map to R. If we now add a new prefix range, R', that is contained in R but still contains R_1 through R_k, then all the spaces between the ranges R_k must be changed to map to the new range, R'. Since k can be $O(n)$, this could lead to a $O(n)$ update. Try to avoid this problem by storing the binary search database as a tree and storing information about precomputed prefixes that cover the space between ranges as high as possible in the tree, as opposed to storing in the leaves. Details can be found in Warkhede et al. (2001).
12. **Counterexamples for Binary Search on Prefix Lengths:** Even in industry, it is often useful to show by counterexample that worst cases can actually exist. This ensures that we are not doing unnecessary work, and it also silences people who say that the worst case will never be too bad. Imagine that Hugh Hopeful is working for the same startup building an IP lookup chip. The company is now considering using binary search on prefix lengths.

- Suppose we use only markers and no precomputation. This would make insertion a lot faster. Hugh Hopeful suggests that backtracking can only lead to a logarithmic number of extra accesses. Find an example that leads to linear time.
- Hugh Hopeful finds that in practice real databases add only 25% extra marker storage, much less than the $\log_2 W$ multiplicative factor that we claimed. This is important because he would like to boast of a larger number of prefixes that his chip can handle for the given amount of memory. Give a worst-case example to show that we can add $\log_2 W$ entries per marker.

13. **Rope Search:** Binary search on prefix lengths can be improved by what is called *rope search* in Waldvogel et al. (1997). If we ever get a match with some entry M at length m, we only search further among the set of lengths corresponding to prefixes that are extensions of M. The basic technique we studied earlier will continue to search among all lengths greater than m in the current set of lengths R. However, many of the lengths $l > m$ may not have a prefix that is an extension of M. Thus this optimization can result in more than halving the set of possible lengths on each match. It may not help the worst case, but it can considerably help the average case. Try to work out the details of such a scheme. In particular, a naive approach would keep a list of all potentially matching lengths ($O(W)$ space, where W is the length of an address) with each prefix. Find a way to reduce the state kept with each marker to $O(\log W)$. Details can be found in Waldvogel et al. (1997).
14. **Invariant for Binary Search on Prefix Ranges:** Designing and proving algorithms that correct via invariants is a useful technique even in network algorithmics. The standard invariant for ordinary binary search when searching for key K is: "K is not in the table, or K is in the current range R." Standard binary search starts with R equal to the entire table and constantly halves the range while maintaining the invariant. Find a similar invariant for binary search on prefix ranges.
15. **Semiperfect Hashing:** Hardware chips can fetch up to 1000 bits at a time using wide buses. Exploit this observation to allow up to X collisions in each hash table entry, where the X colliding entries are stored at adjacent locations. Code up a perfect hashing implementation (of 1000 IP addresses using a set of random hash functions), and compare the amount of memory needed with an implementation based on semiperfect hashing.
16. **Removing Redundancies in Lookup Tables:** Besides the use of compressed structures, another technique to reduce the size of IP lookup tables (especially when the tables are stored in on-chip SRAM) is to remove redundancy. One simple example of redundancy is when a prefix P is longer than a prefix P' and they both have the same next hop. Which prefix can be removed from the table? Can you think of other examples of removing redundancy? How would you implement such compression? Draves et al. (1999) describe a dynamic programming algorithm for compression, but even simpler alternatives can be effective.
17. **Alternative SAIL implementation:** Since there are 4 cases considered in the description of SAIL, the basic SAIL algorithm uses pivot pushing and an extra lookup to the netx hop array at the pivot level to manage with a bitmap (that has only 1 bit and hence 2 possibilities for each bit). Suppose we use 2 bits for each possible prefix P at the pivot length. Can we avoid pivot pushing? What are the tradeoffs in terms of on-chip memory, speed of lookup and insertion costs.
18. **Implementing Lookups in P4:** Go to https://github.com/p4lang/ and download the latest P4 compiler and behavioral model and Mininet if necessary. Implement the DXR, SAIL, Tree bitmap, and Mashup Algorithms and compare them. Can CAM be used to simplify or make more efficient the SAIL and DXR algorithms? Compare these algorithms with respect to a P4 implementation both for IPv4 and IPv6.
19. **Implementing Tries for Best Matching Prefix:** (Due to V. Srinivasan.) The problem is to use tries to implement a file name completion routine in C or C++, similar to ones found in many shells. Given a unique prefix, the query should return the entire string. For example, with the words *angle*, *epsilon*, and *eagle*: Search(a) should return *angle*, Search(e) should return "No unique completion," Search(ea), Search(eag), etc. should return *eagle*; and Search(b) should return "No matching entries found." Assume all lowercase alphabets. To obtain an index into a trie array, use:

```
index= charVariable - ‘a’.
```

The following definition of a trie node may be helpful.

```
\#defineALPHA26
structTRIENODE
{
intcompletionStatus;
charcompletion[MAXLEN];
structTRIENODE*next[ALPHA];
}
```

Can other techniques discussed in the text (e.g., binary search) be applied to this problem? Are insertion costs significant?