Routing Lookups in Hardware at Memory Access Speeds

Pankaj Gupta, Steven Lin, and Nick McKeown
Computer Systems Laboratory, Stanford University
Stanford, CA 94305-9030
{pankaj, sclin, nickm}@stanford.edu

Abstract

Increased bandwidth in the Internet puts great demands on network routers; for example, to route minimum sized Gigabit Ethernet packets, an IP router must process about 1.5×10^6 packets per second per port. Using the "rule-of-thumb" that it takes roughly 1000 packets per second for every 10^6 bits per second of line rate, an OC-192 line requires 10×10^6 routing lookups per second; well above current router capabilities. One limitation of router performance is the route lookup mechanism. IP routing requires that a router perform a longest-prefix-match address lookup for each incoming datagram in order to determine the datagram's next hop. In this paper, we present a route lookup mechanism that when implemented in a pipelined fashion in hardware, can achieve one route lookup every memory access. With current 50ns DRAM, this corresponds to approximately 20×10^6 packets per second; much faster than current commercially available routing lookup schemes. We also present novel schemes for performing quick updates to the forwarding table in hardware. We demonstrate using real routing update patterns that the routing tables can be updated with negligible overhead to the central processor.

1 Introduction

This paper presents a mechanism to perform fast longest-matching-prefix route lookups in hardware in an IP router. Since the advent of CIDR in 1993 [1], IP routes have been identified by a <route prefix, prefix length> pair, where the prefix length is between 0 and 32 bits, inclusive. For every incoming packet, a search must be performed in the router's forwarding table to determine which next hop the packet is destined for. With CIDR, the search may be decomposed into two steps. First, we find the set of routes with prefixes that match the beginning of the incoming IP destination address. Then, among this set of routes, we select the one with the longest prefix. This is the route that we use to identify the next hop.

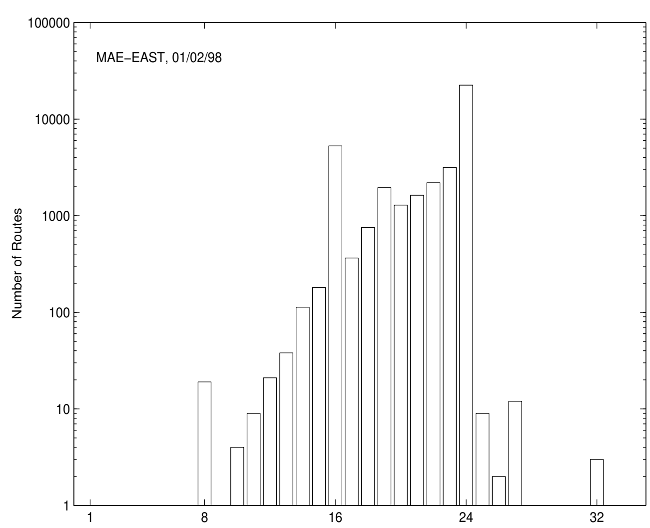
Our work is motivated by the need for faster route lookups; in particular, we are interested in fast, hardware-implementable lookup algorithms. We desire a lookup mechanism that achieves the following goals:

1) The lookup procedure should be easily implementable in hardware using simple logic.
2) Ideally, the route lookup procedure should take exactly one memory access time.
3) If it takes more than one memory access, then (a) the number of accesses should be small, (b) the number of accesses should be bounded by a small value in all cases, and (c) the memory accesses should occur in different physical memories, enabling pipelined implementations (and hence help us achieve goal 2).
4) Practical considerations involved in a real implementation, such as cost, are an important concern.
5) The overhead to update the forwarding table should be small.

The technique that we present here is based on the following assumptions:

1) Memory is cheap. A very quick survey at the time of writing indicates that $16\text{MB} = 2^{24}$ bytes of 60ns DRAM is available for about \$50. The cost per byte is approximately halving each year.
2) The route lookup mechanism will be used in routers where speed is a premium; for example those routers that need to process at least 10 million packets per second.
3) On backbone routers there are very few routes with prefixes longer than 24-bits. This is verified by an examination of the MAE-EAST backbone routing tables [2]. A plot of prefix length distribution is shown in Figure 1; note the logarithmic scale on the y-axis. In this example, 99.93% of the prefixes are 24-bits or less.
4) IPv6 is still some way off — IPv4 is here to stay for the time

Figure 1: Prefix length distributions.



This work was funded by the Center for Integrated Systems at Stanford University. Steven Lin is funded by an NSF Graduate Research Fellowship. Nick McKeown is funded by the Alfred P. Sloan Foundation, Sumitomo Electric Industries and a Robert N. Noyce Faculty Fellowship.

being. Thus, a hardware scheme optimized for IPv4 routing lookups is useful today.

5) There is a single general-purpose processor participating in routing table exchange protocols and constructing a full routing table (including protocol-specific information such as route lifetime, etc. for each route entry). The next hop entries from this routing table are downloaded by the general purpose processor into each forwarding table, which are used to make per-packet forwarding decisions.

In the remainder of the paper we discuss the construction and usage of the forwarding tables, and the process of efficiently updating the tables using the general-purpose processor.

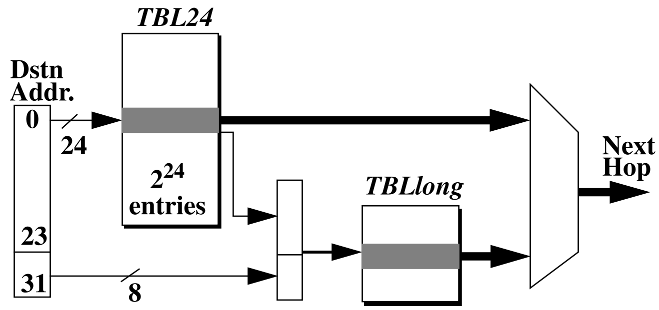


Figure 2: Proposed *DIR-24-8-BASIC* architecture. The next hop result comes from either *TBL24* or *TBLlong*.

2 Previous Work

The current techniques for performing longest matching prefix lookups, for example CAMs [3] and Tries [4], do not seem to be able to meet the goals set forth above. CAMs are generally small (1K x 64 bits is a typical size), expensive, and dissipate a lot of power when compared to DRAM. Tries, in general, have a worst case searching time of 32 memory accesses (for a 32-bit IP address), leading to a wasteful 32-stage pipeline if we desire one lookup per memory access time. Furthermore, if we wish to fully pipeline the design, each layer of the trie needs to be implemented in a different physical memory. This leads to problems because the memory cannot be shared among layers; it could happen that a single layer of the trie exhausts its memory while other layers have free space.

Label swapping techniques, including IP Switching [5] and Multiprotocol Label Swapping (MPLS) [6] have been proposed, to replace the longest-prefix match with a simple direct-lookup based on a fixed-length field. While these concepts show some promise, they also require the adoption of new protocols to work effectively. In addition, they do not completely take away the need for routing lookups.

Recently, several groups have proposed novel data structures to reduce the complexity of longest-prefix matching lookups [7][8]. These data structures and their accompanying algorithms are designed primarily for implementation in software, and cannot guarantee that a lookups will complete in one memory-access-time.

We take a different, more pragmatic approach that is designed for implementation in dedicated hardware. As mentioned in assumption (1), we believe that DRAM is so cheap (and continues to get cheaper), that using large amounts of DRAM inefficiently is advantageous if it leads to a faster, simpler, and cheaper solution. With this assumption in mind, the technique that follows is so simple that it is almost obvious. Our technique allows for an inexpensive, easily pipelined route lookup mechanism that can process one packet every memory-access time when pipelined.

Since the time of writing this paper, we have learned that the lookup technique outlined here is a special case of an algorithm proposed by V. Srinivasan and G. Varghese, described in [9]. However, we take a more hardware-oriented approach with a view to providing more direct benefit to the designers and implementors of routing lookup engines. In particular, we propose a novel technique for performing routing updates in hardware.

The paper is organized as follows. Section 3 describes the basic route lookup technique. Section 4 discusses some variations to the technique which make more efficient use of memory. Section 5 investigates how route entries can be quickly inserted and removed from the forwarding tables, and Section 6 provides a conclusion.

3 Proposed Scheme

We call the basic scheme *DIR-24-8-BASIC* — it makes use of the two tables shown in Figure 2, both stored in DRAM. The first table (called *TBL24*) stores all possible route prefixes that are up to, and including, 24-bits long. This table has 2^{24} entries, addressed from 0.0.0 to 255.255.255. Each entry in *TBL24* has the format shown in Figure 3. The second table (*TBLlong*) stores all route prefixes in the routing table that are longer than 24-bits.

Assume for example that we wish to store a prefix, *X*, in an otherwise empty routing table. If *X* is less than or equal to 24 bits long, it need only be stored in *TBL24*: the first bit of the entry is set to zero to indicate that the remaining 15 bits designate the next-hop. If, on the other hand, the prefix *X* is longer than 24 bits, then we use the entry in *TBL24* addressed by the first 24 bits of *X*. We set the first bit of the entry to one to indicate that the remaining 15-bits contain a pointer to a set of entries in *TBLlong*.

In effect, route prefixes shorter than 24-bits are

Figure 3: *TBL24* entry format

If longest route with this 24-bit prefix is < 25 bits long:

0	Next Hop
1 bit	15 bits

If longest route with this 24 bits prefix is > 24 bits long:

1	Index into 2nd table
1 bit	15 bits

expanded; e.g. the route prefix 128.23/16[†] will have $2^{24-16} = 256$ entries associated with it in *TBL24*, ranging from the memory address 128.23.0 through 128.23.255. All 256 entries will have exactly the same contents (the next hop corresponding to the routing prefix 128.23/16). By using memory inefficiently, we can find the next hop information within one memory access.

TBLlong contains all route prefixes that are longer than 24 bits. Each 24-bit prefix that has at least one route longer than 24 bits is allocated 2^8=256 entries in *TBLlong*. Each entry in *TBLlong* corresponds to one of the 256 possible longer prefixes that share the single 24-bit prefix in *TBL24*. Note that because we are simply storing the next-hop in each entry of the second table, it need be only 1 byte wide (if we assume that there are fewer than 255 next-hop routers — this assumption could be relaxed if the memory was wider than 1 byte).

When a destination address is presented to the route lookup mechanism, the following steps are taken:

1) Using the first 24-bits of the address as an index into the first table *TBL24*, we perform a single memory read, yielding 2 bytes.
2) If the first bit equals zero, then the remaining 15 bits describe the next hop.
3) Otherwise (if the first bit equals one), we multiply the remaining 15 bits by 256, add the product to the last 8 bits of the original destination address (achieved by shifting and concatenation), and use this value as a direct index into *TBLlong*, which contains the next-hop.

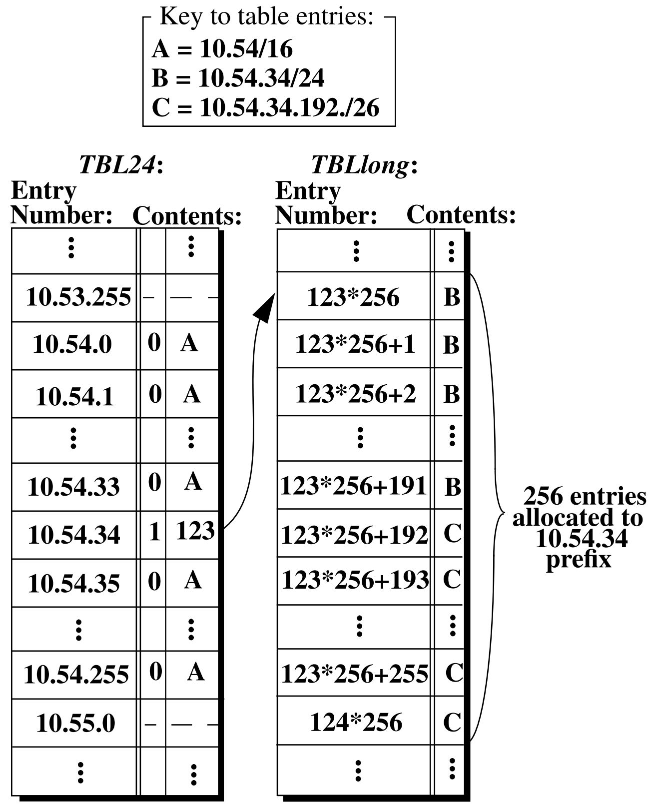


Figure 4: Example of two tables containing three routes.

3.1 Examples

Consider the following examples of how route lookups are performed on the table in Figure 4. Assume that the following routes are already in the table: 10.54/16, 10.54.34/24, 10.54.34.192/26. The first route requires entries in *TBL24* that correspond to the 24-bit prefixes 10.54.0 through 10.54.255 (except for 10.54.34). The 2nd and 3rd routes require that the second table be used (because both of them have the same first 24-bits and one of them is more than 24-bits long). So, in *TBL24*, we insert a one followed by an index (in the example, the index equals 123) into the entry corresponding to the 10.54.34 prefix. In the second table, we allocate 256 entries starting with memory location 123×256. Most of these locations are filled in with the next hop corresponding to the 10.54.34 route, but 64 of them (those from $(123 \times 256) + 192$ to $(123 \times 256) + 255$) are filled in with the next hop corresponding to the 10.54.34.192 route.

Now assume that a packet arrives with the destination address 10.54.22.147. The first 24 bits are used as an index into *TBL24*, and will return an entry with the correct next hop (A). If a second packet arrives with the destination address 10.54.34.23, the first 24 bits are used as an index into the first table, which indicates that the second table must be consulted. The lower 15 bits of the entry (123 in this example) are combined with the lower 8 bits of the destination address, and used as an index into the second table. After two memory accesses, the table returns the next hop (B). Finally, let's assume that a packet arrives with the destination address 10.54.34.194. Again, *TBL24* indicates that *TBLlong* must be consulted, and the lower 15 bits of the entry are combined with the lower 8 bits of the address to form an index into the second table. This time the index an entry associated with the 10.54.34.192/26 prefix (C).

We recommend that the second memory be about 1MByte in size. This is inexpensive and has enough space for 4096 routes longer than 24 bits. (To be precise, we can store 4096 routes longer than 24 bits with distinct 24-bit prefixes.) We see from Figure 1 that the number of routes with length above 24 is much smaller than 4096 (only 28 for this router). Because we use 15 bits to index into the second table, we can, with enough memory, support 32K distinct 24-bit-prefixed long routes with prefixes longer than 24 bits.

As a summary, let's review some of the pros and cons associated with the basic *DIR-24-8-BASIC* scheme.

Pros:

1) Although (in general) two memory accesses are

† Throughout this paper, when we refer to specific examples, a route entry will be written as dotted-decimal-prefix/prefix-length. For example, 10.34.153/24 refers to a 24-bit long route with prefix (in dotted decimal) of 10.34.153.

required, these accesses are in separate memories, allowing the scheme to be pipelined.

2) Except for the limit on the number of distinct 24-bit-prefixed routes with length greater than 24 bits, this infrastructure will support an unlimited number of routes.
3) The total cost of memory in this scheme is the cost of 33 MB of DRAM. No exotic memory architectures are required.
4) The design is well-suited to hardware implementation.
5) When pipelined, 20×10^6 packets per second can be processed with currently available 50ns DRAM. The lookup time is equal to one memory access time.

Cons:

1) Memory is used inefficiently.
2) Insertion and deletion of routes from this table may require many memory accesses. This will be discussed in detail in Section 5.

4 Variations on the theme

There are a number of refinements that can be made to the basic technique. In this section, we discuss two variations that decrease the memory size while adding one or more pipeline stages.

Adding an intermediate "length" table: Observe that, of those routes longer than 24 bits, very few are a full 32 bits. In the basic scheme, we allocated an entire block of 256 entries for each routing prefix longer than 24 bits. For example, if we insert a 26-bit prefix into the table, 256 entries in *TBLlong* are used although only four are required.

We can improve the efficiency of *TBLlong* using a scheme called *DIR-24-8-INT.* In addition to the two tables *TBL24* and *TBLlong*, *DIR-24-8-INT* maintains an additional "intermediate" table, *TBLint*. Basically, by using one additional level of indirection *TBLint* allows us to use a smaller number of entries in *TBLlong*. To do this, we store an i-bit long index (where $i < 15$) value in *TBL24*, instead of the 15-bit value used in the basic scheme. The new index points to an intermediate table (*TBLint*) with 2^i entries as shown in Figure 5; for example, if $i = 12$, *TBLint* contains 4096 entries. Each entry in *TBLint* is pointed to by exactly one entry in *TBL24*, and therefore corresponds to a unique 24-bit prefix. *TBLint* entries contain a 20-bit index into the final table (*TBLlong*), as well as a length field. The index is the absolute memory address in *TBLlong* at which the set of entries associated with this 24-bit prefix begins. The length field indicates the longest route with this particular 24-bit prefix (encoded in three bits since it must be in the range 25-32). The length field also indicates how many entries in *TBLlong* are allocated to this 24-bit prefix. For example, if the longest route with this prefix is a 30-bit route, then the length field will indicate 6 (30-24), and *TBLlong* will have

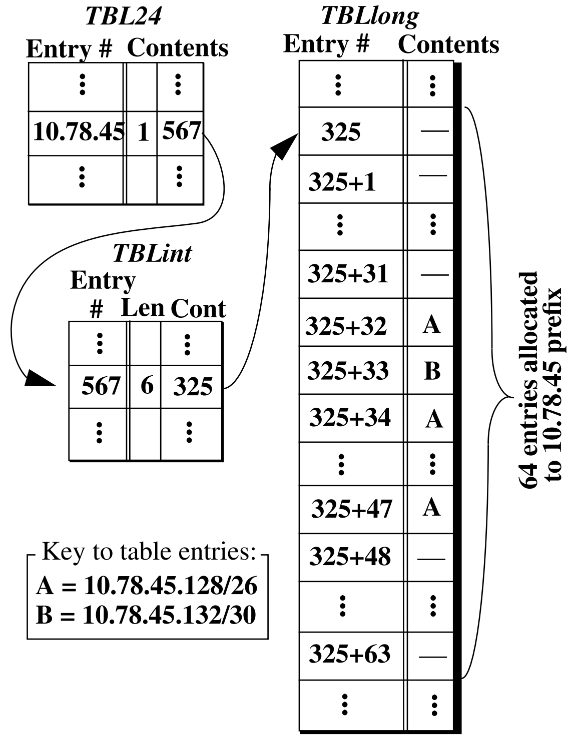


Figure 6: "Intermediate Table" scheme

$2^6 = 64$ entries allocated to this 24-bit prefix.

To clarify, consider the example in Figure 6. Assume that the routes 10.78.45.128/26 and 10.78.45.132/30 are stored in the table. The first table's entry corresponding to 10.78.45 will contain an index to an entry in *TBLint* (in the example, the index equals 567). Entry 567 in *TBLint* indicates a length of 5, and an index into *TBLlong* (in the example, the index equals 325) pointing to 64 entries. One of these entries, the 33rd, contains the next hop for the 10.78.45.132/30 route. Entry 32 and entries 34 through 47 will contain the next hop for the 10.78.45.128/26 route. The others will contain the next-hop value designated to mean "no entry".

The modification requires an additional memory access, extending the pipeline to three stages, but saves some space in the final table by not expanding every "long" route to 256 entries.

Multiple table scheme: Another modification can be made to reduce memory usage, with the addition of a constraint. For simplicity, we present this scheme as an extension of the two table scheme (*DIR-24-8-BASIC*) presented earlier. In this scheme, called *DIR-n-m*, we extend the original scheme

Figure 5: *TBLint* Entry Format

index into 2nd table	max length
20 bits	3 bits

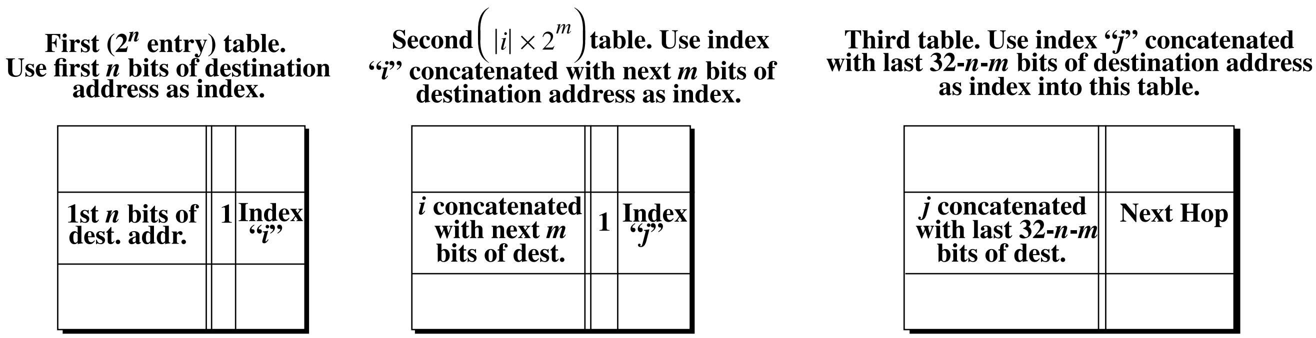


Figure 7: Three table scheme in the worst case, where the route is longer than (n+m) bits long. In this case, all three levels must be used, as shown.

to use three smaller tables, instead of one large table (*TBL24*) and one small table (*TBLlong*). The aim is to split the 32-bit space so as to minimize memory usage.

Let us replace tables *TBL24* and *TBLlong* in scheme *DIR-24-8-BASIC* by a 2^{21} entry table (the "first" table, *TBLfirst21*), another 2^{21} entry table (the "second" table, *TBLsec21*), and a 2^{20} entry table (the "third" table, *TBLthird20*).

The first 21 bits of the packet's destination address are used to index into *TBLfirst21*, which has entries of width 19 bits. The first bit of the entry will, as before, indicate whether the rest of the entry can be used as the "next-hop" identifier, or if the rest of the entry must be used as an index into another table (*TBLsec21* in this case).

If the rest of the entry in *TBLfirst21* is used as an index, we concatenate this 18-bit index with the next 3 bits (bit numbers 22 through 24) of the packet's destination address, and use this concatenated number as an index into *TBLsec21*. *TBLsec21* has entries of width 13 bits. As before, the first bit indicates whether the rest of the entry can be considered as a "next-hop" identifier, of if the rest of the entry must be used as an index into the third table (*TBLthird20*).

Again, if the rest of the entry must be used as an index, we use this value, concatenated with the last 8 bits of the packet's destination address, to index into *TBLthird20*.

TBLthird20, like *TBLlong*, contains entries of width 8 bits, storing the next-hop identifier. These three tables are shown in Figure 7 (with $n = 21$ and $m = 3$ in this case).

The *DIR-21-3* has the advantage that only 9 MB of memory is required: $(2^{21} \cdot 19) + (2^{21} \cdot 13) + (2^{20} \cdot 8)$ bits. The disadvantage is that we have added another constraint to the system. In addition to only supporting 4096 routes of length 25 or greater with distinct 24-bit prefixes, we can now only support 2^{18} routes of length 22 or greater, with distinct 21-bit prefixes. Although this constraint is easily met in current routing environments, in the long term this constraint may pose a problem.

The scheme can, of course, be extended to an arbitrary number of table levels, at the cost of an additional constraint per additional table level. Table 1 indicates the total amount of memory that would be required for various numbers of table levels.[†] Although not shown in the table, memory requirements vary significantly with the choice of the number of bits to use per level. Table 1 shows only the *lowest* memory requirement. As an alternative, a three level split using (16,8,8) bits per level requires 105 MBytes.

Number of Levels	Bits used per level	Min. Memory Requirement
3	**21, 3, and 8**	**9 MB**
4	**20, 2, 2, and 8**	**7 MB**
5	**20, 1, 1, 2, and 8**	**7 MB**
6	**19, 1, 1, 1, 2, and 8**	**7 MB**

Table 1: Memory required as a function of number of levels.

As we increase the number of levels, we achieve diminishing memory savings coupled with increased hardware logic complexity to manage the deeper pipeline.

5 Routing Table Updates

As the topology of the network changes, new routing information is disseminated among the routers, leading to changes in routing tables. As a result of a change, one or more entries must be added, updated, or deleted from the table. Because the action of modifying the table can interfere with the process of forwarding packets, we need to consider the frequency and overhead caused by changes to the table. Shortly, we will consider a number of different techniques for updating the routing table: each comes with a different cost to the forwarding function.

But first, let's consider how often routing tables are

† For the figures in this table, we assume that at each level, only 2^{18} routes can be accommodated by the next higher level table, except the *last* table, which we assume supports only 4096 routes. This is because it seems unlikely that there will be a need to support more than 4096 routes of length 25 bits or greater, with distinct 24-bit prefixes.

changed. Measurements and anecdotal evidence suggest that routing tables change very frequently [11]. Trace data collected from a major ISP backbone router† indicate that a few hundred updates can occur per second. A potential drawback of the 16-million entry *DIR-24-8-BASIC* scheme is that changing a single route prefix can affect a large number of entries in the table. For instance, if an 8-bit route prefix is added to an empty forwarding table, this would require changes to 2^{16} consecutive forwarding entries. With our data, if every routing table change affected 2^{16} entries, it would lead to 20×10^6 entry changes per second!‡

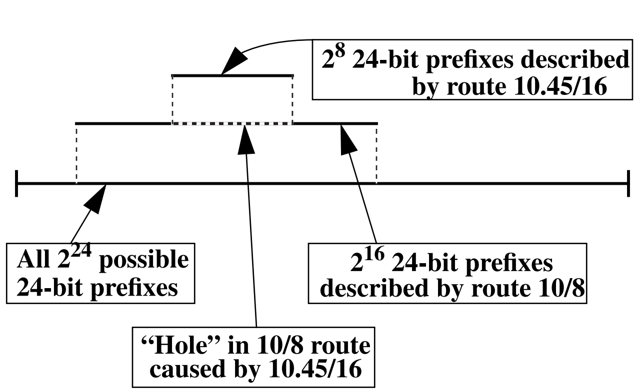
Furthermore, changing the entries for one prefix is not always as simple as changing consecutive entries; longer prefixes create “holes” that must be avoided by the update mechanism. This is illustrated in Figure 8 where a route entry of 10.45/16 exists in the forwarding table. If the new route entry 10/8 is added to the table, we need to modify only a portion of the 2^{16} entries described by the 10/8 route, and leave the 10.45/16 “hole” unmodified.

In what follows, we focus on schemes to update the large *TBL24* table in the *DIR-24-8-BASIC* scheme. The smaller *TBLlong* table requires much less frequent updates and is ignored here.

5.1 Dual Memory Banks

A simple but costly solution, this scheme uses two banks of memory. Periodically, the processor creates and downloads a new forwarding table to one bank of memory. During this time (which in general will take much longer than one lookup time), the other bank of memory is used for forwarding. Banks are switched when the new bank is ready. This provides a mechanism for the processor to update the tables in a simple and timely manner, and has been used in at least one high-performance router [12].

Figure 8: A “hole” in consecutive forwarding entries.



† The router is part of the Sprint network. The trace had a total of 3737 BGP routing updates, with an average of 1.04 updates per second and a maximum of 291 updates per second.

‡ In practice, of course, the number of 8-bit prefixes is limited to just 256, and it is extremely unlikely that they will all change at the same time.

5.2 Single Memory Bank

In general, we can avoid doubling the memory by making the processor do more work. The processor can calculate exactly which entries in the hardware forwarding tables need to be updated and can instruct the hardware accordingly. An important consideration is: how many messages must flow from the processor to update a route prefix? If the number of messages is too high, then the performance will become limited by the processor. We now describe three different update schemes, and compare their performance when measured by the number of update messages that the processor must generate.

Update Mechanism 1: *Row-Update*.
In this scheme, the processor sends one message for each entry that is changed in the forwarding table. For example, if a route of 10/8 is to be added to a table which already has a prefix of 10.45/16 installed, the processor will send $65536 - 256 = 65280$ separate messages to the hardware, each message instructing the hardware to change the next hop of the corresponding entry.

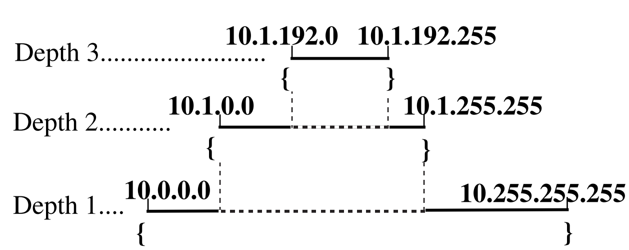
While this scheme is simple to implement in hardware, it places a tremendous burden on the processor.

Update Mechanism 2: *Subrange-Update*.
The presence of “holes” partitions the range of updated entries into a series of subranges. Instead of sending one instruction per entry, the processor can find the bounds of each subrange, and send one instruction per subrange. The messages from the processor to the line cards are now equivalent to “Change *X* entries starting at *Y* to *Z*” where *X* is the number of entries in the subrange, *Y* is the starting entry number, and *Z* is the new next-hop identifier. In our example above, the updates caused by the new route addition could have been performed with just two messages: update 10.0.0 through 10.44.255, and update 10.46.0 through 10.255.255.

This scheme works well when entries have few “holes”. However, in the worst case many messages are still required: it is *possible* (though unlikely) that every other entry must be updated. An 8-bit prefix therefore requires up to 32,768 update messages, i.e. roughly 3.2 million update messages per second.

Update Mechanism 3: *One-Instruction-Update.*
This scheme requires only one instruction from the processor

Figure 9: The balanced parentheses property of prefixes.



for each updated prefix, regardless of the number of holes. One simple way to do this is to include a five bit length field in every table entry indicating the length of the prefix to which the entry belongs.

Consider again our example of a routing table containing the prefixes 10.45/16 and 10/8. The entries in the "hole" created by the 10.45/16 route contain 16 in the length field; the other entries associated with the 10/8 route contain the value 8. Hence, the processor only needs to send a single message for each route update. The message would be similar to: "Change entries starting at number *X* for a *Y*-bit long route to next-hop *Z*." The hardware then examines 2^{24-Y} entries beginning with entry *X*. For each entry whose length field is less than or equal to *Y*, the new next-hop is entered. Those entries with length field greater than *Y* are left unchanged. As a result, "holes" are skipped within the updated range.

One problem is that a five bit length field needs to be added to all 16 million entries in the table; an additional 10 MB (about 30%) of memory.

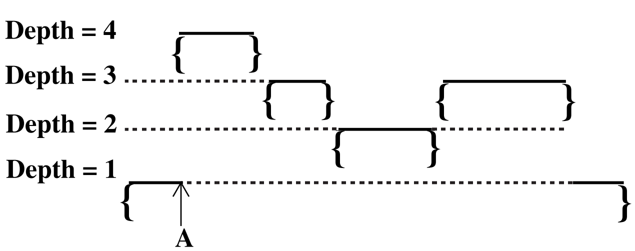
Update Mechanism 4: ***Optimized One-Instruction-Update*.**

Fortunately, we can eliminate the length field from each prefix entry in *TBL24*. First note that for any two distinct prefixes, either one is completely contained in the other, or the two prefixes have no entries in common. This structure is very similar to that of parenthetical expressions where the scope of an expression is delimited by balanced opening and closing parentheses: for example, the characters "{" and "}" used to delimit expressions in the 'C' programming language.

Figure 9 shows an example with three "nested" route prefixes. Suppose that we scan an expression having balanced parentheses from a point with a nesting depth *d*. By keeping track of the number of opening and closing parentheses seen so far, we can determine the current depth. This can then be applied to performing route updates: the central processor provides the hardware with the *first* memory entry to be updated. The hardware scans the memory sequentially, updating only those entries at depth *d*.

Under this scheme, each entry in *TBL24* can be classified as one of the following types: an opening parenthesis (start of route), a closing parenthesis (ending of route), no parenthesis (middle of route), or both an opening and closing parenthesis (if the route contains only a single entry). This classification can be represented by a 2-bit field in each entry.

Figure 10: Moving the parentheses markers.



Care must be taken when a single entry in *TBL24* correspond to the start or end of multiple routes, as shown in Figure 10. With our 2-bit encoding, we cannot adequately describe all the routes that begin and end at memory location 'A'. The problem is readily fixed by shifting the opening and closing markers to the start (end) of the first (last) entry in memory that the route affects. The same update algorithm can then be used without change.

Note that unlike the Row- and Subrange-update schemes, this scheme requires a read-modify-write operation for each scanned entry. This can be reduced to a parallel read and write if the marker field is stored in a separate memory.

5.3 Simulation Results

To evaluate the different update schemes, we simulated the behavior of each when presented with the sequence of routing updates collected from the ISP backbone router. We evaluate the update schemes using two criteria: (i) The number of messages per second sent by the processor, and (ii) The number of memory accesses per second required to be performed by the hardware. The simulation results are shown in Table 2.†

Update Scheme	# of Msgs from Processor per second (Avg/ Max)	# of Memory Accesses per second (Avg/ Max)
Row	43.4/17545	43.4/17545
Subrange	1.14/303	43.4/17545
One-instruction	1.04/291	115.3/40415

Table 2: Simulation results for three update mechanisms

The results corroborate our intuition that the row-update scheme puts a large burden on the processor: up to 17,545 messages per second. At the other extreme, the one-instruction-update scheme is optimal in terms of the number of messages required to be sent by the processor, with a maximum of just 291. But unless we use a separate marker memory, it requires more than twice as many memory accesses as the other schemes. However, this still represents less than 0.2% of the routing lookup capacity available from the scheme. In this simulation, we find that the subrange-update scheme performs well by both measures. The small number of messages from the processor can be attributed to the fact that the routing table contained few holes. We expect this to

† For the one-instruction-update (optimized scheme) we assume that the extra 2-bits to store the opening/closing marker fields mentioned above are *not* stored in a separate memory.

be the case for most routing tables in the near term. But it is too early to tell whether routing tables will become more fragmented, and contain more holes in the future.

6 Conclusions

The continued decreasing cost of DRAM means that it is now feasible to perform an IPv4 routing lookup in dedicated hardware in the time that it takes to execute a single memory access. Today, this corresponds to approximately 20×10^6 lookups per second; enough to process the packets on a 20Gb/s line. The lookup rate will improve in the future as memory speeds increase. The scheme operates by expanding the prefixes and throwing lots of cheap memory at the problem. Yet still the total memory cost today is less than $100, and will (presumably) continue to decrease by roughly 50% each year. For those applications where low cost is paramount, we have mentioned several multilevel variations on the basic scheme that use memory more efficiently.

Using a trace of routing table updates from a major backbone router, we find that care must be taken when designing the hardware update mechanism. We have found and evaluated two update mechanisms (Subrange-update and One-instruction-update) that perform efficiently and quickly in hardware with little burden on the central routing processor. Our results indicate that with either scheme, updates steal less than 0.2% of the lookup capacity.

7 Acknowledgments

We would like to thank Quaizar Vohra and Mark Ross for useful comments on a draft of this paper. We also thank Sprint and Cisco Systems for providing the trace of routing updates used in Section 5.

References

[1] Y. Rekhter, T. Li. "An Architecture for IP Address Allocation with CIDR." *RFC 1518*, Sept. 1993.

[2] Merit Networks, Inc. See http://www.merit.edu

[3] A. McAuley, P. Francis. "Fast Routing Table Lookup Using CAMs." *Proc. IEEE INFOCOM 1993*, Vol. 3, pp 1382-1391, San Francisco, USA.

[4] W. Doeringer, G. Karjoth, M. Nassehi. "Routing on Longest-Matching Prefixes." *IEEE/ACM Trans. Networking*, Vol. 4, No. 1. Feb. 1996.

[5] P. Newman, T. Lyon, G. Minshall. "Flow Labelled IP: A Connectionless Approach to ATM." *Proc. IEEE INFOCOM 1996*, pp. 1251-1260, San Francisco, USA.

[6] Y. Rekhter, B. Davie, D. Katz, E. Rosen, G. Swallow. "Cisco Systems' Tag Switching Architecture Overview." *RFC 2105*, February 1997.

[7] A. Brodnik, S. Carlsson, M. Degermark, S. Pink. "Small Forwarding Tables for Fast Routing Lookups." *Proc. ACM SIGCOMM 1997*, pp. 3-14, Cannes, France.

[8] M. Waldvogel, G. Varghese, J. Turner, B. Plattner. "Scalable High-Speed IP Routing Lookups." *Proc. ACM SIGCOMM 1997*, pp. 25-36, Cannes, France.

[9] V. Srinivasan and G. Varghese. "Efficient Best Matching Prefix Using Tries." pre-publication manuscript, January 1997.

[10] C. Labovitz, G. R. Malan, F. Jahanian. "Internet Routing Instability." *Proc. ACM SIGCOMM 1997*, pp. 115-126, Cannes, France.

[11] C.Partridge et al. "A Fifty Gigabit Per Second IP Router," Accepted for publication in *IEEE/ACM Trans. on Networking*.