Exhibit 1018

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:

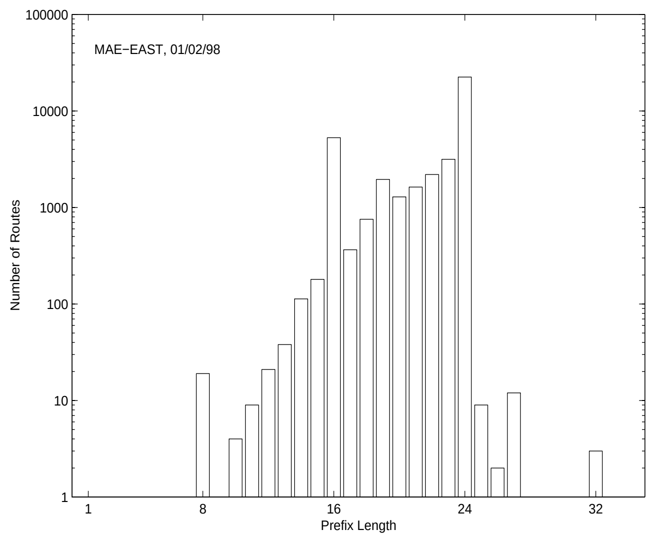
1) The lookup procedure should be easily implementable in hardware using simple logic.
2) Ideally, the route lookup procedure should take exactly one memory access time.
3) If it takes more than one memory access, then (a) the number of accesses should be small, (b) the number of accesses should be bounded 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 16MB = 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

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.

Figure 1: Prefix length distributions.



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.

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.

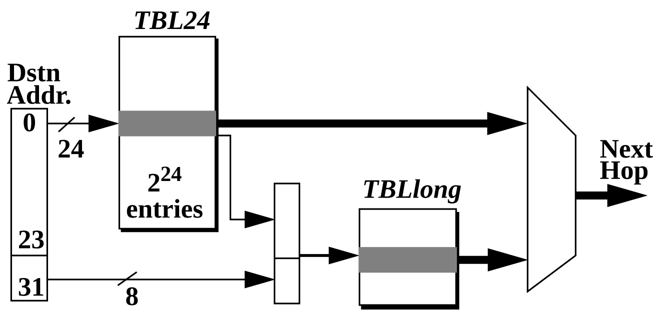


Figure 2: Proposed *DIR-24-8-BASIC* architecture. The next hop result comes from either *TBL24* or *TBLlong*.

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.

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

[†] 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.

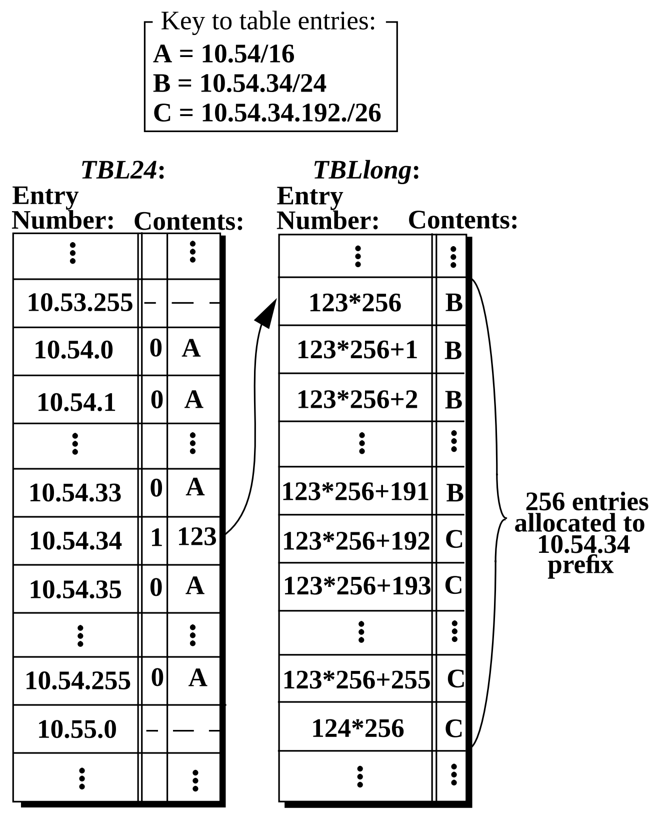


Figure 4: Example of two tables containing three routes.

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

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

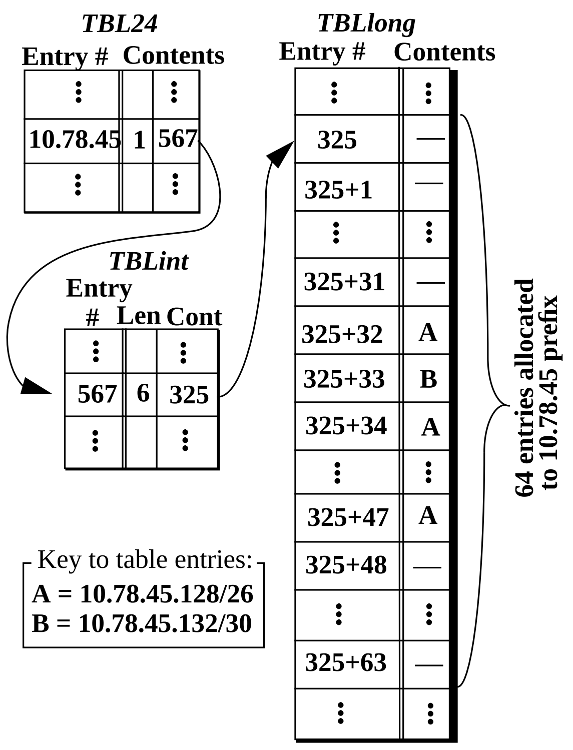


Figure 6: "Intermediate Table" scheme

$2^6 = 64$ 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

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.