US 20100098081A1

(19) **United States**

(12) **Patent Application Publication**

(10) **Pub. No.: US 2010/0098081 A1**

(43) **Pub. Date: Apr. 22, 2010**

Dharmapurikar et al.

(54) **LONGEST PREFIX MATCHING FOR NETWORK ADDRESS LOOKUPS USING BLOOM FILTERS**

(76) Inventors: **Sarang Dharmapurikar**, St. Louis, MO (US); **Praveen Krishnamurthy**, St. Louis, MO (US); **David Edward Taylor**, University City, MO (US)

Correspondence Address:
Radlo & Su LLP
Embarcadero Corporate Center, Suite 800, 2479 East Bayshore Road
PALO ALTO, CA 94303 (US)

(21) Appl. No.: **12/566,150**

(22) Filed: **Sep. 24, 2009**

Related U.S. Application Data

(62) Division of application No. 11/055,767, filed on Feb. 9, 2005, now Pat. No. 7,602,785.

(60) Provisional application No. 60/543,222, filed on Feb. 9, 2004.

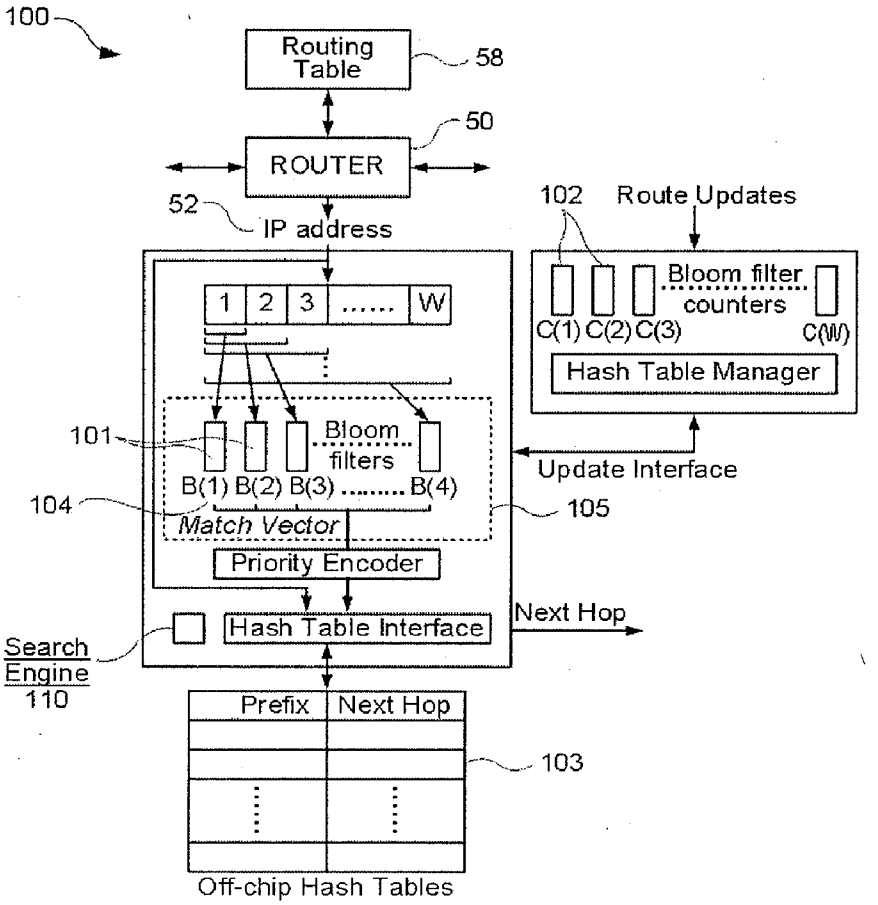
Publication Classification

(51) **Int. Cl.**
H04L 12/56 (2006.01)
H04L 12/28 (2006.01)

(52) **U.S. Cl.** **370/392**; 370/395.32

(57) **ABSTRACT**

Methods and systems for performing parallel membership queries to Bloom filters for Longest Prefix Matching, where address prefix memberships are determined in sets of prefixes sorted by prefix length. Hash tables corresponding to each prefix length are probed from the longest to the shortest match in the vector, terminating when a match is found or all of the lengths are searched. The performance, as determined by the number of dependent memory accesses per lookup, is held constant for longer address lengths or additional unique address prefix lengths in the forwarding table given that memory resources scale linearly with the number of prefixes in the forwarding table. For less than 2 Mb of embedded RAM and a commodity SRAM, the present technique achieves average performance of one hash probe per lookup and a worst case of two hash probes and one array access per lookup.



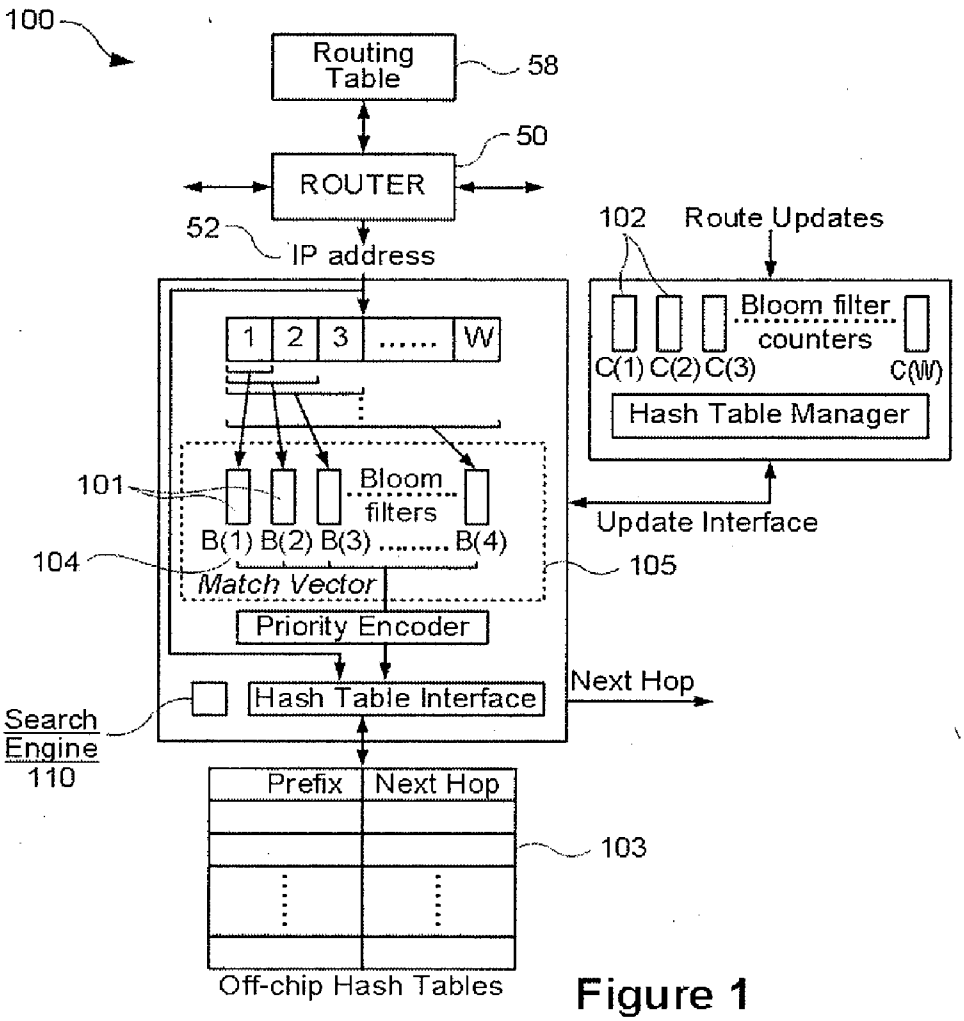


Figure 1

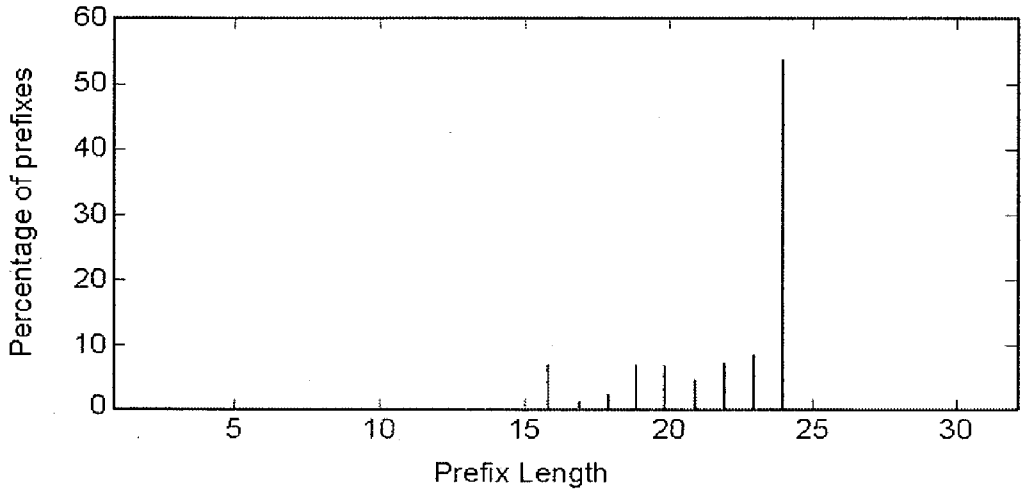


Figure 2

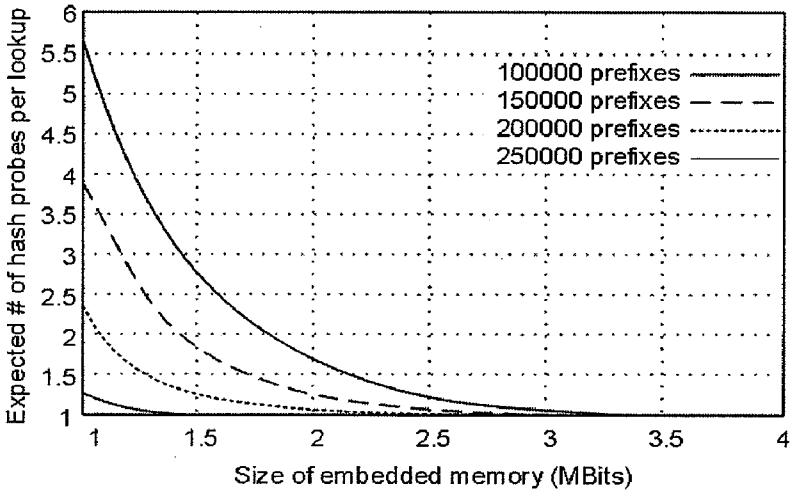


Figure 3

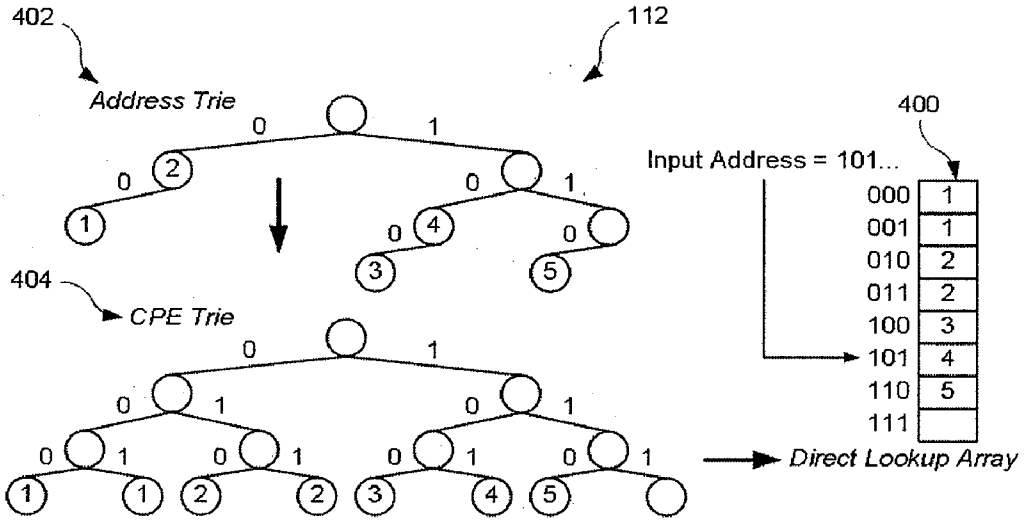


Figure 4

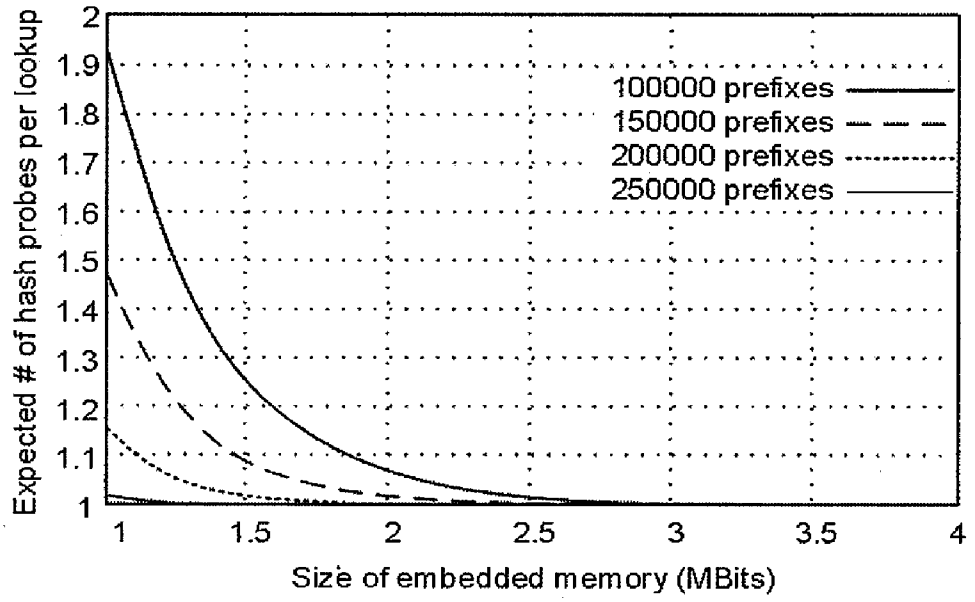


Figure 5

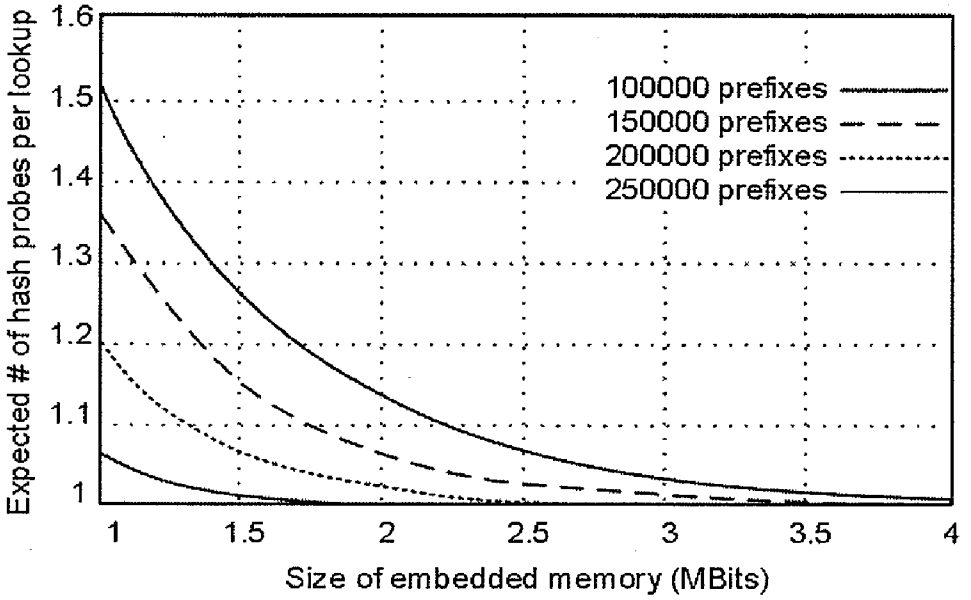


Figure 6

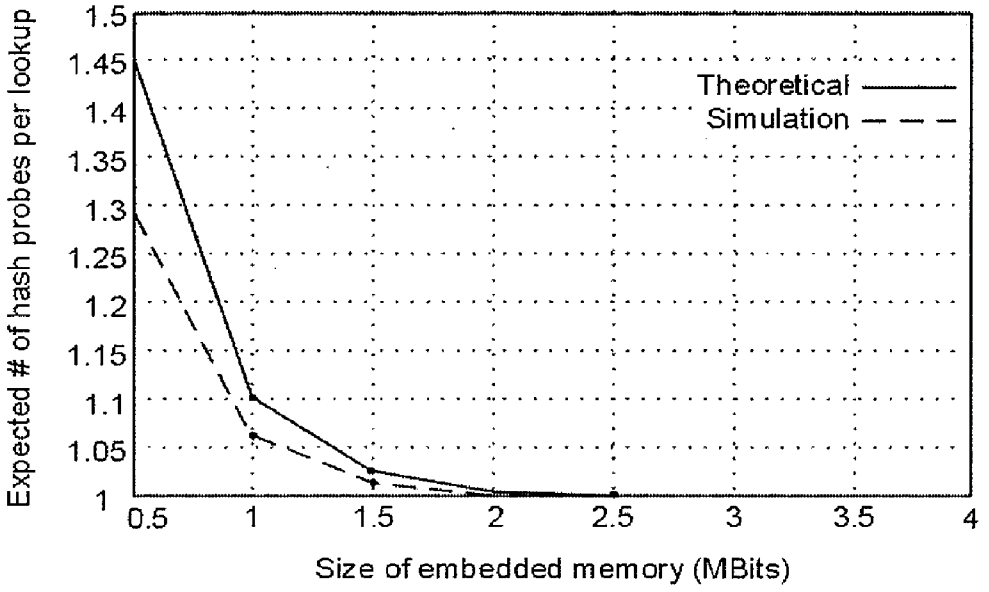


Figure 7

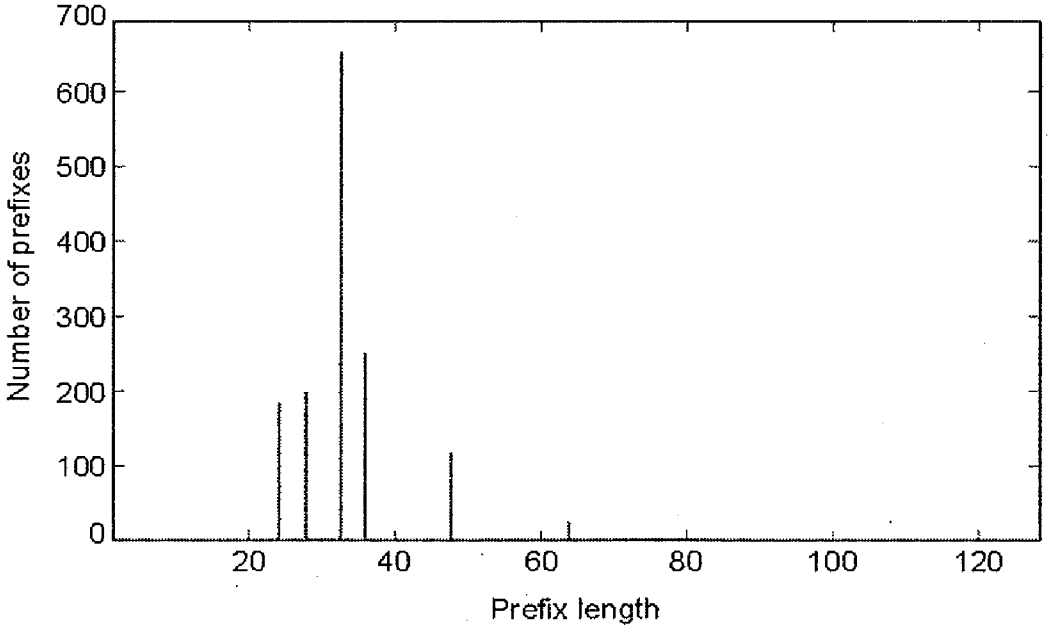


Figure 8

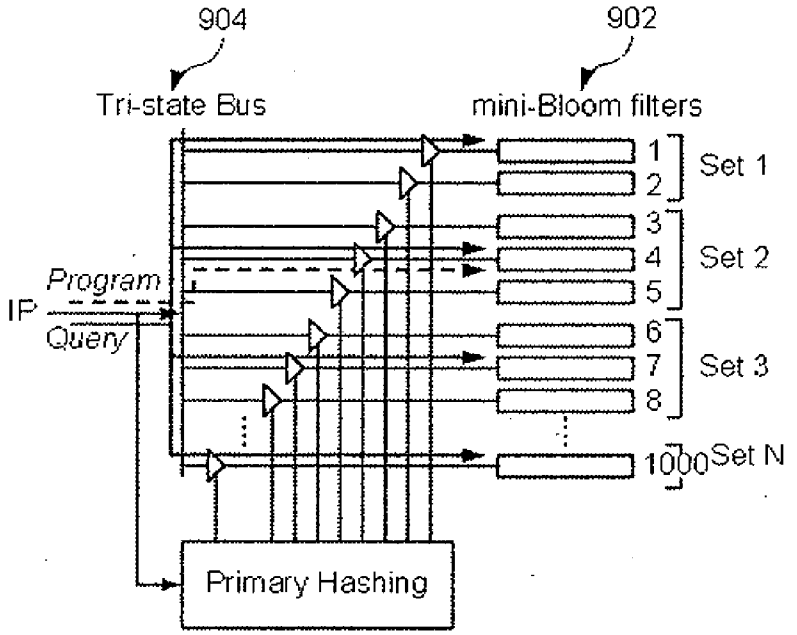


Figure 9

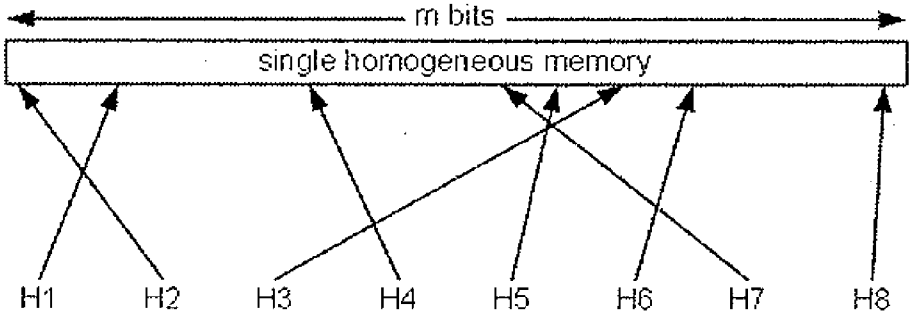


Figure 10a

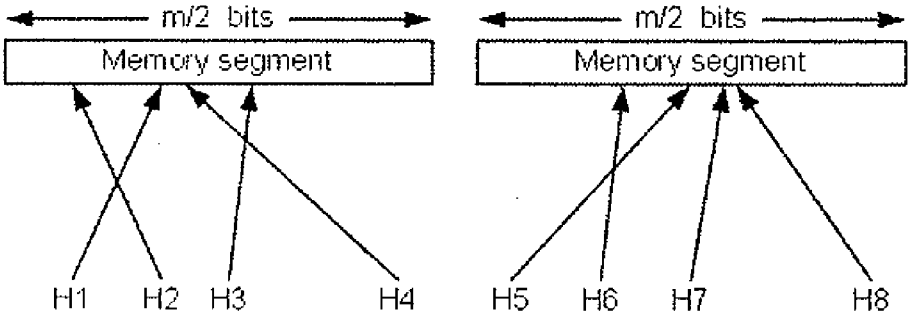


Figure 10b

LONGEST PREFIX MATCHING FOR NETWORK ADDRESS LOOKUPS USING BLOOM FILTERS

PRIOR PATENT APPLICATIONS

[0001] This patent application is a divisional of U.S. patent application Ser. No. 11/055,767, entitled "Method and System for Performing Longest Prefix Matching for Network Address Lookup Using Bloom Filters", filed Feb. 9, 2005, which claims the benefit of U.S. provisional patent application 60/543,222, entitled "Method And Apparatus For Performing Longest Prefix Matching For In Packet Payload Using Bloom Filters," filed on Feb. 9, 2004, each of which prior patent applications is incorporated herein by reference to the fullest extent allowable by law.

STATEMENT OF GOVERNMENTAL INTEREST

[0002] This invention was made with government support under grants ACI-0203869, ANI-9813723, and ANI-0096052 awarded by the National Science Foundation. The government has certain rights in the invention.

BACKGROUND OF THE INVENTION

[0003] The present invention relates to network communication routing and, in particular, to a method and system of performing longest prefix matching for network address lookup using Bloom filters.

[0004] Longest Prefix Matching (LPM) techniques have received significant attention due to the fundamental role LPM plays in the performance of Internet routers. Classless Inter-Domain Routing (CIDR) has been widely adopted to prolong the life of Internet Protocol Version 4 (IPv4). This protocol requires Internet routers to search variable-length address prefixes in order to find the longest matching prefix of the network destination address of each product traveling through the router and retrieve the corresponding forwarding information. This computationally intensive task, commonly referred to as network address lookup, is often the performance bottleneck in high-performance Internet routers due to the number of off-chip memory accesses required per lookup.

[0005] Although significant advances have been made in systemic LPM techniques, most commercial router designers use Ternary Content Addressable Memory (TCAM) devices in order to keep pace with optical link speeds despite their larger size, cost, and power consumption relative to Static Random Access Memory (SRAM).

[0006] However, current TCAMs are less dense than SRAM, and have access times of 100M random accesses per second, which are over 3.3 times slower than SRAMs (which are capable of performing 333,000,000 random accesses per second) due to the capacitive loading induced by their parallelism. Further, power consumption per bit of storage is four orders of magnitude higher than SRAM.

[0007] Techniques such as the Trie-based systems, Tree Bitmap, Multiway and Multicolumn Search, and Binary Search on Prefix Length techniques may make use of commodity SRAM and SDRAM devices. However, these techniques have not met the criteria to provide advantages in performance that are independent of IP address length or to provide improved scalability.

[0008] Therefore, a need exists for a method and system that overcome the problems noted above and others previously experienced.

DISCLOSURE OF INVENTION

[0009] Methods and systems consistent with the present invention employ Bloom filters for Longest Prefix Matching. Bloom filters, which are efficient data structures for membership queries with tunable false positive errors, are typically used for efficient exact match searches. The probability of a false positive is dependent upon the number of entries stored in the filter, the size of the filter, and the number of hash functions used to probe the filter. Methods consistent with the present invention perform a network address lookup by sorting forwarding table entries by prefix length, associating a Bloom filter with each unique prefix length, and "programming" each Bloom filter with prefixes of its associated length. A network address lookup search in accordance with methods consistent with the present invention begins by performing parallel membership queries to the Bloom filters by using the appropriate segments of the input IP address. The result of this step is a vector of matching prefix lengths, some of which may be false matches. A hash table corresponding to each prefix length may then be probed in the order of longest match in the vector to shortest match in the vector, terminating when a match is found or all of the lengths represented in the vector are searched.

[0010] One aspect of the present invention is that the performance, as determined by the number of dependent memory accesses per lookup, may be held constant for longer address lengths or additional unique address prefix lengths in the forwarding table given that memory resources scale linearly with the number of prefixes in the forwarding table.

[0011] Methods consistent with the present invention may include optimizations, such as asymmetric Bloom filters that dimension filters according to prefix length distribution, to provide optimal average case performance for a network address lookup while limiting worst case performance. Accordingly, with a modest amount of embedded RAM for Bloom filters, the average number of hash probes to tables stored in a separate memory device approaches one. By employing a direct lookup array and properly configuring the Bloom filters, the worst case may be held to two hash probes and one array access per lookup while maintaining near optimal average performance of one hash probe per lookup.

[0012] Implementation with current technology is capable of average performance of over 300M lookups per second and worst case performance of over 100M lookups per second using a commodity SRAM device operating at 333 MHz. Methods consistent with the present invention offer better performance, scalability, and lower cost than TCAMs, given that commodity SRAM devices are denser, cheaper, and operate more than three times faster than TCAM-based solutions.

[0013] Specifically, in accordance with methods consistent with the present invention, a method of performing a network address lookup is provided. The method comprises: grouping forwarding entries from a routing table by prefix length; associating each of a plurality of Bloom filters with a unique prefix length; programming said plurality of Bloom filters with said associated set of prefixes; and performing membership probes to said Bloom filters by using predetermined prefixes of a network address.

[0014] In accordance with systems consistent with the present invention, a system is provided for performing a net-

work address lookup. The system comprises means for sorting forwarding entries from a routing table by prefix length, means for associating each of a plurality of Bloom filters with a unique prefix length, means for programming said plurality of Bloom filters with said associated set of prefixes, and means for performing membership queries to said Bloom filters by using predetermined prefixes of an network address.

[0015] Other systems, methods, features, and advantages of the present invention will be or will become apparent to one with skill in the art upon examination of the following figures and detailed description. It is intended that all such additional systems, methods, features, and advantages be included within this description, be within the scope of the invention, and be protected by the accompanying claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0016] FIG. **1** depicts an exemplary system for performing longest prefix matching using Bloom filters according to one embodiment consistent with the present invention;

[0017] FIG. **2** depicts an average prefix length distribution for IPv4 Border Gate Protocol ("BGP") table snapshots according to one embodiment consistent with the present invention;

[0018] FIG. **3** depicts an expected number of hash probes per lookup, Eexp, versus total embedded memory size, M, for various values of total prefixes, N, using a basic configuration for IPv4 with 32 asymmetric Bloom filters, according to one embodiment consistent with the present invention;

[0019] FIG. **4** depicts a direct lookup array for the first three prefix lengths according to one embodiment consistent with the present invention;

[0020] FIG. **5** depicts an expected number of hash probes per lookup, Eexp, versus total embedded memory size, M, for various values of total prefixes, N, using a direct lookup array for prefix lengths **1** . . . **20** and **12** Bloom filters for prefix lengths **21** . . . **32**, according to one embodiment consistent with the present invention;

[0021] FIG. **6** depicts an expected number of hash probes per lookup, Eexp, versus total embedded memory size, M, for various values of total prefixes, N, using a direct lookup array for prefix lengths **1** . . . **20**, and two Bloom filters for prefix lengths **21** . . . **24** and **25** . . . **32**, according to one embodiment consistent with the present invention;

[0022] FIG. **7** depicts an average number of hash probes per lookup for Scheme 3 programmed with database **1**, where N=116,819 for various embedded memory sizes M, according to one embodiment consistent with the present invention;

[0023] FIG. **8** depicts a combined prefix length distribution for Internet Protocol Version 6 ("IPv6") BGP table snapshots, according to one embodiment consistent with the present invention;

[0024] FIG. **9** depicts a plurality of Mini-Bloom filters which allow the system, according to one embodiment consistent with the present invention, to adapt to prefix distribution. The dashed line shows a programming path for a prefix of length **2**, and the solid line illustrates query paths for an input IP address;

[0025] FIG. **10***a* depicts a Bloom filter with single memory vector with k=8, according to one embodiment consistent with the present invention; and

[0026] FIG. **10***b* depicts two Bloom Filters of length m/2 with k=4, combined to realize an m-bit long Bloom filter with k=8, according to one embodiment consistent with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0027] Methods consistent with the present invention employ a LPM technique that provides better performance and scalability than conventional TCAM-based techniques for IP network address lookup. The present invention exhibits several advantages over conventional techniques, since the number of dependent memory accesses required for a lookup is virtually independent of the length of the IP network address and the number of unique prefix lengths (in other words, statistical performance may be held constant for arbitrary address lengths provided ample memory resources). Scaling the present invention to IPv6 does not degrade lookup performance and requires more on-chip memory for Bloom filters only if the number of stored unique prefix lengths increases. Although logic operations and accesses to embedded memory increase operating costs, the amount of parallelism and embedded memory employed by the present invention are well within the capabilities of modern Application-Specific Integrated Circuit ("ASIC") technology. Finally, by avoiding significant precomputation, such as typically exhibited using a known "leaf pushing" technique, the present invention is able to retain its network address lookup performance even when the network prefix databases are incrementally updated.

[0028] FIG. **1** depicts an exemplary system **100** consistent with the present invention for performing a network address lookup using longest prefix matching that employs Bloom filters. In the implementation shown in FIG. **1**, the system **100** is operatively connected to a router **50** to receive an IP address **50**, such as a destination network address, from a packet payload (not shown in figures) that is being traversed through the router **50**. In one implementation, the system **100** may be incorporated into the router **50**. The system **100** includes a group of Bloom filters **101** that are operatively configured to determine IP network address prefix memberships in sets of prefixes that are sorted by prefix length. The system **100** may also include a group of Counting Bloom filters **102** each of which are operatively connected to a respective Bloom filter **101** and a hash table **103**, preferably an off-chip hash table, that is operatively connected to the Bloom filters **101**. As discussed below, a network address lookup search executed by the system **100** in accordance with methods consistent with the present invention begins by performing parallel membership queries to the Bloom filters **101**, which are organized by prefix length. The result is a vector **104** in FIG. **1** of matching prefix lengths, some of which may be false matches. The hash table **103** has all the prefixes in the routing table and is operatively configured to be probed in order of the longest match in the vector **104** to the shortest match in the vector **104**, terminating when a match is found or all of the lengths represented in the vector are searched. In one implementation, the hash table **103** may be one of a multiple of hash tables, each containing prefixes of a particular length, operatively configured to be probed. For a modest amount of on-chip resources for Bloom filters **101**, the expected number of off-chip memory accesses required by the system **100** per network address lookup approaches one, providing better performance, scalability, and lower cost than TCAMs, given

that commodity. SRAM devices are denser, cheaper, and operate more than three times faster than TCAM-based solutions.

[0029] In general, each Bloom filter **101** is a data structure used for representing a set of messages succinctly (See B. Bloom, in "Space/time trade-offs in hash coding with allowable errors", *ACM,* 13(7):422-426, May 1970). Each Bloom filter **101** includes a bit-vector of length m used to efficiently represent a set of messages, such as IP addresses that the router **50** may be expected to receive in a packet payload. Given a set of messages X with n members, for each message x, in X, the Bloom filter **101** may compute k hash functions on x_i, producing k hash values each ranging from 1 to m. Each of these values address a single bit in the m-bit vector, hence each message x, causes k bits in the m-bit long vector to be set to 1. Note that if one of the k hash values addresses a bit that is already set to 1, that bit is not changed. This same procedure is repeated for all the members of the set, and is referred to herein as "programming" the Bloom filter.

[0030] Querying the Bloom filters **101** for membership of a given message x in the set of messages is similar to the programming process. Given message x, the Bloom filter generates k hash values using the same hash functions it used to program the filter. The bits in the m-bit long vector at the locations corresponding to the k hash values are checked. If at least one of these k bits is 0, then the message is declared to be a non-member of the set of messages. If all the k bits are found to be 1, then the message is said to belong to the set with a certain probability. If all the k bits are found to be 1 and x is not a member of X, then it is said to be a false positive. This ambiguity in membership comes from the fact that the k bits in the m-bit vector may be set by any of the n members of X. Thus, finding a bit set to 1 does not necessarily imply that it was set by the particular message being queried. However, finding a 0 bit certainly implies that the string does not belong to the set, since if it were a member then all the k bits would definitely have been set to 1 when the Bloom filter **103** was programmed with that message.

[0031] In the derivation of the false positive probability (i.e., for a message that is not programmed, all k bits that it hashes to are 1), the false probability that a random bit of the m-bit vector is set to 1 by a hash function is simply 1/m. The probability that it is not set is 1−(1/m). The probability that it is not set by any of the n members of X is $(1-(1/m))^n$. Since each of the messages sets k bits in the vector, it becomes $(1-(1/m))^{nk}$. Hence, the probability that this bit is found to be 1 is $1-(1-(1/m))^{nk}$. For a message to be detected as a possible member of the set, all k bit locations generated by the hash functions need to be 1. The probability that this happens, f, is given by:

$$f=(1-(1-(1/m))^{nk})^k \tag{1}$$

[0032] For large values of m, the above equation approaches the limit:

$$f\approx(1-e^{(-nk/m)})^k \tag{2}$$

[0033] This explains the presence of false positives in this scheme, and the absence of any false negatives.

[0034] Because this probability is independent of the input message, it is termed the "false positive" probability. The false positive probability may be reduced by choosing appropriate values for m and k for a given size of the member set, n. It is clear that the size of the bit-vector, m, needs to be quite large compared to the size of the message set, n. For a given ratio of m/n, the false positive probability may be reduced by increasing the number of hash functions, k. In the optimal case, when false positive probability is minimized with respect to k, the following relationship is obtained:

$$k=\left(\frac{m}{n}\right)\ln 2 \tag{3}$$

[0035] The ratio m/n may be interpreted as the average number of bits consumed by a single member of the set of messages. It should be noted that this space requirement is independent of the actual size of the member. In the optimal case, the false positive probability is decreased exponentially with a linear increase in the ratio m/n. In addition, this implies that the number of hash functions k, and hence the number of random lookups in the bit vector required to query membership of one message in the set of messages is proportional to m/n.

[0036] The false positive probability at this optimal point (i.e., false positive probability ratio) is:

$$f=\left(\frac{1}{2}\right)^k \tag{4}$$

[0037] If the false positive probability is to be fixed, then the amount of memory resources, m, needs to scale linearly with the size of the message set, n.

[0038] One property of Bloom filters is that it is not possible to delete a member stored in the filter. Deleting a particular message entry from the set programmed into the Bloom filter **103** requires that the corresponding k hashed bits in the bit vector (e.g., vector **104**) be set to zero. This could disturb other members programmed into the Bloom filter which hash to (or set to one) any of these bits.

[0039] To overcome this drawback, each Counting Bloom filter **102** has a vector of counters corresponding to each bit in the bit-vector. Whenever a member or message (e.g., IP address **52** prefix) is added to or deleted from the set of messages (or prefixes) programmed in the filter **102**, the counters corresponding to the k hash values are incremented or decremented, respectively. When a counter changes from zero to one, the corresponding bit in the bit-vector is set. When a counter changes from one to zero, the corresponding bit in the bit-vector is cleared.

[0040] The counters are changed only during addition and deletion of prefixes in the Bloom filter. These updates are relatively less frequent than the actual query process itself. Hence, counters may be maintained in software and the bit corresponding to each counter is maintained in hardware. Thus, by avoiding counter implementation in hardware, memory resources may be saved.

[0041] An important property of Bloom filters is that the computation time involved in performing the query is independent from the number of the prefixes programmed in it, provided, as stated above, that the memory m used by the data structure varies linearly with the number of strings n stored in it. Further, the amount of storage required by the Bloom filter for each prefix is independent from its length. Still further, the computation, which requires generation of hash values, may be performed in special purpose hardware.

[0042] The present invention leverages advances in modern hardware technology along with the efficiency of Bloom fil-

ters to perform longest prefix matching using a custom logic device with a modest amount of embedded SRAM and a commodity off-chip SRAM device. A commodity DRAM (Dynamic Random Access Memory) device could also be used, further reducing cost and power consumption but increasing the "off-chip" memory access period. In the present invention, by properly dimensioning the amount and allocation of embedded memory for Bloom filters **101**, the network address lookup performance is independent of address length, prefix length, and the number of unique prefix lengths in the database, and the average number of "off-chip" memory accesses per lookup approaches one. Hence, lookup throughput scales directly with the memory device access period.

[0043] In one implementation, the plurality of IP address **52** prefixes (e.g., forwarding prefixes) from a routing table **58** in FIG. **1** that are expected to be received by the system are grouped into sets according to prefix length. As shown in FIG. **1**, the system **100** employs a set of W Bloom filters **101**, where W is the number of unique prefix lengths of the prefixes in the routing table, and associates one filter **101** with each unique prefix length. In one embodiment, the Bloom filters **101** areCounting Bloom filters. Each filter **101** is "programmed" with the associated set of prefixes according to the previously described procedure.

[0044] Although the bit-vectors associated with each Bloom filter **101** are stored in embedded memory **105**, the counters **102** associated with each filter **101** may be maintained, for example, by a separate control processor (not shown in figures) responsible for managing route updates. Separate control processors with ample memory are common features of high-performance routers.

[0045] The hash table **103** is also constructed for all the prefixes where each hash entry is a [prefix, next hop] pair. Although it is assumed, for example, that the result of a match is the next hop for the packet being traversed through the router **50**, more elaborate information may be associated with each prefix if desired. As mentioned above, the hash table **103** may be one of a group of hash tables each containing the prefixes of a particular length. However, a single hash table **103** is preferred. The single hash table **103** or the set of hash tables **103** may be stored off-chip in a separate memory device; for example, a large, high-speed SRAM.

[0046] Using the approximation that probing a hash table **103** stored in off-chip memory requires one memory access, minimizing the number of hash probes per lookup is described as follows.

[0047] A network address lookup search executed by the system **100** in accordance with methods consistent with the present invention may proceed as follows. The input IP address **52** is used to probe the set of W Bloom filters **101** in parallel. The one-bit prefix of the address **52** is used to probe the respective filter **101** associated with length one prefixes, the two-bit prefix of the address is used to probe the respective filter **101** associated with length two prefixes, and so on. Each filter **101** indicates a "match" or "no match." By examining the outputs of all filters **101**, a vector **104** of potentially matching prefix lengths for the given address is composed, referenced herein as the "match vector."

[0048] For example, for packets following IPv4, when the input address produces matches in the Bloom filters **101** associated with prefix lengths **8**, **17**, **23**, and **30**; the resulting match vector would be [**8**,**17**,**23**,**30**]. Bloom filters may produce false positives, but never produce false negatives; therefore, if a matching prefix exists in the database, it will be represented in the match vector.

[0049] The network address lookup search executed by the system **100** in accordance with methods consistent with the present invention then proceeds by probing the hash table **103** with the prefixes represented in the vector **104** in order from the longest prefix to the shortest until a match is found or the vector **104** is exhausted.

[0050] The number of hash probes required to determine the correct prefix length for an IP address is determined by the number of matching Bloom filters **101**. In one implementation of system **100**, all Bloom filters **101** are tuned to the same false positive probability, f. This may be achieved by selecting appropriate values for m for each filter **101**. Let B_l represent the number of Bloom filters **101** for the prefixes of length greater than l. The probability P that exactly i filters associated with prefix lengths greater than l will generate false positives is given by:

$$P_l = \binom{B_l}{i} f^i (1-f)^{B_l - i} \tag{5}$$

[0051] For each value of i, i additional hash probes are required. Hence, the expected number of additional hash probes required when matching a length l prefix is:

$$E_l = \sum_{i=1}^{B_l} i \binom{B_l}{i} f^i (1-f)^{B_l - i} \tag{6}$$

[0052] which is the mean for a binomial distribution with B_l elements and a probability of success f. Hence,

$$E_l = B_l f \tag{7}$$

[0053] The equation above shows that the expected number of additional hash probes for the prefixes of a particular length is equal to the number of Bloom filters for the longer prefixes times the false positive probability (which is the same for all the filters). Let B be the total number of Bloom filters in the system for a given configuration. The worst case value of E_l, which is denoted as E_{add}, may be expressed as:

$$E_{add} = Bf \tag{8}$$

[0054] This is the maximum number of additional hash probes per lookup, independent of input address (e.g., IP address **52**). Since these are the expected additional probes due to the false positives, the total number of expected hash probes per lookup for any input address is:

$$E_{exp} = E_{add} + 1 = Bf + 1 \tag{9}$$

[0055] where the additional one probe accounts for the probe at the matching prefix length. However, there is a possibility that the IP address **52** may create a false positive matches in all the filters **101** in the system **100**. In this case, the number of required hash probes is:

$$E_{worst} = B + 1 \tag{10}$$

[0056] Thus, Equation 9 gives the expected number of hash probes for a longest prefix match, and Equation 10 provides the maximum number of hash probes for a worst case lookup.

[0057] Since both values depend on B, the number of filters **101** in the system **100**, reducing B is important to limit the worst case. In one implementation of the system **100**, the value of B is equal to W.

[0058] Accordingly, the system **100** provides high performance independent of prefix database characteristics and input address patterns, with a search engine (e.g., search engine **110** in FIG. **1**) that achieves, for example, an average of one hash probe per lookup, bounds the worst case search, and utilizes a small amount of embedded memory.

[0059] Several variables affect system performance and resource utilization:

[0060] N, the target amount of prefixes supported by the system;

[0061] M, the total amount of embedded memory available for the Bloom filters;

[0062] W, the number of unique prefix lengths supported by the system;

[0063] m_i, the size of each Bloom filter;

[0064] k_i, the number of hash functions computed in each Bloom filter; and

[0065] n_i, the number of prefixes stored in each Bloom filter.

[0066] For clarity in the discussion, IPv4 addresses (e.g., IP address **52**) are assumed to be 32-bits long. Therefore, in the worst case, W=32. Given that current IPv4 BGP tables are in excess of 100,000 entries, N=200,000 may be used in one implementation of system **100**. Further, the number of hash functions per filter **101** may be set, for example, such that the false positive probability f is a minimum for a filter **101** of length m. The feasibility of designing system **100** to have selectable values of k is discussed below.

[0067] As long as the false positive probability is kept the same for all the Bloom filters **101**, the system **100** performance is independent from the prefix distribution. Let f_i be the false positive probability of the i^{th} Bloom filter. Given that the filter is allocated m_i bits of memory, stores n_i prefixes, and performs $k_i=(m_i/n_i)\ln 2$ hash functions, the expression for f_i becomes,

$$f_i = f = \left(\frac{1}{2}\right)^{\left(\frac{m_i}{n_i}\right)\ln 2} \quad \forall\, i \in [1 \ldots 32] \tag{11}$$

[0068] This implies that:

$$m_1/n_1 = m_2/n_2 = \ldots = m_i/n_i = \ldots = m_{32}/n_{32} = \Sigma m_i/\Sigma n_i = M/N \tag{12}$$

[0069] Therefore, the false positive probability f_i for a given filter i may be expressed as:

$$f_i = f = (1/2)^{(M/N)\ln 2} \tag{13}$$

[0070] Based on the preceding analysis, the expected number of hash probes executed by the system **100** per lookup depends only on the total amount of memory resources, M, and the total number of supported prefixes, N. This is independent from the number of unique prefix lengths and the distribution of prefixes among the prefix lengths.

[0071] The preceding analysis indicates that memory (not shown in figures) may be proportionally allocated to each Bloom filter **101** based on its share of the total number of prefixes. Given a static, uniform distribution of prefixes, each Bloom filter **101** may be allocated m=M/B bits of memory. Examining of standard IP forwarding tables reveals that the distribution of prefixes is not uniform over the set of prefix lengths. Routing protocols also distribute periodic updates; hence, forwarding tables are not static. For example, with 15 snapshots of IPv4 BGP tables, and for gathered statistics on prefix length distributions, as expected, the prefix distributions for the IPv4 tables demonstrated common trends such as large numbers of 24-bit prefixes and few prefixes of length less than 8-bits. An average prefix distribution for all of the tables in this example, is shown in FIG. **2**.

[0072] In an exemplary static system configured for uniformly distributed prefix lengths to search a database with non-uniform prefix length distribution, some filters are "over-allocated" to memory while others are "under-allocated." Thus, the false positive probabilities for the Bloom filters are no longer equal. In this example, the amount of embedded memory per filter is proportionally allocated based on its current share of the total prefixes and the number of hash functions is adjusted to maintain a minimal false positive probability. This exemplary configuration is teemed "asymmetric Bloom filters", and a device architecture capable of supporting it is discussed below. Using Equation 9 for the case of IPv4, the expected number of hash probes per lookup, E_{exp}, may be expressed as:

$$E_{exp} = 32 \times (1/2)^{(M \ln 2/N)} + 1 \tag{14}$$

[0073] Given the feasibility of asymmetric Bloom filters, the expected number of hash probes per lookup, E_{exp}, is plotted versus total embedded memory size M for various values of N in FIG. **3**. With a modest 2 Mb embedded memory, for example, the expected number of hash probes per lookup is less than two for 250,000 prefixes. The present exemplary system **100** is also memory efficient as it only requires 8 bits of embedded memory per prefix. Doubling the size of the embedded memory to 4 Mb, for example, provides near optimal average performance of one hash probe per lookup. Using Equation 10, the worst case number of dependent memory accesses is simply 33. The term for the access for the matching prefix may be omitted, because the default route may be stored internally. Hence, in this implementation of system **100**, the worst case number of dependent memory accesses is 32.

[0074] The preceding analysis illustrates how asymmetric Bloom filters **101** consistent with the present invention may achieve near optimal average performance for large numbers of prefixes with a modest amount of embedded memory.

[0075] Since the distribution statistics shown in FIG. **2** indicate that sets associated with the first few prefix lengths are typically empty and the first few non-empty sets hold few prefixes, the system **100** may use a direct lookup array device (**112** in FIG. **1**) for the first few prefix lengths as an efficient way to represent shorter prefixes while reducing the number of Bloom filters **101**. For every prefix length represented in the direct lookup array device **112**, the number of worst case hash probes is reduced by one. Use of the direct lookup array device **112** also reduces the amount of embedded memory required by the Bloom filters **101** to achieve optimal average performance, as the number of prefixes represented by Bloom filters is decreased.

[0076] One implementation of the direct lookup array device **112** for the first a=3 prefixes is shown in FIG. **4**. This implementation of the direct lookup array device includes a direct lookup array **400** that is operatively connected to a binary trie device **402** and a controlled prefix expansion (CPE) trie **404**. The prefixes of length ≦a are stored in the

binary trie **402**. CPE trie **404** performs a CPE on a stride length equal to a. The next hop associated with each leaf at level a of the CPE trie is written to a respective array slot of the direct lookup array **400** addressed by the bits labeling the path from the root to the leaf. The direct lookup array **400** is searched by using the first a bits of the IP destination address **52** to index into the array **400**. For example, as shown in FIG. **4**, an address **52** with initial bits **101** would result in a next hop of 4. The direct lookup array **400** requires $2^a \times NH_{len}$ bits of memory, where NH_{len} is the number of bits required to represent the next hop.

[0077] For example, a=20 results in a direct lookup array **400** with 1M slots. For a 256 port router (e.g., router **50**) where the next hop corresponds to the output port, 8 bits are required to represent the next hop value and the direct lookup array **400** requires 1 MB of memory. Use of a direct lookup array **400** for the first 20 prefix lengths leaves prefix lengths **21** . . . **32** to Bloom filters **101**. Thus, the expression for the expected number of hash probes per lookup performed by the search engine **110** of the system **100** becomes:

$$E_{exp} = 12 \times (1/2)^{(M \ln 2/N - N)}_{[1:20]} \tag{15}$$

[0078] where $N_{[1:20]}$ is the sum of the prefixes with lengths [1:20].

[0079] On average, the $N_{[1:20]}$ prefixes constitute 24.6% of the total prefixes in the sample IPv4 BGP tables. Therefore, 75.4% of the total prefixes N are represented in the Bloom filters **101** in this implementation. Given this distribution of prefixes, the expected number of hash probes per lookup versus total embedded memory size for various values of N is shown in FIG. **5**. The expected number of hash probes per lookup for databases containing 250,000 prefixes is less than two when using a small 1 Mb embedded memory. Doubling the size of the memory to 2 Mb, for example, reduces the expected number of hash probes per lookup to less than 1.1 for 250,000 prefix databases. Although the amount of memory required to achieve good average performance has decreased to only 4 bits per prefix, for example, the worst case hash probes per lookup is still large. Using Equation 10, the worst case number of dependent memory accesses becomes E_{worst}=(32−20)+1=13. For an IPv4 database containing the maximum of 32 unique prefix lengths, for example, the worst case is 13 dependent memory accesses per lookup.

[0080] A high-performance implementation option for the system **100** is to make the direct lookup array device **112** the final stage in a pipelined search architecture. IP destination addresses **52** that reach this stage with a null next hop value would use the next hop retrieved from the direct lookup array **400** of the device **112**. A pipelined architecture requires a dedicated memory bank or port for the direct lookup array **400**.

[0081] The number of remaining Bloom filters **101** may be reduced by limiting the number of distinct prefix lengths via further use of Controlled Prefix Expansion (CPE). It is desirable to limit the worst case hash probes to as few as possible without prohibitively large embedded memory requirements. Clearly, the appropriate choice of CPE strides depends on the prefix distribution. As illustrated in the average distribution of IPv4 prefixes shown in FIG. **2**, for example, in all of the sample databases that may be used to hold a routing table **58** of IP address **52** prefixes, there is a significant concentration of prefixes from lengths **21** to **24**. On average, 75.2% of the N prefixes fall in the range of 21 to 24.

[0082] Likewise, it is shown for example, in all of the sample databases, that prefixes in the 25 to 32 range are extremely sparse. Specifically, 0.2% of the N prefixes fall in the range 25 to 32. (Note that 24.6% of the prefixes fall in the range of 1 to 20.)

[0083] Based on these observations, in one implementation of the system **100**, the prefixes not covered by the direct lookup array **400** are divided into 2 groups, G_1 and G_2, for example, corresponding to prefix lengths **21-24** and **25-32**, respectively. Each exemplary group is expanded out to the upper limit of the group so that G_1 contains only length **24** prefixes and G_2 contains only length **32** prefixes. For example, $N_{[21:24]}$ is the number of prefixes of length **21** to **24** before expansion and $N_{[25:32]}$ is the number of prefixes of length **25** to **32** before expansion. Use of CPE operations by the system **100**, such as shown in FIG. **4**, increases the number of prefixes in each group by an "expansion factor" factor $\alpha_{[21:24]}$ and $\alpha_{[25:32]}$, respectively. In one example, Applicants observed an average value of 1.8 for $\alpha_{[21:24]}$, and an average value of 49.9 for $\alpha_{[25:32]}$ in the sample databases. Such a large value of $\alpha_{[25:32]}$ is tolerable due to the small number of prefixes in G_2. By dividing the prefixes not covered by the direct lookup array **400** and using CPE operations with the direct lookup array **400**, the system **100** may have two Bloom filters **101** and the direct lookup array **400**, bounding the worst case lookup to two hash probes and one array lookup. The expression for the expected number of hash probes per lookup becomes:

$$E_{exp} = 2 \times \left(\frac{1}{2}\right)^{\left(\frac{M \ln 2}{\alpha_{[21:24]} N_{[21:24]} + \alpha_{[25:32]} N_{[25:32]}}\right)} \tag{16}$$

[0084] Using the observed average distribution of prefixes and observed average values of $\alpha_{[21:24]}$ and $\alpha_{[25:32]}$, the expected number of hash probes per lookup versus total embedded memory M for various values of N is shown in FIG. **6**. In this example, the expected number of hash probes per lookup for databases containing 250,000 prefixes is less than 1.6 when using a small 1 Mb embedded memory. Doubling the size of the memory to 2 Mb reduces the expected number of hash probes per lookup to less than 1.2 for 250,000 prefix databases. The use of CPE to reduce the number of Bloom filters **101** allows the system **100** to perform a maximum of two hash probes and one array access per network address lookup, for example, while maintaining near optimal average network address lookup performance with modest use of embedded memory resources.

[0085] The following provides simulation results for each of three embodiments of system **100** consistent with the present invention, each of which use forwarding or routing tables (e.g., table **58**) constructed from standard IPv4 BGP tables. The exemplary embodiments of the present invention are termed:

[0086] Scheme 1: This first exemplary scheme is the system **100** configuration which uses asymmetric Bloom filters **101** for all prefix lengths as described previously;

[0087] Scheme 2: This second exemplary scheme that may be employed by system **100** uses a direct lookup

array device **112** for prefix lengths [**1** . . . **20**] and asymmetric Bloom filters **101** for prefix lengths [**21** . . . **32**] as described previously; and

[0088] Scheme 3: This third exemplary scheme that may be employed by system **100** uses a direct lookup array device **112** for prefix lengths [**1** . . . **20**] and two asymmetric Bloom filters **101** for CPE prefix lengths **24** and **32** which represent prefix lengths [**21** . . . **24**] and [**25** . . . **32**], respectively, as described above.

shown in Table 1. The maximum number of memory accesses (hash probes and direct lookup) per lookup was recorded for each test run of all the schemes. While the theoretical worst case memory accesses per lookup for Scheme 1 and Scheme 2 are 32 and 13, respectively, the worst observed lookups required less than four memory accesses in all test runs. For scheme 3, in most of test runs, the worst observed lookups required three memory accesses.

TABLE 1

		Scheme 1		Scheme 2		Scheme 3	
Database	Prefixes	Theoretical	Observed	Theoretical	Observed	Theoretical	Observed
1	116,819	1.008567	1.008047	1.000226	1.000950	1.000504	1.003227
2	101,707	1.002524	1.005545	1.000025	1.000777	1.002246	1.001573
3	102,135	1.002626	1.005826	1.000026	1.000793	1.002298	1.001684
4	104,968	1.003385	1.006840	1.000089	1.000734	1.00443	1.003020
5	110,678	1.005428	1.004978	1.000100	1.000687	1.003104	1.000651
6	116,757	1.008529	1.006792	1.000231	1.000797	1.004334	1.000831
7	117,058	1.008712	1.007347	1.000237	1.000854	1.008014	1.004946
8	119,326	1.010183	1.009998	1.000297	1.001173	1.012303	1.007333
9	119,503	1.010305	1.009138	1.000303	1.001079	1.008529	1.005397
10	120,082	1.010712	1.009560	1.000329	1.001099	1.016904	1.010076
11	117,221	1.008806	1.007218	1.000239	1.000819	1.004494	1.002730
12	117,062	1.008714	1.006885	1.000235	1.000803	1.004439	1.000837
13	117,346	1.008889	1.006843	1.000244	1.000844	1.004515	1.000835
14	117,322	1.0008874	1.008430	1.000240	1.001117	1.004525	1.003111
15	117,199	1.008798	1.007415	1.000239	1.000956	1.004526	1.002730
Average	114,344	1.007670	1.007390	1.000204	1.000898	1.006005	1.003265

[0089] For each of the three schemes, M=2 Mb, for example, and m, is adjusted for each asymmetric Bloom filter **101** according to the distribution of prefixes of the database under test. Fifteen IPv4 BGP tables were collected, and for each combination of database and system **100** configuration, the theoretical value of E_{exp} was computed using Equations 14, 15, and 16. A simulation was run for every combination of database and system **100** configuration. The ANSI C rand function was used to generate hash values for the Bloom filters **101**, as well as the prefix hash tables **103**. The collisions in the prefix hash tables **103** were around 0.8% which is negligibly small.

[0090] In order to investigate the effects of input addresses on system **100** network address lookup performance, various traffic patterns varying from completely random addresses to only addresses with a valid prefix in the database were placed under test. In the latter case, the IP addresses **52** were generated in proportion to the prefix distribution. Thus, IP addresses corresponding to a 24 bit prefix in the database dominated the input traffic. One million IP addresses were applied for each test run. Input traffic patterns with randomly generated IP addresses generated no false positives in any of the tests for the three schemes or system **100** configurations. The false positives increased as the traffic pattern contained more IP addresses corresponding to the prefixes in the database.

[0091] Maximum false positives were observed when the traffic pattern consisted of only the IP addresses corresponding to the prefixes in the database. Hence, the following results correspond to this input traffic pattern. The average number of hash probes per lookup from the test runs with each of the databases on all three schemes or system **100** configurations, along with the corresponding theoretical values, are

[0092] Using Scheme 3 or the third system **100** configuration, the average number of hash probes per lookup over all test databases was found to be 1.003, which corresponds to a lookup rate of about 332 million lookups per second with a commodity SRAM device operating at 333 MHz. This is an increase in speed of 3.3 times over state-of-the-art TCAM-based solutions.

[0093] At the same time, Scheme 3 had a worst case performance of 2 hash probes and one array access per lookup. Assuming that the array **400** is stored in the same memory device as the tables **103**, worst case performance is 110 million lookups per second, which exceeds current TCAM performance. Note that the values of the expected hash probes per lookup as shown by the simulations generally agree with the values predicted by the equations.

[0094] A direct comparison was made between the theoretical performance and observed performance for each scheme or system **100** configuration. To see the effect of total embedded memory resources (M) for Bloom filters **101**, Scheme 3 was simulated with database **1** and N=116189 prefixes for various values of M between 500 kb and 4 Mb. FIG. **7** shows theoretical and observed values for the average number of hash probes per lookup for each value of M. Simulation results show slightly better performance than the corresponding theoretical values. This improvement in the performance may be attributed to the fact that the distribution of input addresses **52** has been matched to the distribution of prefixes in the database under test. Since length **24** prefixes dominate real databases, arriving packets are more likely to match the second Bloom filter **101** and less likely to require an array **400** access.

[0095] Thus, the number of dependent memory accesses per lookup may be held constant given that memory resources scale linearly with database size. Given this characteristic of the system **100**, and the memory efficiency demonstrated for

IPv4, a network address lookup system and method consistent with the present invention is suitable for high-speed IPv6 route lookups.

[0096] In order to assess the current state of IPv6 tables, five IPv6 BGP table snapshots were collected from several sites. Since the tables are relatively small, a combined distribution of prefix lengths was computed. FIG. **8** shows the combined distribution for a total of 1,550 prefix entries. A significant result is that the total number of unique prefix lengths in the combined distribution is 14, less than half of the number for the IPv4 tables studied.

[0097] IPv6 unicast network addresses may be aggregated with arbitrary prefix lengths like IPv4 network addresses under CIDR. Although this provides extensive flexibility, the flexibility does not necessarily result in a large increase in unique prefix lengths.

[0098] The global unicast network address format has three fields: a global routing prefix; a subnet ID; and an interface ID. All global unicast network addresses, other than those that begin with 000, must have a 64-bit interface ID in the Modified EUI-64 format. These interface IDs may be of global or local scope; however, the global routing prefix and subnet ID fields must consume a total of 64 bits. Global unicast network addresses that begin with 000 do not have any restrictions on interface ID size; however, these addresses are intended for special purposes such as embedded IPv4 addresses. Embedded IPv4 addresses provide a mechanism for tunneling IPv6 packets over IPv4 routing infrastructure. This special class of global unicast network addresses should not contribute a significant number of unique prefix lengths to IPv6 routing tables.

[0099] In the future, IPv6 Internet Registries must meet several criteria in order to receive an address allocation, including a plan to provide IPv6 connectivity by assigning /48 address blocks. During the assignment process, /64 blocks are assigned when only one subnet ID is required and /128 addresses are assigned when only one device interface is required. Although it is not clear how much aggregation will occur due to Internet Service Providers assigning multiple /48 blocks, the allocation and assignment policy provides significant structure. Thus, IPv6 routing tables will not contain significantly more unique prefix lengths than current IPv4 tables.

[0100] Accordingly, systems and methods consistent with the present invention provide a longest prefix matching approach that is a viable mechanism for IPv6 routing lookups. Due to the longer "strides" between hierarchical boundaries of IPv6 addresses, use of Controlled Prefix Expansion (CPE) to reduce the number of Bloom filters **101** may not be practical. In this case, a suitable pipelined architecture may be employed to limit the worst case memory accesses.

[0101] The ability to support a lookup table of a certain size, irrespective of the prefix length distribution is a desirable feature of the system **100**. Instead of building distribution dependent memories of customized size, for example, a number of small fixed-size Bloom filters called mini-Bloom filters (**902** in FIG. **9**) may be built for the system **100** in lieu of Bloom filters **101**. For example, let the dimensions of each mini-Bloom filter **902** be an m' bit long vector with a capacity of n' prefixes. The false positive probability of the mini-Bloom filter **902** is:

$$f'=(1/2)^{(m'/n')\ln 2} \quad (17)$$

[0102] In this implementation, instead of allocating a fixed amount of memory to each of the Bloom filters **101**, multiple mini-Bloom filters were proportionally allocated according to the prefix distribution. In other words, on-chip resources were allocated to individual Bloom filters in units of mini-Bloom filters **902** instead of bits. While building the database, the prefixes of a particular length across the set of mini-Bloom filters **902** allocated to it were uniformly distributed, and each prefix is stored in only one mini-Bloom filter **902**. This uniform random distribution of prefixes was achieved within a set of mini-Bloom filters by calculating a primary hash over the prefix. The prefix is stored in the mini-Bloom filter **902** pointed to by this primary hash value, within the set of mini-bloom filters, as illustrated by the dashed line in FIG. **9**.

[0103] In the membership query process, a given IP address is dispatched to all sets of mini-Bloom filters **902** for distinct prefix lengths on a tri-state bus **904**. The same primary hash function is calculated on the IP address to find out which one of the mini-Bloom filters **902** within the corresponding set should be probed with the given prefix. This mechanism ensures that an input IP address probes only one mini-Bloom filter **902** in the set associated with a particular prefix length as shown by the solid lines in FIG. **9**.

[0104] Since the prefix is hashed or probed in only one of the mini-Bloom filters **902** in each set, the aggregate false positive probability of a particular set of mini-Bloom filters **902** is the same as the false positive probability of an individual mini-Bloom filter. Hence, the false positive probability of the present embodiment remains unchanged if the average memory bits per prefix in the mini-Bloom filter **902** is the same as the average memory bits per prefix in the original scheme. The importance of the scheme shown in FIG. **9** is that the allocation of the mini-Bloom filters for different prefix lengths may be changed unlike in the case of hardwired memory. The tables which indicate the prefix length set and its corresponding mini-Bloom filters may be maintained on-chip with reasonable hardware resources. The resource distribution among different sets of mini-Bloom filters **902** may be reconfigured by updating these tables. This flexibility makes the present invention independent from prefix length distribution.

[0105] The number of hash functions k, is essentially the lookup capacity of the memory storing a Bloom filter **101**. Thus, k=6 implies that 6 random locations must be accessed in the time allotted for a Bloom filter query. In the case of single cycle Bloom filter queries, on-chip memories need to support at least k reading ports. Fabrication of 6 to 8 read ports for an on-chip Random Access Memory is attainable with existing embedded memory technology.

[0106] For designs with values of k higher than what may be realized by technology, a single memory with the desired lookups is realized by employing multiple smaller memories, with fewer ports. For instance, if the technology limits the number of ports on a single memory to 4, then 2 such smaller memories are required to achieve a lookup capacity of 8 as shown in FIG. **10***b*. The Bloom filter **101** allows any hash function to map to any bit in the vector. It is possible that for some member, more than 4 hash functions map to the same memory segment, thereby exceeding the lookup capacity of the memory. This problem may be solved by restricting the range of each hash function to a given memory. This avoids collision among hash functions across different memory segments.

[0107] In general, if h is the maximum lookup capacity of a RAM as limited by the technology, then k/h such memories of size m/(k/h) may be combined to realize the desired capacity of m bits and k hash functions. When only h hash functions are allowed to map to a single memory, the false positive probability may be expressed as:

$$f'=[1-(1-1/m/(k/h))^{hm}]^{(k/h)h}\approx[1-e^{-nk/m}]^{k} \quad (18)$$

[0108] Comparing equation 18 with equation 2, restricting the number of hash functions mapping to a particular memory, does not affect the false positive probability provided the memories are sufficiently large.

[0109] Accordingly, a Longest Prefix Matching (LPM) system consistent with the present invention employs Bloom filters to efficiently narrow the scope of the network address lookup search. In order to optimize average network address lookup performance, asymmetric Bloom filters **101** may be used that allocate memory resources according to prefix distribution and provide viable means for their implementation. By using a direct lookup array **400** and Controlled Prefix Expansion (CPE), worst case performance is limited to two hash probes and one array access per lookup. Performance analysis and simulations show that average performance approaches one hash probe per lookup with modest embedded memory resources, less than 8 bits per prefix. The future viability for IPv6 route lookups is assured with the present invention. If implemented in current semiconductor technology and coupled with a commodity SRAM device operating at 333 MHz, the present system could achieve average performance of over 300 million lookups per second and worst case performance of over 100 million lookups per second. In comparison, state-of-the-art TCAM-based solutions for LPM provide 100 million lookups per second, consume 150 times more power per bit of storage than SRAM, and cost approximately 30 times as much per bit of storage than SRAM.

[0110] It should be emphasized that the above-described embodiments of the invention are merely possible examples of implementations set forth for a clear understanding of the principles of the invention. Variations and modifications may be made to the above-described embodiments of the invention without departing from the spirit and principles of the invention. All such modifications and variations are intended to be included herein within the scope of the invention and protected by the following claims.

What is claimed is:

1. A method for performing a network address lookup, said method comprising the steps of:

grouping forwarding prefixes from a routing table by prefix length;

associating each of a plurality of Bloom filters with a unique prefix length;

programming each of said plurality of Bloom filters with said prefixes corresponding to said associated unique prefix length; and

performing membership probes to said Bloom filters by using predetermined prefixes of a network address.

2. The method according to claim **1**, further comprising:

storing said prefixes in a hash table.

3. The method according to claim **2**, wherein said hash table comprises a single hash table containing all of the prefixes.

4. The method according to claim **2**, wherein said hash table comprises a plurality of hash tables, each containing prefixes of a particular length.

5. The method according to claim **1** wherein the Bloom filters comprise a bit vector of a plurality of bits.

6. The method according to claim **5** further comprising providing a plurality of counting Bloom filters, each corresponding to one of the plurality of Bloom filters and each counting Bloom filter comprising a plurality of counters corresponding to the plurality of bits in its corresponding Bloom filter.

7. The method according to claim **1**, wherein said Bloom filters are characterized by a false positive probability greater than 0 and a false negative probability of zero.

8. The method according to claim **2**, wherein the step of performing membership probes comprises the step of probing the hash table corresponding to said prefix lengths represented in a match vector in an order of longest prefix to shortest prefix.

9. The method according to claim **8**, wherein probing of said hash tables is terminated when a match is found, and all of said prefix lengths represented in said match vector are searched.

10. The method according to claim **7**, wherein the false positive probability is the same for all of said Bloom filters such that performance is independent of prefix distribution.

11. The method according to claim **5**, further comprising:

providing asymmetric Bloom filters by proportionally allocating an amount of an embedded memory per Bloom filter based on said Bloom filter's current share of a total number of prefixes while adjusting a number of hash functions of said Bloom filters to maintain a minimal false positive probability.

12. The method according to claim **9**, wherein a number of hash probes to said hash table per lookup is held constant for network address lengths in said routing table that are greater than a predetermined length.

13. The method according to claim **9**, wherein a number of dependent memory accesses per network lookup is held constant for additional unique prefix lengths in a forwarding table, provided that memory resources scale linearly with a number of prefixes in said routing table.

14. The method according to claim **1**, further comprising:

utilizing a direct lookup array for initial prefix lengths and asymmetric Bloom filters for the rest of the prefix lengths.

15. The method according to claim **14**, wherein for every prefix length represented in said direct lookup array, a number of worst case hash probes is reduced by one.

16. The method according to claim **5**, further comprising:

uniformly distributing prefixes of a predetermined length across a set of mini-Bloom filters; and

storing each of said prefixes in only one of said mini-Bloom filters.

17. The method according to claim **16**, further comprising:

calculating a primary hash value over said one of said prefixes.

18. The method according to claim **17**, further comprising:

storing said one of said prefixes in said one of said mini-Bloom filters pointed to by said primary hash value, within said set.

19. The method according to claim **18**, further comprising:

dispatching a given network address to all sets of mini-Bloom filters for distinct prefix lengths on a tri-state bus in said probing process.

20. The method according to claim **18**, wherein a same primary hash value is calculated on said network address to

determine which of said mini-Bloom filters within a corresponding set should be probed with a given prefix.

21. A system for performing a network address lookup, comprising:

means for sorting forwarding prefixes from a routing table by prefix length;

means for associating each of a plurality of Bloom filters with a unique prefix length;

means for programming each of said plurality of Bloom filters with said prefixes corresponding to said associated unique prefix length; and

means for performing membership queries to said Bloom filters by using predetermined prefixes of an network address.

22. The system according to claim **21**, further comprising a hash table operable to store said prefixes.

23. The system according to claim **22** wherein said hash table comprises a single hash table containing all of the prefixes.

24. The system according to claim **22**, wherein said hash table comprises a plurality of hash tables, each containing prefixes of a particular length.

25. The system according to claim **21**, wherein the Bloom filters comprise a bit vector of a plurality of bits.

26. The system according to claim **25** further comprising a plurality of counting Bloom filters, each corresponding to one of the plurality of Bloom filters and each counting Bloom filter comprising a plurality of counters corresponding to the plurality of bits in its corresponding Bloom filter.

27. The method according to claim **22**, wherein the means for performing membership queries comprises the means for probing the hash table corresponding to said prefix lengths represented in a match vector in an order of longest prefix to shortest prefix.

28. The system according to claim **21**, further comprising:

a direct lookup array for initial prefix lengths and asymmetric Bloom filters for the rest of the prefix lengths.

29. The system according to claim **28**, wherein for every prefix length represented in said direct lookup array, a number of worst case hash probes is reduced by one.

30. The system according to claim **28**, further comprising:

means for utilizing CPE to reduce a number of said Bloom filters such that a maximum of two hash probes and one array access per network lookup is achieved.

31. The system according to claim **21**, wherein multiple mini-Bloom filters are proportionally allocated according to a prefix distribution.

32. The system according to claim **31**, wherein on-chip resources are allocated to individual Bloom filters in units of mini-Bloom filters instead of bits.

33. The system according to claim **32**, further comprising:

means for uniformly distributing prefixes of a predetermined length across a set of mini-Bloom filters; and

means for storing each of said prefixes in only one of said mini-Bloom filters.

34. The system according to claim **33**, further comprising:

means for calculating a primary hash value over said one of said prefixes.

35. The system according to claim **34**, further comprising:

means for storing said one of said prefixes in said one of said mini-Bloom filters pointed to by said primary hash value, within said set.

36. The system according to claim **35**, further comprising:

means for dispatching a given network address to all sets of mini-Bloom filters for distinct prefix lengths on a tri-state bus in said probing process.

* * * * *