Longest Prefix Matching Using Bloom Filters

Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor, *Member, IEEE*

***Abstract*—We introduce the first algorithm that we are aware of to employ Bloom filters for longest prefix matching (LPM). The algorithm performs parallel queries on Bloom filters, an efficient data structure for membership queries, in order to determine address prefix membership in sets of prefixes sorted by prefix length. We show that use of this algorithm for Internet Protocol (IP) routing lookups results in a search engine providing better performance and scalability than TCAM-based approaches. The key feature of our technique is that the performance, as determined by the number of dependent memory accesses per lookup, can 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. Our approach is equally attractive for Internet Protocol Version 6 (IPv6) which uses 128-bit destination addresses, four times longer than IPv4. We present a basic version of our approach along with optimizations leveraging previous advances in LPM algorithms. We also report results of performance simulations of our system using snapshots of IPv4 BGP tables and extend the results to IPv6. Using less than 2 Mb of embedded RAM and a commodity SRAM device, our technique achieves average performance of one hash probe per lookup and a worst case of two hash probes and one array access per lookup.**

***Index Terms*—Bloom filter, computer networking, IP lookup, Longest Prefix Matching.**

I. INTRODUCTION

LONGEST prefix matching (LPM) techniques have received significant attention in the literature over the past ten years. This is due to the fundamental role it plays in the performance of Internet routers. Due to the explosive growth of the Internet, classless inter-domain routing (CIDR) was widely adopted to prolong the life of Internet Protocol Version 4 (IPv4) [9]. CIDR requires Internet routers to search variable-length address prefixes in order to find the longest matching prefix of the IP destination address and retrieve the corresponding forwarding information for each packet traversing the router. This computationally intensive task, commonly referred to as IP Lookup, is often the performance bottleneck in high-performance Internet routers. While significant advances have been made in algorithmic LPM techniques, most commercial router designers have resolved to 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). The performance bottleneck in LPM algorithms employing RAM is typically the number of dependent memory accesses required per lookup. Dependent memory accesses must be performed sequentially, whereas independent memory accesses may be performed in parallel. Some algorithms allow dependent memory accesses to be masked via pipelining, with each stage accessing an independent memory bank or port; however, this quickly becomes an expensive option. We provide an overview of the prominent LPM algorithmic developments and a comparison of TCAM and SRAM technologies in Section II.

In this paper, we introduce the first algorithm that we are aware of to employ Bloom filters for LPM, as Bloom filters are typically used for efficient exact match searches. A Bloom filter is an efficient data structure for membership queries with tunable false positive errors [3]. The probability of a false positive is dependent upon the number of entries stored in a filter, the size of the filter, and the number of hash functions used to probe the filter. Background on Bloom filter theory is presented in Section III. Our approach begins by sorting the 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 search 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. Hash tables corresponding to each prefix length are 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. The key feature of our technique is that the performance, as determined by the number of dependent memory accesses per lookup, can 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. An overview of the basic technique as well as an analysis of the effects of false positives is provided in Section IV.

In Section V, we introduce optimizations to achieve optimal average case performance and limit the worst case, including asymmetric Bloom filters which dimension filters according to prefix length distribution. We show that 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. We also show that by employing a direct lookup array and properly configuring the Bloom filters, the worst case can be held to two hash probes and one array access per lookup while maintaining near optimal average performance of one hash probe per lookup. We report simulation results for IPv4 using routing databases constructed from publicly available BGP tables in Section VI.

It is important to note that our approach is equally attractive for Internet Protocol Version 6 (IPv6) which uses 128-bit destination addresses, four times longer than IPv4. Based on anal-

Manuscript received December 9, 2003; revised April 17, 2005; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor B. Prabhakar.

The authors are with the Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130 USA (e-mail: sarang@arl.wustl.edu).

Digital Object Identifier 10.1109/TNET.2006.872576

ysis of publicly available IPv6 BGP tables and address allocation and assignment policies for IPv6 deployment, we provide evidence for the suitability of our system for IPv6 route lookups in Section VII. Finally, we discuss considerations for hardware implementation in Section VIII. A system configured to support 250 000 IPv4 prefixes requires 2 Mb of embedded memory, only 8 bits per prefix, to achieve near optimal average performance while bounding the worst case. 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. We assert that this approach offers 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.

II. Related Work

Due to its essential role in Internet routers, IP lookup is a well-researched topic. While a broad spectrum of algorithmic approaches to the problem exist in the literature, most high-performance routers employ TCAM devices in order to perform lookups at optical link speeds. We examine both the prominent algorithmic developments as well as the costs associated with a pure hardware approach utilizing TCAMs.

A. Content Addressable Memories (CAMs)

Content addressable memories (CAMs) minimize the number of memory accesses required to locate an entry. Given an input key, the CAM device compares it against all memory words in parallel; hence, a lookup effectively requires one clock cycle. While binary CAMs perform well for exact match operations and can be used for route lookups in strictly hierarchical addressing schemes [14], the wide use of address aggregation techniques like CIDR requires storing and searching entries with arbitrary prefix lengths. In response, TCAMs were developed with the ability to store an additional "Don't Care" state, thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes at the cost of storage density, access time, and power consumption. Since the input key is compared against every memory word, each bit of storage requires match logic to drive a match word line which signals a match for the given key. The extra logic and capacitive loading due to the massive parallelism lengthens access time and increases power consumption.

We can make a first-order comparison between SRAM and TCAM technology by observing that SRAM cells typically require six transistors to store a binary bit of information, consume between 20 and 30 nW per bit of storage, and operate with 3 ns clock periods (333 MHz). Current generation dual-data-rate (DDR) SRAMs are capable of retrieving two words per clock cycle; however, these devices often have a minimum burst length of two words [15]. Hence, the DDR feature effectively doubles the I/O bandwidth but retains the same random access time. A typical TCAM cell requires additional six transistors to store the mask bit and four transistors for the match logic, resulting in a total of 16 transistors and a cell 2.7 times larger than a standard SRAM cell [17]. In addition to being less dense than SRAM, current generation TCAMs achieve 100 million lookups per second, resulting in access times over 3.3 times longer than SRAM due to the capacitive loading induced by the parallelism [18]. Additionally, power consumption per bit of storage is on the order of 3 μW per "bit" [16]. In summary, TCAMs consume 150 times more power per bit than SRAM and currently cost about 30 times more per bit of storage. Therefore, a lookup technique that employs standard SRAM requires less than four memory accesses per lookup and utilizes less than 11 bytes per entry for IPv4 and less than 44 bytes per entry for IPv6 not only matches TCAM performance and resource utilization, but also provides a significant advantage in terms of cost and power consumption.

B. Trie-Based Schemes

One of the first IP lookup techniques to employ *tries* was the radix trie implementation in the BSD kernel. Optimizations requiring contiguous masks bound the worst case lookup time to $O(W)$ where W is the length of the address in bits [19]. In order to speed up the lookup process, multi-bit trie schemes were developed which perform a search using multiple bits of the address at a time. Srinivasan and Varghese introduced two important techniques for multi-bit trie searches, controlled prefix expansion (CPE) and Leaf Pushing [20]. CPE restricts the set of distinct prefixes by "expanding" prefixes shorter than the next distinct length into multiple prefixes. This allows the lookup to proceed as a direct index lookup into tables corresponding to the distinct prefix lengths, or stride lengths, until the longest match is found. The technique of Leaf Pushing reduces the amount of information stored in each table entry by "pushing" best match information to leaf nodes such that a table entry contains either a pointer or information. While this technique reduces memory usage, it also increases incremental update overhead. Variable length stride lengths, optimal selection of stride lengths, and dynamic programming techniques are discussed as well. Gupta, Lin, and McKeown simultaneously developed a special case of CPE specifically targeted to hardware implementation [10]. Arguing that DRAM is a plentiful and inexpensive resource, their technique sacrifices large amounts of memory in order to bound the number of off-chip memory accesses to two or three. Their basic scheme is a two level "expanded" trie with an initial stride length of 24 and second level tables of stride length eight. Given that random accesses to DRAM may require up to eight clock cycles and current DRAMs operate at less than half the speed of SRAMs, this technique fails to outperform techniques utilizing SRAM and requiring less than 10 memory accesses.

Other techniques such as Lulea [5] and Eatherton and Dittia's Tree Bitmap [7] employ multi-bit tries with compressed nodes. The Lulea scheme essentially compresses an expanded, leaf-pushed trie with stride lengths 16, 8, and 8. In the worst case, the scheme requires 12 memory accesses; however, the data structure only requires a few bytes per entry. While extremely compact, the Lulea scheme's update performance suffers from its implicit use of Leaf Pushing. The Tree Bitmap technique avoids Leaf Pushing by maintaining compressed representations of the prefixes stored in each multi-bit node. It also employs a clever indexing scheme to reduce pointer storage to two pointers per multi-bit node. Storage requirements for Tree Bitmap are on the order of 10 bytes per entry, worst case memory accesses can be held to less than eight with optimizations, and updates require modifications to a few memory words resulting in excellent incremental update performance.

The fundamental issue with trie-based techniques is that performance and scalability are fundamentally tied to address length. As many in the Internet community are pushing to widely adopt IPv6, it is unlikely that trie-based solutions will be capable of meeting performance demands.

C. Other Algorithms

Several other algorithms exist with attractive properties that are not based on tries. The Multiway and Multicolumn Search techniques presented by Lampson, Srinivasan, and Varghese require $O(W + \log N)$ time and $O(2N)$ memory [13]. Again, the primary issue with this algorithm is its linear scaling relative to address length.

Another computationally efficient algorithm that is most closely related to our technique is Binary Search on Prefix Lengths introduced by Waldvogel *et al.* [21]. This technique bounds the number of memory accesses via significant precomputation of the database. First, the database is sorted into sets based on prefix length, resulting in a maximum of W sets to examine for the best matching prefix. A hash table is built for each set, and it is assumed that examination of a set requires one hash probe. The basic scheme selects the sequence of sets to probe using a binary search on the sets beginning with the median length set. For example: for an IPv4 database with prefixes of all lengths, the search begins by probing the set with length 16 prefixes. Prefixes of longer lengths direct the search to its set by placing "markers" in the shorter sets along the binary search path. Going back to our example, a length 24 prefix would have a "marker" in the length 16 set. Therefore, at each set the search selects the longer set on the binary search path if there is a matching marker directing it lower. If there is no matching prefix or marker, then the search continues at the shorter set on the binary search path. Use of markers introduces the problem of "backtracking": having to search the upper half of the trie because the search followed a marker for which there is no matching prefix in a longer set for the given address. In order to prevent this, the best-matching prefix for the marker is computed and stored with the marker. If a search terminates without finding a match, the best-matching prefix stored with the most recent marker is used to make the routing decision. The authors also propose methods of optimizing the data structure to the statistical characteristics of the database. For all versions of the algorithm, the worst case bounds are $O(\log W_{\text{dist}})$ time and $O(N \times \log W_{\text{dist}})$ space where W_{dist} is the number of unique prefix lengths. Empirical measurements using an IPv4 database resulted in memory requirement of about 42 bytes per entry.

Like the Binary Search on Prefix Lengths technique, our approach begins by sorting the database into sets based on prefix length. We assert that our approach exhibits several advantages over the previously mentioned technique. First and foremost, we show that the number of dependent memory accesses required for a lookup can be held constant given that memory resources scale linearly with the size of the forwarding table. We show that our approach remains memory efficient for large databases and provide evidence for its applicability to IPv6. Second, by avoiding significant precomputation like "markers" and "leaf pushing" our approach retains good incremental update performance.

III. Bloom Filter Theory

The Bloom filter is a data structure used for representing a set of messages succinctly. A filter is first "programmed" with each message in the set, then queried to determine the membership of a particular message. It was formulated by B. H. Bloom in 1970 [3] and is widely used for different purposes such as web caching, intrusion detection, and content-based routing [4]. For the convenience of the reader, we explain the theory behind Bloom filters in this section.

A. Bloom Filters

A Bloom filter is essentially a bit-vector (which we call *Vector* in this paper) of length m used to efficiently represent a set of messages. Given a set of messages X with n members, the Bloom filter is "programmed" as follows. For each message x_i in X, k hash functions, $h_1(), \ldots, h_k()$, are computed on x_i producing k values each ranging from 1 to m. Each of these values addresses a single bit in the m-bit vector and sets it 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. The following pseudo-code describes adding a message x to a Bloom filter.

```
BFAdd ($x$)
  1) for ($i = 1$ to $k$)
  2)     Vector[$h_i(x)$] $\leftarrow 1$
```

Querying the filter for set membership of a given message x is similar to the programming process. Given message x, k hash values are generated using the same hash functions 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 the k bits is 0, then the message is declared to be a nonmember of the set. If all the 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 a false positive. The following pseudo-code describes the query process.

```
BFQuery ($x$)
  1) for ($i = 1$ to $k$)
  2)     if (Vector[$h_i(x)$] $= 0$) return false
  3) return true
```

The ambiguity in membership comes from the fact that the k bits in the m-bit vector can be set by any subset 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 message does not belong to the set, since if it were a member, then all k-bits would have been set to 1 when the Bloom filter was programmed.

Now we look at the step-by-step derivation of the false positive probability (i.e., for a message that is not programmed, we find that all k bits that it hashes to are 1). The 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

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k. \quad (1)$$

For large values of m, the above equation reduces to

$$f \approx \left(1 - e^{\frac{-nk}{m}}\right)^k. \quad (2)$$

Since this probability is independent of the input message, it is termed the *false positive* probability. The false positive probability can 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 can be reduced by increasing the number of hash functions, k. In the optimal case, when false positive probability is minimized with respect to k, we get the following relationship:

$$k = \frac{m}{n} \ln 2. \quad (3)$$

The false positive probability at this optimal point is given by

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

It should be noted that if the false positive probability is to be fixed, then the size of the filter, m, needs to scale linearly with the size of the message set, n.

B. Counting Bloom Filters

A property of Bloom filters is that it is not possible to delete a message stored in the filter. Deleting a particular entry requires that the corresponding k hashed bits in the bit vector be set to zero. This could disturb other messages programmed into the filter which hash to any of these bits. In order to solve this problem, the idea of the *Counting Bloom Filters* was proposed in [8]. A counting Bloom Filter maintains a vector of counters (which we call *Counter* in this paper) instead of a bit-vector. Whenever a message is added to or deleted from the filter, the counters corresponding to the k hash values are incremented or decremented, respectively. The following pseudo-codes, CBFAdd and CBFDelete, describe the addition and deletion process, respectively.

```
CBFAdd ($x$)
1) for ($i = 1$ to $k$)
2)     Counter[$h_i(x)$] $++$
```

```
CBFDelete ($x$)
1) for ($i = 1$ to $k$)
2)     Counter[$h_i(x)$] $--$
```

As will be clear later, we use a composite data structure consisting of a counting Bloom filter and an ordinary Bloom filter to support the incremental updates on the set of prefixes in our LPM algorithm.

IV. Our Approach

The performance bottleneck in algorithmic LPM techniques is typically the number of dependent memory accesses required per lookup. Due to the continued scaling of semiconductor technology, logic resources are fast and plentiful. Current application-specific integrated circuits (ASICs) operate at clock frequencies over 500 MHz, are capable of massively parallel computation, and support embedded SRAMs as large as 8 Mb. While logic operations and accesses to embedded memory are not "free," we show that the amount of parallelism and embedded memory employed by our system are well within the capabilities of modern ASIC technology. Given that current ASICs posses an order of magnitude speed advantage to commodity memory devices, approximately ten clock cycles are available for logic operations and embedded memory accesses per "off-chip" memory access. As previously mentioned, commodity SRAM devices are capable of performing 333M random accesses per second while state-of-the-art TCAMs are capable of 100M lookups per second. While the performance ratio may not remain constant, SRAMs will always provide faster accesses than TCAMs which suffer from more capacitive loading due to the massive parallelism inherent in TCAM architecture.

Our approach seeks to leverage advances in modern hardware technology along with the efficiency of Bloom filters to perform longest prefix matching using a custom logic device with a modest amount of embedded SRAM and a commodity "off-chip" SRAM device. Note that a commodity dynamic random access memory (DRAM) device could also be used, further reducing cost and power consumption but increasing the "off-chip" memory access period. We show that by properly dimensioning the amount and allocation of embedded memory for Bloom filters, the average number of "off-chip" memory accesses per lookup approaches one; hence, lookup throughput scales directly with the memory device access period. We also provide system configurations that limit worst case lookup time.

A. Basic Configuration

A naive way to find the longest matching prefix is by using a hash table. We can construct a hash table from all the prefixes and with each prefix we can store the associated next hop information. Given an IP address, we can query the hash table with each of the possible prefixes for its presence, starting from the longest prefix. We stop when the matching prefix is found and retrieve the associated next hop information. The problem of constructing hash tables to minimize collisions with reasonable amounts of memory is well-studied. For the purpose of our discussion, we assume that probing a hash table stored in off-chip memory requires one memory access [21]. Clearly this naive approach would require W_{dist} hash table lookups (and hence as many off-chip memory accesses) in the worst

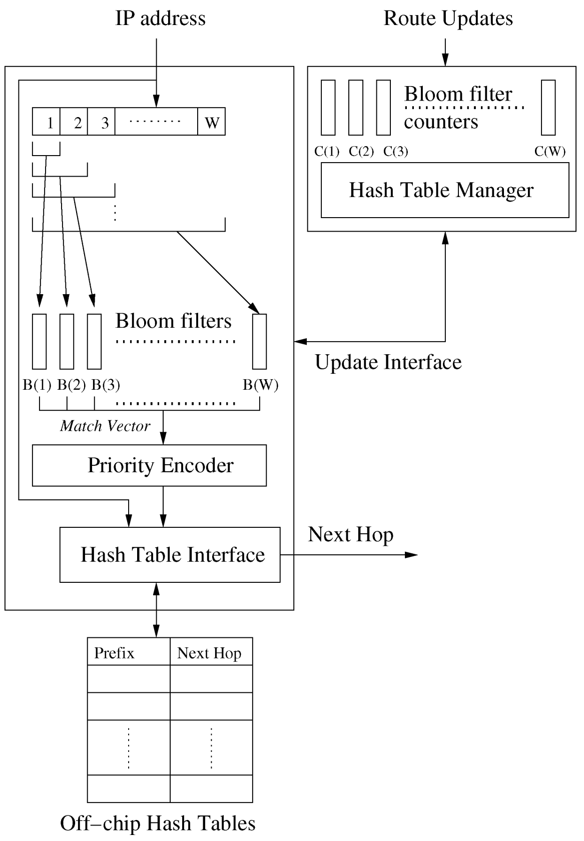


Fig. 1. Basic configuration of longest prefix matching using bloom filters.

case. However, we can reduce the memory accesses to almost a single access by using Bloom filters implemented in *on-chip* memory. The basic configuration of our system is shown in Fig. 1.

We group all the prefixes in our prefix set having the same length and store this group in a Bloom filter. We maintain W parallel Bloom filters in the embedded memory each corresponding to a unique prefix length. Note that we do not need to maintain a Bloom filter corresponding to a prefix length for which there are no prefixes in our set. Before we query the hash table with a prefix, we query the corresponding Bloom filter to check if the prefix is indeed present in the hash table. Only if the Bloom filter shows a match, we proceed to query the hash table. Remember that a Bloom filter can produce a false positive match but never a false negative. If the match of a Bloom filter was a false positive, then we would discover it eventually after the hash table probe results in an unsuccessful search. Thus, in case of a false positive, we simply waste a memory access and proceed to check the next prefix showing a match in the corresponding Bloom filter.

Due to the parallelism offered by embedded memories, we can perform all the Bloom filter lookups concurrently. A carefully designed Bloom filter can give a lookup result in a single cycle of the system clock. We produce a *MatchVector* with W bits in which each bit is set by a Bloom filter if the corresponding prefix queried in it shows a match. Then, we use a priority encoder which simply walks through the MatchVector from longest to shortest prefix and executes the hash table lookups for the matching prefixes. The following pseudo-code formally describes the LPM on an IP address I.

```
LPM ($I$)
1) for ($j = W$ down to 1)
2)     MatchVector[$j$] $\leftarrow$ BFQuery$_j(I_j)$
3) for ($j = W$ down to 1)
4)     if (MatchVector[$j$] = true)
5)         {prefix, NextHop} $\leftarrow$ HashTableLookup$(I_j)$
6)         if (prefix = $I_j$) return {prefix, NextHop}
7) return {NULL, DefaultHop}
```

In the pseudo-code above, I_j denotes the j bit prefix of I and BFQuery$_j$ denotes the process of querying Bloom filter for this prefix length. The loop in line 1–2 is executed in parallel as explained before. By designing Bloom filters to yield a very small false positive probability (explained in Section IV-C) we can make sure that only the HashTableLookup corresponding to the longest matching prefix is performed with very high probability.

B. Supporting Incremental Updates

Now we explain how we support addition and deletion of prefixes to the prefix table using our data structures. As we mentioned before, only ordinary Bloom filters are not sufficient to support deletion of prefixes. Hence, corresponding to each Bloom filter we maintain a counting Bloom filter. Unfortunately, a counting Bloom filter consumes more memory than an ordinary Bloom filter and keeping it in the on-chip memory is not cost-effective. At the same time we note that addition and deletion of routes is relatively infrequent compared to the actual lookup process and hence need not be performed at the same rate. Therefore, we keep the counting Bloom filters in another off-chip memory along with a control processor which is responsible for updates, as shown in Fig. 1.

When we want to add a prefix to the set, first we add it to the counting Bloom filter corresponding to its length. During this addition, if any counter of this filter changes value from zero to one then we must set the corresponding bit of the associated on-chip Bloom filter as well. When a modified counter is already nonzero before addition, we will definitely have the corresponding bit of the associated Bloom filter already set and hence it need not be modified. After modifying Bloom filters, we also need to add the {prefix, NextHop} pair to the hash table.

Likewise, when we want to delete an existing prefix from the set, we first delete it from the counting Bloom filter (i.e., decrement the corresponding counters). If a counter changes its value from one to zero, then the corresponding on-chip Bloom filter must be updated by resetting the bit. Also, the prefix must be deleted from the hash table as well which incurs another memory access. The following pseudo-codes describe addition and deletion of prefixes.

```
AddPrefix ($I_j$, NextHop)
1) for ($i = 1$ to $k$)
2)     Counter$_j[h_i(I_j)]++$
3)     if (Counter$_j[h_i(I_j)] = 1$) Vector$_j[h_i(I_j)] \leftarrow 1$
4) InsertHashTable({$I_j$, NextHop})
```

```
**DeletePrefix** $(I_j)$
1) for ($i = 1$ to $k$)
2)     Counter$_j[h_i(I_j)] - -$
3)     if (Counter$_j[h_i(I_j)] = 0$) Vector$_j[h_i(I_j)] \leftarrow 0$
4) DeleteHashTable$(I_j)$
```

Clearly, the complexity of an add/delete operation is simply k memory accesses to increment/decrement the counter and at the most k *parallel* memory accesses to update the associated Bloom filter, plus one more memory accesses to insert/delete the prefix in the hash table.

C. Effect of False Positives

We now show the relationship between false positive probability and its effect on the throughput of the system. We measure the throughput of the system as a function of the average number of hash probes per lookups, E_{avg}. The worst case performance for the basic configuration of our system is W_{dist} hash probes per lookup. We later show how the worst case can be limited to two hash probes and one array access per lookup by trading off some memory efficiency. Our goal is to optimize average performance and bound the worst case such that our system provides equal or better performance than TCAM based solutions under all traffic conditions using commodity memory technology.

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. For an address which matches a prefix of length l, we first examine any prefix lengths greater than l represented in the *match vector*. For the basic configuration of the system, we assume that all Bloom filters share the same false positive probability, f. We later show how this can be achieved by selecting appropriate values for m for each filter. Let B_l represent the number of Bloom filters for the prefixes of length greater than l. Hence, the average number of additional hash probes required when matching a length l prefix is

$$E_l = B_l f. \tag{5}$$

Let B be the total number of Bloom filters in the system for a given configuration. The worst case value of E_l, which we denote as E_{add}, can be expressed as

$$E_{\text{add}} = Bf. \tag{6}$$

This is the maximum number of additional hash probes per lookup, independent of input address. Since these are the average additional probes due to the false positives, the total number of average hash probes per lookup for any input address is upper bounded as

$$E_{\text{avg}} \leq E_{\text{add}} + 1 = Bf + 1 \tag{7}$$

where the additional one probe accounts for the probe at the matching prefix length. Note that there is the possibility that an IP address creates false positive matches in all the filters in the system. In this case, the number of required hash probes is

$$E_{\text{worst}} = B + 1. \tag{8}$$

Thus, while (7) gives the average number of hash probes for a longest prefix match, (8) provides the maximum number of hash probes for a worst case lookup. Since both values depend on B, the number of filters in the system, reducing B is important for limiting the worst case. Note that for the basic configuration the value of B is simply W_{dist}. The remainder of this paper addresses the issues of filter dimensioning, design trade-offs, and bounding the worst case.

V. Configuration and Optimization

In this section, we seek to develop a system configuration that provides high performance independent of prefix database characteristics and input address patterns. The design goal is to architect a search engine that achieves an average of one hash probe per lookup, bounds the worst case search, and utilizes a small amount of embedded memory. Several variables affect system performance and resource utilization:

- N, the target amount of prefixes supported by the system
- M, the total amount of embedded memory available for Bloom filters
- W_{dist}, the number of unique prefix lengths supported by the system
- m_i, the size of each Bloom filter
- k_i, the number of hash functions computed in each Bloom filter
- n_i, the number of prefixes stored in each Bloom filter.

For clarity, we will use the case of IPv4 throughout the following sections. Implications for IPv6 will be discussed in Section VII. IPv4 addresses are 32 bits long; therefore, $W = 32$ and W_{dist} depend on the characteristics of the database. Given that current IPv4 BGP tables are in excess of 100 000 entries, we use $N = 200\,000$, unless otherwise noted, to illustrate the viability of our approach for future use. For all of our analysis, we set the number of hash functions per filter such that the false positive probability f is a minimum for a filter of length m. The feasibility of designing a system with selectable values of k is discussed in Section VIII.

We have established a basic intuition for system performance assuming that each individual Bloom filter has the same false positive f. We also note that as long as the false positive probability is kept the same for all the Bloom filters, the system performance is independent of the prefix distribution. Let f_i be the false positive probability of the ith 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

$$f_i = f = \left(\frac{1}{2}\right)^{\left(\frac{m_i}{n_i}\right)\ln 2} \quad \forall i \in [1, \ldots, 32]. \tag{9}$$

This implies that

$$\frac{m_1}{n_1} = \frac{m_2}{n_2} = \ldots = \frac{m_{32}}{n_{32}} = \frac{\sum m_i}{\sum n_i} = \frac{M}{N}. \tag{10}$$

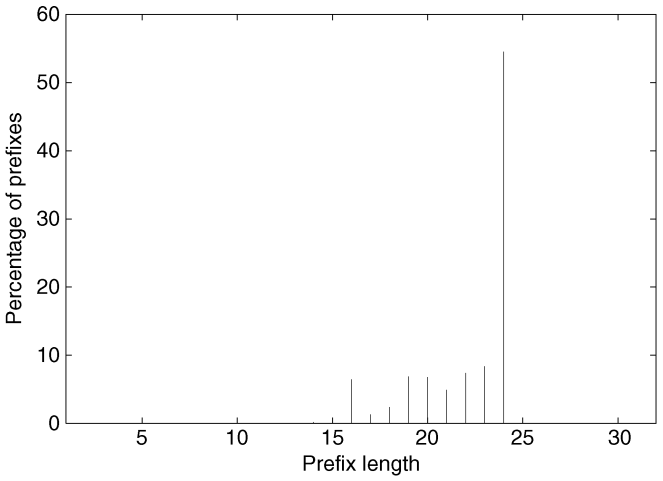


Fig. 2. Average prefix length distribution for IPv4 BGP table snapshots.

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

$$f_i = f = \left(\frac{1}{2}\right)^{\left(\frac{M}{N}\right)\ln 2}. \tag{11}$$

Based on the preceding analysis, the average number of hash probes per lookup depends only on the total amount of memory resources, M, and the total number of supported prefixes, N. It is important to note that this is independent of the number of unique prefix lengths and the distribution of prefixes among the prefix lengths.

A. Asymmetric Bloom Filters

The preceding analysis implies that memory be proportionally allocated to each Bloom filter based on its share of the total number of prefixes. Given a static, uniform distribution of prefixes, each Bloom filter would simply be allocated $m = M/B$ bits of memory. Examination of real 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. We collected 15 snapshots of IPv4 BGP tables from [1] and 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 we collected is shown in Fig. 2.

If we use a static system configured for uniformly distributed prefix lengths to search a database with nonuniform prefix length distribution, some filters are over-allocated memory while others are under-allocated; thus, the false positive probabilities for the Bloom filters are no longer equal. Clearly, we need to proportionally allocate the amount of embedded memory per filter based on its current share of the total prefixes while adjusting the number of hash functions to maintain minimal false positive probability. We refer to this configuration as "asymmetric Bloom filters" and describe a device architecture capable of supporting it in Section VIII. Using (7) for the case

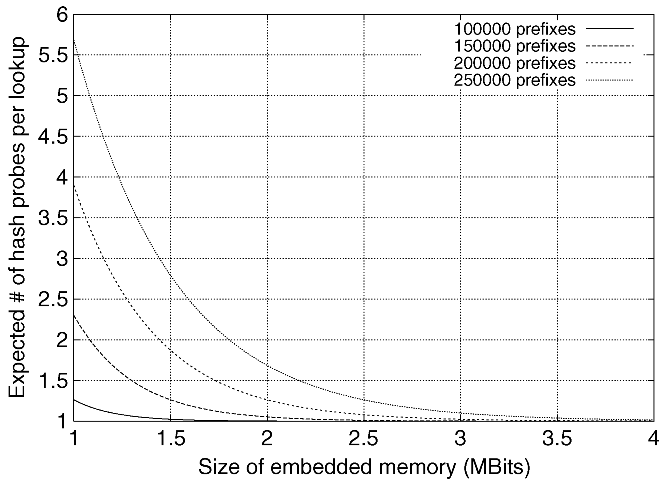


Fig. 3. Average number of hash probes per lookup, E_{avg}, versus total embedded memory size, M, for various values of total prefixes, N, using a basic configuration for IPv4 with 32 asymmetric Bloom filters.

of IPv4, the average number of hash probes per lookup, E_{avg}, may be expressed as

$$E_{\text{avg}} = 32 \times \left(\frac{1}{2}\right)^{\frac{M \ln 2}{N}} + 1. \tag{12}$$

Given the feasibility of asymmetric Bloom filters, we plot the average number of hash probes per lookup, E_{avg}, versus total embedded memory size M for various values of N in Fig. 3. With a modest 2-Mb embedded memory, the average number of hash probes per lookup is less than two for 250 000 prefixes. We assert that such a system 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 provides near optimal average performance of one hash probe per lookup. Using (8), the worst case number of dependent memory accesses is simply 33. Note that the term for the access for the matching prefix may be omitted, as the default route can be stored internally; hence, the worst case number of dependent memory accesses is 32.

B. Direct Lookup Array

The preceding analysis showed how asymmetric Bloom filters can achieve near optimal average performance for large numbers of prefixes with a modest amount of embedded memory. We now examine ways to bound the worst case number of hash probes without significantly affecting average performance. We observe from the distribution statistics that sets associated with the first few prefix lengths are typically empty and the first few nonempty sets hold few prefixes as shown in Fig. 2. This suggests that utilizing a direct lookup array for the first a prefix lengths is an efficient way to represent shorter prefixes while reducing the number of Bloom filters. For every prefix length we represent in the direct lookup array, the number of worst case hash probes is reduced by one. Use of a direct lookup array also reduces the amount of embedded memory required to achieve optimal average performance, as the number of prefixes represented by Bloom filters is decreased.

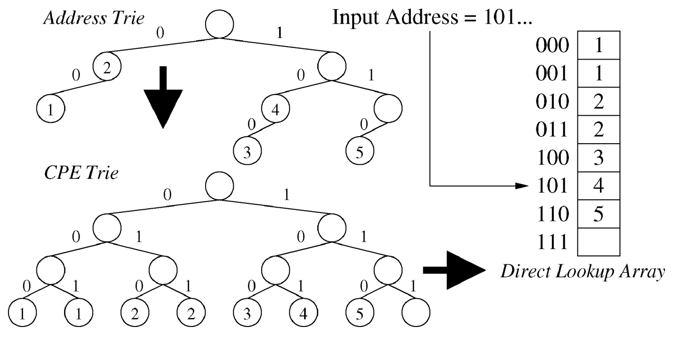


Fig. 4. Example of direct lookup array for the first three prefix lengths.

We now clarify what we mean by a direct lookup array. An example of a direct lookup array for the first $a = 3$ prefixes is shown in Fig. 4. We begin by storing the prefixes of length $\leq a$ in a binary trie. We then perform CPE for a stride length equal to a [20]. The next hop associated with each leaf at level a is written to the array slot addressed by the bits labeling the path from the root to the leaf. The structure is searched by using the first a bits of the IP destination address to index into the array. For example, an address with initial bits 101 would result in a next hop of 4. Note that this data structure requires $2^a \times NH_{\text{len}}$ bits of memory where NH_{len} is the number of bits required to represent the next hop.

For the purpose of a realistic analysis, we select $a = 20$ resulting in a direct lookup array with 1M slots. For a 256-port router where the next hop corresponds to the output port, 8 bits are required to represent the next hop value and the direct lookup array requires 1 MB of memory. Use of a direct lookup array for the first 20 prefix lengths leaves prefix lengths 21,. . .,32 to Bloom filters; hence, the expression for the average number of hash probes per lookup becomes

$$E_{\text{avg}} = 12 \times \left(\frac{1}{2}\right)^{\frac{M \ln 2}{N - N_{[1:20]}}} + 1 \tag{13}$$

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

On average, the $N_{[1:20]}$ prefixes constituted 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. Given this distribution of prefixes, the average number of hash probes per lookup versus total embedded memory size for various values of N is shown in Fig. 5. The average 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 reduces the average number of hash probes per lookup to less than 1.1 for 250 000 prefix databases. While the amount of memory required to achieve good average performance has decreased to only 4 bits per prefix, the worst case hash probes per lookup is still large. Using (8), the worst case number of dependent memory accesses becomes $E_{\text{worst}} = (32 - 20) + 1 = 13$.

For an IPv4 database containing the maximum of 32 unique prefix lengths, the worst case is 13 dependent memory accesses per lookup. A high-performance implementation option is to make the direct lookup array the final stage in a pipelined search

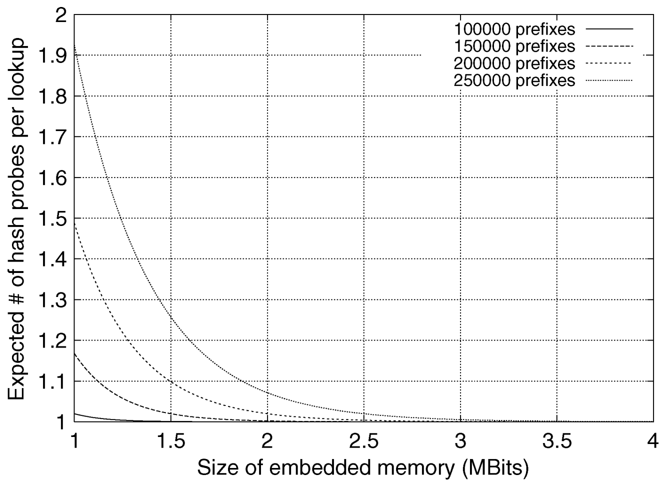


Fig. 5. Average number of hash probes per lookup, E_{avg}, versus total embedded memory size, M, for various values of total prefixes, N, using a direct lookup array for prefix lengths 1,. . .,20 and 12 Bloom filters for prefix lengths 21,. . .,32

architecture. IP destination addresses which reach this stage with a null next hop value would use the next hop retrieved from the direct lookup array. A pipelined architecture does require a dedicated memory bank or port for the direct lookup array. The following section describes additional steps to further improve the worst case.

C. Reducing the Number of Filters

We can reduce the number of remaining Bloom filters by limiting the number of distinct prefix lengths via further use of CPE. We would like 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, we observe in all of our sample databases that 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. Likewise, we observe in all of our 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. (Remember that 24.6% of the prefixes fall in the range of 1 to 20.)

Based on these observations, we chose to divide the prefixes not covered by the direct lookup array into 2 groups, corresponding to prefix lengths 21–24 and 25–32, respectively. Each group is expanded out to the upper limit of the group so that the first group contains only length 24 prefixes and the second group contains only length 32 prefixes. Let $N_{[21:24]}$ be the original number of prefixes of length 21 to 24 and let $N_{[25:32]}$ be the original number of prefixes of length 25 to 32. Using CPE increases the number of prefixes in each group by an "expansion factor" factor $\alpha_{[21:24]}$ and $\alpha_{[25:32]}$, respectively. We observed an average value of 1.8 for $\alpha_{[21:24]}$ in our sample databases, and an average value of 49.9 for $\alpha_{[25:32]}$. Such a large value of α_{32} is tolerable due to the small number of prefixes of length 25 to 32.

Use of this technique results in two Bloom filters and a direct lookup array, bounding the worst case lookup to two hash probes

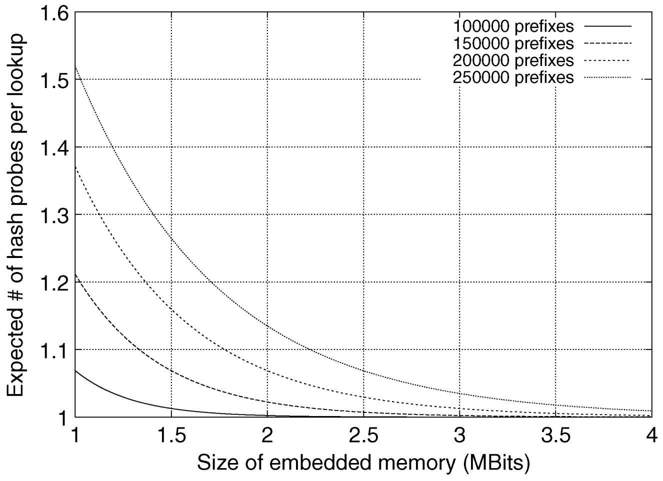


Fig. 6. Average number of hash probes per lookup, E_{avg}, versus total embedded memory size, M, for various values of total prefixes, N, using a direct lookup array for prefix lengths 1,. . .,20 and two Bloom filters for prefix lengths 21,. . .,24 and 25,. . .,32

and an array lookup. The expression for the average number of hash probes per lookup becomes

$$E_{\text{avg}} = 2 \times \left(\frac{1}{2}\right)^{\frac{M \ln 2}{\alpha_{[21:24]} N_{[21:24]} + \alpha_{[25:32]} N_{[25:32]}}} + 1. \qquad (14)$$

Using the observed average distribution of prefixes and observed average values of $\alpha_{[21:24]}$ and $\alpha_{[25:32]}$, the average number of hash probes per lookup versus total embedded memory M for various values of N is shown in Fig. 6. The average 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 average 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 provides for a maximum of two hash probes and one array access per lookup while maintaining near optimal average performance with modest use of embedded memory resources.

VI. Performance Simulations

In the discussion so far we have described three system configurations, each offering an improvement over the earlier in terms of worst case performance. In this section, we present simulation results for each configuration using forwarding tables constructed from real IPv4 BGP tables. We will refer to each configuration as follows.

- Scheme 1: This scheme is the basic configuration which uses asymmetric Bloom filters for all prefix lengths as described in Section V-A.
- Scheme 2: This scheme uses a direct lookup array for prefix lengths [0,. . .,20] and asymmetric Bloom filters for prefix lengths [21,. . .,32] as described in Section V-B.
- Scheme 3: This scheme uses a direct lookup array for prefix lengths [0,. . .,20] and two asymmetric Bloom filters for CPE prefix lengths 24 and 32 which represent prefix lengths [21,. . .,24] and [25,. . .,32], respectively. This configuration is described in Section V-C.

For all schemes we set $M = 2$ Mb and adjusted m_i for each asymmetric Bloom filter according to the distribution of prefixes of the database under test. We collected 15 IPv4 BGP tables from [1]. For each combination of database and system configuration, we computed the theoretical value of E_{avg} using (12)–(14).

A simulation was run for every combination of database and system configuration. The ANSI C function $rand()$ was used to generate hash values for the Bloom filters as well as the prefix hash tables. The collisions in the prefix hash tables were around 0.8% which is negligibly small. In order to investigate the effects of input addresses on system performance, we used various traffic patterns varying from completely random addresses to only addresses with a valid prefix in the database under test. In the latter case, the IP addresses 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 completely randomly generated IP addresses generated no false positives in any of our tests. The false positives increased as the traffic pattern contained more IP addresses corresponding to the prefixes in the database. 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 system configurations, along with the corresponding theoretical values, are shown in Table I. 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.

Using Scheme 3, the average number of hash probes per lookup over all test databases is found to be 1.003, which corresponds to a lookup rate of about 332M lookups per second with a commodity SRAM device operating at 333 MHz. This is a speedup of 3.3X over state-of-the-art TCAM-based solutions. At the same time, the scheme has a worst case performance of two hash probes and one array access per lookup. Assuming that the array is stored in the same memory device as the tables, worst case performance is 110M lookups per second, which exceeds current TCAM performance.

Note that the values of the average hash probes per lookup as shown by the simulations generally agree with the values predicted by the equations. We now provide a direct comparison between theoretical performance and observed performance. To see the effect of total embedded memory resources, M, for Bloom filters, we simulated Scheme 3 with database 1 with $N = 116\,189$ 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 can be attributed to the fact that the distribution of

TABLE I
OBSERVED AVERAGE NUMBER OF HASH PROBES PER LOOKUP FOR 15 IPv4 BGP TABLES ON VARIOUS SYSTEM CONFIGURATIONS DIMENSIONED WITH $M = 2$ Mb

		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	1.005404	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	1.004443	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	117,211	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	1.008874	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**

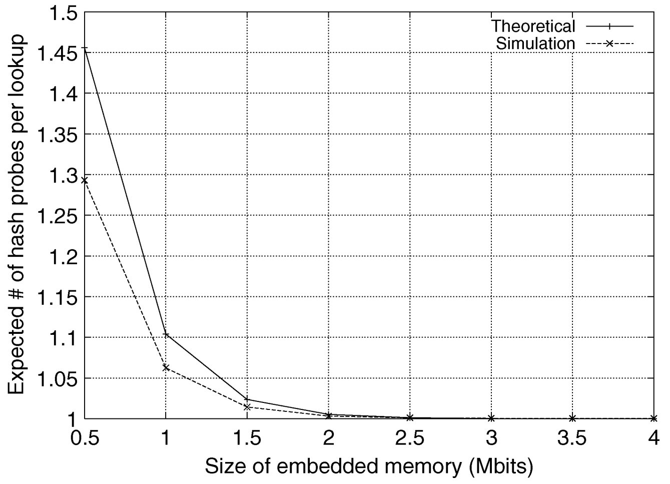


Fig. 7. Average number of hash probes per lookup for Scheme 3 programmed with database 1, $N = 116,819$, for various embedded memory sizes, M.

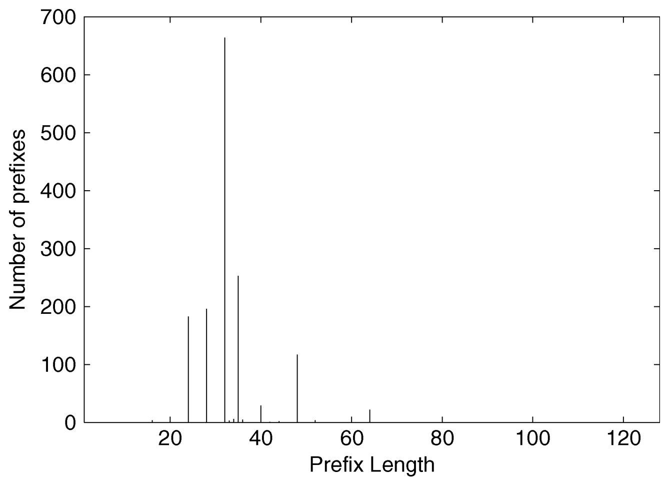


Fig. 8. Combined prefix length distribution for IPv6 BGP table snapshots.

input addresses 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 and less likely to require an array access.

VII. IPv6 PERFORMANCE

We have shown that the number of dependent memory accesses per lookup can be held constant given that memory resources scale linearly with database size. Given this characteristic of our algorithm and the memory efficiency demonstrated for IPv4, we assert that our technique is suitable for high-speed IPv6 route lookups. The primary issue for IPv6 scalability of our technique is not lookup performance, but memory usage. In order for our technique to easily scale to IPv6, the number of unique prefix lengths in IPv6 databases must be manageable.

In order to assess the current state of IPv6 tables, we collected five IPv6 BGP table snapshots from several sites [2]. Since the tables are relatively small, we computed a combined distribution of prefix lengths. Fig. 8 shows the combined distribution for a total of 1550 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 we studied. We now investigate IPv6 address architecture and deployment policies to gain a sense of whether or not the number of unique prefix lengths is expected to grow significantly.

A. Address Architecture

The addressing architecture for IPv6 is detailed in RFC 3513 [11]. In terms of the number of prefix lengths in forwarding tables, the important address type is the global unicast address which may be aggregated. RFC 3513 states that IPv6 unicast addresses may be aggregated with arbitrary prefix lengths like IPv4 addresses under CIDR. While this provides extensive flexibility, we do not foresee that this flexibility necessarily results in an explosion of unique prefix lengths. The global unicast address format has three fields: a global routing prefix, a subnet ID, and an interface ID. All global unicast addresses, other than those that begin with 000, must have a 64-bit interface ID in the Modified EUI-64 format. These identifiers may be of global or local scope; however, we are only interested in the structure they impose on routing databases. In such cases, the global routing prefix and subnet ID fields must consume a total of 64 bits.

Global unicast 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. We anticipate that this special class of global unicast addresses will not contribute many unique prefix lengths to IPv6 routing tables.

B. Address Allocation & Assignment

On June 26, 2002, in a memo titled, "IPv6 Address Allocation and Assignment Policy," the Internet Assigned Numbers Authority (IANA) announced initial policies governing the distribution or "licensing" of IPv6 address space [12]. One of its stated goals is to distribute address space in a hierarchical manner so as to "permit the aggregation of routing information by ISPs, and to limit the expansion of Internet routing tables." To that end, the distribution process is also hierarchical. IANA has made initial distributions of /16 address blocks to existing Regional Internet Registries (RIRs). The RIRs are responsible for allocating address blocks to National Internet Registries (NIRs) and Local Internet Registries (LIRs). The LIRs and NIRs are responsible for assigning addresses and address blocks to end users and Internet Service Providers (ISPs).

The minimum allocation of address space to Internet Registries is in units of /32 blocks. IRs 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 when only one device interface is required. While it is not clear how much aggregation will occur due to ISPs assigning multiple /48 blocks, the allocation and assignment policy does provide significant structure. If these policies are followed, we anticipate that IPv6 routing tables will not contain significantly more unique prefix lengths than current IPv4 tables.

We assert that our longest prefix matching approach is a viable mechanism for IPv6 routing lookups. Due to the longer "strides" between hierarchical boundaries of IPv6 addresses, use of CPE to reduce the number of Bloom filters may not be practical. In this case, a suitable pipelined architecture may be employed to limit the worst case memory accesses.

VIII. Implementation Considerations

We now provide a brief discussion of relevant implementation issues when targeting our approach to hardware. The two most important issues to address are supporting variable prefix length distributions and supporting multiple hash probes to the embedded memory.

A. Variable Prefix Length Distributions

From previous discussions, it is clear that Bloom filters which are designed to suit a particular prefix length distribution tend to perform better. However, an ASIC design optimized for a particular prefix length distribution, will have sub-optimal performance if the distribution varies drastically. Note that this can happen even if the new distribution requires the same aggregate embedded memory resources as before. Thus, in spite of the available embedded memory resources, this inflexibility in

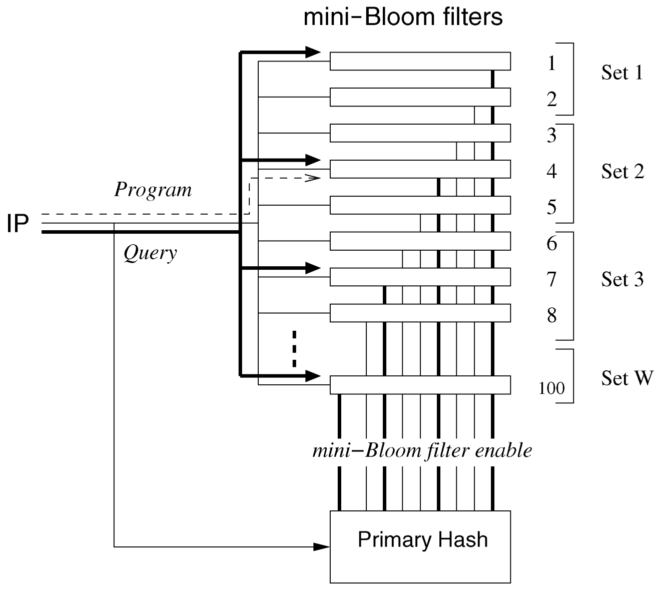


Fig. 9. Mini-Bloom filters allow system to adapt to prefix distribution. Dashed line shows programming path for a prefix of length 2. Solid line illustrates query path for input IP address.

allocating resources to different Bloom filters can lead to poor system performance. The ability to support a lookup table of certain size, irrespective of the prefix length distribution is a desirable feature of this system.

Instead of building distribution dependent memories of customized size, we propose building a number of small fixed-size Bloom filters called *mini-Bloom filters*. Let the dimensions of a mini-Bloom filter be an m'-bit-long vector with a capacity of n' prefixes. The false positive probability of the mini-Bloom filter is

$$f' = \left(\frac{1}{2}\right)^{\left(\frac{m'}{n'}\right) ln2}. \tag{15}$$

Instead of allocating a fixed amount of memory to each of the Bloom filters, we now proportionally allocate multiple mini-Bloom filters according to the prefix distribution. In other words, we allocate on-chip resources to individual Bloom filters in units of mini-Bloom filters instead of bits. While building the database, we uniformly distribute the prefixes of a particular length across the set of mini-Bloom filters allocated to it, storing each prefix in only one mini-Bloom filter. We achieve this uniform random distribution of prefixes within a set of mini-Bloom filters by calculating a primary hash over the prefix. The prefix is stored in the mini-Bloom filter pointed to by this primary hash value, within the set, as illustrated by the dashed line in Fig. 9.

In the query process, a given IP address is dispatched to all sets of mini-Bloom filters for distinct prefix lengths on a bus. The same primary hash function is calculated on the IP address to find out which one of the mini-Bloom filters 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 in the set associated with a particular prefix length as shown by the solid lines in Fig. 9.

Now we analyze the implications of this modification. Since the prefix is hashed or probed in only one of the mini-Bloom fil-

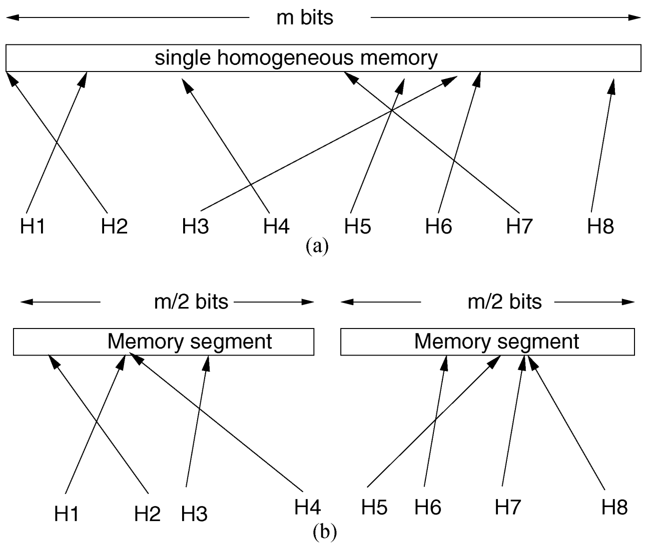


Fig. 10. (a) A bloom filter with single memory vector with $k = 8$. (b) Two bloom filters of length $m/2$ with $k = 4$, combined to realize a m-bit-long bloom filter with $k = 8$.

ters in each set, the aggregate false positive probability of a particular set is the same as the false positive probability of an individual mini-Bloom filter. Hence, the false positive probability of the new system remains unchanged if the average memory bits per prefix in the mini-Bloom filter is the same as the average memory bits per prefix in the original scheme. The importance of this scheme is that the allocation of the mini-Bloom filters for different prefix lengths can be changed unlike in the case of hardwired memory. The tables which indicate the prefix length set and its corresponding mini-Bloom filters can be maintained on-chip with reasonable hardware resources. The resource distribution among different sets can be reconfigured by updating these tables. This flexibility makes this design independent of prefix length distribution.

B. Multi-Port Embedded Memories

The number of hash functions k, is essentially the lookup capacity of the memory storing a Bloom filter. Thus, $k = 6$ implies that six random locations must be accessed in the time alloted 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. We assert that fabrication of six to eight *read ports* for an on-chip RAM is attainable with today's embedded memory technology [6].

For designs with values of k higher than what can 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 four, then two such smaller memories are required to achieve a lookup capacity of eight as shown in Fig. 10(b). The basic Bloom filter allows any hash function to map to any bit in the vector. It is possible that for some members, more than four hash functions map to the same memory segment, thereby exceeding the lookup capacity of the memory. This problem can be solved by restricting the range of each hash function to a given memory. This avoids collision among hash functions across different memory segments.

In general, if h is the maximum lookup capacity of a RAM as limited by the technology, then k/h such memories of size $m/(k/h)$ can 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 can be expressed as

$$f' = \left[1 - \left(1 - \frac{1}{\frac{m}{\left(\frac{k}{h}\right)}}\right)^{hn}\right]^{\left(\frac{k}{h}\right)h} \approx \left[1 - e^{-\frac{nk}{m}}\right]^{k}. \quad (16)$$

Comparing (16) with (2), we see that restricting the number of hash functions mapping to a particular memory does not affect the false positive probability provided the memories are sufficiently large.

IX. Conclusion

We have introduced a Longest Prefix Matching (LPM) algorithm that employs Bloom filters to efficiently narrow the scope of the search. In order to optimize average performance, we introduce asymmetric Bloom filters which allocate memory resources according to prefix distribution and provide viable means for their implementation. We show that via the use of a direct lookup array and use of 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. We provided evidence for the future viability of our approach for IPv6 route lookups.

If implemented in current semiconductor technology and coupled with a commodity SRAM device operating at 333 MHz, our algorithm could achieve average performance of over 300M lookups per second and worst case performance of over 100M lookups per second. In comparison, state-of-the-art TCAM-based solutions for LPM provide 100M 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. While the cost of TCAMs may decrease and power-saving features may emerge, we assert that SRAM technologies will always provide faster access times and lower power consumption due to the massive parallelism inherent in TCAM architectures. As a result, algorithms such as ours that employ commodity RAM devices and achieve comparable or better performance will continue to provide an attractive alternative to TCAM-based solutions.

Acknowledgment

The authors would like to thank Dr. J. Lockwood, Dr. J. Turner, and Dr. R. Chamberlain for their support, and R. Janakiraman for insightful discussions on Bloom filters. The authors would also like to thank the anonymous reviewers and their SIGCOMM'03 shepherd, Dr. C. Partridge, for their thoughtful suggestions.

REFERENCES

[1] BGP Table Data. Feb. 2003 [Online]. Available: http://bgp.potaroo.net/

[2] IPv6 Operational Report. Feb. 2003 [Online]. Available: http://net-stats.ipv6.tilab.com/bgp/bgp-table-snapshot.txt/

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[4] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: a survey," in *Proc. 40th Annu. Allerton Conf.*, Oct. 2002.

[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, 1997, pp. 3–14.

[6] B. Dipert, "Special purpose SRAMs smooth the ride," *EDN*, pp. 93–104, Jun. 1999.

[7] W. N. Eatherton, "Hardware-based Internet Protocol prefix lookups," M.S. thesis, Electr. Eng. Dept., Washington Univ., St. Louis, MO, 1998. [Online]. Available: http://www.arl.wustl.edu/

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[9] S. Fuller, T. Li, J. Yu, and K. Varadhan, Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. RFC 1519, Sep. 1993.

[10] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," in *Proc. IEEE INFOCOM*, 1998, pp. 1240–1247.

[11] R. Hinden and S. Deering, Internet Version 6 (IPv6) addressing architecture. RFC 3513, Apr. 2003.

[12] IANA, IPv6 Address Allocation and Assignment Policy, Jun. 2002. [Online]. Available: http://www.iana.org/ipaddress/ipv6-allocation-policy-26jun02

[13] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Trans. Netw.*, vol. 7, no. 3, pp. 324–334, Jun. 1999.

[14] A. J. McAulay and P. Francis, "Fast routing table lookup using CAMs," in *Proc. IEEE INFOCOM*, 1993, pp. 1382–1391.

[15] Micron Technology Inc., 36 Mb DDR SIO SRAM 2-Word Burst. Datasheet, Dec. 2002.

[16] Micron Technology Inc., Harmony TCAM 1Mb and 2Mb. Datasheet, Jan. 2003.

[17] R. K. Montoye, " Apparatus for storing "Don't Care" in a content addressable memory cell," U.S. Patent 5,319,590, Jun. 7, 1994.

[18] SiberCore Technologies Inc., SiberCAM Ultra-18M SCT1842. Product Brief, 2002.

[19] K. Sklower, A tree-based routing table for Berkeley Unix Univ. California, Berkeley, 1993, Technical report.

[20] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *Proc. ACM SIGMETRICS*, 1998, pp. 1–10.

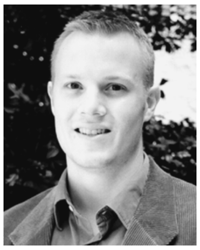
[21] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing table lookups," in *Proc. ACM SIGCOMM*, Sep. 1997, pp. 25–36.

Sarang Dharmapurikar received the B.E. degree in electrical and electronics engineering from Birla Institute of Technology and Science (BITS), Pilani, India, in 1999.

He worked for Wipro Global R&D, Bangalore, India, from June 1999 to June 2000 as a VLSI Systems Designer, where he was part of a team which designed Gigabit router chips. He is currently pursuing the Ph.D. degree in the Department of Computer Science and Engineering, Washington University, St. Louis, MO. He is a member of the Applied Research Laboratory (ARL) where he is doing research on different aspects of very high-speed networking devices. Particularly, his research interests include algorithm design and hardware implementation of high-speed deep packet processing systems.

Praveen Krishnamurthy received the B.E. degree in electronics and communication engineering from the University of Madras, India, in 2000, and the M.S degree in computer engineering from Washington University, St. Louis, MO, in 2002. He is currently pursuing the Ph.D. degree in the Department of Computer Science and Engineering at Washington University, where he is also a member of the Computer Communications Research Center (CCRC).

His research interests include modeling of complex computer systems, micro-architecture design, and design and analysis of resource-aware algorithms for pipelined multi-processors.



David E. Taylor (M'04) received the B.S. degree in electrical and computer engineering in December 1998, the M.S. degree in electrical and computer engineering in May 2002, and the Doctor of Science degree in computer engineering in August 2004 from Washington University, St. Louis, MO.

He is currently developing high-performance reconfigurable hardware systems at Exegy Inc. Prior to joining Exegy, he was a Visiting Assistant Professor in the Department of Computer Science and Engineering and was actively involved in computer communications research at the Applied Research Laboratory at Washington University, St. Louis. His research interests include the design and analysis of scalable searching algorithms and architectures, IP lookup and packet classification algorithms, high-performance reconfigurable hardware systems, programmable routers, and network processors. He held a Research Internship with the network processor hardware group at the IBM Zurich Research Laboratory during the summer of 2002.

Dr. Taylor has been a member of the ACM since 2004.