This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2009 proceedings.

IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards

Haoyu Song, Fang Hao, Murali Kodialam, T.V. Lakshman
Bell Labs, Alcatel-Lucent
Holmdel, NJ, 07733, USA
{haoyusong, fangh, muralik, lakshman}@alcatel-lucent.com

***Abstract*—Internet line speeds are expected to reach 100Gbps in a few years. To match these line rates, a single router line card needs to forward more than 150 million packets per second. This requires a corresponding amount of longest prefix match operations. Furthermore, the increased use of IPv6 requires core routers to perform the longest prefix match on several hundred thousand prefixes varying in length up to 64 bits. It is a challenge to scale existing algorithms simultaneously in the three dimensions of increased throughput, table size and prefix length. Recently, Bloom filter-based IP lookup algorithms have been proposed. While these algorithms can take advantage of hardware parallelism and fast on-chip memory to achieve high performance, they have significant drawbacks (discussed in the paper) that impede their use in practice. In this paper, we present the Distributed and Load Balanced Bloom Filters to address these drawbacks. We develop the practical IP lookup algorithm for use in 100Gbps line cards. The regular and modular hardware architecture of our scheme directly maps to the state-of-art ASICs and FPGAs with reasonable resource consumption. Also, our scheme outperforms TCAMs on most metrics including cost, power dissipation, and board footprint.**

I. Introduction

To keep up with ever-increasing optical transmission rates, Internet core routers need to forward packets as fast as possible. This requires faster and faster implementations of packet-processing functions such as IP lookup – the Longest Prefix Matching (LPM) operation needed to determine the next-hop for incoming packets. The next hop is determined by first performing a longest prefix match of the destination IP address of incoming packets against a set of prefixes stored in a prefix table. Once a match is found, the next-hop information associated with the matched prefix is retrieved. The prefix table can typically have a few hundred thousand prefixes, with prefix lengths varying from 8 to 32 for IPv4 addresses. For IPv6, the prefix lengths can vary from 16 to 64 bits.

The challenges in implementing the lookup operation are in accommodating large table sizes, achieving the 150 million or more lookups per second needed for the new 100Gbps interfaces while keeping memory, power consumption and board footprint low. The number of IPv4 prefixes in a core router BGP table has exceeded 250K recently, and is increasing at a rate of a few tens of thousands of prefixes a year [13]. While the current IPv6 table is still relatively small, the envisaged large scale deployment of IPv6 will result in table sizes at least as large as that for IPv4. High-end core routers such as Cisco's CRS-1 [14] and Juniper's T640 [15] currently can have 40G line cards with a packet forwarding rate of about 50Mpps. Meanwhile, 100GbE standards are expected to be completed by 2010. To meet ever-growing traffic needs, pre-standard 100GbE products (needing a forwarding rate of 150 Mpps) are expected to be available in the market in about the same time frame.

One possibility for IP lookups is the use of TCAMs. TCAMs support 250M+ lookups per second. However, their high power dissipation and large footprint are major disadvantages. To limit the overall system power consumption, only a few hundred watts are typically budgeted for each line card. A TCAM chip alone easily consumes more than 20W of power. For 100Gbps line cards it is conceivable that many other necessary components such as the high speed transceivers will inevitably consume more power necessitating a corresponding reduction in power consumption elsewhere in order to stay within the allotted power budget. In addition, more and more functions and modules need to fit on a line card making footprint an important metric. It is desirable to avoid the use of low density chips when possible. It has long been argued that the brute-force way of searching prefixes in TCAM results in very inefficient storage and uses too many bits per prefix. For example, a TCAM uses 16 transistors to store a bit while an SRAM uses only six. IPv6 results in a two-fold worsening over IPv4 because of the much longer prefixes that need to be stored. In addition to these disadvantages, an incremental prefix update in TCAM involves as many memory operations as the number of unique prefix lengths. This causes an extra performance penalty for IPv6. Because of these disadvantages, TCAMs are better suited when the performance of algorithmic alternatives cannot match as in the case of packet classification and deep packet inspection.

For IP lookup, algorithmic approaches can achieve high rates with compact storage needs. Compact storage permits the use of fast memories such as SRAMs economically and also the effective use of on-chip memory to achieve high throughputs.

The first IP lookup algorithms were implemented in software running on host-based routers with slow SDRAMs used for storage. The next generation of chassis-based routers had dedicated line cards for packet forwarding, with faster but smaller SRAMs on the line cards used to store the prefix table. As the throughput needs outpaced SRAM speeds, on-chip memory began getting used as caches of the prefix table

978-1-4244-3513-5/09/$25.00 ©2009 IEEE

to facilitate faster IP lookups. Currently, a few tens of megabits of fast memory on a chip is feasible and effective use of on-chip memory has proved to be very critical in satisfying the throughput requirements of the next generation network applications [8].

In this paper, we present a new lookup scheme based on a new data structure called Distributed and Load Balanced Bloom Filters (DLB-BF). This scheme is scalable to very high line speeds, can handle large table sizes, and many prefix lengths. Its hardware architecture can be directly mapped to state-of-art ASICs and FPGAs with reasonable resource consumption and can support the non-stop line-speed forwarding needed in the next generation 100Gbps core router line cards.

The paper is organized as follows: Section II discusses related work, drawbacks of some existing schemes and the motivation for our work. Section III describes our new data structure and lookup algorithm in detail. The performance of the algorithm is analyzed in Section IV. Some practical hardware implementation issues are discussed and an FPGA prototype is evaluated in Section V. Concluding remarks are in Section VI.

II. Background

A. Conventional IP Lookup Algorithms

IP lookup algorithms have been well studied in the past. The trie-based algorithms, such as LC-trie [22], Lulea [7], Multibit Trie [27], Tree Bitmap [10], and Shape Shifting Trie [26], are simple and memory efficient. However, their performance degrades linearly as the tree depth increases and this makes them unsuitable for 100Gbps IPv6 lookups.

Another approach to fast IP lookups is the use of memory pipelines [12], [18], [19]. A deep pipeline can be used to produce one lookup result every clock cycle but this approach has several drawbacks. Although the problem of imbalanced memory size for each pipeline stage has been solved recently, the aggregated memory bandwidth needed by all the pipeline stages is very high making it difficult to implement with commodity memory devices. The longer prefix lengths of IPv6 worsen this problem and even the use of a dedicated lookup engine with embedded memory in the pipeline stages does not reduce the complexity sufficiently.

Caching recent look-up results using on-chip memory is discussed in [6]. This approach works fine if there is sufficient temporal locality in the lookups. However, such locality is low in core routers where flows are well intermixed and the cache hit rate is low. In addition, it is preferable to use schemes that are deterministic and not subject to the variability in look-up times resulting from cache misses. Although caching using on-chip memory may not be desirable, the idea of using fast on-chip memory to achieve speed-up has motivated the use of Bloom filters as compact data-structures that can be used for implementing fast look-ups.

B. Bloom Filter

The use of Bloom filters [1] in networking applications has been of much recent research interests [3]. Numerous variations of the basic Bloom filter have been proposed for different applications [4], [5], [11]. The basic Bloom filter is a memory-efficient data structure that stores a "signature" of an item using just a few bits, regardless of the size of the item itself. Given a set of n items and an m-bit array, each item sets up to k bits in the array using k independent hashes to index into the array. Due to hash collisions, a bit can be set by multiple items. When querying the membership of an item, the item is first hashed using the same set of hash functions and then the k bits to which the hash values point are examined. If any bit is zero, one can be certain that the item is not a member. If all the k bits are one, the item can be claimed to be a member with a small false positive probability. The false positive probability p_f is a function of m, n, and k and can be computed as below:

$$p_f = (1 - e^{-kn/m})^k \tag{1}$$

Bloom filters have the advantage that they can represents a set of items very compactly and hence making them amenable for on-chip storage. However, several issues need to be considered in making Bloom filters viable for fast IP lookups.

First, to minimize false positive probabilities, the number of hash functions needed by the Bloom filter can be large (specifically, $k = m \ln 2/n$ is the optimal value). If the Bloom filter is implemented using a single-port memory block, one Bloom filter lookup takes as many memory accesses as the number of hash functions. This can make the achievable throughput far below what is desired. Multi-port memories can resolve this issue by permitting multiple simultaneous accesses to memory. An N-port memory leads to $N\times$ speedup over a single-port memory. However, several practical constraints limit the number of memory-ports. Multiple ports increase the pin count, the power consumption, and the footprint of the memory module. Therefore, N cannot be large. Although memories with 3 or more ports are possible, the most practical option is a 2-port memory. This port constraint has to be accounted for in the design of fast Bloom-filter based lookup schemes.

Second, a good universal hash function is computationally intensive and can lower throughput. Ideally, the hash function must be computable in one clock cycle with low logic needs and without the use of pipelines that can introduce latency. Fast computation of a large number of good, independent hash functions is a challenge.

Third, the possibility of false positives requires a second verification step. This second step is also needed to access the next-hop and other associated information since the Bloom filter itself cannot directly store this information. This second step must also be performed in an efficient manner to avoid performance bottlenecks that might offset the gains from use of the Bloom filter.

We address all these issues in our proposed Bloom-filter based method for fast IP lookups.

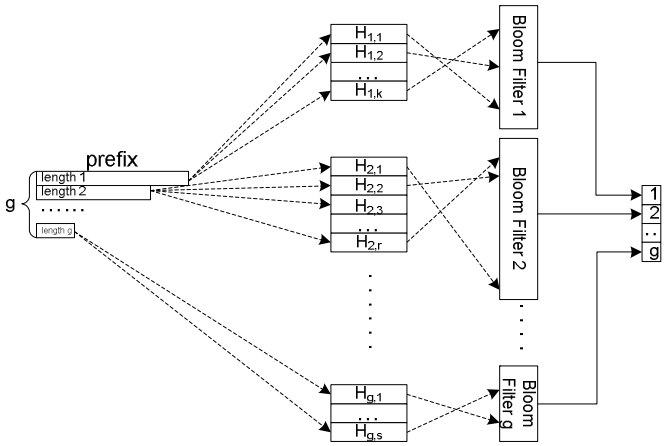

Fig. 1. Using Bloom Filters for IP Lookup

C. IP Lookup using Bloom Filters

The idea of using Bloom filters for IP lookup was first proposed in [8]. We refer to this scheme as PBF in the rest of the paper. PBF essentially is a hash-based algorithm. IP lookup using hash tables has been proposed before [21], [24], but Bloom filters use on-chip memory much better and exploit the intrinsic hardware parallelism.

Figure 1 shows the basic architecture of PBF. First, IP prefixes are partitioned into groups based on their prefix lengths. Next, each group is assigned a Bloom filter that stores the prefixes in that group. For IP lookup, the Bloom filters for all groups are checked in parallel. A subset of the Bloom filters may report a match. This could be due to finding matching prefixes in the Bloom filters or due to false positives. To verify that there is an actual match, we start with the Bloom filter corresponding to the longest prefix (since we are interested in finding the longest matching prefix) and check the match using an off-chip prefix table. If a true match is found, we retrieve the necessary next-hop information. Otherwise, we continue the check using the next longest match indicated by the Bloom filters. Typically, we expect only one off-chip lookup using a hash table to find the real match since the false positive probabilities are low by design. This scheme is simple and can have good average case performance. However, as we discuss below, it has several drawbacks that makes practical use difficult.

One issue is that when multiple Bloom filters indicate false positives, the off-chip prefix table has to be searched multiple times (equal to the number of prefix lengths in the worst case). Although this may happen very infrequently, the resulting variability in forwarding rate could be an issue since worst-case performance is an important metric in router design. One method for improving the worst-case performance here is to reduce the number of prefix-length groups. This requires aggregating prefixes of different lengths into a single Bloom filter so that the number of groups (or Bloom filters) is reduced. This aggregation can be done by using prefix expansion [27]. The tradeoff is that the improvement in the worst-case performance comes at the cost of higher memory use because prefix expansion can significantly increase the size of the prefix table [27]. Experiments show a greater than five-fold table size expansion when the number of prefix-length groups is reduced to 3 (with thresholds of 20, 24, and 32) for a moderate sized IPv4 table with about 100K prefixes [8]. The expansion is worse for larger tables. In addition, prefix expansion makes routing updates, which happen fairly frequently in core routers, much more time-consuming and complex. Multiple expanded prefixes need to be modified when only a single original prefix is inserted or deleted. Hence, the algorithm is not well suited for larger tables and longer prefixes.

Another issue is that the number of prefixes in each prefix-length group is highly variable and changes dynamically with routing updates. A recent snapshot of the IPv4 BGP table contains about 134K prefixes for the largest group (prefix length of 24) but only 9 for the smallest group (prefix length of 9). Meanwhile, the average prefix length keeps shifting between 21 and 24. The percentage of prefixes in the largest group of length 24 can vary from 40% to 70% [13]. To reduce the false positive rate and to best utilize scarce on-chip memory, we need to customize the size of each Bloom filter as well as the number of hash functions based on the number of prefixes in that group. We also need to be able to adapt to the current prefix distribution by adjusting the memory allocation dynamically. Engineering such a system is difficult and expensive – it requires over-provisioning of memory or the capability to reconfigure hardware. Neither of this is feasible since on-chip memory is scarce and reconfiguring FPGAs in practice takes several seconds. Also, forwarding functions are often hard-coded in ASICs which cannot be reconfigured.

A third issue is that one-cycle lookups assume that the Bloom filters are implemented using k-port memory where k is the number of hash functions. As discussed earlier, this is impractical unless k is 2 or 3.

In the next section, we propose a new data structure, Distributed and Load Balanced Bloom Filters (DLB-BF), that we use to address all these issues. Our proposed lookup scheme is well-suited for hardware implementation and can be used in 100Gbps line cards performing IPv6 lookups.

III. DLB-BF for IP Lookups

A. Basic Architecture

While the total number of prefixes is relatively stable, their distribution on length is highly dynamic. Over-provisioning multiple variable-sized Bloom filters results in inefficient memory use. It is desirable to use just one optimized Bloom filter to store all the n prefixes. Also, if we use just one Bloom filter we do not have the prefix expansion problem nor the problem of managing memory allocation for Bloom filters that vary widely in the number of elements stored (from a few prefixes to several hundreds of thousands).

Figure 2(a) illustrates the scheme using a single Bloom filter (SBF). Even though a single Bloom filter is used, each prefix group has its own set of hash functions. A nice feature is that

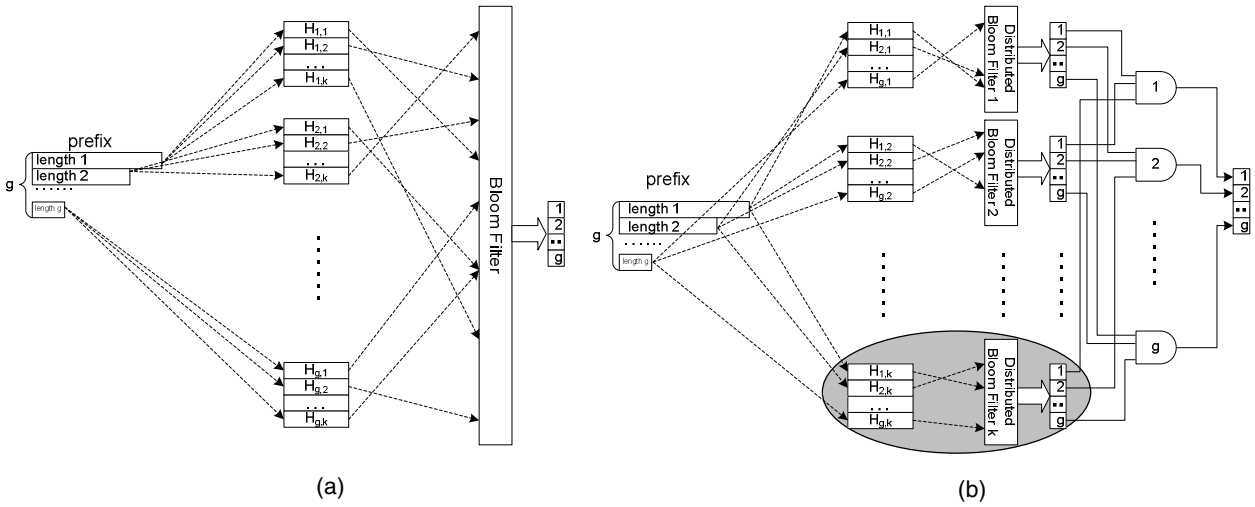

Fig. 2. Distributed and Load Balanced Bloom Filters

all prefix groups need the same number of hash functions, k, that is optimal for the SBF. Let there be x_i prefixes in prefix group i. Each prefix is hashed k times to set at most k bits in the Bloom filter. Overall, at most $\Sigma_{i=1}^{g} x_i k = nk$ bits can be set. Although this SBF architecture has many desirable properties it has a serious drawback in that it does not permit parallel searches and a sequential search is needed on each prefix length. In the worst case, up to g sequential searches are needed if g distinct prefix-length groups are present.

To overcome this drawback while retaining the advantages of SBF, we develop the DLB-BF architecture. As shown in Figure 2(b), we partition the Bloom filter into k equal sized distributed Bloom filters. We name each partition a distributed Bloom filter even though each partition by itself is no longer a real Bloom filter in the classical sense. We further regroup the g hash function groups, each with k hash functions, to k new hash function groups, each with g hash functions. This is done as described below: the i-th hash function of each original group forms the i-th new group, where i ranges from 1 to k. Each group of g hash functions is associated with a distributed Bloom filter.

Now in any new hash function group, each hash function maps to a different prefix length. This means that in the prefix storage process each prefix will set exactly one bit in the corresponding distributed Bloom filter and so at most n bits can be set. Since we have k distributed Bloom filters, kn bits can be set overall. This is the same number of bits as the SBF case.

We call the new architecture DLB-BF. This is because we have to consider all the distributed Bloom filters as a whole to check prefix membership and because all the distributed Bloom filters have the same load. This load balancing is important for a regular and modular implementation. In Section IV, we will formally prove that the false positive probability of DLB-BF is identical to the SBF as well as to that of [8] in the ideal configuration.

The new data structure allows IP lookups to be performed in parallel on all the DLB-BFs. Each DLB-BF search outputs a g-bit vector. If bit i is set, it indicates a prefix match (or possibly a false positive) in prefix-length group i. When we perform a bit-wise AND operation on all the k-bit vectors, the resulting bit vector indicates the real match with a very high probability just as in the SBF case.

B. Further Improvements

Since all the DLB-BFs can be searched in parallel, the system throughput is determined by the lookup rate of a single DLB-BF, which in turn is determined by the hash function calculation speed and the number of memory ports, as discussed in Section II.

We defer the hash function issue to Section V. Here, We focus on the DLB-BF access speed problem assuming that a single-cycle hash-function computation is possible.

Assume that only r-port SRAM blocks are available, we further partition a DLB-BF into t partial DLB-BFs with each being implemented in one r-port SRAM block. To store prefixes in the DLB-BF, each value generated by a hash function comprises two parts: the SRAM block ID and the bucket address in the SRAM block. By doing this, the load of each partial DLB-BF is balanced statistically, accepting about n/t hashes.

The shaded area in Figure 2 is enlarged in Figure 3 to show the details. For IP lookups, the g hash values in each DLB-BF is sent to the scheduler first. The task of the scheduler is to maximize the number of SRAM blocks and ports used in one cycle. The collector is responsible for reorganizing the SRAM outputs and generating the g-bit matching vector.

Ideally, if each SRAM block receives no more than r access requests, the search on each DLB-BF can be done in just one clock cycle. However, in the worst-case when all the g accesses to a DLB-BF happen to be concentrated to the same SRAM block, $\lceil g/r \rceil$ cycles are needed. We analyze the average number of accesses needed to finish one DLB-BF lookup in Section IV and show that this scheme indeed

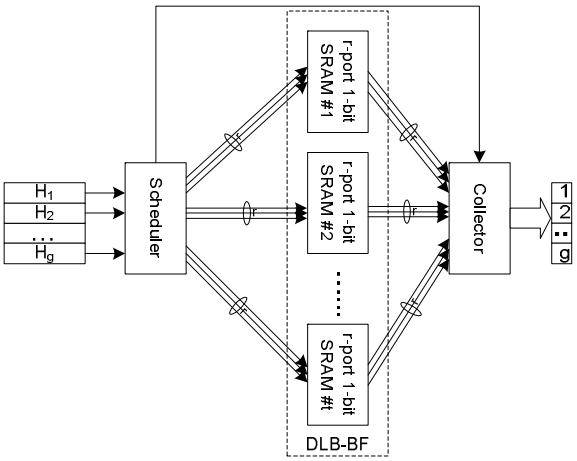

Fig. 3. Partition DLB-BF for Speedup

significantly improves the lookup performance.

Note that the functionality of this module actually mimics a $g \times t$ switch scheduler with the speedup of r, where each input port has only one request to an output port. Implementing such a switch scheduler is not trivial. We will introduce in Section V a simplified scheduling scheme which can still provide satisfactory Bloom filter lookup accuracy while being easy to implement.

C. Ad Hoc Prefix Expansion

One major argument against the use of Bloom filters for IP lookups is its poor worst-case performance when packets come at the highest possible rate and all the Bloom filters show false positives. Although this is highly unlikely, we still have to address this issue to comply with the performance requirements imposed on routers.

The real concern is that once a packet (or more specifically, an address prefix) happens to hit a bad case (i.e. multiple false positives are generated), all subsequent packets having the same address prefix are also problematic unless a longer prefix hits a real match. These packets can slow down the packet lookup rate and might eventually cause packet drops. Next, we present a scheme that can reduce consecutive false positives regardless of the packet arrival pattern.

Our design assumes a certain margin for toleration of a few false positives. For example, assume a 400MHz clock rate, the lookup budget is 400M/150M = 2.7 cycles for the maximum packet rate that can be seen on a 100GbE port, which means we have 1.7 cycles per packet to deal with the Bloom filter false positives in the worst case.

If a particular packet results in many false positives, we need to prevent subsequent packets belonging to the same flow from causing false positives, since the forwarding rate can slow down if these packets are consecutive. We use an ad-hoc prefix expansion scheme for this. When a packet causes several false positives and the longest false positive match is of length k, we extract the k-bit prefix of the packet's IP address and insert it into the off-chip prefix table along with the next hop information. For example, let us say we only allow two false positive matches, but a packet with address 192.168.11.4 results in three false matches and the first (also the longest) false match happens at length of 24. To cope with this bad case, we insert a new "expanded" prefix 192.168.11.0/24 into the prefix table. This new prefix is associated with the same next hop information as the real matching prefix. Any subsequent packets from the same flow will then be guaranteed to find the correct next hop information in just one Bloom filter access.

This scheme has three advantages: (1) Unlike the original prefix expansion scheme which is pre-determined and can exponentially increase the table size, our prefix expansion scheme is invoked dynamically, generates only one new prefix, and is used only when absolutely necessary. (2) More importantly, the on-chip Bloom filters remain intact. We only need to insert one new prefix in the off-chip prefix table. The new prefix does not change the Bloom filter load nor affect its false positive probability. (3) The actual false positive rate observed is not a function of the arrived packets but a function of the unique flows (i.e. unique destination IP addresses). When no new flow is seen, there are no more false positives.

The entire expansion scheme is managed by the system software. An expanded prefix can be revoked at any time if it is no longer necessary. The scheme significantly reduces the overall table size, simplifies the work load of incremental updates, and supports faster packet lookups.

When the number of expanded prefixes becomes too large (after the system has been operational for a long time) and causes performance degradation in the off-chip table lookup, reprogramming the DLB-BFs using the current set of prefixes (excluding those expanded ones) will help reset the state. However, due to the small Bloom filter false positive probability it is unlikely that this will need to be done anyway.

D. Off-chip Prefix Table Optimization

After the on-chip DLB-BF is searched, the off-chip prefix table also needs to be searched to verify the match and to fetch the next hop information. The off-chip prefix table is typically organized as a hash table. In PBF, it is assumed that the hash table lookup takes just one memory access [8]. This is not always true due to hash collisions. Unbounded hash collisions can cause serious performance degradation.

We can take advantage of high memory bandwidths of SRAMS to alleviate this problem. We arrange multiple collided nodes into one hash bucket such that they can be retrieved with one memory access and avoiding traversal of linked lists when hash collision happens.

Currently, 500+ MHz QDR-III SRAMs which support 72-bit read and write operations per clock cycle are available. A burst read access using two clock cycles can retrieve 144 bits. This is sufficient to pack three IPv4 prefixes or two IPv6 prefixes plus the next hops. With a 144-bit bucket size, a 72-Mbit memory can contain 500K buckets which can hold 1.5 million IPv4 prefixes or one million IPv6 prefixes.

A key problem is avoiding bucket overflow entirely or keep its occurrence to a minimum. The Fast Hash Table [25] and Peacock Hash [20] were designed to improve hash table performance. To simplify design, we adopt the scheme in [2] where each prefix is hashed using two hash functions and the prefix is stored in the less loaded bucket. Consequently, a prefix lookup needs to access the hash table two times using two hash functions. All prefixes stored in the two accessed buckets need to be compared to find the match.

Although each prefix has two choices and each bucket can store 2 or 3 prefixes, bucket overflow can still happen. However, analysis and simulation show that the overflows are extremely rare. Given our configuration, when we insert 250K IPv6 prefixes in the table, only 3 prefixes cause overflow. These overflow prefixes can be put in a small on-chip CAM with a handful of entries. For 250K IPv4 prefixes, a 36-Mbit memory can achieve the same performance.

Since each lookup needs to access memory two times and each memory access takes two clock cycles, a 500MHz SRAM can support 125M lookups per second. This is a little short of the worst-case 150Mpps lookup rate required for 100GbE line cards. We can get around this problem in two ways: (1) We can use faster SRAM devices. A 600MHz SRAM device can satisfy the worst-case requirement. (2) We can use two 36 or 18-Mbit SRAM devices in parallel, with each addressed by a hash function. This scheme provides 250M lookups per second which is way beyond the worst-case requirement and leaving more than 67% of memory bandwidth to deal with the DLB-BF false positive matches. Note that this scheme doubles the memory bandwidth but does not increase the memory size.

E. Non-stop Forwarding

Changing network conditions or configuration changes cause the routing information to be updated. The control processor, which runs the routing protocols, computes the new prefix, next-hop information and updates the forwarding table data structure on line cards accordingly. The updates must happen without interrupting the packet forwarding or causing packet mis-routing.

The update procedure for our scheme is simple: We first insert or delete the prefix to be updated from the off-chip hash table. We then modify the on-chip DLB-BFs. This can guarantee the error-free updates. For a prefix update, there is at most one memory access to each DLB-BF, and all the memory accesses can be conducted in parallel. So the impact to the system throughput is minimized. The off-chip hash table is stored in QDR SRAM, where a separate writing port is dedicated for table updates.

The modification to the on-chip DLB-BF for prefix inserting and deleting needs to use the off-line mirror Counting Bloom Filters, just as described in [8].

IV. Performance Analysis

All previous IP lookup solutions based on TCAM, Trie, and pipeline architectures are sensitive to the longest prefix length. Therefore, when used for IPv6, they suffer a performance degradation of up to two times that for the IPv4 case. The storage is also significantly increased. A key feature of our algorithm is that it is insensitive to both the prefix length as well as to the number of unique prefix lengths. Only prefix table size is important. Hence our algorithm works equally well for both IPv6 and IPv4 and is well-suited to be used for IPv6 lookups in core routers.

Now, we prove that our scheme is equivalent to PBF in its ideal configuration. The proof is split into two parts.

Theorem 1: The SBF is identical to the DLB-BF in terms of the false positive probability.

Proof: In SBF, although prefixes with different lengths use different sets of hash functions, each prefix is hashed k times. So for any prefix lookup, the false positive probability is exactly the same as that shown in Equation 1: $SBF_{p_f} = (1 - e^{-kn/m})^k$, where n is the total number of prefixes and m is the total number of Bloom filter buckets.

In DLB-BF there are k distributed Bloom filters. Each prefix is hashed just once, in each distributed Bloom filter, by using one of the g hash functions. Therefore, for a prefix lookup, a bit is set in a distributed Bloom filter with the probability of $1 - e^{-n/m'}$, where m' is the size of the distributed Bloom filter. Since there are k independent distributed Bloom filters, the false positive probability of DLB-BF is $DLBBF_{p_f} = (1 - e^{-n/m'})^k$.

Since $m' = m/k$, we get $SBF_{p_f} = DLBBF_{p_f}$. ■

Similarly, we can prove the following theorem:

Theorem 2: The DLB-BF is identical to the PBF in its ideal configuration in terms of the false positive probability and the number of hash functions used.

Proof: In the PBF algorithm, assume there are g unique prefix lengths, each with n_i prefixes, and each prefix length is assigned a Bloom filter with m_i buckets and k_i hash functions. It was proposed to assign m_i proportional to the prefix distribution, that is $m_i/n_i = m/n$ while $\Sigma_{i=1}^{g} n_i = n$ and $\Sigma_{i=1}^{g} m_i = m$. The false positive probability of each Bloom filter is $p_{f_i} = (1 - e^{-k_i n_i/m_i})^{k_i} = (1 - e^{-k_i n/m})^{k_i}$. The optimal value of k_i is equal for all the Bloom filters, which is $m \ln 2/n$. In total $gm \ln 2/n$ hash functions need to be implemented.

We have shown that the SBF achieves the same false positive probability. The optimal number of hash functions for each prefix length is also $m \ln 2/n$, so the total number of hash functions is $gm \ln 2/n$ too.

Since DLB-BF has the same number of hash functions and the same false positive probability as SBF, the theorem is proved. ■

Although at the first glance our algorithm seems to have the same resource consumption and lookup false positive probability as the PBF algorithm in its ideal configuration, in practice the performance is much better since we avoid large prefix expansion and dynamic resource allocation. Also, the modular architecture of DLB-BF greatly simplifies the actual implementation.

We also analyze the performance of partitioned DLB-BF implementation using multiple 2-port memory blocks. Figure 4

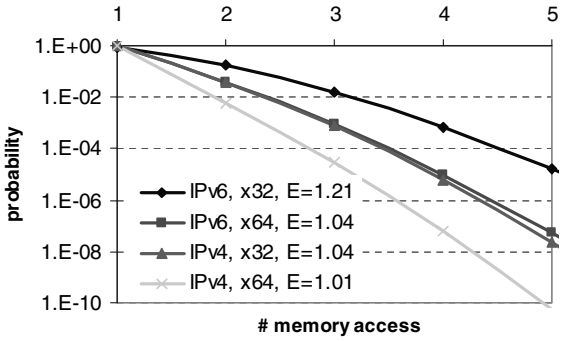

Fig. 4. Distribution of the Number of Sequential Block Memory Accesses for Different Scenarios

shows the probability distribution for the different number of sequential accesses to a same memory block. Each plot shows the result for a combination of prefix type (IPv6 or IPv4) and number of memory blocks (32 or 64). It also shows the expected number of accesses (E) for each case. In general, we expect slightly more than one clock cycle to finish the DLB-BF lookup. The results are in line with our expectation. In the next Section, we will show how a simplified scheduler can be used that results in just one clock cycle being used to finish the DLB-BF lookup.

A. Comparison with TCAMs

The cost per bit of TCAM is about $15\times$ greater than that of SRAM, and TCAM consumes more than $50\times$ as much power as SRAM does for each access [9]. Our algorithm uses about 320 SRAM bits (both on-chip and off-chip) per IPv6 prefix. The equivalent TCAM-based solution uses 72 TCAM bits plus 8 SRAM bits. Hence, compared to the TCAM solution, the DLB-BF algorithm has more than $3\times$ cost advantage and $11\times$ power advantage.

Because of the low cell density and the heat dissipation requirement of TCAMs, the dimension of a typical TCAM chip is $31mm \times 31mm$ [17] while a QDR SRAM chip's size is as small as $13mm \times 15mm$ [16]. The TCAM-based solution requires one TCAM chip plus one SRAM chip, and our algorithm requires one or two SRAM chips. Hence, the footprint of the TCAM-based solution is at least $3\times$ larger than that of the DLB-BF algorithm.

V. Implementation Considerations

A. Reference Design

A reference design uses 8-Mbit on-chip SRAMs for implementing 16 DLB-BFs. Each DLB-BF is further realized with 32 or 64 2-port SRAM blocks. There are 512 or 1024 blocks in total. Each block is 16 or 8Kb in size, configured as 1-bit array. This design is feasible in FPGA devices available in 2008. For example, Altera's Stratix IV FPGA includes more than 22Mb embedded memory including 1280 9-Kbit modules. One or two 9-Kbit modules can be combined to form one memory block. With this configuration, each prefix is hashed 16 times and we need $16 \times 48 = 768$ hash functions in place for IPv6 and $16 \times 24 = 384$ hash functions for IPv4. For 250K prefixes, we achieve a false positive probability as low as 3.3×10^{-7}. When the ad-hoc prefix expansion is applied, this basically means that there are less than 4 expanded prefixes for every 10 million flows.

B. Hardware Hash Functions

A Bloom filter requires k independent hash functions to be computed for each lookup. The performance critically depends on the performance of the hash function computation. A family of hash functions H_3 studied in [23] is shown to be suitable for fast hardware implementation with performance close to the theoretical bound. However, the logic needs can be large. If the input key has r bits and the hash table has s buckets, the hash function needs to register $r \lg_2 s$ bits and perform logic AND and XOR operation on them. When r is large, we need to break the logic operations into multiple pipeline stages to improve the circuit timing. This will introduce extra $\lg_2 s$ flip-flops per pipeline stage.

Our architecture requires the use hundreds of hash functions (i.e. generate hundreds of independent hash values for each key). Using existing hash function to implement the Bloom filters requires too much resources. Hence, we develop an area efficient hash scheme that can produce n hash values using just $O(\lg n)$ seed hash functions for the same hash key. The hash operations use only simple logic operations and are fast enough for the lookup application.

Given a key λ, we can generate n hash values $H_1, ..., H_n$ by using only m seed universal hash functions $S_1, ..., S_m$, as if n unique hash functions are used, where

$$m = \begin{cases} \lg_2 n + 1 & n = 2^k, k \in N \\ \lceil \lg_2 n \rceil & n \neq 2^k, k \in N \end{cases}$$

and each S_i generates an address between 0 and $t-1$, where t is power of 2 (i.e. the hash result can be represented as a bit-vector with $\lg_2 t$ bits).

The construction of H_i, where $i \in [1, n]$, is as follows:

Since for any i, we have a unique representation of

$$i = r_m 2^{m-1} + r_{m-1} 2^{m-2} + ... + r_2 2 + r_1 \quad \forall r_i \in \{0, 1\}$$

We let

$$H_i = (r_m S_m) \oplus (r_{m-1} S_{m-1}) \oplus ... \oplus (r_1 S_1)$$

where $\oplus$ is bit-wise XOR operation. H_i has the exactly same address space as S_i.

The following is an example that uses three seed hash functions to produce seven new hash values.

$$\begin{aligned} H_1 &= S_1 \\ H_2 &= S_2 \\ H_3 &= S_2 \oplus S_1 \\ H_4 &= S_3 \\ H_5 &= S_3 \oplus S_1 \\ H_6 &= S_3 \oplus S_2 \\ H_7 &= S_3 \oplus S_2 \oplus S_1 \end{aligned}$$

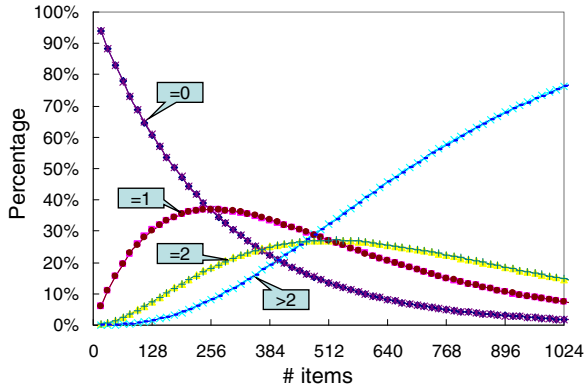

Fig. 5. Hash Table Bucket Load with 64K Buckets and 256 Hash Functions

Note that we can choose any fast hash function as seed functions. In this paper we use H_3 hash functions for the implementation and simulations.

The correlation performance of these new hash values cannot be analyzed theoretically. Therefore, we use simulations to compare their performance with the theoretical results.

First, we test each new hash function to see if it is universal and random by inserting a number of keys into the hash table and measure the hash collisions and the average load of non-empty buckets. Specifically, we implement a counting Bloom filter with different number of hash functions and insert some number of items into the filter. We then track the percentage of buckets with different loads. We compare this number with the theoretical value calculated from random insertions. Figure 5 shows one of the results. Note that the theoretical values match the experimental values so well that the corresponding curves are almost not distinguishable from each other.

Second, we build a Bloom filter with these hash functions to see if these hash functions are independent. Each key is 64-bit long and the Bloom filter has $m = 64$K buckets. We vary the number of keys n to be programmed into the Bloom filter. Table I summarizes the Bloom filter false positive rate with different m/n ratio and different number of hash functions (k). The simulation results meet the theoretical values very well.

In our reference design, we can use just five seed hash functions to generate a group of 16 hash functions for each prefix length, which account for 69% hardware resource savings overall.

C. DLB-BF Memory Port Scheduling

Mapping the 24 or 48 DLB-BF read requests to different memory ports turns out to be the most resource consuming part of the design. Since we have only 2-port memory, when more than two read requests are to the same memory block, we need more than one clock cycle to schedule these requests. This can significantly increase the design complexity and negatively impact the system throughput. We, instead, use only one clock cycle to schedule the requests. When more than two requests are for a memory block, only the two requests for the top two longest prefixes are granted. The remaining requests are simply discarded. However, we assume they all find a match, possibly false, and as though they have all been granted memory accesses. The corresponding bits in the bit vector are thus directly set to one without actually looking up the DLB-BF.

Recall that we have 16 DLB-BFs working in parallel, and each prefix length generates one read request in each DLB-BF using independent hash functions. Even when a request for a prefix length is discarded in one DLB-BF, the requests in the other DLB-BFs are likely to be granted. So the effect is that for a prefix length, a reduced number of hash functions are used to search the Bloom filter. Although the false positive rate is then not as good as that when all the hash functions are used, we trade this off for a smaller and faster implementation.

In addition, the simplified scheduler shows some preference to the longer prefixes. So generally the longer the prefix, the more the hash functions that are applied and in turn the better false positive probability that is achieved (Note that the requests for the top two prefix lengths, 32 and 30, are always granted, according to our scheduling strategy). This arrangement complies with the search order of the last step when multiple matches are found in the DLB-BFs.

D. Prototype and Simulation

We prototype the design using the Altera Stratix III FPGA EP3SL340. The design is aimed at 40G line card with an IPv4 lookup rate of 60Mpps. Each one of the 16 DLB-BFs include 32 8K-bit 2-port memory blocks. The design uses 50% of the logic resource and 25% of the block memory resource. The synthesized circuit can run at 150MHz clock rate. A 36-Mbit 250MHz QDR-II SRAM is used to store the off-chip hash table. Each table bucket has 144 bits, storing up to three prefixes and the corresponding next hops. As discussed before, each prefix is stored in one of two candidate buckets for load balancing.

A snapshot of IPv4 BGP Table, which contains about 180K prefixes and 24 unique prefix lengths, is used to test our design. We combine the prefix value and its length as the key to the off-chip hash table, and find no overflow at all in the hash table.

To test the lookup performance, we first use a synthesized packet trace which contains the same number of packets as prefixes and matches each prefix exactly once. Due to our simplified memory port scheduling algorithm, the observed false positive rate is 8.3×10^{-5}, which is slightly higher than the theoretical value 1.3×10^{-5}. However, after the 15 prefixes that causes the false positives are stored in the off-chip hash table as expanded prefixes, for the same set of flows, no more false positives happen again.

In the second experiment, we look up about 180 million unique IP addresses and find less than 5K false positive occurrences. The expanded prefixes cause only a small prefix set inflation of 2.6%.

Finally, we test the algorithm with a real Internet packet trace. For this case, we observed 38 false positive occurrences and a false positive rate of 1.4×10^{-7}. This means for seven

	k=2		*k=4*		*k=8*		*k=16*	
m/n	simulation	theory	simulation	theory	simulation	theory	simulation	theory
2	4.01e-1	4.00e-1	5.60e-1	5.59e-1	8.58e-1	8.63e-1	9.95e-1	9.95e-1
4	1.53e-1	1.55e-1	1.60e-1	1.60e-1	3.16e-1	3.12e-1	7.49e-1	7.44e-1
8	4.91e-2	4.89e-2	2.44e-2	2.40e-2	2.53e-2	2.55e-2	9.62e-2	9.76e-2
16	1.37e-2	1.38e-2	2.33e-3	2.38e-3	5.40e-4	5.74e-4	6.14e-4	6.50e-4
32	3.66e-3	3.67e-3	1.88e-4	1.91e-4	6.00e-6	5.73e-6	3.40e-7	3.30e-7

TABLE I
BLOOM FILTER FALSE POSITIVE RATE WITH FAST HASH FUNCTIONS

million flows, only one can cause false positive with its first packet.

In all of our tests, we observed at most one false positive for a prefix. We conducted each of the above experiments multiple times, each with a different hash function configuration (i.e. varying the random number set used to implement the seed H_3 hash functions). The results were all consistent.

VI. CONCLUSIONS

With the continued growth in Internet traffic, driven by the surge in media-rich traffic, deployment of 100Gbps line cards in core routers is very likely as soon as standardization efforts are completed. Moreover, Internet routing tables continue to grow and IPv6 use is growing as well. It has also become increasingly important to minimize the power consumption of line cards to the maximum extent possible. These factors lead us reexamine algorithmic approaches to IP lookup. Algorithmic approaches can exploit the increasing amounts of on-chip memory using new data-structures and can considerably improve upon the high power-consumption of TCAMs. However, it is still a challenge to devise lookup schemes usable for 100Gbps line cards supporting IPv6 and large routing tables in an economical, power-efficient manner.

We make the following four major contributions to address this challenge: (1) We design a novel data structure, DLB-BF, based on Bloom filters which is suitable for on-chip storage and makes possible fast prefix-length independent IPv4 and IPv6 lookups at 100Gbps line speed. The proposed scheme addresses several major drawbacks of previously proposed schemes including the need to use large multi-port memories. (2) We avoid the large a priori prefix expansions that are needed to reduce the number of distinct prefix lengths and instead use an ad-hoc prefix expansion method that efficiently copes with the Bloom Filter false positive problem without any changes to the on-chip Bloom filter itself. (3) We design an area-efficient algorithm for computing a large number of fast and high performance hash functions in hardware. These are ideal for implementing Bloom filters. (4) We design a simple and efficient memory-port scheduling scheme that allows us to use prevalent 2-port memory blocks for implementing DLB-BFs without much degradation in attainable false positive probabilities. The comparison shows that our algorithm significantly outperforms the TCAM-based solutions in power consumption, footprint, and cost.

REFERENCES

[1] B. Bloom. Space/Time Trade-offs in Hash Coding With Allowable Errors. *Communications of the ACM*, July 1970.
[2] A. Broder and M. Mitzenmacher. Using Multiple Hash Functions to Improve IP Lookups. In *IEEE INFOCOM*, 2001.
[3] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, 2005.
[4] J. Bruck, J. Gao, and A. Jiang. Weighted Bloom Filter. In *IEEE ISIT*, 2006.
[5] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *The Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004.
[6] I. Chvets and M. MacGregor. Multi-zone Caches for Accelerating IP Routing Table Lookups. In *Proceedings of High-Performance Switching and Routing*, 2002.
[7] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *ACM SIGCOMM*, 1997.
[8] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest Prefix Matching using Bloom Filters. In *ACM SIGCOMM*, 2003.
[9] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. Fast Packet Classification Using Bloom Filters. In *ACM ANCS*, 2006.
[10] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: hardware/software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communication Review*, 2004.
[11] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, Mar. 2000.
[12] J. Hasan and T. N. Vijaykumar. Dynamic Pipelining: Making IP Lookup Truly Scalable. In *ACM SIGCOMM*, 2005.
[13] http://bgp.potaroo.net. *BGP Routing Table Analysis Reports*. 2008.
[14] http://www.cisco.com/en/US/products. *Cisco CRS*. 2007.
[15] http://www.juniper.net/products/tseries. *Juniper Networks T-series Routing Platforms*. 2007.
[16] http://www.necel.com. *NEC Electronics*.
[17] http://www.netlogicmicro.com. *NetLogic*.
[18] W. Jiang and V. K. Prasanna. Beyond TCAMs: An SRAM-based Multi-Pipeline Architecture for Terabit IP Lookup. In *IEEE INFOCOM*, 2008.
[19] S. Kumar, M. Becchi, P. Crowley, and J. S. Turner. CAMP: Fast and Efficient IP Lookup Architecture. In *ACM/IEEE ANCS*, 2006.
[20] S. Kumar, J. Turner, and P. Crowley. Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms. In *IEEE INFOCOM*, 2008.
[21] J. T. M. Waldvogel, G. Varghese and B. Plattner. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM*, 1997.
[22] S. Nilsson and G. Karlsson. IP Address Lookup using LC-Tries. *IEEE Journal on Selected Areas in Communications*, June 1999.
[23] M. Ramakrishna, E. Fu, and E. Bahcekapili. A Performance Study of Hashing Functions for Hardware Applications. In *Proc. 6th Int'l Conf. Computing and Information*, 1994.
[24] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani. Scalable, Memory Efficient, High-Speed IP Lookup Algorithms. *IEEE/ACM Transactions on Networking*, Aug. 2005.
[25] H. Song, S. Dharmapurikar, J. S. Turner, and J. W. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: an Aid to Network Processing. In *ACM SIGCOMM*, 2005.
[26] H. Song, J. Turner, and J. Lockwood. Shape Shifting Tries for Faster IP Lookup. In *IEEE ICNP*, 2005.
[27] V. Srinivasan and G. Varghese. Faster IP Lookups Using Controlled Prefix Expansion. In *ACM SIGMETRICS*, 1998.