Hash, Don't Cache: Fast Packet Forwarding for Enterprise Edge Routers

Minlan Yu
Princeton University
minlanyu@cs.princeton.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

As forwarding tables and link speeds continue to grow, fast packet forwarding becomes increasingly challenging for enterprise edge routers. Simply building routers with ever larger amounts of ever faster memory is not appealing, since high-speed memory is both expensive and power hungry. Instead, we believe future enterprise routers should leverage a hierarchical memory architecture consisting of a small, fast memory and a large, slow memory. However, the conventional approach of caching popular forwarding-table entries in the fast memory does not perform well in practice, especially under worst-case workloads with a wide range of destination IP addresses. Instead, the small memory could be used to store one Bloom filter of the address blocks associated with each outgoing link. In this paper, we present techniques to make the use of Bloom filters practical for enterprise edge routers, including optimizing the sizes of Bloom filters with limited fast memory, handling routing changes and dynamically tuning Bloom filter sizes using counting Bloom filters in slow memory, and handling the small number of false positives. Our evaluation shows that our scheme works well with less than 1 MB of fast memory.

Categories and Subject Descriptors

C.2.6 [**Internetworking**]: Routers

General Terms

Algorithms, Design, Performance

Keywords

Packet forwarding, Enterprise edge routers, Bloom filter

1. INTRODUCTION

Fast packet forwarding is a challenge today due to the significant growth of the forwarding table and the increasing link speeds. To keep up with link speed, the large forwarding table must be stored in larger and faster memory. Enterprises (especially small or mid-size enterprises) are more cost conscious than Internet service providers, making them reluctant to use expensive, power-hungry fast memory such as TCAM (Ternary Content Addressable Memory). Therefore, our design goal is to reduce memory cost and power consumption of packet forwarding by leveraging a small fast memory.

Enterprise edge routers introduce unique opportunities to optimize packet forwarding compared with core routers. First, enterprise edge routers usually have only a few outgoing links. We leverage this fact and propose a solution that maintains a small data structure for each next hop. Second, multi-homed enterprises can reach most destination prefixes through multiple upstream providers, allowing them to occasionally direct packets to a less-preferred outgoing link.

To provide fast packet forwarding using low-cost memory, we assume a hierarchical memory structure consisting of a small, fast memory and a large, slow memory. The fast memory could be embedded SRAM in the line card of hardware routers, or the processor cache in a software router. For multi-core platforms, the fast memory could be a group of caches associated with different cores, or the shared cache among cores. Fast memory is expensive, so we keep its size small, usually less than 1 MB. Slow memory is cheap and can be large enough (e.g., 10-100 MB) to store a conventional forwarding-table data structure (such as a trie) in its entirety. This can be a DRAM placed in line card or near the control plane processor in hardware routers, or the main memory in a software router.

Using the small, fast memory as a cache is a seemingly natural way to leverage the hierarchical memory architecture. The basic idea is to store the most frequently used entries of the forwarding table in the fast memory. In fact, route caching *was* once commonly used in routers [8]. However, during cache misses, the router experiences low throughput and high packet loss. In addition, when routing changes or link failures happen, many of the cached routes are simultaneously invalidated. Malicious traffic with a wide range of destination addresses may significantly increase the cache miss rate, making route caching highly inefficient. Due to its bad performance under *worst-case workloads*, route caching cannot keep up with the increasing link speeds and thus is not used in most routers today.

Instead, a Bloom filter, a hash-based compact data structure to store a set of elements, is a more suitable way to capitalize on the small, fast memory. In fact, several pre-

WREN'09, August 21, 2009, Barcelona, Spain.

vious studies [9, 13] have proposed ways to leverage Bloom filters in packet forwarding. Their basic idea is to use small fast memory to *assist* packet address lookup by reducing the number of accesses to the slow memory. For example, in [9], the authors use Bloom filters to determine the *length* of the longest matching prefix for an address, and then they refer to the entire forwarding table stored in the slow large memory to determine the outgoing link. In these works, every address lookup still must access the slow memory at least once. Instead, we advocate performing the *entire* lookup in the fast memory. Similar to the work in resource routing [4, 12][1], one Bloom filter is constructed for each next hop (i.e., outgoing link), and is used to store all the addresses that are forwarded to that next hop. T Bloom filters are constructed in the router where T is the number of next hops. This scheme works well under *a wide range of workloads* at the expense of a few false positives.

In this paper, we provide practical techniques to apply the basic Bloom filter idea to fast packet forwarding in enterprise edge routers:

- To make efficient use of limited fast memory, we *optimize* the sizes and number of hash functions of the Bloom filters. Surprisingly, we show that to reach the optimal *overall false-positive rate*, Bloom filters with fewer elements must have fewer false positives than those with more elements. We also prove that a small T in the enterprise edge router will lead to a small overall false-positive rate. To obtain a false-positive rate of 1%, we need only 300 KB of fast memory to store the FIB of 165K entries obtained from an edge router with 10 next hops.
- To adapt Bloom filters for routing changes, which happen on a much longer time scale than packet forwarding, we store counting Bloom filters in the large, slow memory. To reduce the false-positive rate under routing changes, we *dynamically* adjust the size and number of hash functions of Bloom filters in fast memory by keeping large fixed-size counting Bloom filters in slow memory.
- Since enterprise edge routers usually have multiple upstream providers, a few false positives are allowable. We also propose multiple methods to handle false positives for enterprise edge routers.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to Bloom filters. Section 3 describes our solution to perform the entire packet address lookup in small fast memory, and our enhancements for reducing computational overhead and false positives. Section 4 evaluates the false-positive rate of our solution under various settings. Section 5 shows how we leverage counting Bloom filters in slow memory to handle routing changes. Section 6 discusses our solutions to handle false positives. Sections 7 and 8 discuss related work and conclude the paper.

2. BACKGROUND ON BLOOM FILTERS

In this section, we give a brief introduction to Bloom filters and counting Bloom filters, which are the basis of our address lookup solutions.

[1]These work design the algorithms of locating resources by using one Bloom filter to store a list of resources that can be accessed through each neighboring node.

A Bloom filter [4] is a compact data structure to store a set of elements. A Bloom filter supports two operations – inserting an element into the set and checking whether an element is a member of the set. A Bloom filter consists of an array of bits. To insert an element into a Bloom filter, we compute k hash functions on the element, and get k values each denoting a position in the array. All the k positions are set to 1 in the array. By repeating the same procedure for all the elements in the set, the Bloom filter is constructed to represent the summary of the set of elements with constant space. It is easy to check if an element belongs to the set with Bloom filter. Given an element, we calculate the same k hash functions and check the bits in the corresponding k positions of the array. If all the bits are 1, we say that the element is in the set; otherwise it is not.

Bloom filters have no false negatives — if one of the k positions is set to 0, the element must not belong to the set. However, Bloom filters can have false positives — an element can absent from the set even if all k positions are set to 1, since each position could be set by the other elements in the set. Assume a Bloom filter with an array of m bits stores n elements. The false-positive rate of a Bloom filter is:

$$f = (1 - (1 - 1/m)^{kn})^k \approx (1 - e^{-kn/m})^k$$

To store a set of n elements in a Bloom filter, the larger m is the smaller the false-positive rate is. For a Bloom filter with fixed size m, to minimize false-positive rate, the number of hash functions k and the minimum false-positive rate f are:

$$k = m \ln 2/n, \quad f = (0.5)^k$$

We can add elements to standard Bloom filters, but there is no way to delete an element. A *counting Bloom filter* [10] is an extension of the Bloom filter to allow adding and deleting elements in a set. A counting Bloom filter stores a counter rather than a bit in each slot of the array. To add an element to the counting Bloom filter, we increment the counters at the positions calculated by the hash functions; to delete an element, we decrement the counters. If the counters do not overflow, counting Bloom filters do not have any false negatives and their false-positive rates can be calculated in the same way as standard Bloom filters.

3. BLOOM FILTER FORWARDING

In this section, we consider a forwarding table that contains a fixed set of routes. We focus on the data plane in the router, i.e., how to perform packet lookup and forwarding. Our goal is to store the forwarding table in an M-bit fast memory. We assume the router connects to T next hops, which is typically small for enterprise edge routers. For simplicity, we assume flat addresses first, and then extend the solution to addresses with various prefix lengths.

We first give an overview of the basic idea of using one Bloom filter for each next hop. We then discuss our practical solutions of making the basic idea work for enterprise edge routers, including the sizing of the Bloom filters and the analysis of false positives for addresses with different prefix lengths.

3.1 Basic Idea of Using Bloom Filters

For simplicity, we first consider using Bloom filters to perform FIB lookups for flat addresses, such as the MAC ad-

dresses used in Ethernet. New protocols with flat address spaces (e.g., ROFL [5], AIP [1]) have been proposed to facilitate the Internet's growth and configuration. Even with CIDR (variable length) prefixes, we can convert each prefix into small, fixed-length (i.e., /24) sub-prefixes that do not overlap.

We set one Bloom filter for each next hop (or outgoing link), and use it to store all the addresses that are forwarded to that next hop. For a router with T next hops, we need T Bloom filters. An address with next-hop t is stored in Bloom filter $BF(t)$. To perform address lookup for an address p, we check which $BF(t)$ ($t \in [1..T]$) p is in, and get the corresponding next hop p should be forwarded to. To reduce the computational overhead of address lookup, we apply the same group of hash functions to all the T Bloom filters.

This scheme is not affected by worst-case workloads as opposed to route caching. The false positive of one address lookup does not affect the lookup of other addresses. In order to send packets that lead to false positives, attackers have to know all the hash functions used to construct Bloom filters. We can change hash functions over time to avoid such attack.

3.2 Variable Size to Reduce False Positives

Since there are different numbers of addresses per next hop, we should use different sizes for the Bloom filters according to the number of addresses stored in them, in order to minimize the *overall false-positive rate* with the small, M-bit fast memory.

We first give the definition of the overall false-positive rate. If any one of the T Bloom filters has a false positive, an address will hit in multiple Bloom filters. In this case, we will get multiple next hops for the address and cannot decide which next hop to forward the packet to. We define the overall false-positive rate as the probability that any one of the T Bloom filters has a false positive. Let $f(t)$ denote the false-positive rate of Bloom filter $BF(t)$. Since Bloom filters for different next hops store independent sets of addresses, and thus are independent of each other, the overall false-positive rate of T Bloom filters is

$$\begin{aligned} F(T) &= 1 - \prod_{t=1}^{T}(1 - f(t)) \\ &\approx \sum_{t=1}^{T} f(t) \quad (when\ f(t) \ll 1/T, \forall t = 1..T) \end{aligned}$$

From the equation, we can see that the false-positive rate is relatively low for edge routers in enterprise networks, since the number of next hops T is usually small.

The overall false-positive rate could be minimized by varying the sizes and the number of hash functions of Bloom filters. If we do not consider route changes, we are aware of the exact number of addresses to be stored in each Bloom filter. Let $n(t)$ denote the number of addresses stored in Bloom filter $BF(t)$. We optimize the false-positive rate $F(T)$ by choosing the best $m(t)$ (the number of bits in $BF(t)$) and $k(t)$ (the number of hash functions used in $BF(t)$), with the constraint that Bloom filters must not take more space than the size of the fast memory. The problem to minimize the overall false-positive rate is formulated as:

$$\begin{aligned} Minimize \quad & F(T) = \sum_{t=1}^{T} f(t) \\ s.t. \quad & f(t) = (1 - e^{-k(t)n(t)/m(t)})^{k(t)} \\ & \sum_{t=1}^{T} m(t) = M \\ given \quad & M \ and\ n(t)(\forall t \in [1..T]) \end{aligned}$$

Through calculus calculation, F(T) is minimized when we size each Bloom filter with

$$\begin{aligned} m(t) &= \frac{(\lambda - \ln n(t))n(t)}{\ln^2 2} \qquad (1) \\ \lambda &= \ln \frac{M \ln^2 2 + \sum_{t=1}^{T}(n(t) \ln n(t))}{\sum_{t=1}^{T} n(t)} \end{aligned}$$

The number of functions used in each Bloom filter is:

$$\begin{aligned} k(t) &= \frac{m(t)}{n(t)} \ln 2 \\ &= \frac{\lambda - \ln n(t)}{\ln 2} \qquad (2) \end{aligned}$$

The optimal $F(T)$ is

$$F(T) = \sum_{t=1}^{T} (\frac{1}{2})^{k(t)}$$

Now given a forwarding table and fast memory size, we are able to construct T Bloom filters that minimize the overall false-positive rate by calculating $m(t)$ and $k(t)$ with Equations (1) and (2). Interestingly, the false-positive rate on each Bloom filter is not equal in the optimal solution. In fact, Bloom filters with fewer elements obtain smaller false-positive rate. This is because we have memory size constraint, adding the same amount of memory for the Bloom filter with fewer addresses reduces its false-positive rate more than for the one with more addresses. We evaluate both equal false-positive rate solution and optimal solution in Section 4.

We've discussed that T Bloom filters could share the same group of hash functions to reduce computational overhead. Though we have a different number of hash functions for each Bloom filter, the Bloom filters can still share part of the hash functions. We calculate the same group of k_{max} hash functions h_i ($i \in [1..k_{max}]$) to reduce the computational overhead, where

$$k_{max} = \max_{t=1}^{T} k(t).$$

To look up an address p, we use the first $k(t)$ hash values for Bloom filter $BF(t)$ ($t \in [1..T]$). Similar to the single Bloom filter case, we check in Bloom filter $BF(t)$, whether the bits in positions

$$h_i(p) \mod m(t), \forall i \in [1..k(t)]$$

are all 1. If so, p is in $BF(t)$; otherwise, it is not.

3.3 False Positive Analysis on Longest Prefix Match

To perform longest prefix match, similar to the solution in [9], we use one Bloom filter for each prefix length. We construct $32T$ Bloom filters with $BF^l(t)(l \in [1..32], t \in [1..T])$

storing the addresses with prefix length l and outgoing link t. We optimize the size of each Bloom filter $m^l(t)$ and the number of hash functions in them $k^l(t)$ to obtain minimal overall false-positive rate.

Now we describe the algorithm to perform lookup for address p. Let p_l denote the l bit prefix of address p. We first calculate k_{max} hash functions h_i ($i \in [1..k_{max}]$), where k_{max} is the maximum number of hash functions in all the Bloom filters.

$$k_{max} = \max_{l \in [1..32], t \in [1..T]} k^l(t)$$

We pick a next hop t^*, we search for the longest prefix length l, such that prefix p_l belongs to $BF^l(t^*)$. If we cannot find p in any $BF^l(t^*)$, $\forall l \in [1..32]$, we know that p should not be forwarded to t^*. Repeating the same procedure for all t ($t \in [1..T]$), we will finally find out the outgoing link that p should be forwarded to.

Using one Bloom filter per (next hop, prefix length) pair has both good and bad effects on the correctness of our decision on which next hop to output.

Good Effect: *If a packet should be forwarded to t^*, the false positives of Bloom filters $BF^l(t^*)$ ($\forall l \in [1..32]$) do not influence the overall false-positive rate.* For example, when an address hits only two Bloom filters $BF^{l_1}(t^*)$ and $BF^{l_2}(t^*)$, even if one of the hits is false positive, we can still determine t^* is the correct next hop. Therefore if there are not any false positives in the other Bloom filters $BF^l(t)$ ($\forall t \in [1..T], t \neq t^*$), we are sure that t^* is the correct next hop. In general, if an address p matches at least one entry in the FIB and all the matching entries[2] have the same next hop t^*, the probability that the Bloom filter mechanism does not output t^* (i.e., the address hits Bloom filters for different next hops due to false positives) is :

$$Prob(\exists t \in [1..T], t \neq t^*, \exists l \in [1..32], p_l \text{ is in } BF^l(t))$$

$$= 1 - \prod_{t \neq t^*} \prod_{l=1}^{32} (1 - f^l(t)) \approx \sum_{t \neq t^*} \sum_{l=1}^{32} f^l(t)$$

This property reduces false positives significantly for enterprise edge routers where T is small.

Bad Effect: *It is difficult to distinguish a false positive from the case that an address matches multiple entries with different prefix lengths in the FIB.* Sometimes an address may hit multiple Bloom filters, though there are not any false positives in all the $32T$ Bloom filters. For example, in a router's forwarding table, there are two entries: "15.0.0.0/16 $\rightarrow A$", "15.128.0.0/17 $\rightarrow B$", while A and B are two next hops belonging to different providers. We thus store "15.0.0.0/16" in Bloom filter $BF^{16}(A)$ and "15.128.0.0/17" in $BF^{17}(B)$. In this case, the address "15.128.0.8" may hit both Bloom filters. Using longest prefix match, we should send the packet to B. However, with our Bloom filter mechanism, we cannot be sure the hit in $BF^{17}(B)$ is not a false positive. Therefore, when we find multiple next hops matched in the Bloom filters, we are not able to distinguish the false positives and the many valid matches with different prefix lengths.

Our solution is to reconstruct the FIB such that the prefixes from different entries in the FIB do not overlap. In the previous example, if we know that the address "15.128.0.8" hits only one entry in the FIB, when it hits multiple Bloom filters with different next hops, we are sure a false positive has happened. One way to reconstruct the FIB is to use flat addresses. We can expand the prefix into a group of non-overlapping prefixes with a fixed prefix length say /24. However, using flat addresses may increase the number of prefixes significantly. Therefore, we choose to expand the prefixes to a group of non-overlapping prefixes with various lengths. In the previous example, we use the two entries: "15.0.0.0/**17** $\rightarrow A$", "15.128.0.0/17 $\rightarrow B$", so "15.128.0.8" only hits the second entry.

[2]We mean the matches of prefixes of all the lengths, not just longest prefix match.

4. PERFORMANCE EVALUATION

We evaluate the false-positive rate with real forwarding tables and packet traces from FUNET and Internet2, and compare them to the analytical results.

4.1 Experimental Setup

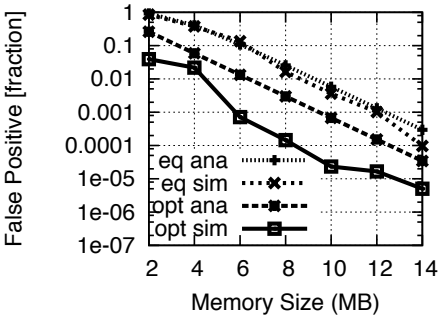
We use routing table and packet traces from an *edge router* in the Finnish University Network (FUNET) in March 2005.[3] The edge router is located in Helsinki University of Technology, which carries FUNET international traffic. The routing table has 165K entries and 10 next hops. The packet trace contains 20 million packets. For Internet 2, we use the routing table from the router located in Chicago and the corresponding netflow data for January 22, 2009. The routing table contains 12K entries and 34 next hops. The netflow trace contains 12 million packets.

4.2 Evaluation of False Positives

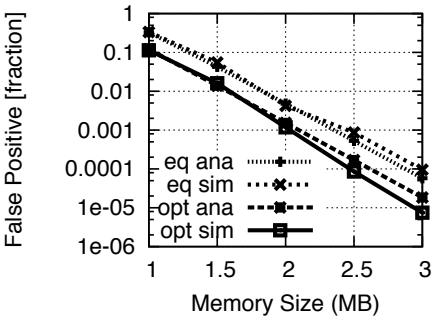
To study our mechanism on flat addresses, we convert the prefixes in the routing tables into /24 prefixes. The new FUNET and Internet2 table have 5400K and 900K flat addresses respectively. Figure 1 shows the analytical false-positive rate (labeled as ana in Figure 1) and the experimental result (labeled as sim) with the increase of fast memory size. We test both the optimal setting based on Equations (1) and (2) (labeled as opt) and the setting that has the same false-positive rate for all the Bloom filters (labeled as eq). Compared with equal false-positive rate setting, the optimal setting of the same fast memory size reduces the overall false-positive rate by 90% in FUNET and 70% in Internet2. With optimal setting, to reach the false-positive rate of 1%, we need only 6 MB fast memory in FUNET (i.e., 9 bits/address) and 1.5 MB (i.e., 13 bits/address) in Internet2. Since FUNET edge router has fewer next hops than Internet2 router, it needs less memory space for each address.

We then study the effect of longest prefix match on false positives. We modified the original FIB by converting nested prefixes to non-overlapping prefixes with various prefix lengths. The modified FUNET table has 250K entries and Internet2 table contains 15K entries. Figure 2 shows the false-positive rate with both original and modified FIBs in the optimal setting. Since the modified FIB has more entries than the original one, when the memory size is small, the false-positive rate on the modified FIB is larger than the original FIB. However, with the original FIB, the false-positive rate remains around 0.8% with the increase of fast memory size.

[3]This trace is confidential due to the sensitive nature of packets.

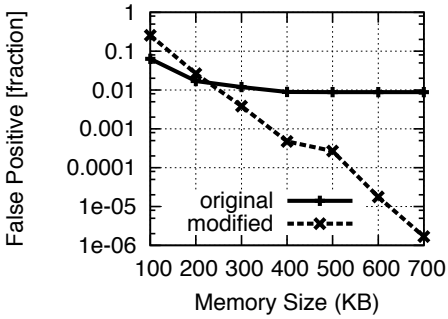


(a) FUNET

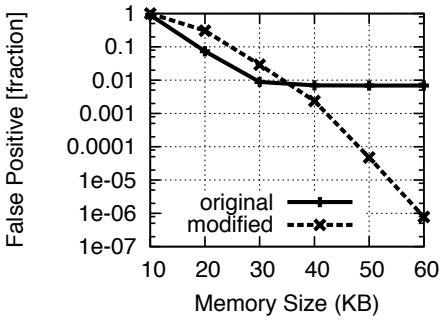


(b) Internet2

Figure 1: False positives with flat address



(a) FUNET



(b) Internet2

Figure 2: False positives with longest prefix match

This is because we are not able to distinguish false positives and multiple valid matches on nested prefixes (the "bad effect" discussed in Section 3). With the modified FIB where prefixes are not overlapping, the false-positive rate is further reduced with the increase of fast memory size. To achieve 0.1% false-positive rate with modified FIBs, we need 400 KB fast memory in FUNET and 40 KB in Internet2.

5. HANDLING ROUTING CHANGES

With standard Bloom filters (BF), we cannot delete elements from the set. We use counting Bloom filters (CBF), which allow both addition and deletion operations, to handle the dynamics of routing tables. However, CBFs require more space than BFs since they store a counter rather than a bit in each spot of the array. Fortunately, since routing changes do not happen very often and thus allow more computational overhead, we can store CBFs in slow memory, and update BFs in small fast memory based on CBFs. By using both CBFs and BFs, we make an efficient use of small fast memory without losing the flexibility to support changes in the FIB.

In this section, we first describe the use of CBFs in slow memory to keep track of changes in the forwarding table. We then discuss how to update the BF from the CBF and how to change the size of the BF without reconstructing the CBF.

5.1 Maintaining CBF in Slow Memory

In a router, the control plane maintains the RIB (Routing Information Base) and updates the FIB (Forwarding Information Base) in the data plane. We implement a group of T CBFs, each containing the prefixes associated with one next hop, corresponding to the BFs described in Section 3. If the control plane adds a route to the FIB for a new address p with next hop t, we will insert p to $CBF(t)$. Similarly, to delete a route is we delete p in $CBF(t)$.[4] The insertion and deletion operations on the CBF are described in Section 2. Since CBFs are maintained in slow memory, we set the sizes of CBFs large enough, so that even with routing changes, the false-positive rates on CBFs are low.

After the $CBF(t)$ is updated, we update the corresponding $BF(t)$ based on the new $CBF(t)$. If the CBF and BF are of the same size, we can easily update the BF by checking if each position in the CBF is 0 or not. However, we have to dynamically adjust the size of the BF to reduce the overall false-positive rate.

5.2 Adjust BF Size without Reconstructing CBF

When the forwarding table changes over time, the number of prefixes in the BF changes, so the size of the BF and the number of hash functions to achieve the optimal false-positive rate also change (see Equations (1) and (2)). We leverage the nice property that to halve the size of a Bloom filter, we just OR the first and second halves together [4]. In general, the same trick applies to reducing the size of a Bloom filter by a constant c. This works well in reducing the BF size when the number of prefixes in the BF decreases. However, when the number of prefixes increases, it is hard to expand the BF.

Fortunately, we maintain a large, fixed size CBF in the slow memory. We can dynamically *increase or decrease* the size of the BF by mapping multiple positions in the CBF to one position in the BF. For example in Figure 3, we can

[4]The control plane must make sure that p was forwarded to t. Otherwise, deleting an element not in the Bloom filter would cause errors on future queries.

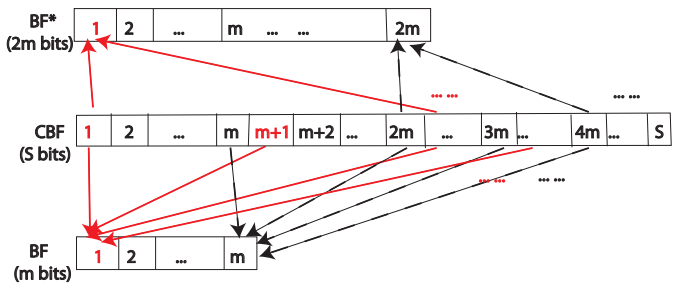


Figure 3: Adjust BF size from m to $2m$ based on CBF

easily expand the BF with size m to BF^* with size $2m$ by collapsing the same CBF.

To minimize the overall false-positive rate under routing changes, we monitor the number of prefixes in each CBF, and periodically reconstruct BFs to be of the optimal sizes and number of hash functions. The procedure of reconstructing a BF with an optimal size from the corresponding CBF is described in three steps:

Step 1: Calculate the optimal BF size and the number of hash functions. Using Equation (1), we first get the optimal size of each BF and denote it by m^*. Then we round m^* to m', which is a factor of S,

$$m' = S/c, where\ c = \lceil S/m^* \rceil.$$

Finally we calculate the optimal number of hash functions to minimize false positive with size m' and the number of prefixes n in the BF: $k' = m' \ln 2/n$.

Step 2: Change the number of hash functions in the CBF from k to k'. There are two ways to change the number of functions with either more computation or more space: (i) If $k' > k$, we calculate the hash values with the $k' - k$ new hash functions on all the prefixes currently in the BF, and update the CBF by incrementing the counters in corresponding positions. If $k' < k$, we also calculate $k - k'$ hash values, and decrementing the counters in corresponding positions. (ii) Instead of doing the calculation on the fly, we can pre-calculate the values of these hash functions with all the elements and store them in the slow memory.

Step 3: Construct the BF of size $m' = S/c$ based on the CBF of size S. As shown in Figure 3, the value of the BF at position x ($x \in [1..m']$) is updated by c positions in CBF x, $2x$, ... cx. If all the counters in the c positions of CBF are 0, we set the position x in BF to 0; otherwise, we set it to 1. During routing changes, the BFs can be updated based on CBFs in the same way.

6. HANDLING FALSE POSITIVES

In this section, we first describe three ways to handle packets that experience false positives in the Bloom filters. Then we discuss our caching solution to avoid subsequent packets from experiencing false positives. Finally, we combine these techniques to produce a practical solution for enterprise edge routers.

6.1 Handling Packets with False Positives

When a packet has a match in multiple Bloom filters, we detect that the packet is experiencing a false positive.[5] We provide three techniques to forward the packet that experiences false positives.

Redirection: If a lookup returns multiple next hops, we can redirect the packet to another place that knows the answer. For example, we can redirect the packet to another core in the multi-core system, which will do a conventional look up on the FIB. Similar to ViAggre [2], if we have a set of routers in the network each responsible for part of the address space, the packets could be redirected to the responsible router. Similar to Ethane [6], we can also have a centralized server to handle packets that experience false positives from any edge routers in the network. With packet redirection, we are guaranteed to forward the packet correctly through a slower decision path.

Send duplicate packets out: To ensure the packets experiencing false positives finally reach their destinations without any delay in forwarding, we can just send a copy of the packet out to each next hop that the packet hits. Since false positives are rare, it is highly unlikely that two Bloom filters experience false positives at the same time. Therefore in most cases, we send only two packets out knowing that one of them is sent to the correct next hop. The next-hop router may either forward the packet if it knows how to reach the destination or drop the packet if it does not. This solution also guarantees the packet will reach the destination, though with the risk of introducing extra traffic and packet loops.

Send to a random next hop: An alternative way to handle false positives, which neither delays the packet nor adds extra traffic, is to randomly pick one of the Bloom filters the packet hit in, and send the packet to the corresponding next hop. Since most edge routers are multi-homed, the packets are likely to get to the destination finally through any next hop we choose. The problem of this method is that sometimes the packet gets lost. However, a few packet losses are tolerable in the Internet especially when false positives are rare.

6.2 Caching for Subsequent Packets

Although the above three methods handle the case where a few packets experience false positives, if a burst of packets to the same destination all experience false positives, the packets will either experience a performance decrease (through redirection), or cause a lot of traffic (if we send multiple copies of the packet out), or significant packet loss (if we direct the packet to just one of the next hops). Therefore, when the first packet experiences a false positive, we perform a conventional lookup and cache the result, so that the subsequent packets will hit in the cache and no longer experience false positives. Note that our caching solution differs from conventional route caching in that it is robust to malicious traffic. With conventional route caching, an at-

[5]Some FIBs do not have default route and assume that addresses that have no match in it should be dropped. However, in our Bloom filter based scheme, the address that has no match in the forwarding table may hit one Bloom filter because of a false positive. To detect such false positives, we can either construct a Bloom filter to store a list of destinations that should be dropped by the router, or just send the packet out which will be finally dropped by the following routers.

tacker can easily send packets with a wide range of destinations resulting in a high cache-miss rate. However, since the attacker does not know the hash functions used for Bloom filters, it is hard to construct a flow of packets that makes our caching mechanism inefficient. Even if the attacker sends lots of packets, detects a few packets that experience false positives, and replay them, the replayed packets will hit in the cache and will not experience false positives.

6.3 Case Study: Enterprise Edge Routers

We apply the techniques discussed above to handle false positives for enterprise edge routers. Enterprise edge routers are connected to routers both in different ISPs and within its own network. There are only a few ISP routers, which are the next hops for most of the addresses. The routers within the enterprise network are next hops for a relatively small number of addresses.

Next hops that are ISP routers: Edge routers are usually home to multiple ISPs for load balancing and greater reliability. Since vast majority of addresses will have these ISP routers as next hops, if a packet with false positive ends up with two ISP next-hop routers, sending it to either of them would be fine in most cases. If the next-hop router we pick does not have a route to forward the packet, it will drop the packet. The end hosts have to retransmit the packet when they detect packet loss.

Next hops that are within the enterprise: Different from ISP routers, if the packets that are destined in the enterprise network are sent to the wrong next hop, they may never reach the destination. Fortunately, the addresses within the enterprise may fall within one or a few address blocks, so it is quick to check whether an address should be forwarded internal or external. Since there are only a small number of destinations inside the enterprise network, we can afford to store the complete forwarding information in a hash table.

We divide the FIB of the enterprise edge router into two parts: for the destinations within the enterprise network, we construct a hash table storing their next hops; for the external destinations, we construct the Bloom filters for all the next hops from different ISPs. To perform a packet lookup on the edge router, we first check whether the address is internal or external. If it is internal, we look up the hash table with all the destinations in its sub-network for the next hop. If the address is external, we perform the lookup with the Bloom filters. If we get multiple hits in the Bloom filters, we randomly pick one next hop and send the packet out through it. At the same time, we redirect the packet to another processor to perform a conventional FIB lookup and cache the result, so the following packets that have the same destination will hit in the cache and get to the correct next hop directly without experiencing a false positive.

7. RELATED WORK

The idea of using small fast memory to improve packet address lookup performance has been applied in route caching. A route cache stores the most popular part of the forwarding table in a small, fast memory. However, as discussed in Section 1, traffic with a wide range of destinations may increase the cache miss rate significantly, leading to both the inefficient packet lookup and the significant CPU consumption due to cache swapping. In contrast, since the attacker does not know the hash functions, the false positive in our Bloom filters cannot be easily generated by malicious traffic.

Bloom filters have been used for longest prefix match [9]. The authors use Bloom filters to determine the *length* of the longest matching prefix for an address, and then perform a direct lookup in a large hash table in slow memory. Different from their work, we perform the *entire* lookup in the fast memory at the expense of a few false positives. In [9], the goal is to reduce false positives in each Bloom filter. Thus they choose to use the size of Bloom filters proportional to the number of elements in it, and propose mini-Bloom filters to deal with various distributions of prefix lengths. In contrast, our goal is to minimize the *overall false-positive rate* rather than the false positive on each Bloom filter. We thus choose the optimal sizes for Bloom filters and dynamically adjust the size upon routing changes. The authors also discussed some alternative solutions to use less than 32 Bloom filters in longest prefix match, such as grouping the prefix lengths. Their techniques can be applied to our solution.

Bloom filters have also been used in resource routing [4, 12], which applies Bloom filters to probabilistic algorithms for locating resources. Our "one Bloom filter per next hop" is similar to their general idea of using one Bloom filter to store the list of resources that can be accessed through each neighboring node. In order to keep up with link speed in packet forwarding with strict fast memory size constraint, we *dynamically* tune the *optimal* size and the number of hash functions of Bloom filters by keeping large fixed-size counting Bloom filters in slow memory.

In general, packet address lookup can be viewed as an application of a *hash table*, where for each key (address), we find out the matching value (next hop). *Bloomier filter* [7] is a generalization of Bloom filter to associate a function value (from a discrete finite set of values) with each element in a *static* set, which uses a group of Bloom filters to represent the values. In theory, if both the set and the function values are static, the Bloomier filter only needs linear space. Our work is similar to the Bloomier filter in that we use a group of Bloom filters, one for each value of a function that maps a prefix to a next hop. Since we focus on the forwarding table on enterprise edge routers, we provide simple, practical solutions to handle false positives and routing changes.

People have proposed to use multiple hash functions to store the matching of keys and values. With d hash functions, we need to decide which of the d positions in the array to store the value and how to handle collisions. The authors in [3] design a *d-left scheme* for IP lookups. In this scheme, there is a bucket corresponding to each position of the array in the slow memory, and the value is stored in one of the d buckets in a load balancing way. However, to perform an IP lookup, we still need to access the slow memory at least d times. In [13], the authors use Bloom filter in the fast memory, and store the values in a linked structure in the slow memory such that the value can be accessed via one access on the slow memory most of the times. In contrast, we perform the *entire* packet lookup in the fast memory.

To handle routing changes, the works in [9, 13] use the counting Bloom filter (CBF) in fast memory, which uses more memory space than the Bloom filter (BF). We leverage the fact that routing changes happen on a much longer time scale than address lookup, and thus store only the BF in fast memory, and use the CBF in slow memory to handle routing changes. Our idea of maintaining both the CBF and BF is

similar to the work in [10], which uses BFs for sharing caches among Web proxies. Since cache contents change frequently, the authors suggest that caches use a CBF to track their own cache contents, and broadcast the corresponding BF to the other proxies. The CBF is used to avoid the cost of reconstructing the BF from scratch when an update is sent; the BF rather than the CBF is sent to the other proxies to reduce the size of broadcast messages. Different from their work, we dynamically adjust the size of the BF without reconstructing the corresponding CBF, which may be useful for other Bloom filter applications.

8. CONCLUSION

To improve packet-forwarding performance, we leverage a hierarchy of small, fast memory and large, slow memory. Leveraging the unique properties in enterprise edge routers, we propose a Bloom filter based mechanism, which performs the *entire* packet address lookup in less 1 MB fast memory. Our mechanism works well under worst-case workloads such as packets with a wide range of destinations, and is also robust to route changes and malicious traffic.

We are implementing our Bloom filter mechanism as a module in Click [11] modular router. We will deploy it on multi-core commodity platforms. To leverage many cores and the hierarchical cache architecture, we divide the FIB into address ranges, where each core is responsible for packet forwarding for one part of the address space. We leverage VMDq (Virtual Machine Device Queues) techniques in modern NICs to demultiplex the packets to the appropriate cores. Each core maintains a Bloom filter for each next hop, which is sized to minimize the overall false-positive rate, subject to fitting in the cache associated with each core. In the next-level cache shared amongst the cores, we store counting Bloom filters to handle routing changes and the entire FIB for forwarding packets that experience false positives. We will evaluate our mechanism across a wide range of workloads and under routing changes.

9. REFERENCES

[1] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol (AIP). In *Proc. ACM SIGCOMM*, 2008.

[2] H. Ballani, P. Francis, T. Cao, and J. Wang. Making routers last longer with ViAggre. In *Proc. NSDI*, 2009.

[3] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In *IEEE INFOCOM*, 2001.

[4] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Internet Mathematics*, volume 1, pages 485–509, 2005.

[5] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. ROFL: Routing on flat labels. In *Proc. ACM SIGCOMM*, 2006.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM*, 2007.

[7] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[8] Cisco Systems. How to choose the best router switching path for your network. `http://www.cisco.com/warp/public/105/20.pdf`.

[9] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using Bloom filters. In *Proc. ACM SIGCOMM*, 2003.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, 2000.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, Aug. 2000.

[12] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *IEEE INFOCOM*, 2002.

[13] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended Bloom filter: An aid to network processing. In *Proc. ACM SIGCOMM*, 2005.