BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations

Minlan Yu Alex Fabrikant Jennifer Rexford
Dept. of Computer Science, Princeton University

Abstract

In enterprise and data center networks, the scalability of the data plane becomes increasingly challenging as forwarding tables and link speeds grow. Simply building switches with larger amounts of faster memory is not appealing, since high-speed memory is both expensive and power hungry. Implementing hash tables in SRAM is not appealing either because it requires significant overprovisioning to ensure that all forwarding table entries fit. Instead, we propose the BUFFALO architecture, which uses a small SRAM to store one Bloom filter of the addresses associated with each outgoing link. We provide a practical switch design leveraging flat addresses and shortest-path routing. BUFFALO gracefully handles false positives without reducing the packet-forwarding rate, while guaranteeing that packets reach their destinations with bounded stretch with high probability. We tune the sizes of Bloom filters to minimize false positives for a given memory size. We also handle routing changes and dynamically adjust Bloom filter sizes using counting Bloom filters in slow memory. Our extensive analysis, simulation, and prototype implementation in kernel-level Click show that BUFFALO significantly reduces memory cost, increases the scalability of the data plane, and improves packet-forwarding performance.

1 Introduction

The Ethernet switches used in today's enterprise and data-center networks do not scale well with increasing forwarding-table size and link speed. Rather than continuing to build switches with ever larger and faster memory in the data plane, we believe future switches should leverage Bloom filters for a more scalable and cost-effective solution.

1.1 Memory Problems in the Data Plane

Enterprises and data centers would be much easier to design and manage if the network offered the simple abstraction of a virtual layer-two switch. End hosts could be identified directly by their hard-coded MAC addresses, and retain these addresses as they change locations (e.g., due to physical mobility or

virtual-machine migration). The hosts could be assigned IP addresses out of a large pool, without the artificial constraints imposed by dividing a network into many small IP subnets. However, traditional Ethernet can only support this abstraction in small network topologies, due to a heavy reliance on network-wide flooding and spanning tree. Recent advances [1, 2, 3] have made it possible to build much larger layer-2 networks, while still identifying hosts by their MAC addresses. These new architectures focus primarily on improving the *control plane*, enabling the use of shortest-path routing protocols (instead of spanning tree) and efficient protocols for disseminating end-host information (instead of flooding data packets).

As these new technologies enable the construction of ever larger "flat" networks, the scalability of the *data plane* becomes an increasingly serious problem. In today's Ethernet and in proposed solutions like TRILL [1, 2], each switch maintains a forwarding-table entry for each active MAC address. Other solutions [3] cache a smaller number of end-host MAC addresses, but still require a relatively large amount of data-plane state to reach every switch in the network. Large networks can easily have tens or hundreds of thousands of end-host MAC addresses, due to the proliferation of PDAs (in enterprises) and virtual machines (in data centers). In addition, link speeds are increasing rapidly, forcing the use of ever-faster—and, hence, more expensive and power-hungry—memory for the forwarding tables. This motivates us to explore new ways to represent the forwarding table that require less memory and (perhaps more importantly) do not require memory upgrades when the number of end hosts inevitably grows.

To store the forwarding table, one simple solution is to use a hash table in SRAM to map MAC addresses to outgoing interfaces. However, this approach requires significant overprovisioning the fast memory for three reasons: First, when switches are out of memory, the network will either drop packets in some architectures [1, 2] or crash in others [3]. Second, it is difficult and expensive to upgrade the memory for all the switches in the networks. Third, collisions in hash tables (i.e., different destination addresses mapped to the same place in the hash table) require extra memory overhead to handle them, and affect the throughput of the switch.

Given these memory problems in the data plane, our goal is to make efficient use of a small, fast memory to perform packet forwarding. Such small fast memory can be the L1 or L2 cache on commodity PCs serving as software switches, or dedicated SRAM on the line cards. When the memory becomes limited with the growth of forwarding table, we ensure that all packet-forwarding decisions are still handled within the SRAM, and thus allow the switches last longer with the increase of forwarding table size.

1.2 The BUFFALO Forwarding Architecture

Most enterprise and data center networks are "SPAF networks", which uses Shortest Path routing on Addresses that are Flat (including conventional link-state, distance-vector, and spanning tree protocols). Leveraging the unique properties in SPAF networks, we propose BUFFALO, a Bloom Filter Forwarding

Architecture for Large Organizations. BUFFALO performs the *entire* address lookup for all the packets in a small, fast memory while occasionally sending the packets through a slightly longer path.

To make all packet-forwarding decisions with a small fast memory, we use a *Bloom filter* [4], a hash-based compact data structure for storing a set of elements, to perform the flat address lookup. Similar to previous work on resource routing [4, 5][1], we construct one Bloom filter for each next hop (i.e., outgoing link), and store all the addresses that are forwarded to that next hop. By checking which Bloom filter the addresses match, we perform the *entire* address lookup within the fast memory for all the packets. In contrast, previous work [6] uses Bloom filters to *assist* packet lookup and every address lookup still has to access the slow memory at least once.

To apply our Bloom filter based solution in practice, we provide techniques to resolve three issues:

Handling false positives: False positives are one key problem for Bloom filters. We propose a simple mechanism to forward packets experiencing false positives without any memory overhead. This scheme works by randomly selecting the next hop from all the matching next hops, excluding the interface where the packet arrived. We prove that in SPAF networks the packet experiencing one false positive (which is the most common case for the packet) is guaranteed to reach the destination with constant bounded stretch. We also prove that in general the packets are guaranteed to reach the destination with probability 1. BUFFALO gracefully degrades under higher memory loads by gradually increasing stretch rather than crashing or resorting to excessive flooding. In fact, in enterprise and data center networks with limited propagation delay and high-speed links, a small increase in stretch would not run the risk of introducing network congestion. Our evaluation with real enterprise and data center network topologies and traffic shows that the expected stretch of BUFFALO is only 0.05% of the length of the shortest path when each Bloom filter has a false-positive rate of 0.1%.

Optimizing memory and CPU usage: To make efficient use of limited fast memory, we *optimize* the sizes and number of hash functions of the Bloom filters to minimize the overall false-positive rate. To reduce the packet lookup time, we let the Bloom filters share the same group of hash functions and reduce the memory access times for these Bloom filters. Through extensive analysis and simulation, we show that BUFFALO reduces the memory usage by 65% compared to hash tables.

Handling routing dynamics: Since routing changes happen on a much longer time scale than packet forwarding, we separate the handling of routing changes from the packet forwarding and use counting Bloom filters in the large, slow memory to assist the update of the Bloom filters. To reduce the false-positive rate under routing changes, we *dynamically* adjust the sizes and number of hash functions of Bloom filters in fast memory based on the large fixed-size

[1]These studies 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.

counting Bloom filters in slow memory.

We implement a prototype in the Click modular router [7] running in the Linux kernel. By evaluating the prototype under real enterprise and data center network topologies and traffic, we show that in addition to reducing memory size, BUFFALO forwards packets 10% faster than traditional hash table based implementation. BUFFALO also reacts quickly to routing changes with the support of counting Bloom filters.

The rest of the paper is organized as follows: Section 2 describes the underlying SPAF networks we focus on in this paper. Section 3 presents an overview of the BUFFALO architecture. Section 4 describes how to handle false positives and proves the packet reachability. In Section 5, we adjust the sizes of Bloom filters to make the most efficient use of limited fast memory. In Section 6, we show how to dynamically adjust the sizes of Bloom filters using counting Bloom filters. Section 7 presents our prototype implementation and the evaluation. Section 8 discusses several extensions of BUFFALO. Section 9 and 10 discuss related work and conclude the paper.

2 Shortest Paths & Flat Addresses

This paper focuses on *SPAF networks*, the class of networks that perform Shortest-Path routing on Addresses that are Flat. In fact, most enterprise and data center networks are SPAF networks.

Flat addresses: Flat addresses are used widely in enterprise and data center networks. For example, MAC addresses in Ethernet are flat addresses. New protocols with flat address spaces (e.g., SEATTLE [3], ROFL [8], AIP [9]) have been proposed to facilitate network configuration and management, because they simplify the handling of topology changes and host mobility without requiring administrators to reassign addresses. Even IP routing can be done based on flat addresses, by converting variable-length IP prefixes into multiple non-overlapping fixed-length (i.e., /24) sub-prefixes.

Shortest path routing: We also assume shortest-path routing on the network topology, based on link-state protocols, distance-vector protocols, or spanning-tree protocols.[2] Recent advances in Ethernet such as Rbridges [1, 2] and SEATTLE [3] all run link-state protocols that compute shortest paths.

Based on the above two properties, we define the SPAF network as a graph $G = (V, E)$, where V denotes the set of switches in the network, and E denotes all the links viewed in the data plane. In the SPAF network we assume all the links in E are actively used, i.e., the weight on link $e(A, B)$ is smaller than that on any other paths connecting A and B. This is because if a link is not actively used, it should not be seen in the data plane. Let $P(A, B)$ denote the set of all paths from A to B. Let $l(A, B)$ denote the length of the shortest path from A to B, i.e., the length of $e(A, B)$, and the length of a path p is $l(p) = \sum_{e \in p} l(e)$.

[2] In today's Ethernet the control plane constructs a spanning tree and the data plane forwards packets along shortest paths within this tree.

We have:

$$\forall A, B \in V, p \in P(A, B), l(A, B) \leq l(p).$$

In this paper, we propose an efficient data plane that supports any-to-any reachability between flat addresses over (near) shortest paths. We do not consider data-plane support for Virtual LAN (VLANs) and access-control lists (ACLs), for three main reasons. First, the new generation of layer-two networks [1, 2, 3] do not perform any flooding of data packets, obviating the need to use VLANs simply to limit the scope of flooding. Second, in these new architectures, IP addresses are opaque identifiers that can be assigned freely, allowing them to be assigned in ways that make ACLs more concise. For example, a data center could use a single block of IP addresses for all servers providing a particular service; similarly, an enterprise could devote a small block of IP addresses to each distinct set of users (e.g., faculty vs. students). This makes ACLs much more concise, making it easier to enforce them with minimal hardware support at the switches. Third, ACLs are increasingly being moved out of the network and on to end hosts for easier management and better scalability. In corporate enterprises, distributed firewalls [10, 11], managed by Active Directory [12] or LDAP (Lightweight Directory Access Protocol), are often used to enforce access-control policies. In data-center networks, access control is even easier since the operators have complete control of end hosts. Therefore, the design of BUFFALO focuses simply on providing any-to-any reachability, though we briefly discuss possible ways to support VLAN in Section 8.

3 Packet Forwarding in BUFFALO

In this section, we describe the BUFFALO switch architecture in three aspects: First, we use one Bloom filter for each next hop to perform the entire packet lookup in the fast SRAM. Second, we leverage shortest-path routing to forward packets experiencing false positives through slightly longer paths without additional memory overhead. Finally, we leverage counting Bloom filters in slow memory to enable fast updates to the Bloom filters after routing changes. Figure 1 summarizes the BUFFALO design.

3.1 One Bloom Filter Per Next Hop

One way to use a small, fast memory is route caching. The basic idea is to store the most frequently used entries of the forwarding table (FIB) in the fast memory, but store the full table in the slow memory. However, during cache misses, the switch experiences low throughput and high packet loss. Malicious traffic with a wide range of destination addresses may significantly increase the cache miss rate. In addition, when routing changes or link failures happen, many of the cached routes are simultaneously invalidated. Due to its bad performance under *worst-case workloads*, route caching is hardly used today.

To provide *predictable* behavior under various workloads, we perform the *entire* packet lookup for *all* the packets in the fast memory by leveraging Bloom

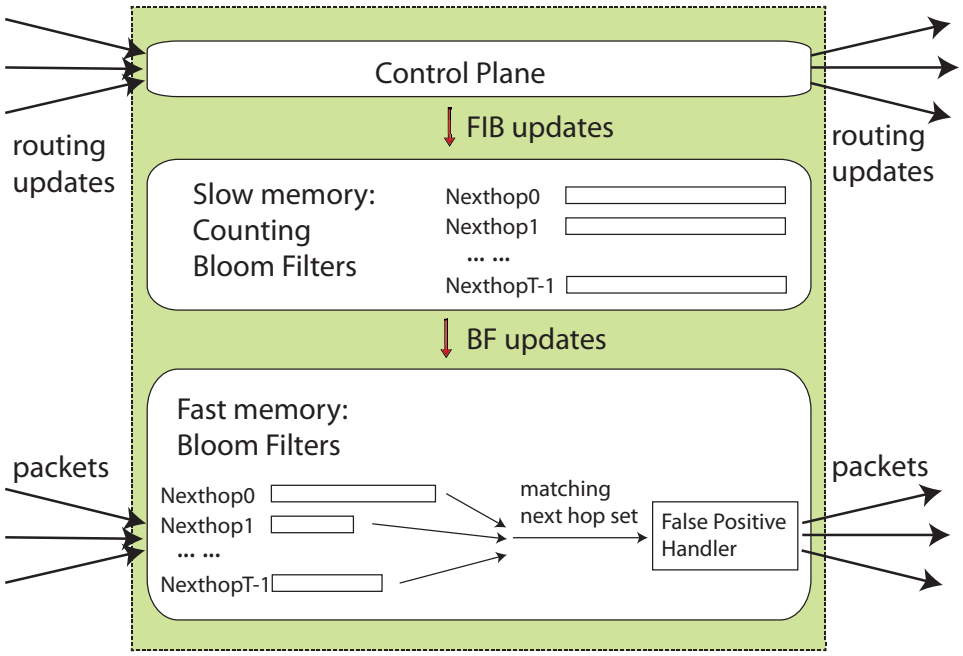


Figure 1: BUFFALO Switch Architecture

filters, a hash-based compact data structure to store a set of elements. We set one Bloom filter $BF(h)$ for each next hop h (or outgoing link), and use it to store all the addresses that are forwarded to that next hop. For a switch with T next hops, we need T Bloom filters. A Bloom filter consists of an array of bits. To insert an address into Bloom filter $BF(h)$, we compute k hash values of the address, 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 addresses with next hop h, Bloom filter $BF(h)$ is constructed.

It is easy to check if an address belongs to the set with Bloom filter $BF(h)$. Given an address, 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. To perform address lookup for an address *addr*, we check which $BF(h)$ contains *addr*, and forward the packet to the corresponding next hop h.

Note that there are different number of addresses associated with each next hop. Therefore we should use different size for each Bloom filter according to the number of addresses stored in it, in order to minimize the overall false-positive rate with a fixed size of fast memory. We formulate and solve the false-positive rate optimization problem in Section 5.

3.2 Handling False Positives in Fast Memory

One key problem with Bloom filters is the false positive — an element can be absent from the set even if all k positions are marked as 1, since each position could be marked by the other elements in the set. Because all the addresses belong to one of the Bloom filters we construct, we can easily detect packets that experience false positives if they match in multiple Bloom filters.[3]

One way to handle packets experiencing false positives is to perform a full packet lookup in the forwarding table stored in the slow memory. However, the lookup time for packets experiencing false positives will be much longer than others, leading to the throughput decrease. Attackers may detect those packets with longer latency and send a burst of them. Therefore, we must handle false positives in fast memory by picking one of the matching next hops.

For the packets that experience false positives, if we do not send them through the next hop on the shortest path, they may experience stretch and even loops. One way to prevent loops is to use a deterministic solution by storing the false positive information in the packets (similar to FCP [13]). When a packet is detected to have false positives in a switch, we store the switch and the next hop it chooses in the packet. So next time the packet travels to the same switch, the switch will choose a different next hop for the packet. However, this method requires modifying the packets and has extra payload overhead.

Instead, we use a probabilistic solution without any modification of the packets. We observe that if a switch sends the packet back to the interface where it comes from, it will form a loop. Therefore, we avoid sending the packet to the incoming interface. To finally get out the possible loops, we randomly pick one from all the remaining matching next hops. In Section 4, we prove that in SPAF networks, the packet experiencing one false positive (which is the most common case for the packet) is guaranteed to reach the destination with constant bounded stretch. In general, packets are guaranteed to reach the destination with probability 1. This approach does not require any help from the other switches in the network and thus is incrementally deployable.

3.3 CBFs for Handling Routing Changes

When routing changes occur, the switch must update the Bloom filters with a new set of addresses. However, with a standard Bloom filter (BF) we cannot delete elements from the set. A *counting Bloom filter* (CBF) is an extension of the Bloom filter that allows adding and deleting elements [14]. 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

[3]The handling of addresses that should have *multiple* matches (e.g., Equal-Cost Multi-Path) are discussed in Section 8. Addresses that have *no match* in the FIB should be dropped. Yet these packets may hit one Bloom filter due to false positives. We cannot detect these addresses, but they will eventually get dropped when they hit a downstream switch that has no false positives. In addition, an adversary cannot easily launch an attack because it is hard to guess which MAC addresses would trigger this behavior.

positions calculated by the hash functions; to delete an element, we decrement the counters.

A simple way to handle routing changes is to use CBFs instead of BFs for packet forwarding. However, CBFs require much more space than BFs. In addition, under routing changes the number of addresses associated with each next hop may change significantly. It is difficult to dynamically increase/decrease the sizes of CBFs to make the most efficient use of fast memory according to routing changes.

Fortunately, since routing changes do not happen very often, we can store CBFs in slow memory, and update the BFs in small fast memory based on the CBFs. We store CBF in slow memory rather than a normal FIB because it is easier and faster to update BFs from CBFs under routing changes. With a CBF layer between BFs and control plane, we can even change the sizes of BFs to the optimal values with low overhead. 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. The details of using CBFs are discussed in Section 6.

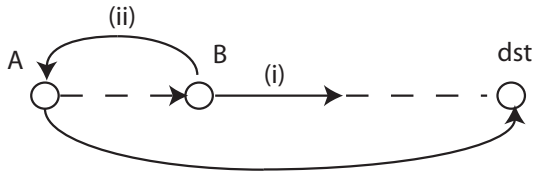
As shown in Figure 1, there are three layers in our switch architecture. The control plane can be either today's Ethernet or new Ethernet techniques such as Rbridges [1, 2] and SEATTLE [3]. CBFs are stored in slow memory and learn about routing changes from the control plane. BFs are stored in the fast memory for packet lookup. During routing changes, the related BFs will be updated according to the corresponding CBFs. If the number of addresses associated with a BF changes significantly, BFs are reconstructed with new optimal sizes from CBFs.

4 Handling False Positives

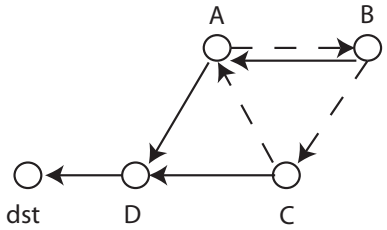
When BUFFALO detects packets that experience false positives, it randomly selects a next hop from the candidates that are different from the incoming interface, as discussed in Section 3.2. In the case of a single false positive, which is the most common, avoiding sending the packet to the incoming interface guarantees that packets reach destination with tightly bounded stretch. In the improbable case of multiple false positives, this randomization guarantees packet reachability with probability 1, with a good bounds on expected stretch.

Notation	Definition
$NH_{sp}(A)$	The next hop for shortest path at switch A
$NH_{fp}(A)$	The matching next hop due to a sole false positive at switch A
$l(A, B)$	The length of the shortest path from switch A to B
$P(A, B)$	All the paths from switch A to B

Table 1: Notations for the false-positive handler



(a) One false positive



(b) Multiple false positives

Figure 2: Loops caused by false positives (⇢ false positive link, ⟶ shortest path link)

4.1 Handling One False Positive by Avoiding Incoming Interface

We first investigate the case that a packet only experiences one false positive at one switch, i.e., at switch A, the packet has two matching next hops. Let $NH_{sp}(A)$ denote the next hop A should forward the packet to on the shortest path, and $NH_{fp}(A)$ denote the additional next hop matched by a Bloom filter false positive. (The notations are summarized in Table 4.) Note that this single false positive case is the most common case, because with reasonable Bloom filter parameters the probability of multiple false positives is much lower than one false positive. Since switch A connects to each end host through one switch port, there is no false positives for the ports that connect to end hosts. Therefore, $NH_{fp}(A)$ must be a switch rather than an end host.

The one false positive case is shown in Figure 2(a). (We hereafter use ⇢ to denote false positive link, and $\rightarrow$ for shortest path link.) Switch A has a false positive, and randomly picks a next hop B ($NH_{fp}(A) = B$). Switch B receives the packet, and may (i) send it to a next hop different from A, which leads to the destination or (ii) send it back to A. For (i), we will prove that there are not any loops. For (ii), when A receives the packet, it will not pick B since the packet comes from B.

In general, we have the following theorem:

Theorem 1. *In SPAF networks, if a packet only experiences one false positive in one switch, it is guaranteed to reach the destination with no loops except a possible single transient 2-hop one.*

Proof. Suppose the packet matches two next hops at switch A: $NH_{sp}(A)$ and some $B = NH_{fp}(A)$. If A picks $NH_{sp}(A)$, there is no loop. If B is selected, it will have a path $B \rightarrow \ldots \rightarrow dst$ to forward the packet to the destination, since the network is connected. With a single false positive, a packet will follow the correct shortest-path hops at all nodes other than A. Thus, the only case that can cause a loop is the packet going through A again ($A \dashrightarrow B \rightarrow \ldots \rightarrow A \rightarrow \ldots \rightarrow dst$). However, in SPAF networks, we assume the link $e(B, A)$ is actively used (This assumption is introduced in Sec. 2), i.e.,

$$l(NH_{fp}(A), A) \leq l(p), \forall p \in P(NH_{fp}(A), A).$$

Thus, on a shortest path from B to dst, if the packet visits A at all, it would be immediately after B. If the packet is sent back to A, A will avoid sending it to the incoming interface $NH_{fp}(A)$, and thus send the packet to $NH_{sp}(A)$, and the shortest path from there to dst can't go through A. Thus, the packet can only loop by following $A \dashrightarrow NH_{fp}(A) \rightarrow A \rightarrow \ldots \rightarrow dst$. So the path contains at most one 2-hop loop. □

Now we analyze the stretch (i.e., latency penalty) the packets will experience. For two switches A and B, let $l(A, B)$ denote the length of the shortest path from A to B. Let $l'(A, B)$ denote the latency the packet experiences on the path in BUFFALO. We define the stretch as:

$$S = l'(A, B) - l(A, B)$$

.

Theorem 2. *In SPAF networks, if a packet experiences just one false positive at switch A, the packet will experience a stretch of at most* $l(A, NH_{fp}(A)) + l(NH_{fp}(A), A)$.

Proof. Since there is one false positive in A, A will choose either $NH_{sp}(A)$ or $NH_{fp}(A)$. If A picks $NH_{sp}(A)$, there is no stretch. If A picks $NH_{fp}(A)$, there are two cases: (i) If $NH_{fp}(A)$ sends the packet to the destination without going through A, the shortest path from $NH_{fp}(A)$ to the destination is followed. Based on the triangle inequality, we have:

$$l(NH_{fp}(A), dst) \leq l(NH_{fp}(A), A) + l(A, dst)$$
$$l'(A, dst) - l(A, dst) \leq l(A, NH_{fp}(A)) + l(NH_{fp}(A), A)$$

(ii) If $NH_{fp}(A)$ sends the packet back to A, the stretch is $l(A, NH_{fp}(A)) + l(NH_{fp}(A), A)$. □

Therefore, we prove that in the one false positive case, packets are guaranteed to reach the destination with bounded stretch in SPAF networks.

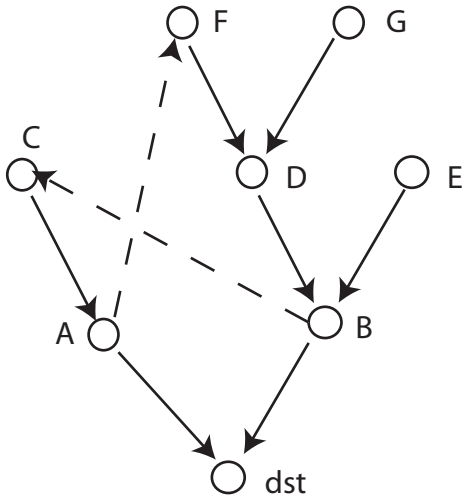


Figure 3: BUFFALO forwarding graph (⇢ false positive link, ⟶ shortest path tree)

4.2 Handling Multiple False Positives with Random Selection

Now we consider the case where all the switches in the network apply our Bloom filter mechanism. We choose different hash functions for different switches so that the false positives are independent among the switches. (We can choose different keys for key-based hash functions to generate a group of hash functions.) Thus it is rare for a packet to experience false positives at multiple switches.

Let us fix a destination *dst*, and condition the rest of this section on the fixed forwarding table and memory size (which, per Section 5, means the Bloom filter parameters are fixed, too). Let $f(h)$ be the probability of Bloom filter h erroneously matching *dst* (a priori independent of *dst* if *dst* shouldn't be matched). Then, k, the number of Bloom filters mistakenly matching d is, in expectation, $F = \sum_h f(h)$. If all values of f are comparable and small, F is roughly binomial, with $\Pr[k > x]$ decays exponentially for $x \gg 2f|E|$.

There may exist loops even when we avoid picking the incoming interface. For example, in Figure 2(b), A may choose the false-positive hop to B at first, and B may choose the false-positive next hop C. However, the packet can eventually get out of the loop if each switch chooses the next hop randomly: any loop must contain a false-positive edge, and the source node of that edge will with some probability choose its correct hop instead. E.g., A will eventually choose the next hop D to get out of the loop. Such random selection may also cause out-of-order packets. This may not be a problem for some applications. For the applications that require in-order packet forwarding, we can still reorder packets in the end hosts.

In general, in SPAF networks there is a shortest path tree for each destination, with each packet to *dst* following this tree. In BUFFALO, packets destined to *dst* are forwarded in the *directed* graph consisting of a shortest path tree for *dst* and a few false positive links. We call this graph *BUFFALO forwarding graph*. An example is shown in Figure 3, where 2 false positives occurred at A and B. Note that, if the shortest-path links are of similar length, the link from A to F can't be less than about twice that length: otherwise F's shortest path would go through A. If all links are the same length (latency), no false positive edge can take more than 1 "step" away from the destination.

If a packet arrives at the switch that has multiple outgoing links in BUFFALO forwarding graph due to false positives, we will randomly select a next hop from the candidates. Thus, the packet actually takes a random walk on the BUFFALO forwarding graph, usually following shortest-path links with occasional "setbacks".

Theorem 3. *With an adversarially designed BUFFALO network with uniform edge latencies, and even worst-case placement of false positives, the expected stretch of a packet going to a destination associated with k false positives is at most $S(k) = \rho \cdot (\sqrt[3]{3})^k$, where $\rho = \frac{529}{54 \cdot \sqrt[3]{3}} < 6.8$.*

Proof sketch: We couple the random walk on the BUFFALO graph with a random walk on a "line" graph which only records the current hop distance to the destination, as shown in Figure 4. The tight bound is produced by the worst-case scenario of the network shaped like a line itself, with three false positives at all but 4-6 steps pushing the packet "back up" the line. The complete proof is shown in the Appendix. □

Though this is exponential, this is counterbalanced by the exponentially low probability of k false positives. Tuning the Bloom filter parameters to optimize memory usage will allow us to bound F, yielding at least a superlinear tail bound on the probability of large stretches, assuming f values are comparable to $1/2m$ or smaller, yielding $F = O(1)$:

Theorem 4. *For any $z \geq 60.3 \cdot 7.3221^F$, the probability of stretch exceeding z is bounded by $2/z$. Asymptotically, the tail will decay as $\tilde{O}(1/z^{1.54})$, to within polylog factors. (The proof is given in the Appendix.)*

This bound characterizes the worst-case configuration the network may end up in after any particular control-plane event. As a description of the *typical* behavior, on the other hand, this bound is quite crude. We expect that an *average-case analysis* over false positive locations, corresponding to the typical behavior of BUFFALO, will yield polynomial expected stretch for fixed k: the exponential worst-case behavior relies on all the false-positives carefully "conspiring" to point away from the destination, and randomly placed false positives, as with real Bloom filters, will make the random walk behave *similarly* to a random walk on an undirected graph, producing polynomial hitting times. This will allow $z = poly(k)$ and hence an *exponentially* decaying stretch distribution.

While our scheme works with any underlying network structure, it works particularly well with a tree topology. Tree topologies are common in the edges

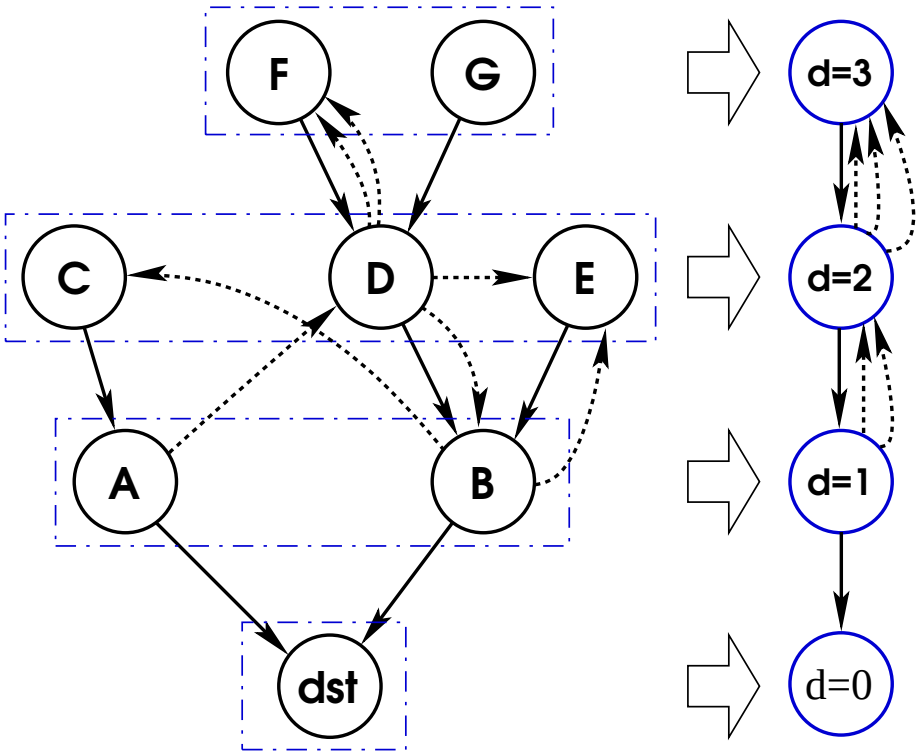


Figure 4: Coupling for the expected stretch proof: all nodes at the same hop distance from *dst* are collapsed into 1 node. Forward false positive links, like D to B, are dropped. The number of backward edges on the line graph is the maximum over all nodes at that distance of the number of backward and same-depth false-positive links (B's edges at $d = 1$, D's edges at $d = 2$). A random walk on the line graph converges no faster than a random walk on the original. The line graph itself is a valid network, thus allowing for tight bounds.

of enterprise and data center networks. A logical tree is usually constructed with the spanning tree protocols in today's Ethernet. Roughly speaking, in a tree, a lot of distinct false positives are needed at each distance from the destination in order to keep "pushing" the packet away from the destination.

Claim 5. *If the underlying network is a tree, with no multiple links between any one pair of routers, the expected stretch with k false positives, even if they are adversarially placed, is at most $2(k-1)^2$. (The proof is given in the Appendix.)*

We believe that similar results should apply when we allow heterogeneous latencies, especially when the per-link latencies are within a small constant factor of each other, as is likely in many geographically-local networks.

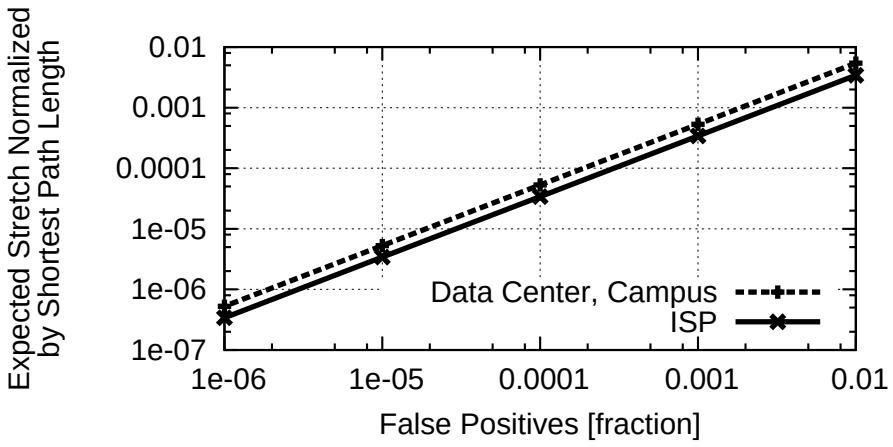


Figure 5: Expected stretch

4.3 Stretch in Realistic Networks

We evaluate the stretch in three representative topologies: Campus is the campus network of a large (roughly 40,000 students) university, consisting of 1600 switches [15]. AS 1239 is a large ISP network with 315 routers [16]. (The routers are viewed as switches in our simulation.) We also constructed a model topology similar to the one used in [3], which represents a typical data center network composed of four full-meshed core routers each of which is connected to a mesh of twenty one aggregation switches. This roughly characterizes a commonly-used topology in data centers [17].

In the three topologies, we first analyze the expected stretch given the false-positive rate. We then use simulation to study the stretch with real packet traces.

Analysis of expected stretch: We pick the false-positive rate of Bloom filters and calculate the expected stretch for each pair of source and destination in the network by analyzing all the cases with different numbers and locations of false positives and all the possible random selections. The expected stretch is normalized by the length of the shortest path. We take the average stretch among all source-destination pairs. We can see that the expected stretch increases linearly with the increase of the false-positive rate. This is because the expected stretch is dominated by the one false-positive case. Since we provide a constant stretch bound for the one false-positive case in BUFFALO, the expected stretch is very small. Even with a false-positive rate of 1%, the expected stretch is only 0.5% of the length of the shortest path (Figure 5).

Simulation on stretch distribution: We also study the stretch of BUFFALO with packet traces collected from the Lawrence Berkeley National Lab campus network by Pang et. al. [18]. There are four sets of traces, each collected over a period of 10 to 60 minutes, containing traffic to and from roughly 9,000 end hosts distributed over 22 different subnets. Since we cannot get the

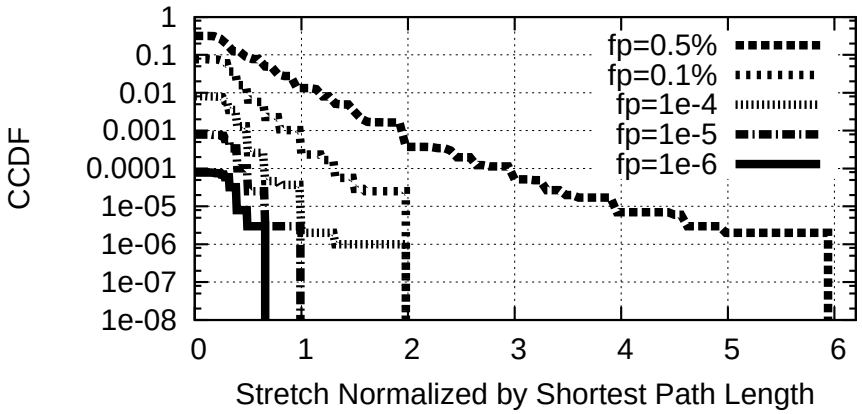


Figure 6: Stretch with real traces in campus network

network topology where the trace is collected, we take the same approach in [3] to map the trace to the above campus network while preserving the distribution of source-destination popularity of the original trace. Figure 6 shows the distribution of stretch normalized by shortest path length. When the false-positive rate is 0.01%, 99% of the packets do not have any stretch and 0.01% of the packets have a stretch that is twice as long as the length of the shortest path. Even when the false-positive rate is 0.5%, only 0.0001% of the packets have a stretch of 6 times of the length of the shortest path. Note that in an enterprise or data center, the propagation delays are small, so the stretch caused by false positives is tolerable.

5 Optimizing Memory Usage

In this section, we consider a switch with M-bit fast memory (i.e., SRAM) and a fixed routing table. We formulate the problem of minimizing the overall false-positive rate through tuning the sizes in the Bloom filters. This optimization is done by a Bloom filter manager implemented in a BUFFALO switch. We then show numerical results of false positives with various sizes of memory and forwarding tables.

5.1 Optimizing Bloom-Filter Sizes

Our goal is to minimize 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 send the packets through a slightly longer path as described in Section 4. To reduce the stretch, we must minimize the false positives in each switch. We define the overall false-positive rate for a switch as the probability that any one of the T Bloom filters has a false positive. As above, let $f(h)$ denote

the false-positive rate of Bloom filter $BF(h)$. 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

$$F = 1 - \prod_{h=1}^{T}(1 - f(h)) \approx \sum_{h=1}^{T} f(h)$$
$$(when\ f(h) \ll 1/T, \forall h = 1..T)$$

Optimizing the sum approximation for F also directly optimizes the applicability threshold for Theorem C.1, expressed in terms of the sum as such.

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 M-bit fast memory.

In addition to constraining the fast memory size, we should also avoid overloading the CPU. We bound the packet lookup time, which consists of hash computation time and memory access time. To reduce the computational overhead of address lookup, we apply the same group of hash functions to all T Bloom filters. Since we use the same hash functions for all the Bloom filters, we need to check the same positions in all the Bloom filters for an address lookup. Therefore, we put the same positions of the Bloom filters in one memory unit (e.g., a byte or a word), so that they can be accessed by one memory access. In this scheme, the packet lookup time is determined by the maximum number of hash functions in T Bloom filters ($k_{max} = \max_{h=1}^{T}(k(h))$, where $k(h)$ denotes the number of hash functions used in $BF(h)$). Let u_{hash} denote the number of hash functions that can be calculated in a second. We need k_{max}/u_{hash} time for hash computation. Let t_{fmem} denote the access time on small, fast memory. Assume there are b bits in a memory unit which can be read by one memory access. We need $\lceil (Tk_{max}/b)t_{fmem} \rceil$ memory access time. Since both hash computation and memory access time are linear in the maximum number of hash functions k_{max}, we only need to bound k_{max} in order to bound the packet lookup time.[4]

We minimize the lookup time for each packet by choosing $m(h)$ (the number of bits in $BF(h)$) and $k(h)$ (the number of hash functions used in $BF(h)$), with the constraint that Bloom filters must not take more space than the size of the fast memory and must have a bounded number of hash functions. Let $n(h)$ denote the number of addresses in Bloom filter $BF(h)$. The optimization

[4]In switch hardware, the 4-8 hash functions can be calculated in parallel. We also assert that fabrication of 6 to 8 read ports for an on-chip Random Access Memory is attainable with today's embedded memory technology [19]. The cache line size on a Intel Xeon machine is about 32 bytes to 64 bytes, which is enough to put all the positions of T Bloom filters in one cache line.

problem is formulated as:

$$Min \quad F = \sum_{h=1}^{T} f(h) \tag{1}$$

$$s.t. \quad f(h) = (1 - e^{-k(h)n(h)/m(h)})^{k(h)} \tag{2}$$

$$\sum_{h=1}^{T} m(h) = M \tag{3}$$

$$k(h) \leq k_{max}, \forall h \in [1..T] \tag{4}$$

$$given \quad T, M, k_{max}, \ and \ n(h)(\forall h \in [1..H])$$

Equation (1) is the overall false-positive rate we need to minimize. Equation (2) shows the false-positive rate for a standard Bloom filter. Equation (3) is the size constraint of the fast memory. Equation (4) is the bound on the number of hash functions. We have proved that this problem is a convex optimization problem.[5] Thus there exists an optimal solution for this problem, which can be found by the IPOPT [20] (Interior Point OPTimizer) solver. Most of our experiments converge within 30 iterations, which take less than 50 ms. Note that the optimization is executed only when the forwarding table has significant changes such as a severe link failure leading to lots of routing changes. The optimization can also be executed in the background without affecting the packet forwarding.

5.2 Analytical Results of False Positives

We study in a switch the effect of forwarding table size, number of next hops, the amount of fast memory, and number of hash functions on the overall false-positive rate.[6] We choose to analyze the false positives with synthetic data to study various sizes of forwarding tables and different memory and CPU settings. We have also tested BUFFALO with real packet traces and forwarding tables. The results are similar to the analytical results and thus omitted in the paper. We studied a forwarding table with 20K to 2000K entries (denoted by N), where the number of next hops (T) varies from 10 to 200. The maximum number of hash functions in the Bloom filters (k_{max}) varies from 4 to 8. Since next hops have different popularity, Pareto distribution is used to generate the number of addresses for each next hop. We have the following observations:

(1) A small increase in memory size can reduce the overall false-positive rate significantly. As shown in Figure 7, to reach the overall false-positive rate of 0.1%, we need 600 KB fast memory and 4-8 hash functions to store a FIB with 200K entries and 10 next hops. If we have 1 MB fast memory, the false-positive rate can be reduced to the order of 10^{-6}.

[5] The proof is omitted due to lack of space.

[6] In Section 4, the false-positive rate is defined for each Bloom filter. Here the overall false-positive rate is defined for the switch because different Bloom filters have different false-positive rates. The overall false-positive rate can be one or two orders of magnitude larger than individual Bloom-filter false-positive rate.

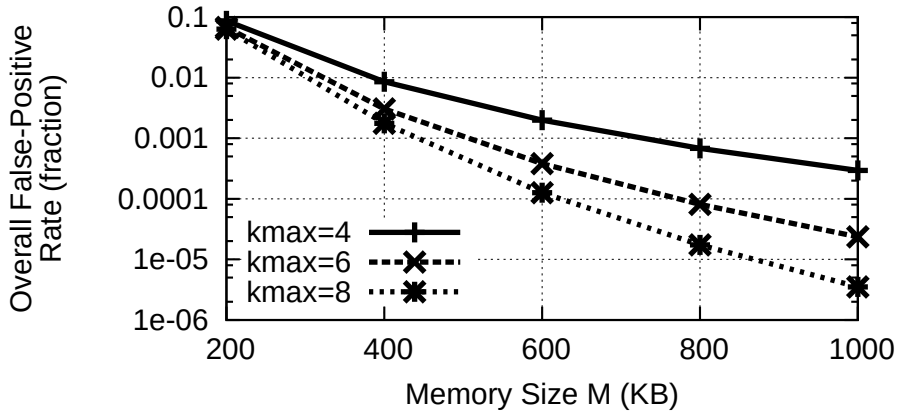


Figure 7: Effect of memory size ($T = 10, N = 200K$)

(2) The overall false-positive rate increases almost linearly with the increase of T. With the increase of T and thus more Bloom filters, we will have larger overall false-positive rate. However, as shown in Figure 8, even for a switch that has 200 next hops and a 200K-entry forwarding table, we can still reach a false-positive rate of 1% with 600KB fast memory ($k_{max} = 6$). This is because if we fix the total number of entries N, with the increase of T, the number of addresses for each next hop drops correspondingly.

(3) BUFFALO switch with fixed memory size scales well with the growth of forwarding table size. For example, as shown in Figure 9, if a switch has a 1MB fast memory, as the forwarding table grows from 20K to 1000K entries, the false-positive rate grows from 10^{-9} to 5%. Since packets experiencing false positives are handled in fast memory, BUFFALO scales well with the growth of forwarding table size.

(4) BUFFALO reduces fast memory requirement by at least 65% compared with hash tables at the expense of a false-positive rate of 0.1%. We assume a perfect hash table that has no collision. Each entry needs to store the MAC address (48 bits) and an index of the next hop ($\log(T)$ bits). Therefore the size of a hash table for an N-entry forwarding table is $(\log(T) + 48)N$ bits. Figure 10 shows that BUFFALO can reduce fast memory requirements by 65% compared with hash tables for the same number of FIB entries at the expense of a false-positive rate of 0.1%. With the increase of forwarding table size, BUFFALO can save more memory. However in practice, handling collisions in hash tables requires much more memory space and affects the throughput. In contrast, BUFFALO can handle false positives without any memory overhead. Moreover, the packet forwarding performance of BUFFALO is independent of the workload.

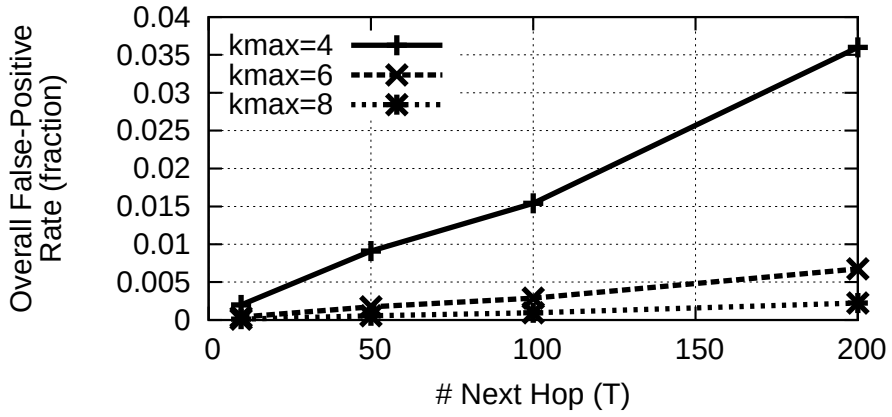


Figure 8: Effect of number of next hops ($M = 600KB, N = 200K$)

6 Handling Routing Changes

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.

6.1 Update BF Based on CBF

In a switch, the control plane maintains the RIB (Routing Information Base) and updates the FIB (Forwarding Information Base) in the data plane. When FIB updates are received, the Bloom filters should be updated accordingly. We use CBFs in slow memory to assist the update of Bloom filters. We implement a group of T CBFs, each containing the addresses associated with one next hop. To add a new route of address *addr* with next hop h, we will insert *addr* to $CBF(h)$. Similarly, to delete a route, we remove *addr* in $CBF(h)$. 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 when the number of addresses in one CBF increases significantly due to routing changes, the false-positive rates on CBFs are kept low.

After the $CBF(h)$ is updated, we update the corresponding $BF(h)$ based on the new $CBF(h)$. We only need to modify a few BFs that are affected by the routing changes without interrupting the packet forwarding with the rest BFs. To minimize the interruption of packet forwarding with the modified BFs, we implement a pointer for each BF in SRAM. We first generate new snapshots of BFs with CBFs and then change the BF pointers to the new snapshots. The extra fast memory for snapshots is small because we only need to modify a few BFs at a time.

If the CBF and BF have the same number of positions, we can easily update the BF by checking if each position in the CBF is 0 or not. The update from

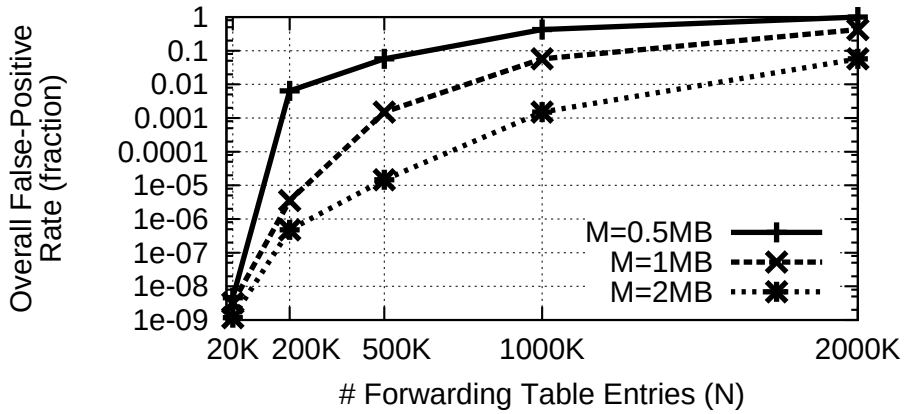


Figure 9: Effect of number of entries ($T = 10, k_{max} = 8$)

CBF to BF becomes more challenging when we have to dynamically adjust the size of the BF to reduce the overall false-positive rate.

6.2 Adjust BF Size Without Reconstructing CBF

When the forwarding table changes over time, the number of addresses 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. 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 addresses in the BF decreases. However, when the number of addresses 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 11, we can 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 addresses in each CBF, and periodically reconstruct BFs to be of the optimal sizes and number of hash functions. Since resizing a BF based on a CBF requires the BF and CBF to use the same number of hash functions. We need to adjust the number of hash functions in the CBF before resizing the BF. 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. Solving the optimization problem in Section 5, we first get the optimal size of each

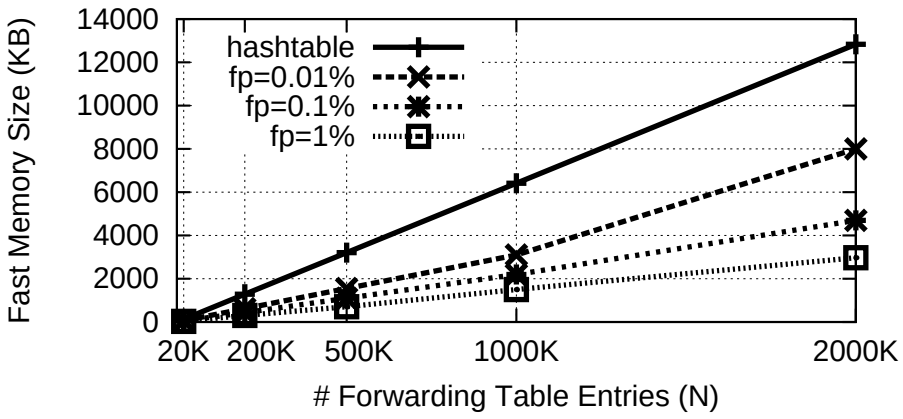


Figure 10: Comparison of hash table and Bloom filters ($T = 10, k_{max} = 8$)

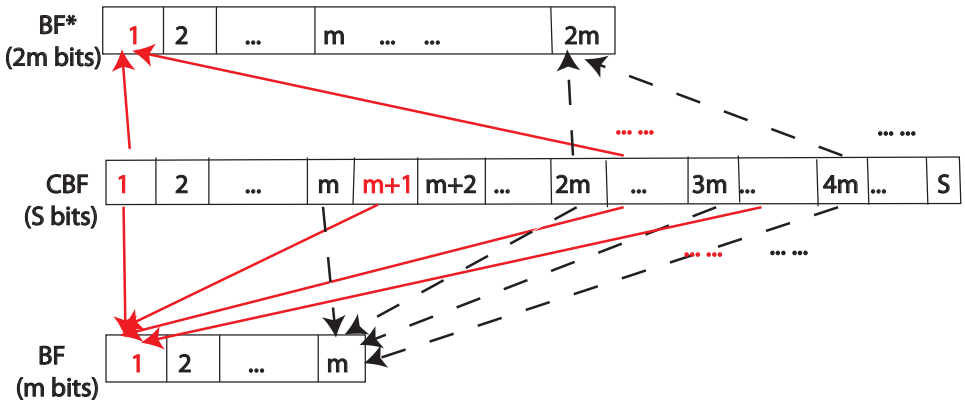


Figure 11: Adjust BF size from m to $2m$ based on CBF

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 positives with size m' and the number of addresses n in the BF, which is $m' \ln 2/n$ based on standard Bloom filter analysis [4]. We also need to bound the number of hash functions by k_{max}. Thus the number of hash functions is $k' = \min(k_{max}, m' \ln 2/n)$.

Step 2: If $k \neq k'$, change the number of hash functions in the CBF from k to k'. The number of hash functions does not always change because routing changes are sometimes not significant and we have the k_{max} bound. When we must change k, there are two ways with either more computation or more space: (i) If $k' > k$, we obtain all the addresses currently in the forwarding table from the control plane, calculate the hash values with the $k' - k$ new hash functions on all the addresses 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 corresponding counters. (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 11, 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.

7 Implementation and Evaluation

To verify the performance and practicality of our mechanism through a real deployment, we built a prototype *BuffaloSwitch* in kernel-level Click [7]. The overall structure of our implementation is shown in Figure 1. *BuffaloSwitch* consists of four modules:

Counting Bloom filters: The counting Bloom filter module is used to receive routing changes from the control plane and increment/decrement the counters in the related CBFs correspondingly.

Bloom filters: The Bloom filter module maintains one Bloom filter for each next hop. It also performs the packet lookup by hash calculation and checking all the Bloom filters. When it finds out the multiple next hop candidates due to false positives, it will call the false positive handler.

Bloom filter manager: The Bloom filter manager monitors the number of addresses in each BF. If the number of addresses in one BF changes significantly (above threshold TH), we recalculate the optimal size of the BF and reconstruct it based on the CBF.

False positive handler: The false positive handler module is responsible for selecting a next hop for the packets that experience false positives.

To evaluate our prototype, we need a forwarding table and real packet traces. We map the Lawrence Berkeley National Lab Campus network traces [18] to the campus network topology [15] as described in Section 4.3. We then calculate shortest paths in the network and construct the forwarding table accordingly. The forwarding table consists of 200K entries.

We run *BuffaloSwitch* on a 3.0 GHz 64-bit Intel Xeon machine with a 8 KB L1 and 2 MB L2 data cache. The main memory is a 2 GB 400 Mhz DDR2 RAM. We take the fast memory size M as 1.5MB to make sure Bloom filters

fit in the L2 cache. The Bloom filter manager optimizes sizes of Bloom filters given the forwarding table and M. To avoid the potential bottleneck at the Ethernet interfaces, we run the Click packet generator on the same machine with *BuffaloSwitch*. We send the packet with constant rate and measure the peak forwarding rate of *BuffaloSwitch*. The packet size is set as 64 bytes, which is the minimum Ethernet packet size, so that the packet payload does not pollute the cache much. For comparison, we also run *EtherSwitch* — a standard Click element which performs Ethernet packet forwarding using hash tables.

Our experiment shows that *BuffaloSwitch* achieves a peak forwarding rate of 365 Kpps, 10% faster than *EtherSwitch* which has 330 Kpps peak forwarding rate. This is because all the Bloom filters in *BuffaloSwitch* fit in the L2 cache, but the hash table in *EtherSwitch* does not and thus takes longer time to access memory. The forwarding rate with *BuffaloSwitch* can be further improved by parallelizing the hash calculations on multiple cores.

To measure the performance of *BuffaloSwitch* under routing changes, we generate a group of routing updates which randomly change FIB entries and replay these updates. It takes 10.7 μsec for *BuffaloSwitch* to update the Bloom filters for one route change. Under significant routing changes, it takes an additional 0.47 seconds to adjust the Bloom filter sizes based on counting Bloom filters. This is because CBFs are very large and takes longer time to scan through them. However, it is much faster than recalculating hash functions for all FIB entries to reconstruct Bloom filters.

8 Extensions to BUFFALO

In this section we discuss the extensions of BUFFALO to support ECMP, VLAN, broadcast and multicast packets, and backup routes.

Supporting ECMP: In shortest-path routing protocols like OSPF and IS-IS, ECMP (Equal-Cost Multi-Path) is used to split traffic among shortest paths with equal cost. When a destination address has multiple shortest paths, the switch inserts the destination address into the Bloom filter of each of the next hops. Since packets with this address match multiple next hops, the BUFFALO false-positive handler will randomly choose one next hop from them, achieving even splitting among these equal-cost multiple paths.

Supporting virtual LANs: VLAN is used in Ethernet to allow administrators to group multiple hosts into a single broadcast domain. A switch port can be configured with one or more VLANs. We can no longer use just a single Bloom filter for each port because due to false positives a packet in VLAN A may be sent to a switch which does not know how to reach VLAN A and thus get dropped. To support VLANs in BUFFALO, we use one Bloom filter for each (VLAN, next hop) pair. For a packet lookup, we simply check those Bloom filters that have the same VLAN as the packet. However, this does not scale well with a large number of VLANs in the network. For future architectures that have simpler network configuration and management methods rather than VLAN, we do not have this problem.

Broadcast and multicast: In this paper, we have focused on packet forwarding for unicast traffic. To support Ethernet broadcast, the switch can identify the broadcast MAC address and forward broadcast packets directly without checking the Bloom filters. Supporting multicast in the layer-2 network is more complex. One way is to broadcast all the multicast packets and let the NIC on the hosts decide whether to accept or drop the packets. Another way is to allow switches to check whether the destination IP address of the packet is a multicast-group packet, and leverage IP multicast solutions such as storing the multicast forwarding information in packets [21, 22].

Fast failover to backup routes: When a significant failure happens such as one of the switch's own links fail, many routes change in a short time. In order to quickly recover from significant failures, we provide an optional optimization of BUFFALO. The control plane calculates backup routes for every link/node failure case in advance, and notifies the data plane about the failure event. In addition to the original routes stored in $CBF(h)$ ($h \in [1..T]$), we pre-calculate backup counting Bloom filters $CBF(h_1, h_2)$ (for all $h_1 \in [1..T], h_2 \in [1..T]$), which denotes the set of addresses that are originally forwarded to next hop h_1, but if h_1 is not accessible, they should be forwarded to next hop h_2. When the failure happens and thus h_1 is not accessible, we simply need to update the Bloom filters based on the original Bloom filters and backup counting Bloom filters. For example, we update $BF(h_2)$ based on the old $BF(h_2)$ and $CBF(h_1, h_2)$. This is fast because merging two Bloom filters is just OR operations.

9 Related Work

Bloom filters have been used for IP packet forwarding, and particularly the longest-prefix match operation [6]. 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. The authors in [23] design a *d-left scheme* using d hash functions for IP lookups. To perform an IP lookup, they still need to access the slow memory at least d times. The paper [24] stores Bloom filter in the fast memory, and stores 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. Different from these works, we focus on flat addresses and perform the *entire* lookup in fast memory at the expense of a few false positives. We also propose a simple scheme that handles false positives within fast memory, and proves its reachability and stretch bound.

Bloom filters have also been used in resource routing [4, 5], which applies Bloom filters to probabilistic algorithms for locating resources. Our "one Bloom filter per next hop" scheme 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. To keep up with link speed in packet forwarding with a 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. We also handle false positives without any memory overhead. BUFFALO is also similar to *Bloomier filters* [25] in that we both use a group of Bloom filters, one for each value of a function that maps the key to the value. However, Bloomier filters only work for a *static* element set.

Bloom filters are also been used for multicast forwarding. LIPSIN [22] uses Bloom filters to encode the multicast forwarding information in packets. False positives in Bloom filters may cause loops in its design. LIPSIN caches packets that may experience loops and send the packets to a different link when a loop is detected. However, they do not show how well they can prevent loops and the cache size they need. In contrast, our loop prevention mechanism is simple and effective, and does not have any memory overhead.

To handle routing changes, both [6] and [24] store counting Bloom filters (CBFs) in fast memory, which uses more memory space than the Bloom filters (BFs). 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 [14], 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.

Our idea of using one Bloom filter per port is similar to SPSwitch [26] which forward packets on flat identifiers in content-centric networks. Our workshop paper [27] applies Bloom filters for enterprise edge routers by leveraging the fact that edge routers typically have a small number of next hops. However, it does not deal with loops caused by false positives. The paper uses one Bloom filter for each (next hop, prefix length) pair and discusses its effect on false positives. It also proposes the idea of using CBF to assist the BF update and resizing. We consider flat address lookup in SPAF networks in this paper, and thus eliminate the effect of various prefix lengths. We also propose a mechanism to handle false positives in the network without extra memory. We perform extensive analysis, simulation, and prototype implementation to evaluate our scheme.

10 Conclusion

With recent advances in improving control plane scalability, it is possible now to build large layer-2 networks. The scalability problem in the data plane becomes challenging with increasing forwarding table sizes and link speed. Leveraging flat addresses and shortest path routing in SPAF networks, we proposed BUFFALO, a practical switch design based on Bloom filters. BUFFALO performs the entire packet forwarding in small, fast memory including those packets ex-

periencing false positives. BUFFALO gracefully degrades under higher memory loads by gradually increasing stretch rather than crashing or resorting to excessive flooding. Our analysis, simulation and prototype demonstrate that BUFFALO works well in reducing memory cost and improving the scalability of packet forwarding in enterprise and data center networks.

11 Acknowledgment

Alex Fabrikant was supported by Princeton University postdoctoral fellowship and a grant from Intel. We would like to thank Matt Caesar, Changhoon Kim Sanjay Rao, and Yi Wang for their comments on improving the paper.

References

[1] "IETF TRILL working group." `http://www.ietf.org/html.charters/trill-charter.html`.

[2] R. Perlman, "Rbridges: Transparent routing," in *Proc. IEEE INFOCOM*, 2004.

[3] C. Kim, M. Caesar, and J. Rexford, "Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises," *ACM SIGCOMM*, 2008.

[4] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2005.

[5] S. C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *Proc. IEEE INFOCOM*, 2002.

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

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

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

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

[10] S. M. Bellovin, "Distributed firewalls," *;login:*, Nov. 1999.

[11] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," *ACM Conference on Computer and Communications Security*, 2000.

[12] "Introduction to server and domain isolation." `technet.microsoft.com/en-us/library/cc725770.aspx`.

[13] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica, "Achieving convergence-free routing using failure-carrying packets," in *Proc. ACM SIGCOMM*, 2007.

[14] 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.

[15] Y.-W. E. Sung, S. Rao, G. Xie, and D. Maltz, "Towards systematic design of enterprise networks," in *Proc. ACM CoNEXT*, 2008.

[16] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP topologies with Rocketfuel," in *IEEE/ACM Transactions on Networking*, 2004.

[17] M. Arregoces and M. Portolani, *Data Center Fundamentals*. Cisco Press, 2003.

[18] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proc. Internet Measurement Conference*, 2005.

[19] B. Dipert, "Special purpose SRAMs smooth the ride," *EDN magazine*, 1999.

[20] "Interior point optimizer." `www.coin-or.org/Ipopt/`.

[21] S. Ratnasamy, A. Ermolinskiy, and S. Shenker, "Revisiting ip multicast," in *ACM SIGCOMM*, 2006.

[22] P. Jokela, A. Zahemszky, C. Esteve, S. Arianfar, and P. Nikander, "LIPSIN: Line speed publish/subscribe inter-networking," in *Proc. ACM SIGCOMM*, 2009.

[23] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proc. IEEE INFOCOM*, 2001.

[24] 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.

[25] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: An efficient data structure for static support lookup tables.," in *Proc. Symposium on Discrete Algorithms*, 2004.

[26] C. Esteve, F. L. Verdi, and M. F. Magalhaes, "Towards a new generation of information-oriented internetworking architectures," *ReArch Workshop*, 2008.

[27] M. Yu and J. Rexford, "Hash, don't cache: Fast packet forwarding for enterprise edge routers," in *Proc. ACM SIGCOMM Workshop on Research in Enterprise Networks*, 2009.

[28] D. Aldous and J. Fill, "Reversible markov chains and random walks on graphs." Monograph manuscript at `http://www.stat.berkeley.edu/~aldous/RWG/book.html`, retrieved on 2009-10-07.

[29] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995.

[30] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[31] J. L. Palacios, "Bounds on expected hitting times for random walk on a connected graph," *Linear Algebra and Its Applications*, vol. 141, pp. 241–252, 1990.

[32] E. Seneta, *Non-negative matrices and Markov Chains*. Springer, 2nd ed., 1981.

Appendix A Model

For all the results here, we make two simplifying assumptions:

1. We assume the underlying SPAF computation is based on "hop count" only, or, equivalently, equal edge weights on all edges. We expect that the bounds we derive under this assumption will not differ qualitatively from the typical case of edge weights that are diverse but roughly on the same order of magnitude.

2. BUFFALO implements a "no-bounce" rule: if a node detects a false positive, it randomly selects a next hop out of the several candidates that the Bloom filters then offer, but *excludes* from consideration the hop that the packet arrived from. We allow the selection of the latter, as well, ignoring the "no-bounce" optimization in BUFFALO. This simplifies the analysis, and we expect that the upper bounds should not get any worse in the presence of this feature.

We represent the underlying network with an undirected graph $G = (V, E)$, with n nodes, each representing a switch or an end host. Consider forwarding to a particular destination $\mathsf{dst} \in V$ with some, possibly all, switches in the network implementing BUFFALO. The underlying protocol builds a shortest-path tree $(V, T \subset E)$, directed toward dst.

After any given FIB update, some BUFFALO-using routers may have false positives for destination dst. A false positive at node v for next hop v' will cause BUFFALO to possibly forward from v to v' — but this can only happen

if the edge $\{v, v'\}$ exists in G. Let (V, B) be the directed "BUFFALO graph," containing all the edges along which BUFFALO switches might forward, given the current false positives. That is, B consists of T, and, for each false positive at node v that matches next hop v' for destination dst, an additional (directed) edge (v, v'). We use P to denote the false-positive edges alone ($P = B \setminus T$).

Observation A.1. *Without the "no-bounce" rule, if some or all routers use BUFFALO, a packet to* dst *performs an unbiased random walk on* (V, B) *until reaching* dst*.*

For background on random walks, we refer the reader to any of a number of standard books, such as [28, 29, 30].

We denote by $d_T(v, w)$ the hop distance from v to w in T, and use $d(v)$ as shorthand for $d_T(v, \mathsf{dst})$, the distance from v to the destination in T, which we call the *depth* of v.

Observation A.2. $d(v) = d_E(v, \mathsf{dst}) = d_B(v, \mathsf{dst})$*, since* T *is a shortest-path tree, and is contained in* B *and* E*. The shortest-path property also ensures that, for any edge* $\{v, w\} \in E$*,* $|d(v) - d(w)| \leq 1$*.*

We say a packet traveling from v to dst experiences stretch S if it traverses $d(v) + S$ hops before reaching dst. The expected stretch of a packet starting at v is directly connected to the random walk's expected hitting time from v to dst: $T_{v,\mathsf{dst}} = d(v) + \mathrm{E}[S]$.

The simulations in Section 4 show that BUFFALO networks almost never admit stretch longer than a couple of hops in the networks we examined there. Here, we present analytic bounds on stretch in worst-case networks and under worst-case locations of false positives.

A.1 Average-case vs worst-case considerations

We evaluate the scaling of BUFFALO as the number of nodes in the network grows, considering the worst-case behavior under any possible graph structure of the network.

The random choice of *which* next-hop to follow, out of the ones that have a Bloom filter hit, is independent at each router and not linked to any quantity that can be influenced by any party outside the router, assuming a secure source of randomness. Our analysis is thus average-case over these random choices in a packet's walk through the network.

Separately, there is the randomness inherent in the Bloom filter behavior that defines where false positives appear.

Since forwarding happens on a much faster time scale than FIB changes, the results here focus exclusively on the behavior of the network between two FIB updates. In this framework, we consider the *worst-case placement of Bloom filter false positives.*

Our upper and lower bounds are tight, but, unsurprisingly, the worst examples that comprise the lower bounds happen when the false positives "conspire"

to push the packet away from dst. This seems very unlikely given that the Bloom filters use independent hashes. We expect that much stronger bounds can be obtained when analyzing the behavior in the *average case over false positive placements*. This would better capture both *typical* behavior of BUFFALO, as well as the *expected long-term average behavior of BUFFALO*, on a time scale long enough to regularly repopulate the Bloom filters.

In short, the results below evaluate BUFFALO's forwarding stretch in worst-case networks, with worst-case false-positive placements, averaged over random walk choices. We believe relevant and likely much stronger bounds may arise from analyzing worst-case networks, averaged over possible false-positive placements, and averaged over random walk choices. We leave the latter as a major open problem of this work.

Appendix B Expected stretch upper bound for k false positives

Theorem B.1 (Theorem 3 in Section 4)**.** *With an adversarially designed BUFFALO network with uniform edge latencies, and even worst-case placement of false positives, the expected stretch of a packet going to a destination associated with k false positives is at most $S(k) = \rho \cdot (\sqrt[3]{3})^k$, where $\rho = \frac{529}{54 \cdot \sqrt[3]{3}} < 6.8$.*

First, "project" the whole graph (the shortest path tree and the "noise" false positive edges) onto a line (multi)graph L where node l_i corresponds to all nodes in the real network at depth i away from dst, as shown in Figure 4. Let's say l_0, corresponding to the destination, is on the "right" while l_x, where x is the equivalent of "diameter" (the length of the longest shortest path to d), is on the left.

l_i has 1 edge pointing to the right, and k_i edges pointing to the left, where k_i is the max, over all nodes v at depth i in the original graph, of the number of false positives at v that point to nodes of depth i or $i+1$.

Lemma B.2. *The expected stretch of an unbiased random walk on B starting at v is at most the expected stretch of an unbiased random walk on L starting at $l_{d(v)}$.*

Proof. The lemma follows from a conventional coupling argument, between the random walk's depth in B and the random walk's depth in L.

Consider the next hop from some node v in B, at depth d, which has $i \geq 1$ edges to nodes of depth $d-1$, j edges to nodes of depth d, and k edges to nodes of depth $d+1$. By construction, the corresponding node l_d has at least $k+j$ edges to l_{d+1} and one edge to l_{d-1}. The probability of the walk moving to depth $d-1$ in L at most $1/(k+j+1)$, while the walk in B moves to depth $d-1$ with probability $i/(i+k+j) \geq 1/(k+j+1)$. This allows a natural coupling for depth of the next hop in B and in L:

If the L walk takes a deeper next hop than the B walk, we can "suspend" the B walk, and let the L walk continue until the next time it reaches a node

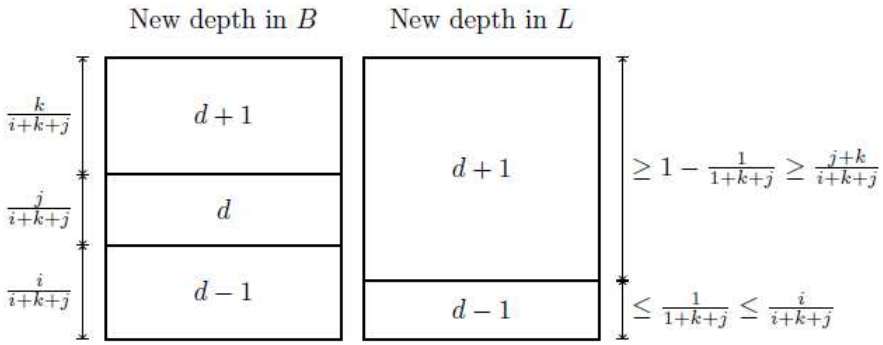


Figure 12: Coupling the depth process in B and L

at the same depth as the B walk — this is guaranteed to happen, since depth never changes by more than 1 hop. After that point, proceed with the next hop on both random walks. Since L and B random walks start at the same depth, this coupling guarantees that L will never move to a lower depth than B, and thus that L will take at least as many hops as B. □

We now analyze the worst-case expected hitting time for an unweighted random walk on L. For any $i > j$, the hitting time from l_i to l_0 must include at least one visit to l_j, guaranteeing that the expected hitting time to l_0 is maximized at l_x. The transition probability matrix (where $P_{i,j}$ is the probability, when at l_i, of going to l_j on the next step) is:

$$P = \begin{pmatrix} 0 & 1 & 0 & \cdots & & & 0 \\ \frac{k_{x-1}}{k_{x-1}+1} & 0 & \frac{1}{k_{x-1}+1} & 0 & \cdots & & 0 \\ 0 & \frac{k_{x-2}}{k_{x-2}+1} & 0 & \frac{1}{k_{x-2}+1} & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & \cdots & 0 & \frac{k_2}{k_2+1} & 0 & \frac{1}{k_2+1} & 0 \\ 0 & & \cdots & 0 & \frac{k_1}{k_1+1} & 0 & \frac{1}{k_1+1} \\ 0 & & & \cdots & 0 & 0 & 1 \end{pmatrix} \tag{5}$$

Following the expected hitting time analysis detailed in, e.g., [31], which cites [32], we let Q be the matrix P with the last row and last column removed. The expected hitting time from v_x to v_0 is then:

$$T = \sum_j (I - Q)^{-1}_{0,j} \tag{6}$$

Inversion of $(I-Q)$ via Gaussian elimination on the augmented matrix yields:

$$
\begin{array}{r}
R_x: \\ R_{x-1}: \\ R_{x-2}: \\ \vdots \\ R_2: \\ R_1:
\end{array}
\left(\begin{array}{ccccccc|c}
1 & -1 & 0 & \cdots & & 0 & 1 \\
\frac{-k_{x-1}}{k_{x-1}+1} & 1 & \frac{-1}{k_{x-1}+1} & 0 & \cdots & 0 & & 1 \\
0 & \frac{-k_{x-2}}{k_{x-2}+1} & 1 & \frac{-1}{k_{x-2}+1} & \cdots & 0 & & & 1 \\
\vdots & & \ddots & \ddots & \ddots & \vdots & & & & \ddots \\
0 & \cdots & 0 & \frac{-k_2}{k_2+1} & 1 & \frac{-1}{k_2+1} & & & & & 1 \\
0 & & \cdots & 0 & \frac{-k_1}{k_1+1} & 1 & & & & & & 1
\end{array}\right)
\begin{array}{l}
\leftarrow R'_{x-1} = (k_{x-1}+1)R_{x-1} + k_{x-1}R'_x \\
\leftarrow R'_{x-2} = (k_{x-2}+1)R_{x-2} + k_{x-2}R'_{x-1} \\
\vdots \\
\leftarrow R'_2 = (k_2+1)R_2 + k_2 R'_3 \\
\leftarrow R'_1 = (k_1+1)R_1 + k_1 R'_2
\end{array}
$$

$$
\begin{array}{r}
R''_x: \\ R'_{x-1}: \\ R'_{x-2}: \\ \vdots \\ R'_2: \\ R'_1:
\end{array}
\left(\begin{array}{ccccccc|ccccccc}
1 & -1 & 0 & \cdots & & 0 & 1 & & 0 & \cdots & & & \\
0 & 1 & -1 & 0 & \cdots & 0 & k_{x-1} & & (k_{x-1}+1) & 0 & \cdots & & \\
0 & 0 & 1 & -1 & \cdots & 0 & k_{x-1}k_{x-2} & & (k_{x-1}+1)k_{x-2} & (k_{x-2}+1) & 0 & \cdots & \\
\vdots & & \ddots & \ddots & \ddots & \vdots & & & & & & \ddots & \\
0 & \cdots & 0 & 0 & 1 & -1 & k_{x-1}k_{x-2}\cdots k_2 & & (k_{x-1}+1)k_{x-2}\cdots k_2 & \cdots & & (k_2+1) & 0 \\
0 & & \cdots & 0 & 0 & 1 & k_{x-1}k_{x-2}\cdots k_2 k_1 & & (k_{x-1}+1)k_{x-2}\cdots k_2 k_1 & \cdots & & (k_2+1)k_1 & (k_1+1)
\end{array}\right)
\begin{array}{l}
\leftarrow R''_x = \sum_i R'_i
\end{array}
\tag{7}
$$

The expecting hitting time, the sum of the top row of the inverse matrix, is thus the sum of all the entries on the right side. We split the sum of each column c except the first into $\sum_{i=1}^{x-c+1} \prod_{j=i}^{x-c+1} k_j + \sum_{i=1}^{x-c} \prod_{j=i}^{x-c} k_j + 1$, and then group the first term with the previous column, $c-1$. This yields:

$$
\begin{aligned}
T &= x + 2\sum_{c=1}^{x} \sum_{i=1}^{x-c} \prod_{j=i}^{x-c} k_j && (8) \\
&= x + 2 \sum_{1 \le a \le b \le x} \prod_{i=a}^{b} k_i && (9)
\end{aligned}
$$

Since, with no false positives, a packet may take up to x hops to get to d, the stretch S is thus upper bounded by $S \le T - x = 2\sum_{1 \le a \le b \le x} \prod_{i=a}^{b} k_i$.

We'll use the symbol S_p^q for $\sum_{p \le a \le b \le q} \prod_{i=a}^{b} k_i$, i.e. the sum of products over each possible substring of integer sequence $\{k_i\}$.

The S_p^q notation, as well as the L^q and R_p notations introduced below, leave implicit their dependence on the sequence $\{k_i\}$. When discussing proposed changes to the sequence, these symbols are to always be interpreted in terms of the sequence *before* the change in question.

B.1 Exact patterns for maximizing S_1^x

The remainder of the proof of Theorem B.1 is a detailed combinatorial treatment of maximizing S_1^x over all sequences whose sum, i.e. the total number of false positives $\sum_i k_i$, is fixed at some k. We proceed by listing a series of transformations of any candidate maximizing sequence that never change the sum, never decrease the sum of substring products, and converge to a very specific family of maximum sequences that allows S to be explicitly computed.

We forewarn the reader that this subsection is relatively technical and is unlikely to be of broader interest outside this proof.

In all the statements below, we implicitly require that $\sum_i k_i = k$.

Observation B.3. *To upper-bound stretch, we can assume, WLOG, that $k_i > 0$ for all i. Otherwise, removing that k_i and thus shrinking x by 1 cannot decrease S: the walk before the edge is removed can never return to the part of L above*

this edge, so contracting the edge only adds possible extra upward traversals, which would still have to return to this node before continuing to dst.

Lemma B.4. *There is a sequence* $\{k_i\}$ *maximizing* S_1^x *that has* $k_i \leq 3$ *for all* i.

Proof. Suppose there is a $k_i \geq 4$ in some sequence $\{k_i\}$ maximizing S_1^x. Then, split k_i into two adjacent sequence elements, 2 and $k_i - 2$. Since $2(k_i - 2) \geq k_i$, each substring of the original sequence that contained k_i will have a corresponding substring including both 2 and $k_i - 2$ in the new sequence, with a product at least as big as the original substring. Any substring that didn't include k_i originally will still exist as-is. Also, the new sum of substring products will also include the substring containing just the new 2 element, which does not correspond to any of the original substrings, thus guaranteeing a strictly greater substring product sum. □

Lemma B.5. *There is a sequence* $\{k_i\}$ *that maximizes* S *that has* $1 \leq k_i \leq 3$ *and is* dip-free*: if* $a < b < c$*, and* $k_a > k_b$*, then* $k_b \geq k_c$*; and if* $a < b < c$ *and* $k_b < k_c$*, then* $k_a \leq k_b$*. That is,* $\{k_i\}$ *matches the regular expression* $1^*2^*3^*2^*1^*$.

Proof. Consider a sequence $\{k_i\}$ maximizing S. We will see that it is dip-free by considering transpositions of two adjacent sequence elements.

Let L^p denote $\sum_{1 \leq a \leq p} \prod_{i=a}^{p} k_i$, the partial sum of substring products covering all substrings ending exactly at p. Similarly, let R_q denote $\sum_{q \leq b \leq x} \prod_{i=q}^{b} k_i$, the partial sum of substring products covering all substrings starting exactly at q. We define $\mathsf{L}^0 = \mathsf{R}_{x+1} = 0$.

Note that, since $k_i \geq 1$, $\mathsf{L}^p = k_p \mathsf{L}^{p-1} + k_p$ monotonically strictly increases with p, and $\mathsf{R}_q = k_q \mathsf{R}_{q+1} + k_q$ monotonically strictly decreases with q.

For any $1 \leq i \leq x - 1$, we can now split the full sum of S_1^x into several categories, based on where the substring lies in relation to k_i and k_{i+1}:

$$\begin{aligned}\mathsf{S}_1^x =& \mathsf{S}_1^{i-1} + \mathsf{S}_{i+2}^x + k_i k_{i+1} + k_i + k_{i+1} + k_i k_{i+1} \left(\mathsf{L}^{i-1} + \mathsf{R}_{i+2} + \mathsf{L}^{i-1}\mathsf{R}_{i+2}\right) \\ &+ k_i \mathsf{L}^{i-1} + k_{i+1}\mathsf{R}_{i+2}\end{aligned}$$

All *except* the last two of the terms add up to an expression that is symmetric with respect to transposing k_i and k_{i+1}. Now, suppose $k_i < k_{i+1}$. Then, since the sequence is maximizing, its S_1^x is greater than it would be if k_i and k_{i+1} were transposed, requiring:

$$\begin{aligned} k_i \mathsf{L}^{i-1} + k_{i+1}\mathsf{R}_{i+2} &\geq k_{i+1}\mathsf{L}^{i-1} + k_i \mathsf{R}_{i+2} \\ (k_{i+1} - k_i)(\mathsf{R}_{i+2} - \mathsf{L}^{i-1}) &\geq 0 \\ \mathsf{R}_{i+2} - \mathsf{L}^{i-1} &\geq 0 \end{aligned}$$

Then, for any $i' < i$, strict monotonicity gives us $\mathsf{R}_{i'+2} - \mathsf{L}^{i'-1} > 0$. If $k_{i'} > k_{i'+1}$, we would have $(k_{i'+1} - k_{i'}) \left(\mathsf{R}_{i'+2} - \mathsf{L}^{i'}\right) < 0$, contradicting maximality

of $\{k_i\}$, since that would allow us to raise S_1^x by transposing $k_{i'}$ and $k_{i'+1}$. Inductively applying this argument shows that, if $a > b > c$ and $k_b < k_c$, then $k_a \leq k_b$. The other side follows by symmetry. □

Lemma B.6. *There exists a maximizing sequence obeying the constraints of Lemma B.5 that starts and ends with a 1, i.e., it matches the regular expression* $11^*2^*3^*2^*1^*1$.

Proof. If $k_1 > 1$ in a maximizing sequence, consider splitting it into adjacent elements 1 and $k_1 - 1$. This transforms the original sum of substring products from $k_1 + k_1\mathsf{R}_2 + \mathsf{S}_2^x$ into $1 + k_1 - 1 + 1(k_1 - 1) + (k_1 - 1)\mathsf{R}_2 + 1(k_1 - 1)\mathsf{R}_2 + \mathsf{S}_2^x = 1 + 2(k_1 - 1) + 2(k_1 - 1)\mathsf{R}_2 + \mathsf{S}_2^x$. Since $k_1 \leq 2(k_1 - 1)$, this increases the total sum of substring products by 1. The same argument holds for k_x. □

Observation B.7. *An adjacent* $(1, 3)$ *pair can always be replaced with a* $(2, 2)$ *pair, strictly increasing* S_1^x. *It changes from* $\mathsf{S}_1^{i-2} + \mathsf{S}_{i+1}^x + (1 + 3)\mathsf{L}^{i-2} + (3 + 3)\mathsf{R}_{i+1} + 3\mathsf{L}^{i-2}\mathsf{R}_{i+1} + 1 + 3 + 3$ *to* $\mathsf{S}_1^{i-2} + \mathsf{S}_{i+1}^x + (2 + 4)\mathsf{L}^{i-2} + (2 + 4)\mathsf{R}_{i+1} + 4\mathsf{L}^{i-2}\mathsf{R}_{i+1} + 2 + 2 + 4$, *an increase by* $2\mathsf{L}^{i-2} + \mathsf{L}^{i-2}\mathsf{R}_{i+1} + 1$, *which is always positive.*

Lemma B.8. *For* $k \geq 7$, *there is an maximizing sequence satisfying B.1 that has at most two 1's on each side.*

Proof. Consider a sequence satisfying B.1 with at least three 1's at the start. If the first non-1 element is some $k_i = 3$, Observation allows us to replace it and the preceding 1 with (2,2).

If the first non-1 element is some $k_i = 2$, consider replacing $(1, 2)$ with a 3. S_1^x changes from $\mathsf{S}_1^{i-2} + \mathsf{S}_{i+1}^x + (1+2)\mathsf{L}^{i-2} + (2+2)\mathsf{R}_{i+1} + 2\mathsf{L}^{i-2}\mathsf{R}_{i+1} + 1 + 2 + 2$ to $\mathsf{S}_1^{i-2} + \mathsf{S}_{i+1}^x + 3\mathsf{L}^{i-2} + 3\mathsf{R}_{i+1} + 3\mathsf{L}^{i-2}\mathsf{R}_{i+1} + 3$. The increment is $\mathsf{L}^{i-2}\mathsf{R}_{i+1} - \mathsf{R}_{i+1} - 2$, non-negative if $(\mathsf{L}^{i-2} - 1)\mathsf{R}_{i+1} \geq 2$.

For $k \geq 7$, this is satisfied, since either:

1. There are four or more ones before k_i. Then, $\mathsf{L}^{i-2} - 1 \geq 2$. By B.1, there is at least one element after k_i, guaranteeing $\mathsf{R}_{i+1} \geq 1$;
2. Or $\sum_{j=i+1}^{x} k_j \geq 2$, guaranteeing $\mathsf{R}_{i+1} \geq 2$. Since the sequence starts with three or more ones, $\mathsf{L}^{i-2} - 1 \geq 1$.

A symmetrical argument applies to the other end of the sequence. □

Lemma B.9. *For* $k \geq 18$, *any maximizing sequence satisfying B.8 has at least one 3.*

Proof. With $k \geq 18$ no 3's, and at most two 1's at each end, the sequence must then have at least seven 2's. Let k_{i-2} be the first 2. We consider replacing the third, fourth, and fifth 2's (k_i, k_{i+1}, and k_{i+2})) with two 3's.

Before the change, S_1^x is $\mathsf{S}_1^{i-1} + \mathsf{S}_{i+3}^x + (2 + 4 + 8)\mathsf{L}^{i-1} + (2 + 4 + 8)\mathsf{R}_{i+3} + 8\mathsf{L}^{i-1}\mathsf{R}_{i+3} + 2 + 2 + 2 + 4 + 4 + 8 = \mathsf{S}_1^{i-1} + \mathsf{S}_{i+3}^x + 14(\mathsf{L}^{i-1} + \mathsf{R}_{i+3}) + 8\mathsf{L}^{i-1}\mathsf{R}_{i+3} + 22$.

After the change, it becomes $\mathsf{S}_1^{i-1}+\mathsf{S}_{i+3}^x+(3+9)\mathsf{L}^{i-1}+(3+9)\mathsf{R}_{i+3}+9\mathsf{L}^{i-1}\mathsf{R}_{i+3}+3+3+9=\mathsf{S}_1^{i-1}+\mathsf{S}_{i+3}^x+12(\mathsf{L}^{i-1}+\mathsf{R}_{i+3})+9\mathsf{L}^{i-1}\mathsf{R}_{i+3}+15$. This is an increase of $\mathsf{L}^{i-1}\mathsf{R}_{i+3}-2(\mathsf{L}^{i-1}+\mathsf{R}_{i+3})-7=(\mathsf{L}^{i-1}-2)(\mathsf{R}_{i+3}-2)-11$. With $k_{i-2}=k_{i-1}=k_{i+3}=k_{i+4}=2$, we are guaranteed that both L^{i-1} and R_{i+3} are at least 6, making the increase positive. □

Lemma B.10. *For $k \geq 18$, any maximizing sequence satisfying B.9 has exactly one 1 on each side.*

Proof. Suppose, WLOG, there are exactly two 1's in the beginning (Lemmas B.1 and B.8 allow us to reach such a maximizing sequence from any other). Use b for the number of 2's between the last 1 and the first 3 ($b = i-3$). Consider replacing $k_2 = 1$ and the first 3, $k_i = 3$ (which Lemma B.9 guarantees to exist) with $k_2 = k_i = 2$. The increment in S_1^x is:

$$\begin{aligned}
&\left(\mathsf{S}_{i+1}^x+\mathsf{S}_3^{i-1}+\mathsf{R}_{i+1}\left(1\cdot 2\cdot 2^b\cdot 2+2\cdot 2^b\cdot 2+2\sum_{j=0}^{b}2^j\right)\right.\\
&\left.+2\cdot\left(\sum_{j=0}^{b}2^j\right)+2\cdot 2^b\cdot 2+2\cdot 2^b\cdot 2\cdot 1+2\cdot\left(\sum_{j=0}^{b}2^j\right)+1\cdot 2\cdot\left(\sum_{j=0}^{b}2^j\right)+1\right)-\\
&\left(\mathsf{S}_{i+1}^x+\mathsf{S}_3^{i-1}+\mathsf{R}_{i+1}\left(1\cdot 1\cdot 2^b\cdot 3+1\cdot 2^b\cdot 3+3\sum_{j=0}^{b}2^j\right)\right.\\
&\left.+3\cdot\left(\sum_{j=0}^{b}2^j\right)+3\cdot 2^b\cdot 1+3\cdot 2^b\cdot 1\cdot 1+1\cdot\left(\sum_{j=0}^{b}2^j\right)+1\cdot 1\cdot\left(\sum_{j=0}^{b}2^j\right)+1\right)\\
&=\mathsf{R}_{i+1}\left((12\cdot 2^b-2)-(12\cdot 2^b-3)\right)+\left((20\cdot 2^b-5)-(16\cdot 2^b-4)\right)\\
&=\mathsf{R}_{i+1}+2^{b+2}-1 \quad (10)
\end{aligned}$$

Since $b \geq 1$, the increment is strictly positive, violating maximality. A symmetric argument applies for two 1's at the end. □

Lemma B.11. *For $k \geq 18$, any maximizing sequence satisfying B.10 has at most three 2's on each side of the 3's.*

Proof. Consider a maximizing sequence with at least four 2's before the 3's. First, apply to ensure that we're considering a maximizing sequence starting and ending with a 1.

By Observation B.1, in a maximizing sequence, there must be at least one 2 after the 3's — else the total is strictly increased by replacing a $(3,1)$ pair with $(2,2)$.

Like in Lemma B.9 above, we consider replacing the last three 2's before the 3's with two new 3's, which will increase the total if $(\mathsf{L}^{i-1}-2)(\mathsf{R}_{i+3}-2)>11$. But the existence of at least one 3, at least one 2, and a 1 after the triple we're

replacing guarantees that $\mathsf{R}_{i+3} - 2 \geq 3 + 6 + 6 - 2 = 13$. Since there must be at least one extra 2 before the triple, and at least one 1, we have $\mathsf{L}^{i-1} \geq 4$, guaranteeing the strict inequality. □

We are now guaranteed, for $k \geq 18$, the existence of a maximizing sequence of form $12^b3^c2^d1$ where $1 \leq b, d \leq 3$ and $c \geq 1$. We can now obtain a convenient closed form for S_1^x for such sequences. First note that the substring products of a homogeneous sequence z^n add up to:

$$\begin{aligned}
m(z,n) &= \sum_{1 \leq a \leq b \leq n} \prod_{i=a}^{b} k_i \\
&= z \sum_{a=1}^{n} \sum_{b=a}^{n} z^{b-a} \\
&= z \sum_{a=1}^{n} \frac{z^{(n-a+1)} - 1}{z - 1} \\
&= \frac{z}{z-1} \sum_{i=1}^{n} (z^i - 1) \\
&= \frac{z}{z-1} \cdot \left(\frac{z^{n+1} - 1}{z - 1} - 1 - n \right) \\
&= \frac{z^{n+2} - (n+1)z^2 + nz}{(z-1)^2}
\end{aligned}$$

Using this, we can split up the computation of S_1^x based on which part of the sequence the substring starts and ends in:

$$
\begin{aligned}
\mathsf{S}_1^x &= 1 + m(2,b) + m(3,c) + m(2,d) + 1 \\
&\quad + 1 \cdot \sum_{i=1}^{b} 2^i + 1 \cdot 2^b \cdot \sum_{i=1}^{c} 3^i + 1 \cdot 2^b \cdot 3^c \cdot \sum_{i=1}^{d} 2^i + 1 \cdot 2^b \cdot 3^c \cdot 2^d \cdot 1 \\
&\quad + \left(\sum_{i=1}^{b} 2^i\right)\left(\sum_{i=1}^{c} 3^i\right) + \left(\sum_{i=1}^{b} 2^i\right) \cdot 3^c \cdot \left(\sum_{i=1}^{d} 2^i\right) + \left(\sum_{i=1}^{b} 2^i\right) \cdot 3^c \cdot 2^d \cdot 1 \\
&\quad + \left(\sum_{i=1}^{c} 3^i\right)\left(\sum_{i=1}^{d} 2^i\right) + \left(\sum_{i=1}^{c} 3^i\right) \cdot 2^d \cdot 1 \\
&\quad + \left(\sum_{i=1}^{d} 2^i\right) \cdot 1 \\
&= 4 \cdot 2^b - 2b - 4 + 4 \cdot 2^d - 2d - 4 + \frac{9}{4}3^c - \frac{9}{4}c - \frac{9}{4} + \frac{3}{4}c \\
&\quad + (2 - 2 + 3) + (2 - \frac{3}{2} - 3)2^b + (2 - \frac{3}{2} - 3)2^d + (-3 + 4 - 3)3^c \\
&\quad + (\frac{3}{2} - 2 + 3 - 4)2^b 3^c + (\frac{3}{2} - 2 + 3 - 4)3^c 2^d + (2 + 1 + 4 + 2)2^b 3^c 2^d \\
&= 9 \cdot 2^b 3^c 2^d - 1.5 \cdot 3^c(2^b + 2^d) + 0.25 \cdot 3^c + 1.5(2^b + 2^d) - 2(b + d) - 1.5c - 6.25
\end{aligned} \tag{11}
$$

Lemma B.12. *For $k \geq 18$, any maximizing sequence satisfying B.11 cannot contain three consecutive 2's.*

Proof. Suppose, WLOG, $b = 3$. We consider two cases for d:

Suppose $d = 1$. Consider "balancing" the sequence, by setting $b = d = 2$, and leaving c fixed. In equation 11, this preserves the value of $b + d$ and $2^b 2^d$. Thus, the only change to S_1^x is in the terms involving $(2^b + 2^d)$, which goes from 9 to 8. Since $c \geq 1$ by Lemma B.9, this guarantees a strict increase in S_1^x, by $1.5 \cdot 3^c - 1.5 \geq 3$.

Suppose $2 \leq d \leq 3$. Consider replacing k_3 and k_4, the second and third 2 at the beginning, and k_{x-2}, the first 2 at the end, with two 3's. This changes b from 3 to 1, d to $d - 1$, and c to $c + 2$. Thus, S_1^x is incremented by:

$$
\begin{aligned}
9 \cdot 3^c(2 \cdot 2^{d-1} \cdot 9 - 8 \cdot 2^d) - 1.5 \cdot 3^c(9(2 + 2^{d-1}) - (8 + 2^d)) + 0.25 \cdot 3^c(9 - 1) \\
+ 1.5(2 + 2^{d-1} - 8 - 2^d) - 2(1 + (d - 1) - (3 + d)) - 1.5 \cdot 2 = \\
= 7.5 \cdot 3^c 2^{d-1} - 13 \cdot 3^c - 1.5 \cdot 2^{d-1} - 6 \\
\geq 15 \cdot 3^c - 13 \cdot 3^c - 1.5 \cdot 4 - 6 \\
= 2 \cdot 3^c - 12
\end{aligned}
$$

Since there are at most three 2's and exactly one 1 on each side of the sequence, $k \geq 18$ requires $c \geq 2$, which renders the increment strictly positive. □

We now know that $1 \le b, d \le 2$ in a maximizing sequence. But, at this point, if $k \bmod 3 = 0$, this only allows $b = d = 1$; if $k \bmod 3 = 1$, this only allows $b = d = 2$; and if $k \bmod 3 = 2$ this only allows $b = 2$ and $d = 1$ or vice versa — and since the formula for S_1^x is symmetric with respect to reversing the sequence, in all three cases, this specifies the exact sequence maximizing S_1^x, and hence the expected stretch.

For $k \ge 18$, Theorem B.1 follows from substituting each of the three possibilities into equation 11:

$$
\begin{aligned}
S(k) &\le 2 \max \begin{cases} (9 \cdot 4 - 6 + 0.25) 3^{(k-6)/3} - 1.5\frac{k-6}{3} + 6 - 4 - 6.25 & (k \bmod 3 = 0) \\ (9 \cdot 16 - 12 + 0.25) 3^{(k-10)/3} - 1.5\frac{k-10}{3} + 12 - 8 - 6.25 & (k \bmod 3 = 1) \\ (9 \cdot 8 - 9 + 0.25) 3^{(k-8)/3} - 1.5\frac{k-8}{3} + 9 - 6 - 6.25 & (k \bmod 3 = 2) \end{cases} \\
&< 2 \cdot 3^{k/3} \max \begin{cases} 121/36 & (k \bmod 3 = 0) \\ 529/(108\sqrt[3]{3}) & (k \bmod 3 = 1) \\ 253/(36\sqrt[3]{9}) & (k \bmod 3 = 2) \end{cases} \\
&= \frac{529}{54\sqrt[3]{3}} \cdot 3^{k/3}
\end{aligned}
$$

For $1 \le k \le 17$, brute force search among sequences adding up to k and obeying Lemma B.5 shows that the sequences in Table 2 maximize stretch, which obeys the bound of Theorem B.1, as well, concluding the proof of the latter.

B.2 Tight lower bound

Observation B.13. *Assuming that there can be multiple (a, b) edges from a particular node a to a particular node b, the line graph L* itself *is then a valid "tree and false positives" configuration, which means the sequence defined by Lemma B.12 corresponds to a concrete, valid example that makes the above bound exactly tight for $k \bmod 3 = 1$.*

Observation B.14. *If multiple edges between the same pair of nodes are not allowed, we expect that the worst-case stretch will be somewhat better, but can still be exponential under worst-case false positive placements. An example of such an arrangement is in Figure 13.*

Appendix C Tail bound on stretch

Theorem C.1 (Theorem 4 in Section 4)**.** *For any $z \ge 60.3 \cdot 7.3221^F$, the probability of stretch exceeding z is bounded by $2/z$. Asymptotically, the tail will decay as $\tilde{O}(1/z^{1.54})$, to within polylog factors.*

k	$\max S(k)$	$\left\lfloor \frac{529}{54\sqrt[3]{3}} 3^{k/3} \right\rfloor$	Maximum stretch at:
1	2	9	1
2	6	14	1 1
3	12	20	1 1 1
4	20	29	1 1 1 1
5	32	42	1 1 2 1
6	52	61	1 2 2 1
7	76	88	1 1 2 2 1
8	120	127	1 2 2 2 1
9	170	183	1 2 3 2 1
10	260	264	1 2 2 2 2 1
11	370	381	1 2 2 3 2 1
12	544	550	1 2 2 2 2 2 1
13	786	793	1 2 2 3 2 2 1
14	1126	1144	1 2 2 3 3 2 1
15	1622	1650	1 2 2 2 3 2 2 1
16	2370	2380	1 2 2 3 3 2 2 1
17	3400	3433	1 2 2 3 3 3 2 1

Table 2: Optimal sequences for $1 \leq k \leq 17$, obtained by brute-force search.

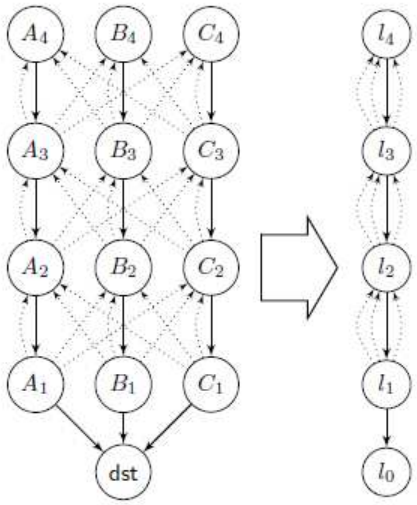


Figure 13: Exponential stretch with k false positives and no multiple edges: an $i+1$-row network like the one on the left has $9i$ false positives, with worst-case stretch equivalent to the line graph on the right: $S(k) = \Theta(3^i) = \Theta(3^{k/9})$

Proof. Assume the worst-case arrangement of false positive locations. Consider packets starting at the source with the worst expected stretch to the destination. With k false positives, by the Markov bound, the stretch will be at most $2S(k)$ with probability at least $1/2$. After every $2S(k)$ steps, any such packet not at the destination can be "reset" to the worst possible starting location without shortening its current expected arrival time. Thus, for any integer α, the probability of the stretch being more than $2\alpha S(k)$ is at most $1/2^{\alpha}$.

We can bound the overall stretch by setting k, with foresight, to the solution of $k3^{k/3} = \frac{z}{\rho(2-1/e)}$ (here and below, e is Euler's constant), which is a unique positive value, by monotonicity[7].

This value of k will allow us to productively apply the following union bound:

$$\Pr[\text{stretch} > z] \leq \Pr[\text{stretch} > z | \leq k \text{ false pos}] + \tag{12}$$

$$+ \Pr[> k \text{ false pos}] \tag{13}$$

The above iterated Markov bound covers the first term, which is then bounded by $1/2^{z/2S(k)} = 1/2^{k(1-1/2e)}$.

For the second term, let $F = \sum f(h)$ be the expected number of false positives anywhere in the system for any fixed destination, i.e. the sum of the probabilities of false positives along each possible edge. If $k > 2eF$ (where e is Euler's constant), we can use the fact that the hashes for each Bloom filter are chosen independently and apply the following form of the Chernoff bound [29]: $\Pr[X > (1+\delta)\,\mathrm{E}[X]] < 2^{-\delta\,\mathrm{E}[X]}$, for any $\delta > 2e-1$. With X as the random variable counting the false positives for our destination, we set $\delta = k/F - 1$ yielding $\Pr[X > k] < 2^{-(k/F-1)F} = 2^{-k+F} < 1/2^{k(1-1/2e)}$, bounding the overall probability by $2/(2^{1-1/2e})^k \leq 2/1.76^k$.

To satisfy the $k > 2eF$ requirement of the Chernoff bound, we need $z = (2-1/e)\rho k 3^{k/3} \geq 60.3 \cdot 7.3221^F$. The same constraint, since $F > 0$, guarantees $z \geq 29$, ensuring that $2/1.76^k \leq 2/((2-1/e)\rho k 3^{k/3}) = 2/z$.

With $z = 3^{k/3+O(log(k))}$, the tail asymptotically decays as $\tilde{O}(1/z^{\log_{\sqrt[3]{3}} 1.76})$, to within polylog factors, yielding the second part of the theorem. □

Appendix D Forwarding on an underlying tree network

Observation D.1 (Claim 5 in Section 4)**.** *If the underlying network is a tree, with no multiple links between any one pair of routers, the expected stretch with k false positives, even if they are adversarially placed, is at most $2(k-1)^2$.*

Proof. Consider any configuration of the shortest path tree and the extra false positive edges. Since there are no multiedges, each edge of the shortest path tree has 1 or 0 antiparallel false positive edges. Contracting all the tree edges without a corresponding false positive edge will not decrease the expected stretch, by an

[7] We omit the details of dealing with rounding and integrality.

argument similar to Observation B.3: Any such edge renders the subtree behind it unreachable after it's traversed. Shrinking this edge only adds the possibility of an extra detour back into the subtree, which would have to pass through this node again before proceeding — at that point, the expected remaining time until destination will be the same as when this node was first visited in the original graph.

The resulting graph is effectively undirected, with each edge having a corresponding antiparallel edge. A random walk is thus identical to a random walk on an undirected tree, shown in, e.g., Sec. 5.3 of [28] to have the expected hitting time of at most $2(k-1)^2$. □