A Dynamic Lookup Scheme for Bursty Access Patterns

Funda Ergun* Suvo Mittra† Süleyman Cenk Şahinalp* Jonathan Sharp* Rakesh K. Sinha†

Abstract—The problem of fast address lookup is crucial to routing and thus has received considerable attention. Most of the work in this field has focused on improving the speed of individual accesses – independent from the underlying access pattern. Recently, Gupta *et al.* [7] proposed an efficient data structure to exploit the bias in access pattern. This technique achieves faster lookups for more frequently accessed keys while bounding the worst case lookup time; in fact it is (near) optimal under constraints on worst case performance. However, it needs to be rebuilt periodically to reflect the changes in access patterns, which can be inefficient for bursty environments.

In this paper we introduce a new *dynamic* data structure to exploit biases in the access pattern which tend to change dynamically. Recent work shows that there are many circumstances under which access patterns change quickly [11], [12]. Our data structure, which we call the *biased skip list* (BSL), has a self-update mechanism which reflects the changes in the access patterns efficiently and immediately, without any need for rebuilding. It improves throughput while keeping the worst case access time bounded by that of the fastest (unbiased) schemes. We demonstrate the practicality of BSL by experiments on data with varying degrees of burstiness.

I. Introduction

A fundamental requirement for a router is to decide, upon the arrival of each packet, on which outgoing line to forward the packet, i.e., to compute the next hop. Given the exponential growth of the Internet, Internet traffic, and the increase in link speeds, it has become crucial that this decision be made at extremely high speeds. In traditional IP this decision is based on the destination address of the packet received. In this paper, we mainly focus on the IP lookup problem, where, given a set of prefixes, the task of a router is to quickly find the longest prefix that matches the destination address of an incoming packet and forward the packet along the outgoing link associated with that prefix. The destination addresses are 32 bits long in the current IPv4 standard and will include 128 bits in the next generation IPv6. Figure I shows an example set of prefixes and the corresponding next hops. In this example, the destination address 0010 matches the first three prefixes, but the longest matching prefix is "001∗". 1010, on the other hand, only matches "∗".

Prefix	Next hop
*	a
0*	b
001*	c
100*	d

Fig. 1. Sample table of prefixes

The most commonly used data structure for the IP lookup problem is the binary trie, a tree where every edge has a bit label and each prefix is represented by a path from the root to a leaf. The internal nodes in a trie can have a single child or two children. The search time in a trie is proportional to the number of bits in the address space and a trie does not distinguish between high frequency (commonly occurring) and low frequency (rarely occurring) prefixes; this may result in poor performance especially for 128 bit IPv6 addresses.

Many data structures have been proposed for improving the performance of tries ([2], [6], [9], [13], [16]. The majority of this work focuses on intelligent schemes to minimize expected or worst case lookup times for individual accesses, independent from the underlying access pattern. It is conceivable, however, that one can improve the throughput of a data structure by tuning it according to lookup biases.

An interesting work by Cheung and McCanne [3] exploits the bias in access with a trie-based approach under space limitations. More recently, Gupta *et al.* [7] proposed an efficient data structure to exploit the bias under worst case performance constraints.

*Dept. of EECS, Case Western Reserve University; email: {afe, cenk, jps17}@eecs.cwru.edu

† Bell Labs, Murray Hill; email:{mittra, rks1}@research.bell-labs.com

The authors are listed alphabetically.

The main features of this method are, (i) the prefixes are treated as intervals (ranges) in the address space $[0 : 2^{32}]$; the number of ranges obtained is shown to be at most twice the number of prefixes, (ii) access counts are maintained for ranges over long periods, (iii) a binary search tree is constructed on the range set. To guarantee worst case performance, the depth of the tree is bounded by a user specified parameter. The tree is then optimized according to access counts under this maximum depth constraint. While this data structure achieves (near) optimal lookup time for a given set of data, it assumes relatively stable access patterns and prefixes; i.e. the addresses and access probabilities do not change much over time. In the event of a change in the form of an addition/removal of an address, or the change of an access probability, the data structure cannot be easily modified (this is the case in [3] as well). Instead, it must be rebuilt from scratch in $O(n \log n)$ time where n is the number of items. (It is suggested that one can wait for a number of such changes to accumulate before reconstructing the tree.) Similar search times are achieved by [10] at the cost of even higher reconstruction time.

Contributions. In this paper we present a new data structure which we call a *Biased Skip List*, or BSL. BSL exploits the access probabilities and achieves a search time similar to the above scheme. In addition, it allows fast updates, and therefore is suitable for highly dynamic environments where the access patterns are bursty and the addresses tend to change often. It has been observed in many circumstances that access patterns do show a highly dynamic character, i.e. the "working set" is quite small and is subject to change, and the access probabilities are not static over long periods of time [11]. In such circumstances, BSL "adjusts itself" to the new environment through insertion and deletion of keys without resorting to reconstruction. The updates can be in the form of a change to an access probability, or the insertion/deletion of a prefix. As a result, any change in the router's data or address set can be reflected in the data structure as soon as it happens, rather than being accumulated.

BSL is based on the Skip List data structure [14]. Skip lists are simple, randomized data structures which do not require complex balancing operations. They are generally regarded as the best performing search data structure in practice. Skip lists support $O(\log n)$ time search, insert and delete operations for uniform access patterns. However, they do not distinguish between keys in terms of search and update times. BSL improves the Skip List by exploiting the bias in the access pattern. This is done by ranking the addresses according to the number of accesses and partitioning them into dynamic classes based on their current rank.

We consider two fundamentally important access patterns, and adapt BSL to each for obtaining efficient solutions.

1. *Independently skewed access patterns.* Certain items are accessed more than others but the access frequencies remain relatively stable over time. This is the type considered in [7], where an elegant solution is given. We present a variant of BSL which provides a solution for this case; this gives insight for our approach for bursty access patterns, which is our main focus.

2. *Bursty access patterns.* A few items get "hot" for short periods of time and are accessed very frequently. Such patterns have been observed by a number of studies ([11], [12]) in various applications. There are a number of data structures ([15], [1]) which are designed to exploit such biases, however they do not support efficient insertion and deletion operations within this context. Also, these data structures perform complex balancing schemes which hinder their practical performance. We present a variant of BSL for these patterns which employs a novel lazy maintenance scheme for fast and immediate reflection of updates to the access pattern or to the data structure. Our maintenance scheme should not be confused with those that do not reflect updates to the data structure immediately, but accumulate them to reduce the overhead.

Both variants of BSL require $O(n)$ space, and can be constructed in $O(n)$ time when keys are given in sorted order. The keys are ranked 1 through n according to how frequently or recently they have been accessed. Searching for a key k takes $O(\log r(k))$ time, where $r(k)$ denotes the rank of the key. The novel feature of BSL is that for bursty patterns, it supports insertions and deletions in $O(\log r_{max}(k))$ time, where $r_{max}(k)$ denotes the maximum rank of k during its lifespan. Because the maximum rank is n, all operations have a worst case bound of $O(\log n)$

Range	Next hop
$[0000 - 0001]$	b
$[0010 - 0011]$	c
$[0100 - 0111]$	b
$[1000 - 1001]$	d
$[1010 - 1111]$	a

Fig. 2. Sample table of ranges

as with skip lists and other efficient data structures – which is optimal.

Organization of the Paper. In the following section, we describe our problem more formally and explain how BSL works. In Section III we describe and analyze the cost of construction and search. In Section IV we investigate dynamic BSL. Section V presents some experimental results.

II. Preliminaries

The Setup. We are given a set of *rules* $R = \{R_1, \ldots, R_k\}$ which are in the form of bit strings corresponding to IP address prefixes, and and an action to be taken for a packet that "matches" a given rule. When multiple prefixes match an address (e.g., the destination address on the packet), the rule with the longest matching prefix is chosen. We represent the prefixes as non-overlapping intervals in the address space, as in [9] and [7]; the number of intervals required is at most twice the number of prefixes. In Figure II we show the translation of the prefixes from Figure I into non-overlapping intervals.

Thus, we represent our rule set as $n \leq 2k$ keys, each key consisting of the two endpoints delimiting the interval. We borrow the process for translating the prefixes to intervals to be inserted and deleted as necessary from the above papers and do not go into the details of this translation. Instead, we concentrate on operations on intervals.

Efficiency. Our main concern is throughput; i.e. we would like to minimize total lookup time over a sequence of lookups. In most applications certain prefixes are accessed much more frequently than others, giving a biased distribution. The distribution may either (i) change rapidly, as in the case of end routers to which connections are short lived but frequent (such as HTTP), or, (ii) may remain relatively stable as in the case of backbone routers. Our data structure, the Biased Skip List, adapts to maximize throughput once the type of traffic pattern is identified.

Regular Skip List in a Nutshell. A skip list is a search data structure for n ordered keys [14]. It allows linear time construction (if keys are given in sorted order) and logarithmic time search, insert and delete operations. To construct a skip list, we first make up level $\log n$, which is a sorted linked list of all of the keys. (for simplicity of notation, throughout the paper we write $\log n$ instead of $\lceil \log_2 n \rceil$.) Levels $(\log n) - 1, \ldots, 0$ are built randomly in a bottom-up fashion as follows. Each level i is a sorted linked list consisting of a subset of the keys in level $i + 1$ obtained as follows: each key in level $i + 1$ is copied to level i independently with probability $1/2$. Each key has vertical pointers to and from its copies (if they exist) on adjacent levels. Because there are $\log n$ levels, we expect to have one key in level 1. Note that since the construction is randomized, it only makes sense to talk about expected values rather than absolute numbers when talking about the number of keys.

To search for a key k, we start from the smallest (leftmost) key in the top level (level 0) . On each level, we go right until we encounter a key which is greater than k. We then take a step to the left and go down one step (which leaves us in a key less than k). The search ends as soon as k is found, or when, at the lowest level, a key greater than k is reached. To delete a key, we perform a search to find its highest occurrence in the structure and delete it from every level it appears. To insert a new key k, we first search for k in the data structure to locate the correct place to insert k in the bottom level. Once the key is inserted into the bottom level, a fair coin is tossed to decide whether to copy it to the level above. If k is indeed copied, the procedure is repeated iteratively for the next level, otherwise the insertion is complete.

III. Biased Skip List (BSL)

Some characteristics. With regular skip lists, since searches start at the top level, it takes less time to search for keys on appearing on higher levels than those appearing only on lower levels. Since the locations of keys in the structure are determined completely randomly, with non-uniform access pattern, some of the most frequently searched keys may end up at low levels and have long search times, making the throughput unnecessarily poor. Our data structure, BSL, maintains the good characteristics of the skip lists while remedying the above shortcoming by making sure that keys with smaller ranks (those frequently accessed) are located close to the top of the data structure, and thus can be quickly accessed.

Interval Search. Even though skip lists are designed to search for exact matches, our task is to locate an interval containing a given address. Each key contains two endpoints, representing an interval. Since the intervals are non-overlapping, we define an ordering between keys as follows. For keys i, j, we say $i < j$ if the right (second) endpoint of i is less than the left (first) endpoint of j, implying that all addresses within i are smaller than those within j. An address can be compared to a a key in a similar way as well; the address is said to be smaller than a key if it is less than the key's left endpoint, and greater if greater than its right endpoint. The address is said to *match* (be equal to) a key if it falls within the key's interval. We then make the following observation.

Observation 1: Given the above ordering between keys and between keys and addresses, it is possible to treat the keys in a BSL as integers and perform a range search as an exact search.

The Data Structure. There are n keys, each of which are assigned distinct ranks $[1 \ldots n]$ depending on how often or how recently they have been accessed. The rank of a key i is denoted by $r(i)$. The keys are partitioned into classes $C_1, C_2 \ldots, C_{\log n}$: if keys x and y are in classes C_k and C_{k+1} respectively, then $r(x) < r(y)$, and the keys in C_k are more "hot".

BSL has multiple levels; since searches start from the top levels, keys located higher than others are accessed faster. Since we would like the keys with smaller ranks (those in classes with small index) to be quickly accessible, we place those classes higher in our data structure. To ensure this, the bottom level contains all keys. As we go up to the higher levels, some of the keys belonging to higher-numbered classes are gradually left out (not promoted to the next level). The intuition is that as we go higher and higher, the remaining keys tend more and more to be from the more popular classes with the result that those keys can be searched faster. A schematic example of how classes containing keys of small rank preserve their initial size and how those containing keys of large rank get smaller in the upper levels is illustrated in Figure 3.

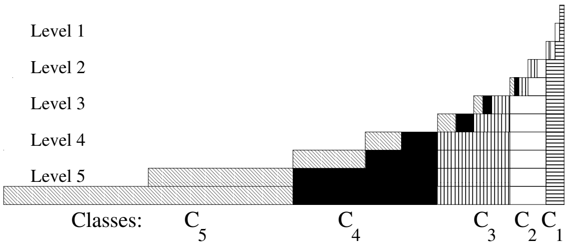

Fig. 3. Schematic display of the sizes of classes through levels of BSL.

We first describe the data structure when the ranks are static.

Construction. In addition to the main randomized data structure described below, BSL maintains a master list of all the keys in ascending rank order, [1] partitioned into classes $C_1, C_2 \ldots, C_{\log n}$. The class sizes are geometric, $|C_i| = 2^{i-1}$. (2 is not special, our analysis holds for $|C_i| = \alpha^{i-1}$ for any $\alpha > 1$.)

BSL is constructed in a randomized way; it contains a sorted doubly linked list of keys for each level. The levels are labeled, starting at the bottom, as $L_{\log n}, L'_{\log n-1}, L_{\log n-1}, \ldots, L_2, L'_1, L_1$, and are constructed in a bottom-up, randomized fashion as follows. The bottom level, $L_{\log n}$, includes all of the keys in the BSL. To obtain the keys on a level L'_i we copy from L_{i+1} to L'_i, (1) all $2^i - 1$ keys from classes $C_i, C_{i-1} \ldots, C_1$ (by definition all should be present in L_{i+1}), and (2) a subset of the remaining keys of L_{i+1} picked independently with probability $1/2$ each. (the expected number of these is 2^{i-1}.) We form L_i slightly differently by copying the following keys from L'_i: (1) all keys from classes $C_{i-1}, C_{i-2} \ldots, C_1$, and (2) a subset of the remaining keys in L'_i picked independently with probability $1/2$ each. An example BSL is presented in Figure 4.

When keys are given in sorted order, a BSL can be constructed in expected linear time.

Lemma 2: A BSL with n keys can be constructed in $O(n)$ expected time.

Proof: Consider how many times a key is copied. Each time a randomized decision is made, the copying probability is 1/2. This process generates an expected single copy (not counting the orig-

[1] This list contains pointers to and from the main data structure as needed.

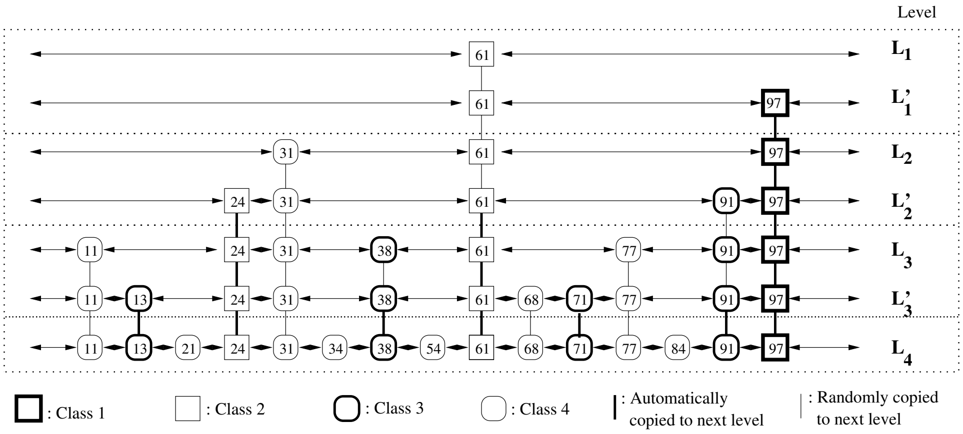

Fig. 4. BSL and its levels. For simplicity, only the left endpoints of the keys are shown.

inal) of each key subjected to it; the expected total for randomly made copies is $O(n)$. Now consider the automatically made copies. None of the $(n+1)/2$ keys in $C_{\log n}$ are automatically copied. The $(n+1)/4$ keys in $C_{\log n-1}$ are automatically copied to two levels. The keys in $C_{\log n-2}$ are copied to four levels. In general, the keys in $C_{\log n-p}$ are copied to $2p$ levels. Summing all automatic copies for each key in each class, we find a total of $O(n)$ copies; adding the expected $O(n)$ randomized copies, the expected total copying (thus construction) time is $O(n)$. ∎

Search. Searching for a match for an address A in a BSL is similar to searching in a regular skip list. We start from the smallest (leftmost) key in the top level. On each level, we follow the linked list to the right until we encounter a key which is greater than A. Then we take a step back (left) and go down one level in constant time using vertical pointers, (ending up on a key less than A). We end the search when we locate an interval containing A.

The following theorem analyzes the search time. Recall that $r(k)$ is the rank of key k.

Theorem 3: A search for an address matched by a key k in a BSL of n keys takes $O(\log r(k))$ expected time.

Proof: To bound the total search time, we bound the number of horizontal and vertical links that we traverse. Consider searching for a key k which belongs to class C_c, with $c \leq \log r(k) + 1$. By construction, all of the keys in C_c are present on level L'_c, which is $2c-1$ steps down from the top, which bounds the vertical distance that we travel. Let us now consider the horizontal links that we traverse on each level. The top level has an expected single key, thus we traverse at most 2 links. Now let our current level be L_m (resp. L'_m). We must have come down from level L'_{m-1} (resp. L_m), following a vertical link on some key s. Let t be the key immediately following s on the level above, i.e. L'_{m-1} (resp. L_m). Then, it must be that $s < k < t$ (recall the ordering between intervals). Thus, on the current level, we need to go right at most until we hit a copy of t. The number of links that we traverse on the current level is then at most 1 plus the number of keys between s and t. Note that these keys do not exist on the level above, even though s and t do. The problem then is to determine the expected number of keys between s and t on this level. In a scheme with no automatic copying (all copying is random with probability 1/2), given two adjacent keys on some level l, the expected number of keys between them on the level below (not copied to l) is 1. This is because, due to the 1/2 probability, one expects on the average to "skip" (not copy upwards) one key for each that one does copy. In our scheme, some keys get copied automatically; therefore, we are less likely to see skipped keys. Thus, in our case, one would expect to see fewer than 1 uncopied key between two copied keys. The expected number of horizontal links that we travel on the current level between s and t then is at most 2. [2] Since we visit at most $2c-1$ levels, traversing an expected 2 links on each level, the total (expected) running time is

[2] In fact, on the average we expect to go only halfway between s and t before going down; for simplicity we ignore this subtlety.

$O(c) = O(\log r(k))$. ■

IV. DYNAMIC BSL

BSL as described above performs well on searches, however, the fact that keys with small rank have many copies makes insertions problematic. For instance, when we insert a key with rank 1 (which will be the case later for our bursty pattern), we need to make $O(\log n)$ copies of the key that we are inserting, one per level. If we know that the key will always stay close to the top of the structure, it is wasteful to copy it all the way to the bottom. The culprit for this problem is the fact that all elements are present at the bottom, for the sake of simplicity. To improve BSL, we propose that elements in class C_i not be present in all the levels below L'_i, but be randomly copied down for only a few levels. (Not copying down from L'_i at all causes gaps which affect efficiency.) This improves the time, as well as the space efficiency of the data structure.

We show below how to incorporate the above changes into BSL so that these updates can be performed efficiently. We call this variant of the data structure a *dynamic BSL*.

Construction. A fundamental difference between the dynamic and the static BSL is in how each of the levels are constructed. In dynamic BSL there is no automatic copying of keys to an upper level. To construct the dynamic BSL we start with the bottommost level $L_{\log n}$ and make our way up. Level $L_{\log n}$ is made up of all the keys that belong to class $C_{\log n}$. For $i < \log n$, an upper level L'_i is constructed from level L_{i+1} by choosing and copying keys from level L_{i+1} independently with probability $1/2$. In addition to those copied from below, L'_i includes all keys in class C_i; these are called the *default keys* of L'_i. To facilitate efficient search some of the default keys of L'_i are chosen and copied to the lower level L_{i+1} independently with probability $1/2$. Each key copied to level L_{i+1} may further be copied to lower levels $L'_{i+1}, L_{i+2}, L'_{i+2}, \ldots$ using the same randomized process. Once we have level L'_i, we construct level L_i by simply choosing and copying keys from level L'_i independently with probability $1/2$ for each key. For an example of a dynamic BSL see Figure 5.

We now show that the construction time for the dynamic BSL is the same as the static version.

Lemma 4: A dynamic BSL with n keys can be constructed in $O(n)$ expected time.

Proof: The expected number of keys on each level is bounded by that in the static BSL. In static BSL, level L'_i contains all keys in classes C_0 through C_i, which number $O(|C_i|)$. L_i contains (an expected) half of those. Thus, L_i and L'_i contain $O(2^i)$ keys.

Let us look at the cost per level. L'_i is formed from L_{i+1} and Class C_i in time $O(|C_i|)$. L_i is formed from L'_i in time $O|C_i|$. The only other cost is for copying down the default keys of L'_i. The expected number of times that a key will be copied down is just under 1. Thus, the expected number of copies from C_i below L'_i is $O(|C_i|)$. To copy a key k down, we go to its left neighbor on the same level, and keep going left until we find a down link, which we take. Then, we go right until we come to a key greater than k and insert k before it. The expected number of left steps is less than 2, because the probability that a key on this level will not exist on the previous level is less than 1/2. Likewise, the expected number of right steps is less than 2. Thus, we spend $O(1)$ time per copy, and $O(|C_i|)$ time for the entire C_i. The costs associated with L'_i and L_i add up to $O(|C_i|)$; summing over i we get $O(n)$. ■

Search. To search for an interval matching an address A in a dynamic BSL we start from the leftmost key in the top level. As in the static BSL, we follow the linked list to the right until we encounter a key greater than A. We then take a step to the left, and attempt to go down one level. However, it is possible that a down pointer may not exist from our current location. In that case, we go left until we hit a key l from which we can go down, [3] follow the link down, and repeat, starting from the copy of l at that level. When we encounter an interval (a key) which matches A, we return the key as the search result.

Rank Assignment. When a key is accessed (through search/insertion/deletion), its rank, along with the ranks of some of the other keys, changes according to a *rank assignment policy*. We focus on two rank assignment policies, each suitable for handling one of the two types of commonly observed access patterns mentioned in Section I.

1. Most Frequently Accessed (MFA) policy is de-

[3]Remember that a key on this level does not necessarily have a copy one level below.

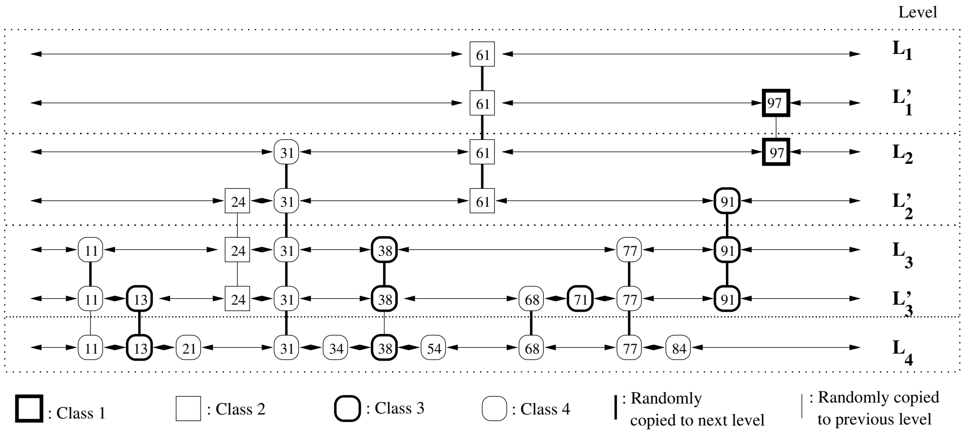

Fig. 5. Dynamic BSL. Only the left endpoints of the keys are shown.

signed for independently skewed access patterns, where the rank of key k reflects how many times it has been accessed. The number of accesses is denoted by $f(k)$. Thus, if $f(i) > f(j)$ for two keys i, j, then $r(i) < r(j)$.

2. Most Recently Accessed (MRA) policy is designed for bursty access patterns, which are our main focus. In this policy the rank of a key k is the number of unique keys accessed since the last access to k.

The next two sections describe how BSL implements these policies.

A. *BSL for MFA*

In this section we show how dynamic BSL can self-adjust to the slight change in the access pattern after each search. An elegant solution is provided for this problem in [7]; our running times are similar. Our main goal is to present a simpler variant of dynamic BSL in this section as a motivation for the variant that we will use in the next section for our main focus area, bursty access patterns. The dynamic data structure that we use in this section is self-adjusting; after each search, if necessary, it readjusts the ranks to reflect the new access statistics. Insertions and deletions must also leave the class sizes intact.

Search. To comply with the MFA policy, BSL may need to update the rank (and possibly the class) of a key k after it is accessed due to a search. We use the rank ordered master list of all keys in the BSL to check, once the access count of k is incremented, whether the new count is more than that of the key ranked $r(k) - 1$. If so, then the rank of k is changed and k is moved to its new rank position in the list. [4]

Let C_i be the class of k before it was accessed. Once the new rank of k is determined, we need to check whether k has now moved to class C_{i-1}. If so, we need to assign L'_{i-1} as the default level of k and shift the bottommost and topmost levels of k up by 2 by manipulating the pointers to and from the copies of k on the levels involved. Once k moves up to class C_{i-1}, another key l from C_{i-1} is moved down to C_i to preserve the class sizes. We need to change the default level of l and push down the topmost and bottommost levels of l by two as well.

Theorem 5: In the MFA implementation of BSL with n keys, search for an address matched by key k takes $O(\log r(k))$ expected time.

Proof: To bound the number of vertical steps taken during the search, note that k will be $O(\log r(k))$ steps down from the top as a default key on that level. Following the arguments for static BSL, in each level the expected number of right steps that we take until we reach a key greater than k is at most 3. We might, however, need to go left to find a key from which we can go down. Again, the expected number of left steps that we need to take is less than 2 with an analogous argument, this time considering downward copies. Therefore, the expected number of horizontal steps per level is $O(1)$, and the whole search takes $O(\log r(k))$ time. ■

Insertion/deletion Since MFA is not our main focus,

[4] If there are multiple keys with access counts same as the key ranked $r(k) - 1$, we need to move k a few steps in the linked list. This can be done in $O(1)$ time by grouping each set of keys with the same access count in a separate linked list.

we skip the details of insertion and deletion, as well as the proof of their running time. We mention very briefly how they are performed and give a theorem stating their complexity.

When inserting a key k to BSL, it is assigned rank $n+1$ according to the MFA policy, and therefore needs to be inserted at the bottom of the structure. If $n = 2^\ell - 1$ for some integer ℓ, then it means class $C_{\log n}$ is full, and we need to establish a new class $C_{\log n+1}$ with k as its sole member. To do this, we create two levels $L'_{\log n}$ and $L_{\log n+1}$ and simply insert k in level $L_{\log n+1}$. For promoting k to upper levels, we again use the independent random process (fair coin) with probability $1/2$.

To delete key k, we first identify its location in BSL, then delete all of its copies from the BSL as well as from the master list. This changes the ranks of keys with rank greater than $r(k)$. We update the default levels of all keys whose classes change – there can be at most $\log n$ of them – by pushing their top- and bottommost levels up by two.

The proof of the theorem below is left out for space considerations; it can be accessed at [8].

Theorem 6: In the MFA implementation of BSL with n keys, insertions and deletions can be done in $O(\log n)$ expected time.

B. Most recently accessed (MRA)

This scheme, to be used with highly burst access patterns, is our main focus. The versions of BSL that we have described so far handle insertions and deletions in $O(\log n)$ time. This is good enough for keys that remain in the data structure for a long time, but in many applications, the majority of the keys have very short lifespans. Once inserted, these keys are rapidly accessed a few times, then deleted after a short period. As a result, their MRA ranks always remain close to 1. In such applications, insertion and deletion of such hot keys are the main bottleneck.

To facilitate more efficient implementation of insertions and deletions, we employ a *lazy updating scheme* of levels and allow flexibility in class sizes. The size of a class C_i is allowed to be in the range $(2^{i-2}, \ldots, 2^i - 1)$; recall that the default size of C_i is 2^{i-1}. The lazy updating allows us to postpone updates until the size of a level becomes too large or too small.

Example: Let the number of elements in classes $C_1, C_2, \ldots, C_5$ be $1, 2, 4, 8, 16$ respectively (the default sizes). A sequence of eight insertions would respectively yield the following class sizes.

1	1	1	1	1	1	1	1	1
2	3	2	3	2	3	2	3	2
4	4	6	6	4	4	6	6	4
8	8	8	8	12	12	12	12	8
16	16	16	16	16	16	16	16	24

Observe that the effects of most insertions and deletions are confined to upper levels. As a result, the insertion or deletion of a key k takes $O(\log r_{max}(k))$ time, where $r_{max}(k)$ is the maximum rank of key k in its lifespan.

Search. In accordance with the MRA policy, when a key k is searched for, its rank becomes 1 and the ranks of all keys whose rank were smaller are incremented by one. The rank changes are reflected in the BSL as described for MFA. Although the class sizes are not rigid, to make the analysis simple, we maintain the class sizes after a search without affecting the overall cost. [5]

Lemma 7: In the MRA implementation of BSL with n keys, search for a key k takes $O(\log r(k))$ expected time.

Proof: We follow the same lines as the dynamic BSL for MFA with a slight difference in level sizes. In this model, the size of a class is always at least half its default size. Thus, the height of the BSL is at most 2 plus its "ideal" height with the default class sizes. The expected number of horizontal steps per level during search is still $O(1)$ using the same argument as the MFA. Since we go down $O(\log r(k))$ steps for a search the lemma follows. ■

Insertion/deletion. When a key k is inserted into the BSL it is assigned a rank of 1 and the ranks of all keys in the data structure are incremented by one. After an insertion, if the size of a class C_i reaches its upper limit of 2^i, then half of the keys in C_i with the largest rank change their default level from L'_i to L'_{i+1}. This is done by moving the topmost and bottommost levels of each such key by two as described for the MFA implementation. One can observe that such an operation can be very costly; for instance, if

[5] This is possible by shifting the largest ranked key down by one class for all the classes numbered lower that the initial class of k; k is moved to the top class.

all classes $C_1, C_2, \ldots, C_l$ are full, an insertion will change the default levels of 1 key in class C_1, 2 keys in class C_2, and in general 2^{i-1} keys from class C_i. However a tighter amortized analysis is possible by charging more costly insertions to less costly ones. The amortized analysis gives the average insertion time for a key k as $O(\log r_{max}(k))$, where $r_{max}(k)$ denotes the maximum rank of k in its lifespan.

To delete a key k, we first search for it in the BSL, which changes its rank to 1 and then we delete it from all the levels where it has copies. If the number of keys in class C_i after the deletion is above its lower limit of 2^{i-2} then we stop. Otherwise, we go to the next class C_{i+1} and move its lowest ranked 2^{i-1} keys to class C_i by moving their default levels upwards by two. We continue this process until all levels have a legal number of keys. As in insertion it is possible to do amortizations to show that the average deletion time for a key k is $O(\log r_{max}(k))$.

The proof for the theorem below stating these running times is presented in the full version of the paper; it can be accessed at [8].

Theorem 8: The MRA implementation of BSL facilitates insertion or deletion of a key k in amortized $O(\log r_{max}(k))$ time, where $r_{max}(k)$ is the maximum rank of key k in its lifespan.

In bursty patterns, when a key is hot, it is accessed a large number of times rapidly. This means that, according to the MRA scheme, the key maintains a small rank compared to the number of keys in the data structure. Thus, having the running times depend on the rank rather than the total number of keys yields significant gains.

V. Experimental Results

In order to evaluate the performance of BSL in practice we provide a comparative experimental study that involves implementations of BSL and other data structures on both real traces and simulated data with varying biases.

Our experiments on *simulated data* compare search times when the keys are accessed with a *geometric distribution* on MRA ranks. At each stap of the simulation, a key k with rank $r(k)$ was accessed with probability $P(k) = p^{r(k)}(1-p)$, where p is the bias parameter. By varying p we changed the average rank of the keys that are accessed and hence simulated the "hot" working set phenomenon observed in packet traces [4], [12]. It was noticed in many contexts [11], [12] that the probability of accessing a key drops exponentially with the time the key is kept inactive; thus geometric distribution provides appropriate means for modeling the bias in bursty patterns.

Our analysis using real data uses the publicly available LBL trace data [5]. This trace contains 1.8 million TCP packets flowing between the Lawrence Berkeley Labs and the rest of the Internet.

There are a number of parameters which affect the practical performance of a BSL implementation. For example, the size of the topmost class C_1 can be tuned to capture the bias in the data for "optimizing" BSL's performance. Figure 6 shows how the size of class C_1 affects the search times for accesses following geometric distributions with different p values. An interesting observation is that with increasing bias (i.e. the average rank of the searched keys), the optimal size for class C_1 increases. For example, when the average rank is 10 the optimal size of C_1 becomes 16, when the average rank is increased to $10^2 = 100$ the optimal size of C_1 increases to $16^2 = 256$.

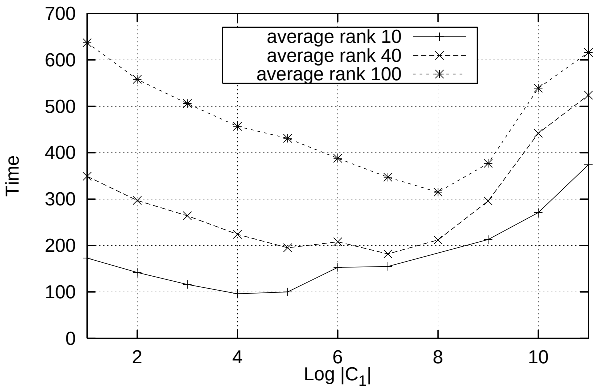

Fig. 6. Search times for different class 1 sizes – 10^6 searches on 98K keys

The summary of our experiments on simulated data with geometric access distribution on MRA ranks is provided in Figure 7. [6] Here we compare performance of several variants of BSL with that of regular skip lists[7] and a simple move-to-front (MTF)

[6]We performed our experiments on a desktop PC with 400MHz Pentium II microprocessor.

[7]The implementation we use was written by William Pugh and is available at
`ftp://ftp.cs.umd.edu/pub/skipLists/`

linked list implementation. The choice of these data structures for our experiments is clear. On unbiased or slightly biased data, the regular skip list is considered to be the best performing search data structure. Thus, it provides excellent means to measure the performance of BSL on data with low bias. An MTF list, on the other hand, is a very simple data structure which provides the best performance on highly biased data. Although it has a worst case search performance of $O(n)$ (which is unacceptable for many applications), it provides a very good benchmark on how well BSL evaluates on highly biased distributions.

The first observation we make is that the BSL implementations consistently outperform the regular skip lists on all bias figures we tested. This shows that the added complexity in maintaining multiple classes in BSL is more than compensated by the gains in efficiency due to the exploitation of bias. Although it is conceivable that skip lists may perform slightly better than BSL on even lower biases than we tested, we can safely predict that for all biases of practical interest BSL is a favorable choice.

Another observation is that BSL implementations outperform the MTF lists by at least one order of magnitude when the bias is low. However MTF lists perform better with high bias. Although this phenomenon is partially due to the simplicity of MTF lists we suspected that an even bigger factor is the cache effects: while running BSL or the regular skip list implementation, it is expected that the caches need to keep some of the intermediate nodes that are encountered during the searches, losing valuable space; this is not the case for MTF lists. To check our hypothesis and also see how well BSL would compare to MTF lists in next generation architectures, where it would be possible to place the whole data structure in an on-chip memory, we tested both BSL and MTF list implementations on the same data after disabling caches. Results of this test are summarized in Figure 8, where one can observe that BSL catches up with the performance of move-to-front lists much earlier. Yet, the initial gains of MTF lists give the idea that it can be combined with BSL in a hybrid scheme where a short MTF list can act as an initial filter before a search is forwarded to BSL; we implemented and used such a hybrid data structure in our tests.

Three different BSL data structures were included in our experiments. One uses 8 for the size of class C_1 and another used a size of 512. We see that setting $|C_1| = 8$ works well for highly biased data, but the overhead of maintaining small classes gives worse performance for accesses with smaller biases. Using $|C_1| = 512$ gives better performance over a larger range and shows improvement over regular skip lists beyond a working set size of 500. The performance of a hybrid scheme where keys are initially searched in a move-to-front linked list can also be found in the plots. We used a list of 50 keys before resorting to regular BSL with 512 keys in C_1 .

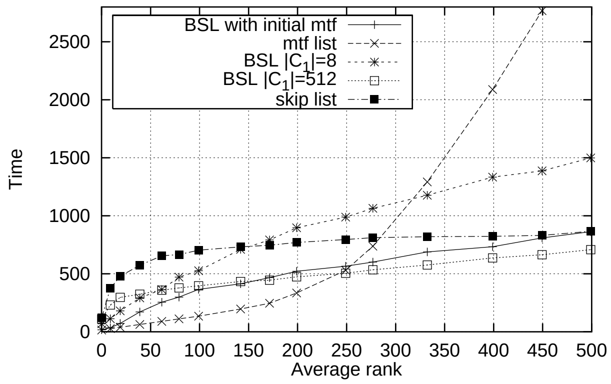

Fig. 7. Search times on geometrically distributed data with varying biases; a total of 10^6 searches were performed on 198K keys.

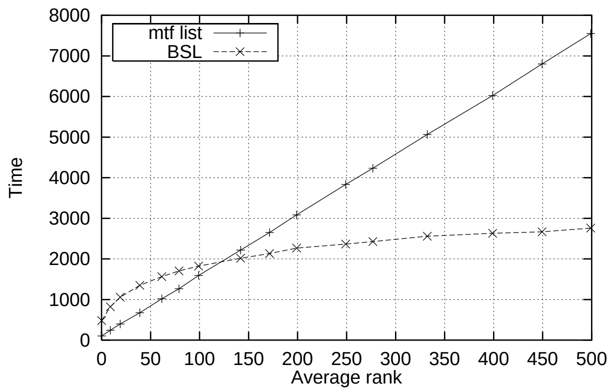

Fig. 8. Comparison of BSL ($|C_1| = 8$) and MTF list performances with disabled caches on geometrically distributed data; a total of 10^5 searches were performed on 98K keys.

Our final experiments compare the data structures mentioned above using the LBL trace data. For ease

Data Structure	Time (secs)
Skip List	4.4
MTF List	1.97
BSL $\|C_1\| = 8$	7.23
BSL $\|C_1\| = 512$	5.87
BSL+mtf(length 50) $\|C_1\| = 512$	3.3
BSL+mtf(length 256) $\|C_1\| = 4096$	2.61

TABLE I
Running times for the LBL trace data.

of implementation the flows are initially identified and mapped to a unique integer key in a preprocessing phase. Since the LBL trace data already uses modified IP addresses for privacy reasons and the data structures in question should not be effected by the values of the keys this should not change the results of the test. The trace uses about 1.8 million packets and includes about 15 thousand insert operations. The timing results of the data structures considered in our study on the LBL data are summarized in table I, which are again quite favorable for (hybrid) BSL implementations.

VI. Discussion and Future Work

We present BSL, a dynamic lookup data structure which exploits the bias in access patterns to provide faster lookup. In addition to supporting fast lookups for more commonly accessed keys, BSL adapts to changes in the access pattern without the need for an explicit and costly reconstruction. An interesting future direction is to make the solution more general so that it can handle multiple fields in the form of prefixes and ranges. Such a data structure is highly desirable in various Layer 4 classification/filtering applications where lookups may involve other fields such as source address, port number, TCP flags, etc., and we need to find the best matching rule for these fields. The results presented here naturally extend to packet classification with one field specified as a prefix (source or destination address), or a range (port number), and all the remaining fields specified exactly. This is achieved by the concatenation of the fields within a single key. Firewall specifications are usually compatible with this format. It remains a highly interesting open question to extend our result to the general packet classification/filtering problems where burstiness may result in higher gains in performance.

References

[1] S. Bent, D. Sleator, and R. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14, 1985.

[2] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM*, 1997.

[3] G. Cheung and S. McCanne. Optimal routing table design for IP address lookups under memory constraints. In *IEEE INFOCOM*, 1999.

[4] K.C. Claffy, H.W.Braun, and G.C.Polyzos. A parameterizable methodology for internet traffic flow profiling. In *IEEE Journal on Selected Areas in Communications*, 1995.

[5] LBL-TCP-3 Trace Data. http://ita.ee.lbl.gov/html/contrib/lbl-tcp-3.html.

[6] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proc. IEEE INFOCOM*, 1998.

[7] P. Gupta, B. Prabhakar, and S. Boyd. Near-optimal routing lookups with bounded worst case performance. In *Proc. IEEE Infocom*, 2000.

[8] http://vorlon.cwru.edu/∼cenk.

[9] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. In *Proc. IEEE Infocom*, 1998.

[10] L. Larmore and T. Przytycka. A fast algorithm for optimal height-limited alphabetic binary trees. *SIAM J. on Computing*, 1994.

[11] S. Lin and N. McKeown. A simulation study of IP switching. In *Proc. ACM SIGCOMM*, 1997.

[12] Suvo Mittra and Anindya Basu. Packet classification: An argument for a working set model. Technical report, 1999.

[13] S. Nilsson and G. Karlsson. Fast address look-up for internet routers. In *Proc. IEEE Broadband Communications*, 1998.

[14] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Comm. of the ACM*, 33,6, 1990.

[15] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32, 1985.

[16] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. In *In Proc. ACM Sigmetrics*, 1998.