Biased Skip Lists for Highly Skewed Access Patterns

Funda Ergun[1], S. Cenk Şahinalp[1,2], Jonathan Sharp[1,2], and Rakesh K. Sinha[3]

[1] Department of Electrical Engineering and Computer Science, Case Western Reserve University, OH. {afe,cenk,jps17}@eecs.cwru.edu
[2] Department of Computer Science, University of Warwick, Coventry, UK.
[3] Bell Laboratories, Murray Hill, NJ.

Abstract. [1] Dynamic tables that support search, insert and delete operations are fundamental and well studied in computer science. There are many well known data structures that solve this problem, including balanced binary trees, skip lists and tries among others. Many of the existing data structures work efficiently when the access patterns are uniform, but in many circumstance access patterns are biased. Various data structures have been proposed that exploit bias in access patterns to improve efficiency for the operations they support.
In this paper we introduce a new data structure, the *biased skip list* (BSL), which is designed to work with biased access distributions. Specifically, given key k, let its rank $r(k)$ be the number of distinct keys accessed since the last access to k. BSL enables one to search for k in $O(\log r(k))$ expected time. Insertions and deletions take $O(\log r_{max}(k))$ expected time where $r_{max}(k)$ denotes the maximum rank of k during its lifespan.
Our work is motivated by recent studies on packet filtering and classification where keys have been found to have geometric (or more skewed) access probabilities as a function of how recently they have been accessed.
We demonstrate the practicality of BSL with experiments on real and synthetic data with various degrees of bias.

1 Introduction

Maintenance of dynamic tables is fundamental to computer science with a rich collection of papers and textbook pages devoted to it. Given a set of keys from a totally ordered universe, the goal is to build a data structure which supports search, insert and delete operations with efficient performance bounds. Three popular families of data structures for implementing dynamic tables are balanced search trees (including the classical examples of AVL-trees, red-black trees and 2-3 trees [21]), tries [10], and skip lists [16]. These data structures provide efficient operations and work well when the access probabilities do not vary significantly across the keys. However, when access patterns are biased, it is possible to improve the efficiency of the operations. In this paper we describe a new data structure, the *biased skip list*, which is designed to exploit bias in access patterns.

Our new data structure is based on skip lists. Skip lists are simple, randomized data structures with small constant factor overheads. Given a set of n keys in sorted

[1] Part of this work will appear at INFOCOM 01.

order, a skip list can be constructed in $O(n)$ expected time. Search, insert, and delete operations each take $O(\log n)$ expected time. Skip lists are often favored over balanced search trees, whose operations have similar asymptotic running times, but suffer worse performance in practice due to the overhead of complex balancing operations. This is especially true when the key distribution is non-uniform over the universe. Since the performance of tries depends (logarithmically) on the universe size, skip lists can also be favored over tries, especially when the universe size is large.

There are many circumstances under which access patterns show considerable bias. It is well known that locality of reference results in high skew in memory/cache [20], disk [14] and buffer [8] management for which a least-recently-used (LRU) policy yields good results. Recent studies on table lookups for Internet packet filtering and classification [11, 12, 5] demonstrate similar skew. In such applications, the access probability of a key k, decreases exponentially (or faster) with its *LRU rank*, written $r(k)$, which is the number of distinct keys accessed since the last access to k. Such bias is observed in many other types of data for which move-to-front based data compression [3] is very effective; examples include source code compression [17], index compression [15] and compression of text after Burrows-Wheeler transform [4].

Several special purpose data structures have been proposed that exploit skew in access distribution, such as biased search trees [2], splay trees [19], treaps [18] and variants of tries [6]. Even though skip lists possess many desirable properties and have several advantages over the alternatives, we are not aware of any work that modifies skip lists to take advantage of such bias.[2]

Our new data structure, the *biased skip list* (BSL) improves the throughput of skip lists in the presence of high locality of reference in access patterns. More specifically, BSL enables one to search for key k in $O(\log r(k))$ expected time. This implies that if the access requests are generated by an ergodic random source, BSL's search time matches the entropy of the source [3] and is therefore optimal in the information theoretic sense. Note that when the bias is low (or non-existent), BSL's performance is similar to that of a regular skip list.

Similar to the best performing data structures for dynamic table lookup, (including biased search trees, splay trees and treaps), one can insert or delete a key k to BSL in $O(\log n)$ expected time. We show a variant of the BSL that uses a lazy updating scheme to improve this running time to $O(\log r_{max}(k))$, where $r_{max}(k)$ denotes the maximum rank that k attains during its lifetime. There are many scenarios where improved insertion and deletion times can lead to considerably higher throughput. One such application is the maintenance of web proxy caches, where a small number of pages (e.g. stock news) stay very "hot", i.e. keep a small rank, throughout their lifespan but are short lived. Other pages (e.g. company web pages) may have less popularity but do live longer. See [1] for empirical evidence and discussion. Another application is firewall maintenance, where one needs to cater to connections with different characteristics, such as HTTP, FTP, and telnet. High bias in insertions and deletions for firewall maintenance and other Layer 4, IP related applications is supported by empirical evidence [12, 11, 5].

[2] One exception is the work of [13] which attempts to optimize skip lists when access probabilities are static and given in advance.

Another variant of BSL, is to enhance performance in the presence of independently skewed access patterns, where access probabilities to keys are biased but relatively stable over time. When the access probability $p(k)$ of each key k is i.i.d., this variant of BSL facilitates accessing a key k in optimal $O(\log \frac{1}{p(k)})$ time. Insertion and deletion times are $O(\log n)$. Further discussion of this variant of BSL is left until the full version of this paper.

Organization of the Paper. We start with a summary of skip lists. In Section 2 we describe BSL and its operations. Following this, Section 3 shows how to modify the basic BSL to improve insert and delete times. Section 4 demonstrates the practicality of BSL with experiments on synthetic data and data from real applications. Finally we end with some conclusions and discussion of further work.

1.1 Preliminaries

Given n keys from an ordered universe, a skip list can be constructed in $O(n)$ expected time if the keys are given in sorted order. Search, insert, and delete operations each take $O(\log n)$ expected time. To construct a skip list, we start with a linked list of all keys in the data structure in sorted order; this makes up level $\log n$ of the skip list.[3] We iteratively build subsequent levels $\log n - 1, \ldots, 2, 1$ in a randomized fashion. 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 links to its copies (if they exist) on adjacent levels. Because there are $\log n$ levels, the expected number of keys in level 1 is only 1.

To search for a key k in a skip list, we start from the smallest (leftmost) key in the highest level. On each level, we follow the linked list to the right until we encounter a key which is greater than k. When that happens, we take a step to the left and go down one level (this leaves us at a key which is less than k). The search ends when k is found or when a key greater than k is found in the lowest level. To delete a key, we perform a search to find its highest occurrence in the structure and delete it from every level it appears in. To insert a new key k, we first search for k in the skip list. This search locates the correct place to insert k in each level. Key k is inserted into the bottommost level and a fair coin is tossed to determine how many subsequent levels k is copied to. On a successful coin toss, k is copied up one level and this process continues until the first failure.

2 Biased Skip List (BSL)

In this section we present a version of BSL which has an $O(\log r(k))$ expected search time for a key k, and has $O(\log n)$ expected insert and delete times. A modification that allows more efficient inserts and deletes is described later in the paper.

The biased skip list is an extension to a regular skip list in the following way. Each key maintained by BSL is given a unique rank in the range $[1 \ldots n]$ and is denoted $r(k)$. BSL keeps all keys in a linked list in ascending rank order. We partition this list

[3] Throughout the paper $\log n$ denotes $\lceil \log_2 n \rceil$.

into *classes* $C_1, C_2, \ldots, C_{\log n}$ contiguously, where the class sizes are geometric with[4] $|C_i| = 2^{i-1}$.

Construction. BSL is constructed randomly in a bottom-up fashion similar to skip lists. It comprises of several levels, each one being a doubly linked list of keys in sorted order. The levels of the data structure are labelled $L_{\log n}, L'_{\log n-1}, L_{\log n-1}, \ldots, L_2, L'_1, L_1$. The bottommost level, $L_{\log n}$, includes the keys from all classes. To obtain the keys in any level L'_i, we copy the following keys from level L_{i+1}: (1) all keys from classes $C_i, C_{i-1} \ldots, C_1$ (by definition all should be present in L_{i+1}) – there will be $2^i - 1$ of them, and (2) a subset of the remaining keys in L_{i+1} picked independently, each with probability $1/2$ – the expected number of these keys will be 2^{i-1}. We obtain the keys in level 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, each with probability $1/2$. See Figure 1 for an example. Notice that a key k in class C_i, is automatically copied from the bottom to all levels up to L'_i and then randomly copied to higher levels.

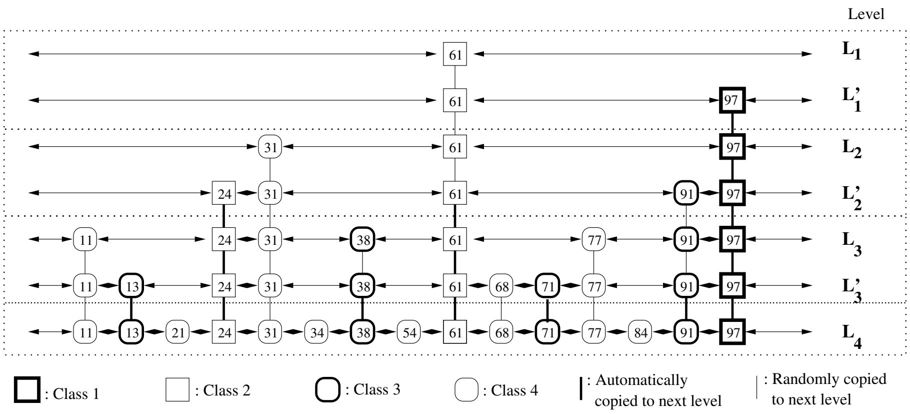

Fig. 1. BSL and its levels.

Lemma 1. *A BSL with n keys can be constructed in $O(n)$ expected time, provided the keys are given in sorted order.*

Proof. Let us look at how many times an element will be copied up. Since the copying probability is 1/2, the expected number of levels on which an element will be copied as a result of a coin toss is 1. Let us count the automatically made copies. $C_{\log n}$, which is of size $(n+1)/2$, is not subject to automatic copying. $C_{\log n-1}$, which is of size $(n+1)/4$, is copied automatically on to two levels. In general, $C_{\log n-p}$ gets copied on to $2p$ levels. Summing up over all elements, the total number of automatically made copies is $O(n)$; combined with the randomly made copies, the total construction time is $O(n)$. □

[4] In fact, with appropriate modifications, class sizes may follow any geometric sequence.

Search. Searching for a key k in a BSL is similar to that in a regular skip list. We start from the smallest key in the topmost level. On each level, we follow the linked list to the right until we encounter a key which is greater than k. Then we take a step back (to the left) and go down one level. We end the search either when we find k, or when we reach a key greater than k in the bottommost level, which indicates that k is not present in the data structure.

If k is found then some adjustments to the data structure are necessary to reflect the changes in rank. The rank of k becomes 1 and thus it is moved to the front of the rank ordered linked list. If necessary, extra copies of k are inserted in levels up to L'_1. If k was in class C_c before the search, then the last (by rank ordering) key in each class C_i, for $i < c$, will move to class C_{i+1}. For each such key, two of the automatically made copies must be removed so that its automatically made copies stop at level L'_{i+1}.

Lemma 2. *A successful search for an element k in a BSL of n keys takes $O(\log r(k))$ expected time, whereas an unsuccessful search takes $O(\log n)$ expected time.*

Proof. First, consider a successful search. Due to the partitioning into classes, key k belongs to class C_c, where $c \leq \log r(k) + 1$. To bound the search time, we bound the horizontal and vertical distance that we travel until we encounter k. The construction method for the BSL dictates that all the keys in Class C_c are present on level L'_c, which is at depth $2c$ from the top of the BSL. Once we reach level L'_c, we are guaranteed to find k, thus we travel at most $2c - 1$ vertical links from the top of the BSL. We now analyze how many (expected) horizontal links are traversed per level. On the top level, there is an expected single key, corresponding to at most 2 links. Let us now consider our actions on level l. We must have come to level l from the level above, say l', from some key k'. Let k'' be the key immediately to the right of k' on level l'. By our search strategy, $k' < k < k''$. Thus, on level l, in the worst case, we need to traverse the links between k' and k'', which number one more than the number of keys on level l between k' and k'' not copied up to level l'. In a situation where all keys are always subject to the randomized process to decide whether they will be copied up, the expected number of keys that are skipped between two keys that are copied is 1. In our scheme, since some keys get copied automatically, the expected number is even less. Thus, the expected number of links on level l, between k' and k'', is less than 2. Since we visit at most $2c - 1$ levels, traversing at most an expected 2 links on each level, the total (expected) running time to find k in the data structure is $O(c)$; substituting the value for c, this is $O(\log r(k))$. The rank adjustments to the data structure can also be done in $O(\log r(k))$ time. To see this, observe that at most $2c - 1$ extra copies of k must be made and $c - 1$ keys (one from each class C_i, $i < c$) must be removed from 2 levels each. Therefore the total running time for a successful search is expect $O(\log r(k))$.

For an unsuccessful search, the number of horizontal links traversed per level stays the same; however, one has to go all the way down to the bottom of the data structure. Therefore, the vertical distance traveled is $O(\log n)$ levels, with an expected constant number of links per level, giving an overall expected running time of $O(\log n)$. □

Insertion and deletion. To insert a new key k, we first search for k knowing it will not be found. This locates the correct place to insert k in each level. Copies of k are made in each level up to L'_1 and k is added to the front of the rank ordered list. One key from each class is removed from 2 levels in the data structure to update the new rank ordering, in the same fashion as for a search. This can be done in $O(\log n)$ expected time.

To delete a key k from BSL, we follow the search procedure, removing any copies of k we find. In this case the first (by rank ordering) key in each class that is numbered higher that the class of k moves to the next lower numbered class. Such keys are copied up two further levels in the data structure to reflect the new rank ordering. Again, this can be accomplished in $O(\log n)$ expect time.

3 Improving the Efficiency of BSL

BSL, as described above, performs well on searches, however, keys with small rank have many copies, which makes insertion and deletion less efficient. For instance, when we insert a key with rank 1, we make $O(\log n)$ copies of the key – 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. To improve this, BSL can be modified so that elements in class C_i are not automatically present in all the levels below L'_i, but are randomly copied down, the same way they are randomly copied up. Notice that not copying down keys from L'_i at all can cause gaps especially when keys in the same class are are grouped together in terms of key values; this would affect efficiency. This modification improves the time, as well as the space efficiency of the data structure.

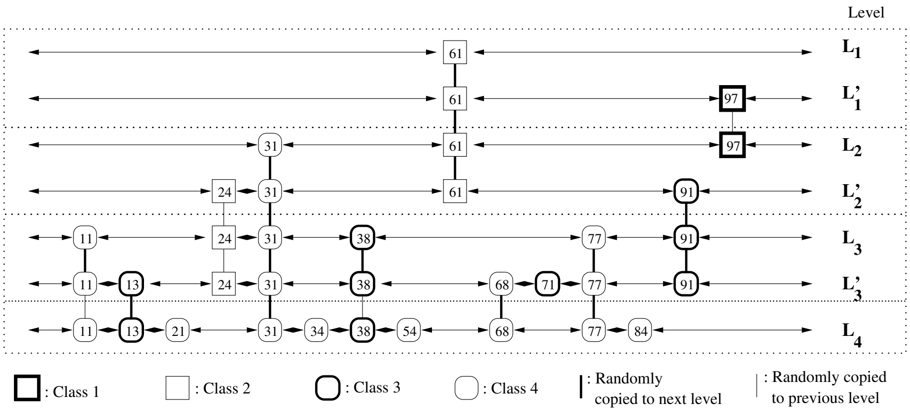

Fig. 2. Improved BSL.

Construction. We start with the bottommost level $L_{\log n}$ and iteratively build up. Level $L_{\log n}$ contains all keys that belong to class $C_{\log n}$. For $i < \log n$, an upper level L'_i is constructed by copying those keys of level L_{i+1} that are chosen independently with probability $1/2$. L'_i also 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 copied to the lower level L_{i+1}. This is done independently with probability $1/2$. Each key copied to level L_{i+1} may further be copied to lower levels $L_{i+1}', L_{i+2}, L_{i+2}', \ldots$ with independent coin tosses. Once we have the keys in level L_i', we construct level L_i by copying those keys in level L_i' which are chosen independently with probability $1/2$. For an example see Figure 2.

Lemma 3. *The improved BSL with n keys can be constructed in $O(n)$ expected time, provided the keys are given in sorted order.*

Proof. The probabilistic copying down of keys does not add significantly to the number of keys per level since the expected number of keys copied down decreases faster than the expected number of keys on each level as we go down. Thus, we still maintain that any level L_i or L_i' will contain $O(2^i)$ keys. We examine the cost per level. Level L_i' is formed from the previous level and the class C_i in time $O(|C_i|)$. Level L_i is formed from L_i', again in $O(|C_i|)$ time. The only remaining cost is the copying down of keys originating from L_i'. Note that, the expected number of times a key will be copied down is just under 1. Thus, the expected total number of copies from C_i that will lie below L_i' is $O(|C_i|)$. To copy a key down, we go to its left neighbor on the same level and attempt to go down or keep going left until we can go down one level. After that, we go right until we come to a key greater than the one we need to copy, and insert our key before it. The expected number of steps that we take to the left is $O(1)$, since the probability that a key on this level will not exist on the previous level is less than 1/2. With a similar argument, the expected number of steps that we take to the right is $O(1)$. Thus, we spend $O(1)$ time copying one key down, and $O(|C_i|)$ time taking care of the copying of the entire C_i. All the costs associated with levels L_i' and L_i hence add up to $O(|C_i|)$. The sum of these costs over all i gives us $O(n)$. ☐

Search. To search for a key k, we start from the smallest key in the topmost level. On each level, we follow the linked list to the right until we encounter a key j which is greater than or equal to k. If $j = k$, we successfully terminate the search. Otherwise we go left until we reach a key l that has a copy one level below.[5] Then we simply go one level down and iterate starting from the copy of l at that level. If the level is the bottommost one and while going right we have encountered a key which is larger than k, then we conclude that k is not present in the BSL, and terminate. Any rank adjustments that must be made after a key is found are done similarly to those described for searches in Section 2.

3.1 Lazy Updating

To facilitate efficient implementation of insertions and deletions, we employ a *lazy updating scheme* of levels and allow flexibility in class sizes. The size of class C_i is allowed to be in the range $(2^{i-2}, 2^i)$, while its default size remains 2^{i-1}. The lazy updating allows us to keep an insertion or deletion local to upper levels for most insertions and deletions. 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.

[5] Remember that a key at level L_i' does not necessarily have a copy in level L_{i+1}.

Search. In accordance with how LRU ranks are assigned, when a key k is accessed 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 without changing the number of keys in any class or level; class sizes may only change after an insertion or a deletion. This enables BSL to search for an existing key k in $O(\log r(k))$ expected time. Searching for a key which is not in the data structure takes $O(\log n)$ expected time, as before. See [9] for the full description and proof.

Insertion and deletion. When a key k is inserted to the BSL it is given rank 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, i.e. 2^i, then the highest ranked half of the keys in C_i 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. One can observe that such an operation can be very costly; if, for example, 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 it is possible to amortize more costly insertions to less costly ones: we show that the average insertion time for a key k is $O(\log r_{max}(k))$, where $r_{max}(k)$ denotes the maximum rank of k in its lifespan.

To delete a key k we first search for k in BSL, which will change its rank to 1 and simply delete it from all the levels 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 up by two. Similar to insertion it is possible to amortize more costly deletions to less costly ones; the main result in this section is that the average deletion time for a key k is $O(\log r_{max}(k))$.

The BSL implementing MTF enables insertion or deletion of a key k in $O(\log r_{max}(k))$ amortized expected time, where $r_{max}(k)$ is the maximum rank of key k during its lifespan. See [9] for the full description and proof.

4 Experimental Results

In order to evaluate the performance of BSL in practice, we provide a comparative, experimental study of BSL implementations and other data structures using data from real applications and synthetic data with varying degrees of bias. The architecture used in these experiments has an AMD K6-2 300MHz processor with 64M RAM, 32K level 1 cache (on the CPU) and 512K level 2 cache.

Our experiments on *synthetic data* compare search times when the keys are accessed with a *geometric distribution* on LRU ranks. At each step 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 [5, 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 a geometric distribution provides appropriate means for modeling the bias in bursty access patterns.

Our experiments on *real data* use the publicly available LBL trace data [7]. This trace contains full address lookups to 1.8 million TCP packets flowing between the

Lawrence Berkeley Labs firewall and the rest of the Internet. Notice that in firewall applications one usually needs to search for full IP addresses (and port numbers) rather than address prefixes.

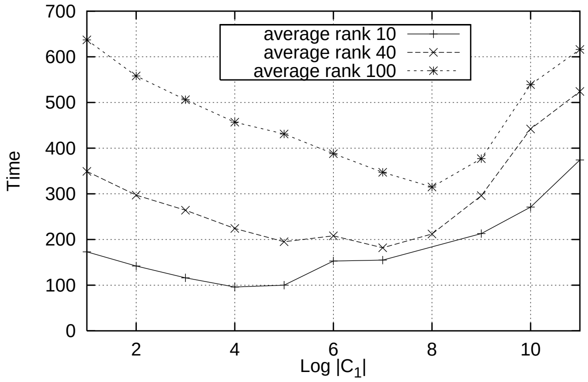

Fig. 3. Search times for different class 1 sizes – 10^6 searches on 98K keys

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, "optimizing" BSL's performance. Figure 3 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 $16^2 = 256$.

The summary of our experiments on simulated data with geometric access distribution on LRU ranks, is provided in Figure 4. Here we compare performance of several variants of BSL with that of regular skip lists[6] and a simple implementation of move-to-front (MTF) via a link list. The choice of these data structures for our experiments is as follows. On unbiased or slightly biased data, the regular skip list is often 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, especially if searches that fail to find a key occur often), 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 degrees of bias we tested, although we can see the skip list search times begin to plateau before BSL. This shows that the added complex-

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

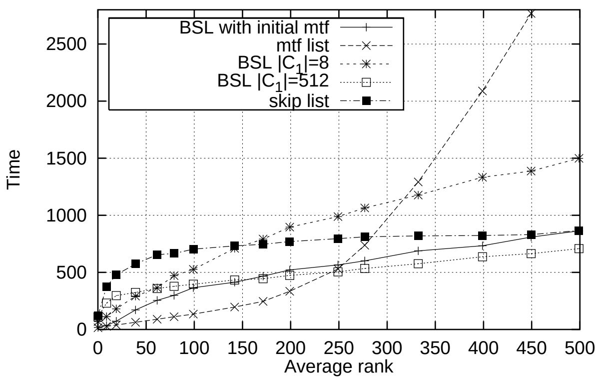

Fig. 4. Search times on geometrically distributed data with varying biases; a total of 10^6 searches were performed on 198K keys.

ity in maintaining multiple classes in BSL is more than compensated by the gains in efficiency with biased accesses. 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 in the packet filtering and classification applications we consider, BSL is a favorable choice.

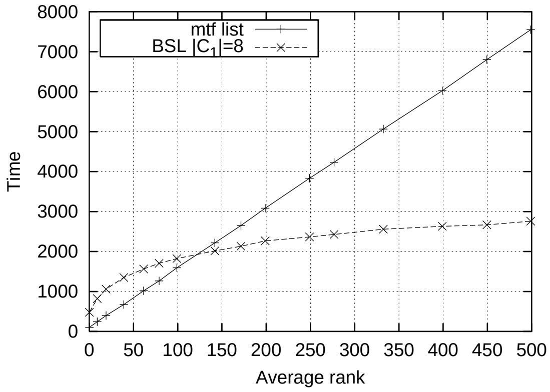

Fig. 5. Comparison of BSL ($|C_1| = 8$) and MTF list performances with disabled caches on geometrically distributed data; a total of 10^5 searches were performed on 98K keys.

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 5, where one can observe that BSL catches up with the performance of MTF lists much earlier. Yet, the initial gains of MTF lists suggest 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 .

To make sure that we are competitive with other dynamic table lookup methods, we compare the performance of BSL with simple binary tries. Figure 6 demonstrates that BSL outperforms tries on 64-bit keys throughout the whole range of biases we considered (for $n = 98K$); it also performs favorably on 32 and 48 bit keys for biases observed in practice. We are going to include performance comparisons of BSL with balanced binary trees in the full version of the paper.

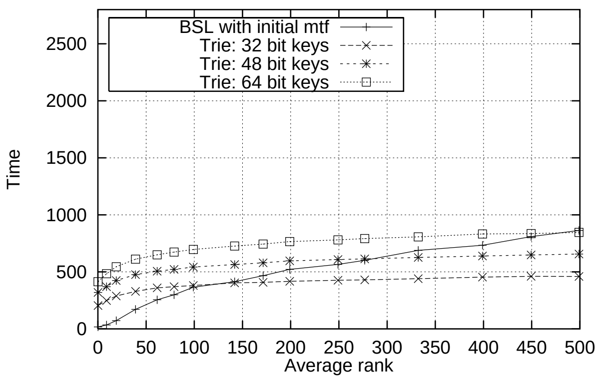

Fig. 6. Comparison of a hybrid BSL (with an MTF list of size 50) and binary trie performances on synthetic data with varying universe sizes. A total of 10^5 searches were performed on 98K keys.

Our final experiments compare the data structures mentioned above using the LBL trace data. For ease of implementation the flows are initially identified and mapped to

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

Table 1. Running times for the LBL trace data.

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 approximately 1.8 million packets and includes around 15 thousand insert operations. The average rank or the searched keys is approximately 25. The timing results of the data structures considered in our study on the LBL data are summarized in Table 1. Given the high bias of the data and the results shown by previous experiments, we would expect BSL to perform better when compared with skip lists. We explain this as follows. The total number of keys in the trace data build gradually to the relatively low total of around 15 thousand. This is favorable to the $O(\log n)$ search and insert times for skip lists. Additionally BSL has a higher overhead during insertion. We hope the modified BSL proposed in Section 3 will improve this. The results also show that MTF lists process the data quickest. However, if the data produced more failed searches the time would dramatically increase. For Internet applications in particular, this is not acceptable since it could be exploited in denial of service attacks. The hybrid BSL performs competitively, suggesting it may prove useful for applications that deal with biased data.

5 Conclusions

We introduced a new alternative to the data structures for maintaining dynamic tables, which is designed for biased access patterns. Its adaptive nature captures changes to the access distribution and keeps the search times efficient, even when the frequently accessed keys change. The improvements described in Section 3 offer fast insert and delete operations for applications where the lifetime of keys is short.

Preliminary experimental results suggest BSL may be a suitable choice for highly biased, dynamic data. The results show that BSL offers fast access to frequently accessed keys, while providing efficient access times for infrequently accessed ones. There are many parameters that affect the practical performance, such as the size of class C_1 and the length of the initial MTF list in the hybrid BSL. A more sophisticated implementation could dynamically choose the most suitable values based on the recent access patterns. This would further increase that adaptive nature of BSL and expand the range over which BSL is competitive. We leave such a study for future work. More experiments are required that compare BSL with the alternatives in order to fully evaluate its practicality. In addition, more tests using data from real applications are needed to help quantify any improvements.

References

1. L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. IEEE Infocom*, 1999.
2. S. Bent, D. Sleator, and R. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14, 1985.
3. J. L. Bentley, D. S. Sleator, R. E. Tarjan, and V.K. Wei. A locally adaptive data compression scheme. *Communication of the ACM*, 29:320–330, 1986.
4. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report*, 124, 1994.
5. 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.
6. G. Cheung and S. McCanne. Optimal routing table design for IP address lookups under memory constraints. In *IEEE INFOCOM*, 1999.
7. LBL-TCP-3 Trace Data. http://ita.ee.lbl.gov/html/contrib/lbl-tcp-3.html.
8. A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proc. ACM SIGMETRICS*, 1990.
9. Jon Sharp Funda Ergun, Cenk Sahinalp and Rakesh Sinha. Rapid fire and hire at top ranks: Move to front with faster updates. Technical report, Case Western Reserve University, 2000.
10. D. E. Knuth. *The Art of Computer Programming, volume 3 – 2nd ed.* Addison-Wesley, 1997.
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, Bell Laboratories, 1999.
13. C. Martinez and S. Roura. Optimal and nearly optimal static weighted skip lists. Technical report, Universitat Politecnica de Catalunya, 1995.
14. E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, 1993.
15. G. Manzini P. Ferragina. Opportunistic data structures with applications. In *Proc. IEEE Symposium on Foundations of Computer Science*, 2000.
16. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM*, 1990.
17. W. Pugh. Compressing java class files. In *Proc. ACM Sigplan Conference on Programming Language Design and Implementation*, 1999.
18. R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16, 1996.
19. D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32, 1985.
20. Andrew Tanenbaum. *Operating Systems: Design and Implementation, 2nd edition*. Prentice-Hall, 1997.
21. C. Leiserson T. Cormen, R. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.