Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond

Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, Zalán Heszberger
Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics
{retvari,tapolcai,korosi,majdan,heszi}@tmit.bme.hu

ABSTRACT

Lately, there has been an upsurge of interest in compressed data structures, aiming to pack ever larger quantities of information into constrained memory without sacrificing the efficiency of standard operations, like random access, search, or update. The main goal of this paper is to demonstrate how data compression can benefit the networking community, by showing how to squeeze the IP Forwarding Information Base (FIB), the giant table consulted by IP routers to make forwarding decisions, into information-theoretical entropy bounds, with essentially zero cost on longest prefix match and FIB update. First, we adopt the state-of-the-art in compressed data structures, yielding a static entropy-compressed FIB representation with asymptotically optimal lookup. Then, we re-design the venerable prefix tree, used commonly for IP lookup for at least 20 years in IP routers, to also admit entropy bounds and support lookup in optimal time and update in nearly optimal time. Evaluations on a Linux kernel prototype indicate that our compressors encode a FIB comprising more than 440K prefixes to just about 100–400 KBytes of memory, with a threefold increase in lookup throughput and no penalty on FIB updates.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Store and forward networks*; E.4 [**Coding and Information Theory**]: Data compaction and compression

Keywords

IP forwarding table lookup; data compression; prefix tree

1. INTRODUCTION

Data compression is widely used in processing large volumes of information. Not just that convenient compression tools are available to curtail the memory footprint of basically any type of data, but these tools also come with theoretical guarantee that the compressed size is indeed minimal, in terms of some suitable notion of *entropy* [8]. Correspondingly, data compression has found its use in basically all aspects of computation and networking practice, ranging from text or multimedia compression [62] to the very heart of communications protocols [58] and operating systems [3].

Traditional compression algorithms do not admit standard queries, like pattern matching or random access, right on the compressed form, which severely hinders their applicability. An evident workaround is to decompress the data prior to accessing it, but this pretty much defeats the whole purpose. The alternative is to maintain a separate index dedicated solely to navigate the content, but the sheer size of the index can become prohibitive in many cases [25, 39].

It is no surprise, therefore, that the discovery of compressed self-indexes (or, within the context of this paper, *compressed data structures*) came as a real breakthrough [18]. A compressed data structure is, loosely speaking, an entropy-sized index on some data that allows the complete recovery of the original content as well as fast queries on it [9, 18, 19, 25, 34, 38, 39, 42, 62]. What is more, as the compressed form occupies much smaller space than the original representation, and hence is more friendly to CPU cache, the time required to answer a query is often far less than if the data had not been compressed [9, 62]. *Compressed data structures, therefore, turn out one of the rare cases in computer science where there is no space-time trade-off.*

Researchers and practitioners working with big data were quick to recognize this win-win situation and came up with compressed self-indexes, and accompanying software tools, for a broad set of applications; from compressors for sequential data like bitmaps (`RRR`), [42]) and text documents (`CGlimpse` [18], wavelet trees [19]); compression frontends to information retrieval systems (`MG4J` [54], `LuceneTransform` [32]) and dictionaries (`MG` [57]); to specialized tools for structured data, like XML/HTML/DOM (`XGRIND` [52], `XBZIPINDEX` [17]), graphs (`WebGraph` [56]), 3D models (`Edgebreaker` [44]), genomes and protein sequences (`COMRi` [50]), multimedia, source and binary program code, formal grammars, etc. [57]. With the advent of replacements for the standard file compression tools (`LZgrep` [40]) and generic libraries (`libcds` [38]), today we might be right at the verge of seeing compressed data structures go mainstream.

Curiously, this revolutionary change has gone mostly unnoticed in the networking community, even though this field is just one of those affected critically by skyrocketing volumes of data. A salient example of this trend is the case of the IP Forwarding Information Base (FIB), used by Internet

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM or the author must be honored. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
Copyright 2013 ACM 978-1-4503-2056-6/13/08 ...$15.00.

routers to make forwarding decisions, which has been literally deluged by the rapid growth of the routed IP address space. Consequently, there has been a flurry of activity to find space-efficient FIB representations [1,5,10–13,21–23,27, 31,35,41,45,47,48,53,55,59,60], yet very few of these go beyond ad-hoc schemes and compress to information-theoretic limits, let alone come with a convenient notion of FIB entropy. Taking the example of compressing IP FIBs as a use case, thus, our aim in this paper is to popularize compressed data structures to the networking community.

1.1 FIB Compression

There are hardly any data structures in networking affected as compellingly by the growth of the Internet as the IP FIB. Stored in the line card memory of routers, the FIB maintains an association from every routed IP prefix to the corresponding next-hop, and it is queried on a packet-by-packet basis at line speed (in fact, it is queried *twice* per packet, considering reverse path forwarding check). Lookup in FIBs is not trivial either, as IP's longest prefix match rule requires the most specific entry to be found for each destination address. Moreover, as Internet routers operate in an increasingly dynamic environment [14], the FIB needs to support hundreds of updates to its content each second.

As of 2013, the number of active IPv4 prefixes in the Default Free Zone is more than 440,000 and counting, and IPv6 quickly follows suit [26]. Correspondingly, FIBs continue to expand both in size and management burden. As a quick reality check, the Linux kernel's `fib_trie` data structure [41], when filled with this many prefixes, occupies tens of Mbytes of memory, takes several minutes to download to the forwarding plane, and is still heavily debated to scale to multi-gigabit speeds [2]. Commercial routers suffer similar troubles, aggravated by the fact that line card hardware is more difficult to upgrade than software routers.

Many have argued that mounting FIB memory tax will, sooner or later, present a crucial data-plane performance bottleneck for IP routers [36]. But even if the scalability barrier will not prove impenetrable [16], the growth of the IP forwarding table still poses compelling difficulties. Adding further fast memory to line cards boosts silicon footprint, heat production, and power budget, all in all, the CAPEX/ OPEX associated with IP network gear, and forces operators into rapid upgrade cycles [30, 61]. Large FIBs also complicate maintaining multiple virtual router instances, each with its own FIB, on the same physical hardware [46] and build up huge control plane to data plane delay for FIB resets [20].

Several recent studies have identified *FIB aggregation* as an effective way to reduce FIB size, extending the lifetime of legacy network devices and mitigating the Internet routing scalability problem, at least temporarily [30, 61]. FIB aggregation is a technique to transform some initial FIB representation into an alternative form that, supposedly, occupies smaller space but still provides fast lookup. Recent years have seen an impressive reduction in FIB size: from the initial 24 bytes/prefix (prefix trees [45]), use of hash-based schemes [1, 55], path- and level-compressed multibit tries [5, 41, 48], tree-bitmaps [13], etc., have reduced FIB memory tax to just about 2–4.5 bytes/prefix [10, 53, 60]. Meanwhile, lookup performance has also improved [41].

The evident questions "Is there an ultimate limit in FIB aggregation?" and "Can FIBs be reduced to fit in fast ASIC SRAM/CPU cache entirely?" have been asked several times before [5, 10, 12, 48]. In order to answer these questions, we need to go beyond conventional FIB aggregation to find *new compressed FIB data structures that encode to entropy-bounded space and support lookup and update in optimal time*. We coined the term *FIB compression* to mark this ambitious undertaking [43]. Accordingly, this paper is dedicated to the theory and practice of FIB compression.

1.2 Our Contributions

Our contributions on FIB compression are two-fold: based on the labeled tree entropy measure of Ferragina *et al.* [17] we specify a compressibility metric called FIB entropy, then we propose two entropy-compressed FIB data structures.

Our first FIB encoder, *XBW-l*, is a direct application of the state-of-the-art in compressed data structures to the case of IP FIBs. *XBW-l* compresses a contemporary FIB to the entropy limit of just 100–300 Kbytes and, at the same time, provides longest prefix match in asymptotically optimal time. Unfortunately, it turns out that the relatively immature hardware and software background for compressed string indexes greatly constrain the lookup and update performance of *XBW-l*. Therefore, we also present a practical FIB compression scheme, called the trie-folding algorithm.

Trie-folding is in essence a "compressed" reinvention of prefix trees, a commonly used FIB implementation in IP routers, and therefore readily deployable with minimal or no modification to router ASICs [15]. We show that trie-folding compresses to within a small constant factor of FIB entropy, supports lookup in strictly optimal time, and admits updates in nearly optimal time for FIBs of reasonable entropy (see later for precise definitions). The viability of trie-folding will be demonstrated on a Linux prototype and an `FPGA` implementation. By extensive tests on real and synthetic IP FIBs, we show that trie-folding supports tens of millions of IP lookups and hundreds of thousands updates per second, in less than 150–500 Kbytes of memory.

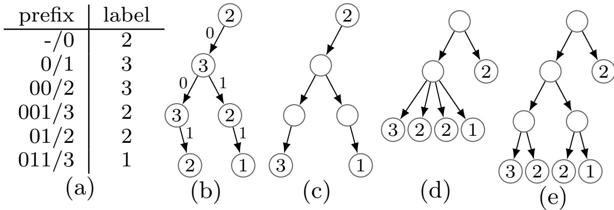
1.3 Structure of the Paper

The rest of the paper is organized as follows. In the next section, we survey standard FIB representation schemes and cast compressibility metrics. In Section 3 we describe *XBW-l*, while in Section 4 we introduce trie-folding and we establish storage size bounds. Section 5 is devoted to numerical evaluations and measurement results, Section 6 surveys related literature, and finally Section 7 concludes the paper.

2. PREFIX TREES AND SPACE BOUNDS

Consider the sample IP routing table in Fig. 1(a), storing address-prefix-to-next-hop associations in the form of an index into a neighbor table, which maintains neighbor specific information, like next-hop IP address, aliases, ARP info, etc. Associate a unique *label*, taken from the alphabet Σ, with each next-hop in the neighbor table. We shall usually treat labels as positive integers, complemented with a special *invalid label* $\perp \in \Sigma$ to mark blackhole routes. Let N denote the number of entries in the FIB and let $\delta = |\Sigma|$ be the number of next-hops. An IP router does not keep an adjacency with every other router in the Internet, thus $\delta \ll N$. Specifically, we assume that δ is $O(\text{polylog}\, N)$ or $O(1)$, which is in line with reality [6, 51]. Finally, let W denote the width of the address space in bits (e.g., $W = 32$ for IPv4).

To actually forward a packet, we need to find the entry that matches the destination address in the packet on the



prefix	label
-/0	2
0/1	3
00/2	3
001/3	2
01/2	2
011/3	1

Figure 1: Representations of an IP forwarding table: tabular form with address in binary format, prefix length and next-hop address label (a); prefix tree with state transitions marked (b); ORTC-compressed prefix tree (c); level-compressed multibit trie (d); and leaf-pushed trie (e).

greatest number of bits, starting from the MSB. For the address 0111, each of the entries $-/0$ (the default route), 0/1, 01/2, and 011/3 match. As the most specific entry is the last one, the lookup operation yields the next-hop label 1. This is then used as an index into the neighbor table and the packet is forwarded on the interface facing that neighbor. This tabular representation is not really efficient, as a lookup or update operation requires looping through each entry, taking $O(N)$ time. The storage size is $(W + \lg \delta)N$ bits[1].

Binary prefix trees, or *tries* [45], support lookup and update much more efficiently (see Fig. 1(b)). A trie is a labeled ordinal tree, in which every path from the root node to a leaf corresponds to an IP prefix and lookup is based on successive bits of the destination address: if the next bit is 0 proceed to the left sub-trie, otherwise proceed to the right, and if the corresponding child is missing return the last label encountered along the way. Prefix trees generally improve the time to perform a lookup or update from linear to $O(W)$ steps, although memory size increases somewhat.

A prefix tree representation is usually not unique, which opens the door to a wide range of optimization strategies to find more space-efficient forms. For instance, the prefix tree in Fig. 1(c) is *forwarding equivalent* with the one in Fig. 1(b), in that it orders the same label to every complete W bit long key, yet contains only 3 labeled nodes instead of 7 (see the ORTC algorithm in [12, 53]). Alternatively, *level-compression* [5, 41, 48] is a technique to remove excess levels from a binary trie to obtain a forwarding equivalent multibit trie that is substantially smaller (see Fig. 1(d)).

A standard technique to obtain *a unique, normalized form* of a prefix tree is *leaf-pushing* [12, 46, 48]: in a first preorder traversal labels are pushed from the parents towards the children, and then in a second postorder traversal each parent with identically labeled leaves is substituted with a leaf marked with the children's label (see Fig. 1(e)). The resultant trie is called a *leaf-labeled* trie since interior nodes no longer maintain labels, and it is also a *proper* binary trie with nice structure: any node is either a leaf node or it is an interior node with exactly two children. Updates to a leaf-pushed trie, however, may be expensive; modifying the default route, for instance, can result in practically all leaves being relabeled, taking $O(N)$ steps in the worst-case.

2.1 Information-theoretic Limit

How can we know for sure that a particular prefix tree representation, from the countless many, is indeed space-efficient? To answer this question, we need information-theoretically justified storage size bounds.

The first verifiable cornerstone of a space-efficient data structure is whether its size meets the *information-theoretic lower bound*, corresponding to the minimum number of bits needed to uniquely identify any instance of the data. For example, there are exactly δ^n strings of length n on an alphabet of size δ, and to be able to distinguish between any two we need at least $\lg(\delta^n) = n \lg \delta$ bits. In this example even a naive string representation meets the bound, but in more complex cases attaining it is much more difficult.

This argumentation generalizes from strings to leaf-labeled tries as follows (see also Ferragina *et al.* [17]).

Proposition 1. *Let T be a proper, binary, leaf-labeled trie with n leaves on an alphabet of size δ. The information-theoretic lower bound to encode T is $4n + n \lg \delta$ bits.*

The bound is easily justified with a simple counting argument. As there are $F_t = \binom{2t+1}{t}/(2t+1)$ proper binary trees on t nodes we need $\lg F_t = 2t - \Theta(\log t)$ bits to encode the tree itself [28]; storing the label map defined on the n leaves of T requires an additional $n \lg \delta$ bits; and assuming that the two are independent we need $2t + n \lg \delta$ bits overall. From here the claim follows, as for a proper binary tree $t = 2n - 1$.

A representation that encodes to within the constant factor of the information-theoretic lower bound (up to lower order terms) and simultaneously supports queries in optimal time is called a *compact data structure*, while if the constant factor is 1 then it is also a *succinct data structure* [28].

2.2 Entropy Bounds

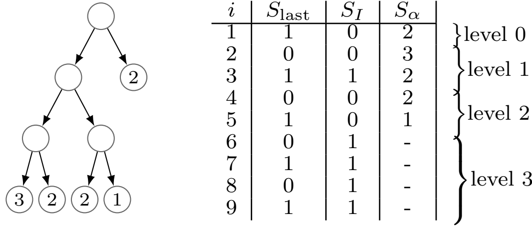
A succinct representation very often contains further redundancy in the form of regularity in the label mapping. For instance, in the sample trie of Fig. 1(e) there are three leaves with label 2, but only one with label 1 or 3. Thus, we could save space by representing label 2 on fewer bits, similarly to how Huffman-coding does for strings. This correspondence leads to the following notion of *entropy* for leaf-labeled tries (on the traces of Ferragina *et al.* [17]).

Proposition 2. *Let T be a proper, binary, leaf-labeled trie with n leaves on an alphabet Σ, let p_s denote the probability that some symbol $s \in \Sigma$ appears as a leaf label, and let H_0 denote the Shannon-entropy of the probability distribution $p_s, s \in \Sigma$: $H_0 = \sum_{s \in \Sigma} p_s \log_2 1/p_s$. Then, the zero-order entropy of T is $4n + nH_0$ bits.*

Intuitively speaking, the entropy of the tree structure corresponds to the information-theoretic limit of $4n$ bits as we do not assume any regularity in this regard. To this, the leaf-labels add an extra nH_0 bits of entropy.

The entropy of a trie depends mainly on the size of the underlying tree and the distribution of labels on it. This transforms to FIBs quite naturally: the more prefixes go to the same next-hop and the more the FIB resembles "a default route with few exceptions", the smaller the Shannon-entropy of the next-hop distribution and the tighter the space limit. Accordingly, we shall define the notions *FIB information-theoretic lower bound* and *FIB entropy* as those of the underlying leaf-pushed prefix tree. Both space bounds are well-defined as the normalized form is unique. Note, however, that in contrast to general trees IP FIBs are of bounded height, so the real bounds should be somewhat smaller. Additionally, for the purposes of this paper our space bounds

[1]The notation $\lg x$ is shorthand for $\lceil \log_2(x) \rceil$.



i	S_{last}	S_I	S_α
1	1	0	2
2	0	0	3
3	1	1	2
4	0	0	2
5	1	0	1
6	0	1	-
7	1	1	-
8	0	1	-
9	1	1	-

Figure 2: A leaf-pushed trie and its *XBW-l* transform.

involve binary leaf-labeled tries only. We relax this restriction in [43] using the generic trie entropy measure of [17].

3. ATTAINING ENTROPY BOUNDS

Our first compressed FIB data structure, the *Burrows-Wheeler transform for leaf-labeled tries* (*XBW-l*), is essentially a mix of the state-of-the-art in succinct level-indexed binary trees of Jacobson [28], succinct k-ary trees, and, most importantly, the XBW transform due to Ferragina *et al.* [17], with some FIB-specific twists (see also [43]).

The basis for the *XBW-l* transform is a normalized, proper, leaf-labeled trie. Accordingly, we assume that the FIB has already been leaf-pushed prior to being subjected to *XBW-l*. In addition, it may or may not have been level-compressed as well; this is completely orthogonal to *XBW-l* but we note that multibit tries will generally yield smaller transforms and faster lookup than binary tries. Herein we restrict ourselves to binary representations; investigation of the impact of level-compression is left for further study.

Let T be a tree on t nodes, let L be the set of leaves with $n = |L|$, and let l be a mapping $V \mapsto \Sigma$ specifying for a node v either that v does not have a label associated with it (i.e., $l(v) = \emptyset$) or the corresponding label $l(v) \in \Sigma$. If T is proper and leaf-labeled, then the following invariants hold:

P1: Either $v \in L$, or v has 2^k children for some $k > 0$.

P2: $l(v) \neq \emptyset \Leftrightarrow v \in L$.

P3: $t < 2n$ and so $t = O(n)$.

The main idea in *XBW-l* is serializing T into two bitstrings S_{last} and S_I that encode the tree structure and a string S_α on the alphabet Σ encoding the labels, and then using a sophisticated lossless string compressor to obtain the storage size bounds. The trick is in making just the right amount of context available to the string compressors, and doing this all with guaranteeing optimal lookup on the compressed form. Correspondingly, the *XBW-l* transform is defined as the tuple $\text{xbwl}(T) = (S_{\text{last}}, S_I, S_\alpha)$, where

- S_{last}: a bitstring of size t with 1 in position i if the i-th node of T in level-order is the last child of its parent, and zero otherwise;
- S_I: a bitstring of size t with zero in position i if the i-th node of T in level-order is an interior node and 1 otherwise; and
- S_α: a string of size n on the alphabet Σ encoding the leaf labels.

For our sample FIB, the leaf-pushed trie and the corresponding *XBW-l* transform are given in Fig. 2.

3.1 Construction and IP lookup

In order to generate the *XBW-l* transform, one needs to fill up the strings S_{last}, S_I, and S_α, starting from the root and traversing T in a breadth-first-search order.

```
i ← 0; j ← 0
BFS-TRAVERSE (node v, integer i, integer j)
    if v is the last child of its parent then
        S_last[i] ← 1 else S_last[i] ← 0
    if v ∉ L then S_I[i] ← 0
        else S_I[i] ← 1; S_α[j] ← l(v); j ← j + 1
    i ← i + 1
```

We assume that the root is "last": $S_{\text{last}}[0] = 1$. The following statement is now obvious.

LEMMA 1. *Given a proper leaf-labeled trie T on t nodes,* $\text{xbwl}(T)$ *can be built in $O(t)$ time.*

The transform $\text{xbwl}(T)$ has some appealing properties. For instance, the children of some node, if exist, are stored on consecutive indices in S_{last}, S_I, and S_α. In fact, *all* nodes at the same level of T are mapped to consecutive indices.

The next step is to actually compress the strings. This promises easier than compressing T directly as $\text{xbwl}(T)$, being a sequential string representation, lacks the intricate structure of tries. An obvious choice would be to apply some standard string compressor (like the venerable `gzip(1)` tool), but this would not admit queries like "get all children of a node" without first decompressing the transform. Thus, we rather use a compressed string self-index [17,19,28,42] to store $\text{xbwl}(T)$, which allows us to implement efficient navigation immediately on the compressed form.

The way string indexers usually realize navigability is to implement a certain set of simple primitives in constant $O(1)$ time in-place. Given a string $S[1,t]$ on alphabet Σ, a symbol $s \in \Sigma$, and integer $q \in [1,t]$, these primitives are as follows:

- $\text{access}(S,q)$: return the symbol at position q in S;
- $\text{rank}_s(S,q)$: return the number of times symbol s occurs in the prefix $S[1,q]$; and
- $\text{select}_s(S,q)$: return the position of the q-th occurrence of symbol s in S.

Curiously, these simple primitives admit strikingly complex queries to be implemented and supported in optimal time. In particular, the IP lookup routine on $\text{xbwl}(T)$ takes the following form.

```
lookup (address a)
    q ← 0, i ← 0
    while q < W
        if access(S_I, i) = 1 then
            return access(S_α, rank_1(S_I, i))
        r ← rank_0(S_I, i)
        f ← select_1(S_last, r) + 1
        l ← select_1(S_last, r + 1)
        j ← bits(a, q, log_2(l − f + 1))
        i ← f + j; q ← q + log_2(l − f + 1)
```

The code first checks if the actual node, encoded at index i in $\text{xbwl}(T)$, is a leaf node. If it is, then $\text{rank}_1(S_I, i)$ tells how many leaves were found in the course of the BFS-traversal *before* this one and then the corresponding label is returned from S_α (recall **P2**). If, on the other hand, the actual node is an interior node, then r tells how many interior nodes precede this one, f gets the index of the last child of the previous interior node plus 1, and l gets the index of the

last child of the actual node. Easily, $l - f + 1$ gives the number of children, and the logarithm (by **P1**) tells how many bits to read from a, starting at position q, to obtain the index j of the child to be visited next. Finally, we set the current index to $f + j$ and carry on with the recursion.

3.2 Memory Size Bounds

First, we show that *XBW-l* is a succinct FIB representation, in that it supports lookup in optimal $O(W)$ time and encodes to information-theoretic lower bound.

LEMMA 2. *Given a proper leaf-labeled trie T with n leaves on an alphabet of size δ,* $\text{xbwl}(T)$ *can be stored on $4n + n \lg \delta$ bits so that* `lookup` *on* $\text{xbwl}(T)$ *terminates in $O(W)$ time.*

PROOF. One can encode S_{last} and S_I on at most $2t \approx 4n$ bits using the `RRR` succinct bitstring index [42], which supports select and rank in $O(1)$. In addition, even the trivial encoding of S_α needs only another $n \lg \delta$ bits and provides access in $O(1)$. So every iteration of `lookup` takes constant time, which gives the result. □

Next, we show that *XBW-l* can take advantage of regularity in leaf labels (if any) and encode *below* the information-theoretic bound to zero-order entropy.

LEMMA 3. *Let T be a proper leaf-labeled trie with n leaves on an alphabet of size $O(\text{polylog}\, n)$, and let H_0 denote the Shannon-entropy of the leaf-label distribution. Then,* $\text{xbwl}(T)$ *can be stored on $4n + nH_0 + o(n)$ bits so that* `lookup` *on* $\text{xbwl}(T)$ *terminates in $O(W)$ time.*

PROOF. S_{last} and S_I can be encoded as above, and S_α can be stored on $nH_0 + o(n)$ bits using generalized wavelet trees so that access is $O(1)$, under the assumption that the alphabet size is $O(\text{polylog}\, n)$ [19]. □

Interestingly, the above *zero-order* entropy bounds can be easily upgraded to *higher-order* entropy. A fundamental premise in data compression is that elements in a data set often depend on their neighbors, and the larger the context the better the prediction of the element from its context and the more efficient the compressor. Higher-order string compressors can use the Burrows-Wheeler transform to exploit this contextual dependency, a reversible permutation that places symbols with similar context close to each other. This argumentation readily generalizes to leaf-labeled tries; simply, the *context of a node corresponds to its level in the tree* and because *XBW-l* organizes nodes at the same level (i.e., of similar context) next to each other, it realizes the same effect for tries as the Burrows-Wheeler transform for strings (hence the name). Deciding whether or not such contextual dependency is present in real IP FIBs is well beyond the scope of this paper. Here, we only note that if it is, then *XBW-l* can take advantage of this and compress an IP FIB to higher-order entropy using the techniques in [17, 43].

In summary, the *XBW-l* transform can be built fast, supports lookup in asymptotically optimal time, and compresses to within entropy bounds. Updates, however, may be expensive. Even the underlying leaf-pushed trie takes $O(n)$ steps in the worst-case to update, after which we could either rebuild the string indexes from scratch (again in $O(n)$) or use a dynamic compressed index that supports updates to the compressed form efficiently. For instance, [34] implements insertion and deletion in roughly $O(\log n)$ time, at the cost of slower rank and select. The other shortcoming of *XBW-l* is that, even if it supports lookups in *theoretically* optimal time, it is just too slow for line speed IP lookup (see Section 5.3). In the next section, therefore, we discuss a somewhat more practical FIB compression scheme.

4. PRACTICAL FIB COMPRESSION

The string indexes that underly *XBW-l* are *pointerless*, encoding all information in compact bitmaps. This helps squeezing *XBW-l* into higher-order entropy bounds but also causes that we need to perform multiple rank and select operations just to, say, enter a child of a node. And even though these primitives run in $O(1)$ the constants still add up, building up delays too huge for line speed IP lookup. In contrast, a traditional *pointer machine*, like a prefix tree, can follow a child pointer in just a single indirection with only one random memory access overhead. The next compressed FIB data structure we introduce is, therefore, pointer-based.

The idea is to essentially re-invent the classic prefix tree, borrowing the basic mechanisms from the Lempel-Ziv (`LZ78`) string compression scheme [8]. `LZ78` attains entropy by parsing the text into unique sub-strings, yielding a form that contains no repetitions. Tree threading is a generalization of this technique to *unlabeled trees*, converting the tree into a Directed Acyclic Graph (DAG) by merging isomorphic subtrees [27, 29, 46, 49]. In this paper, we apply this idea to *labeled trees*, merging sub-tries taking into account both the underlying tree structure *and* the labels [4, 7]. If the trie is highly regular then this will eliminate all recurrent substructures, producing a representation that contains no repetitions and hence, so the argument goes, admits entropy bounds like `LZ78`. Consider the below equivalence relation.

DEFINITION 1. *Two leaf-labeled tries are identical if each of their sub-tries are pairwise identical, and two leaves are identical if they hold the same label.*

We call the algorithmic manifestation of this recursion the *trie-folding algorithm* and the resultant representation a *prefix DAG*. Fig. 3(a) depicts a sample prefix tree, Fig. 3(b) shows the corresponding leaf-pushed trie, and Fig. 3(c) gives the prefix DAG. For instance, the sub-tries that belong to the prefix 0/1 and 11/2 are equivalent in the leaf-pushed trie, and thusly merged into a single sub-trie that is now available in the prefix DAG along two paths from the root. This way, the prefix DAG is significantly smaller than the original prefix tree as it contains only half the nodes.

For the trie-folding algorithm it is again irrelevant whether the trie is binary or level-compressed. What is essential, however, is that it be normalized; for instance, in our example it is easy to determine from the leaf-pushed trie that the two sub-tries under the prefixes 00/2 and 10/2 are identical, but this is impossible to know from the original prefix tree. Thus, leaf-pushing is essential to realize good compression but, at the same time, makes updates prohibitive [46].

To avoid this problem, we apply a simple optimization. We separate the trie into two parts; "above" a certain level λ, called the *leaf-push barrier*, where sub-tries are huge and so common sub-tries are rare, we store the FIB as a standard binary prefix tree in which update is fast; and "below" λ, where common sub-tries most probably turn up, we apply leaf-pushing to obtain good compression. Then, by a cautious setting of the leaf-push barrier we simultaneously realize fast updates and entropy-bounded storage size.

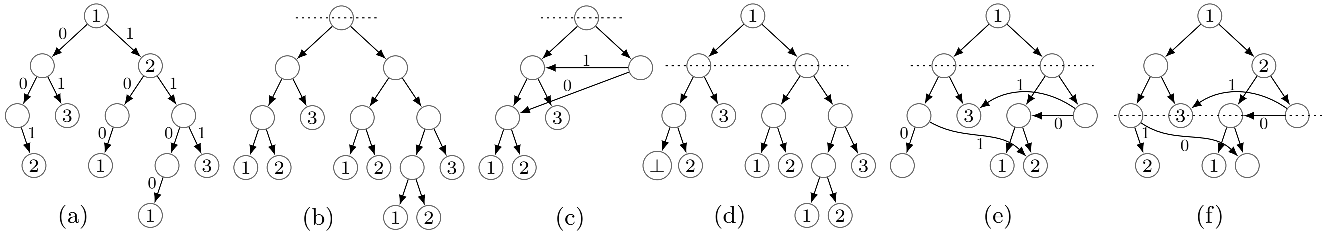


Figure 3: A binary trie for a sample FIB (a); the same trie when leaf-pushing is applied from level $\lambda = 0$ (b); the prefix DAG for leaf-push barrier $\lambda = 0$ (c); the trie (d) and the prefix DAG (e) for $\lambda = 1$; and the prefix DAG for $\lambda = 2$ (f). Dashed lines indicate the leaf-push barrier λ and the invalid label $\perp$ was removed from the corresponding leaf nodes in the prefix DAGs.

The prefix DAGs for $\lambda = 1$ and $\lambda = 2$ are depicted in Fig. 3(e) and 3(f). The size is somewhat larger, but updating, say, the default route now only needs setting the root label without having to cycle through each leaf.

4.1 Construction and IP lookup

We are given a binary trie T (not necessarily proper and leaf-pushed) of depth W, labeled from an alphabet Σ of size δ. Let V_T ($|V_T| = t$) be the set of nodes and L_T ($|L_T| = n$) be the set of leaves. Without loss of generality, we assume that T does not contain explicit blackhole routes. Then, the trie-folding algorithm transforms T into a prefix DAG $D(T)$, on nodes V_D and leaves L_D, with respect to the leaf-push barrier $\lambda \in [0, W]$.

The algorithm is a simple variant of trie threading [29]: assign a unique id to each sub-trie that occurs below level λ and merge two tries if their ids are equal (as of Definition 1). The algorithm actually works on a copy of T and always keeps an intact instance of T available. This instance, called the control FIB, can exist in the DRAM of the line card's control CPU, as it is only consulted to manage the FIB. The prefix DAG itself is constructed in fast memory. We also need two ancillary data structures, the leaf table and the sub-trie index, which can also live in DRAM.

The *leaf table* will be used to coalesce leaves with identical labels into a common leaf node. Accordingly, for each $s \in \Sigma$ the leaf table $\mathrm{lp}(s)$ stores a leaf node (no matter which one) with that label. Furthermore, the *sub-trie index* $\mathcal{S}$ will be used to identify and share common sub-tries. $\mathcal{S}$ is in fact a reference counted associative array, addressed with pairs of ids $(i, j) \in \mathbb{N} \times \mathbb{N}$ as keys and storing for each key a node whose children are exactly the sub-tries identified by i and j. $\mathcal{S}$ supports the following primitives:

- `put`(i, j, v): if a node with key (i, j) exists in $\mathcal{S}$ then increase its reference count and return it, otherwise generate a new id in v.id, store v at key (i, j) with reference count 1, and return v; and
- `get`(i, j): dereference the entry with key (i, j) and delete it if the reference count drops to zero.

In our code we used a hash to implement $\mathcal{S}$, which supports the above primitives in amortized $O(1)$ time.

Now, suppose we are given a node v to be subjected to trie-folding and a leaf-push barrier λ. First, for each descendant u of v at depth λ we normalize the sub-trie rooted at u using label $l(u)$ as a "default route", and then we call the compress routine to actually merge identical leaves and sub-tries below u, starting from the bottom and working upwards until we reach u. Consider the below pseudo-code for the main `trie_fold` routine.

`trie_fold` (node v, integer λ)
    for each λ-level child u of v **do**
        if $l(u) = \emptyset$ **then** leaf_push$(u, \perp)$ **else** leaf_push$(u, l(u))$
        POSTORDER-TRAVERSE-AT-u(`compress`)
    $l(\mathrm{lp}(\perp)) \leftarrow \emptyset$
`compress` (node w)
    if $w \in L_D$ **then** w.id $= l(w)$; $u \leftarrow \mathrm{lp}(w)$
        else $u =$ `put`$(w$.left.id$, w$.right.id$, w)$
    if $u \neq w$ **then** re-pointer the parent of w to u; delete(w)

Here, w.left is the left child and w.right is the right child for w, and w.id is the id of w. As `trie_fold` visits each node at most twice and `compress` runs in $O(1)$ if `put` is $O(1)$, we arrive to the following conclusion.

LEMMA 4. *Given a binary trie T on t nodes, $D(T)$ can be constructed in $O(t)$ time.*

Lookup on a prefix DAG goes exactly the same way as on a conventional prefix tree: follow the path traced out by the successive bits of the lookup key and return the last label found. We need to take care of a subtlety in handling invalid labels, though. Namely, in our sample FIB of Fig. 3(a), the address 000 is associated with label 1 (the default route), which in the prefix DAG for $\lambda = 1$ (Fig. 3(e), derived from the trie on Fig. 3(d)) would become $\perp$ if we were to let leaf nodes' $\perp$ labels override labels inherited from levels above λ. This problem is easy to overcome, though, by removing the label from the leaf $\mathrm{lp}(\perp)$. By our assumption the FIB contains no explicit blackhole routes, thus every traversal yielding the empty label on T terminates in $\mathrm{lp}(\perp)$ on $D(T)$ and, by the above modification, also gives an empty label.

That being said, the last line of the `trie_fold` algorithm renders standard trie lookup correct on prefix DAGs. Since this is precisely the lookup algorithm implemented in many IP routers on the ASIC [15], we conclude that prefix DAGs can serve as compressed drop-in replacements for trie-based FIBs in many router products (similarly to e.g., [53]).

The following statement is now obvious.

LEMMA 5. *The* `lookup` *operation on $D(T)$ terminates in $O(W)$ time.*

In this regard, trie-folding can be seen as a generalization of conventional FIB implementations: for $\lambda = 32$ we get good old prefix trees, and for smaller settings of λ we obtain *increasingly smaller FIBs with exactly zero cost on lookup efficiency.* Correspondingly, there is no memory size vs. lookup complexity "space-time" trade-off in trie-folding.

4.2 Memory Size Bounds

The tries that underlie *XBW-l* are proper and leaf-labeled, and the nice structure makes it easy to reason about the

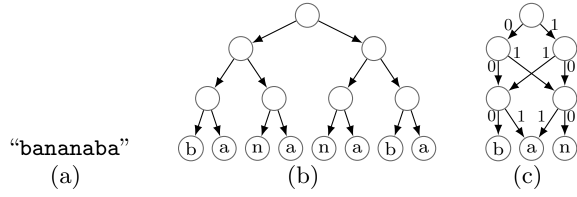


Figure 4: Trie-folding as string compression: a string (a), the complete binary trie (b), and the compressed DAG (c). The third character of the string can be accessed by looking up the key $3-1=010_2$.

size thereof. Unfortunately, the tries that prefix DAGs derive from are of arbitrary shape and so it is difficult to infer space bounds in the same generic sense. We chose a different approach, therefore, in that we *view trie-folding as a generic string compression method and we compare the size of the prefix DAG to that of an input string* given to the algorithm. The space bounds obtained this way transform to prefix trees naturally, as trie entropy itself is also defined in terms of string entropy (recall Proposition 2).

Instead of being given a trie, therefore, we are now given a string S of length n on an alphabet of size δ and zero-order entropy H_0. Supposing that n equals some power of 2 (which we do for the moment), say, $n=2^W$, we can think as if the symbols in S were written to the leaves of a complete binary tree of depth W as labels. Then, trie-folding will convert this tree into a prefix DAG $D(S)$, and we are curious as to how the size of $D(S)$ relates to the information-theoretic limit for storing S, that is, $n \lg \delta$, and the zero order entropy nH_0 (see Fig. 4). Note that every FIB has such a "complete binary trie" representation, and vice versa.

The memory model for storing the prefix DAG is as follows. Above the leaf-push barrier λ we use the usual trick that the children of a node are arranged on consecutive memory locations [41], and so each node holds a single node pointer of size to be determined later, plus a label index of $\lg \delta$ bits. At and below level λ nodes hold two pointers but no label, plus we also need an additional $\delta \lg \delta$ bits to store the coalesced leaves.

Now, we are in a position to state the first space bound. In particular, we show that $D(S)$ attains information-theoretic lower bound up to some small constant factor, and so it is a *compact data structure.*

Theorem 1. *Let S be a string of length $n=2^W$ on an alphabet of size δ and set the leaf-push barrier as*

$$\lambda = \left\lfloor \frac{1}{\ln 2} \mathcal{W}\left(n \lg(\delta) \ln 2\right) \right\rfloor , \tag{1}$$

where $\mathcal{W}()$ denotes the Lambert $\mathcal{W}$-function. Then, $D(S)$ can be encoded on at most $5\lg(\delta)n + o(n)$ bits.

Note that the Lambert $\mathcal{W}$-function (or product logarithm) $\mathcal{W}(z)$ is defined as $z=\mathcal{W}(z)e^{\mathcal{W}(z)}$. The detailed proof, based on a counting argument, is deferred to the Appendix.

Next, we show that trie-folding compresses to within a constant factor of the zero-order entropy bound, subject to some reasonable assumptions on the alphabet.

Theorem 2. *Let S be a string of length n and zero-order entropy H_0, and set the leaf-push barrier as*

$$\lambda = \left\lfloor \frac{1}{\ln 2} \mathcal{W}(nH_0 \ln 2) \right\rfloor . \tag{2}$$

Then, the expected size of $D(S)$ is at most $(7 + 2\lg\frac{1}{H_0} + 2\lg\lg\delta)H_0 n + o(n)$ bits.

Again, refer to the Appendix for the proof.

It turns out that the compression ratio depends on the specifics of the alphabet. For reasonable δ, say, $\delta = O(1)$ or $\delta = O(\text{polylog}\, n)$, the error $\lg\lg\delta$ is very small and the bound gradually improves as H_0 increases, to the point that at maximum entropy $H_0 = \lg\delta$ we get precisely $7H_0 n$. For extremely small entropy, however, the error $2\lg\frac{1}{H_0}$ can become dominant as the overhead of the DAG outweighs the size of the very string in such cases.

4.3 Update

What remained to be done is to set the leaf-push barrier λ in order to properly balance between compression efficiency and update complexity. Crucially, small storage can only be attained if the leaf-push barrier is chosen according to (2). Strikingly, we found that precisely this setting is the key to fast FIB updates as well[2].

Herein, we only specify the `update` operation that changes an *existing* association for prefix a of prefix length p to the new label s. Adding a new entry or deleting an existing one can be done in similar vein.

```
update (address a, integer p, label s, integer λ)
    v ← D(T).root; q ← 0
    while q < p
        if q ≥ λ then v ← decompress(v)
        if bits(a, q, 1) = 0 then v ← v.left else v ← v.right
        q ← q + 1
    if p < λ then l(v) ← s; return
    w ← T.copy(v); re-pointer the parent of v to w; l(w) ← s
    POSTORDER-TRAVERSE-AT-v(u: get(u.left.id, u.right.id))
    trie_fold(w, 0)
    for each parent u of w: level(u) ≥ λ do compress(u)
decompress (node v)
    w ← new_node; w.id ← v.id
    if v ∈ L_D then l(w) ← l(v)
        else w.left ← v.left; w.right ← v.right
            get(v.left.id, v.right.id)
    re-pointer the parent of v to w; return w
```

First, we walk down and decompress the DAG along the path traced out by the successive bits of a until we reach level p. The `decompress` routine copies a node out from the DAG and removes the reference from $\mathcal{S}$ wherever necessary. At this point, if $p < \lambda$ then we simply update the label and we are ready. Otherwise, we replace the sub-trie below v by a new copy of the corresponding sub-trie from T, taking care of calling `get` on the descendants of v to remove dangling references, and we set the label on the root w of the new copy to s. Then, we re-compress the portions of the prefix DAG affected by the change, by calling `trie_fold` on w and then calling `compress` on all the nodes along the upstream path from w towards to root.

Theorem 3. *If the leaf-push barrier λ is set as* (2)*, then* `update` *on $D(T)$ terminates in $O(W(1+\frac{1}{H_0}))$ time.*

Proof. If $p < \lambda$, then updating a single entry can be done in $O(W)$ time. If, on the other hand, $p \geq \lambda$, then `update` visits at most $W + 2^{W-\lambda} \leq W + \frac{W}{H_0}$ nodes, using that $\lambda \geq W - \lg(\frac{W}{H_0})$ whenever λ is as (2). □

In summary, under mild assumptions on the label distribution a prefix DAG realizes the Shannon-entropy up to a

[2]Note that (2) transforms into (1) at maximum entropy.

small factor and allows indexing arbitrary elements and updates to any entry in roughly $O(\log n)$ time. As such, it is in fact a dynamic, entropy-compressed string self-index. As far as we are aware of, this is the first pointer machine of this kind, as the rest of compressed string-indexes are pointerless. Regrettably, both the space bound and the update complexity weaken when the label distribution is extremely biased, i.e., when H_0 is very small. As we argue in the next section though, this rarely causes problems in practice.

5. NUMERICAL EVALUATIONS

At this point, we have yet to demonstrate that the appealing theoretical properties of compressed FIBs indeed manifest as practical benefits. For this reason, we conducted a series of numerical evaluations with the goal to quantify the compressibility of real IP FIBs and see how our compressors fare. It was *not* our intention, however, to compare to other FIB storage schemes from the literature, let alone evince that ours is the fastest or the most efficient one. After all, information-theoretic space bounds are purposed precisely at making such comparisons unnecessary, serving as analytically justified ground truth. Instead, our motivation is merely to demonstrate that FIB compression allows to reduce memory tax without any compromise on the efficiency of longest prefix match or FIB updates.

For the evaluations, we coded up a full-fledged Linux prototype, where FIB compression and update run in user space and IP lookup is performed by a custom kernel module embedded in the kernel's IP stack. The code executed on a single core of a 2.50GHz Intel Core i5 CPU, with 2x32 Kbyte L1 data cache, 256 Kbyte L2 cache, and 3 Mbyte L3 cache.

Research on IP FIB data structures has for a long time been plagued by the unavailability of real data, especially from the Internet core. Alas, we could obtain only 5 FIB instances from real IP routers, each from the access: `taz` and `hbone` are from a university access, `access(d)` is from a default and `access(v)` from a virtual instance of a service provider's router, and `mobile` is from a mobile operator's access (see Table 1). The first 3 are in the DFZ, the rest contain default routes. Apart from these, however, what is available publicly is RIB dumps from BGP collectors, like RouteViews or looking glass servers (named `as*` in the data set). Unfortunately, these only very crudely model real FIBs, because collectors run the BGP best-path selection algorithm on their peers and these adjacencies differ greatly from real next hops on production routers. We experimented with heuristics to restore the original next-hop information (e.g., set next-hop to the first AS-hop), but the results were basically the same. Thus, these FIBs are included in the data set only for reference. We also used two randomly generated FIBs, one of 600,000 (`fib_600k`) and another of 1 million prefixes (`fib_1m`), to future-proof our results. These synthetic FIBs were generated by iterative random prefix splitting and setting next-hops according to a truncated Poisson-distribution with parameter $\frac{3}{5}$ ($H_0 = 1.06$, $\delta = 4$).

5.1 Update Complexity

First, we set out to determine a good setting for the leaf-push barrier λ. Recall that λ was introduced to balance between the compression efficiency and update complexity (also recall that no such compromise exists between compression and *lookup*.). Our theoretical results provide the essential pointers to set λ (see (1) and (2)), but these are for compressing strings over complete binary trees. IP FIBs, however, are not complete.

We exercised the memory footprint vs. update complexity trade-off by varying λ between 0 and 32. The update time was measured over two update sequences: a random one with IP prefixes uniformly distributed on $[0, 2^{32} - 1]$ and prefix lengths on $[0, 32]$, and a BGP-inspired one corresponding to a real BGP router log taken from RouteViews. Here, we treated all BGP prefix announcements as generating a FIB update, with a next-hop selected randomly according to the next-hop distribution of the FIB. The results are mean values over 15 runs of $7,500$ updates, each run roughly corresponding to 15 minutes worth of BGP churn.

Herein, we only show the results for the `taz` FIB instance in Fig. 5. The results suggest that standard prefix trees (reproduced by the setting $\lambda = 32$), while pretty fast to update, occupy a lot of space. Fully compressed DAGs ($\lambda = 0$), in contrast, consume an order of magnitude less space but are expensive to modify. There is a domain, however, at around $5 \leq \lambda \leq 12$, where we win essentially all the space reduction and still handle about $100,000$ updates per second (that's roughly two and a half hours of BGP update load). What is more, *the space-time trade-off only exists for the synthetic, random update sequence, but not for BGP updates.* This is because BGP updates are heavily biased towards longer prefixes (with a mean prefix length of 21.87), which implies that the size of leaf-pushed sub-tries needed to be re-packed per update is usually very small, and hence update complexity is relatively insensitive to λ.

Based on these considerations, we set $\lambda = 11$ for the rest of the evaluations.

5.2 Storage Size

Storage size results are given in Table 1. Notably, real FIBs that contain only a few next-hops compress down to about 100–150 Kbytes with *XBW-l* at 2–4 bit/prefix(!) efficiency, and only about two times more with trie-folding. This is chiefly attributed to the small next-hop entropy, indicating the presence of a dominant next-hop. Core FIBs, on the other hand, exhibit considerably larger next-hop entropy, with *XBW-l* transforms in the range of 200–400 and prefix DAGs in 330–700 KBytes. Recall, however, that these FIBs exhibit unrealistic next-hop distribution. Curiously, even the extremely large FIB of 1 million prefixes shrinks below 500 Kbytes (800 KBytes with trie-folding). In contrast, small instances compress poorly, as it is usual in data compression. Finally, we observe that many *FIBs show high next-hop regularity* (especially the real ones), reflected in the fact that entropy bounds are 20–40% smaller than the information-theoretic limit. *XBW-l very closely matches entropy bounds, with trie-folding off by only a small factor.*

We also studied compression ratios on synthetic FIBs, whose entropy was controlled by us. In particular, we re-generated the next-hops in `access(d)` according to Bernoulli-distribution: a first next-hop was set with probability p and another with probability $1 - p$. Then, varying p in $[0, \frac{1}{2}]$ we observed the FIB entropy, the size of the prefix DAG, and the compression efficiency ν, i.e., the factor between the two (see Fig. 6). We found that *the efficiency is around* 1.7 *and*, in line with our theoretical analysis, *degrades as the next-hop distribution becomes extremely biased*. This, however, never occurs in reality (see again Table 1). We repeated

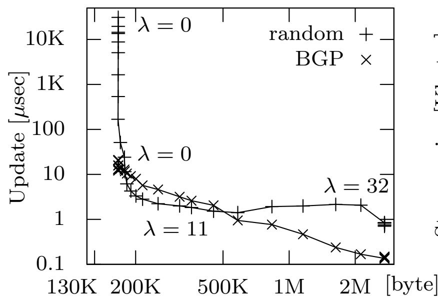


Figure 5: Update time vs. memory footprint on `taz` for random and BGP update sequences.

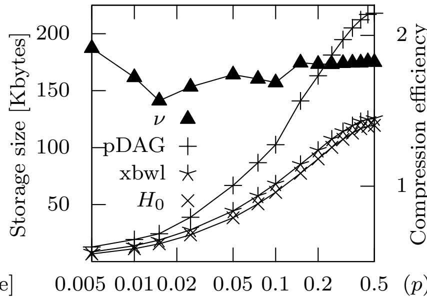


Figure 6: Size and compression efficiency ν over FIBS with Bernoulli distributed next-hops as the function of parameter p.

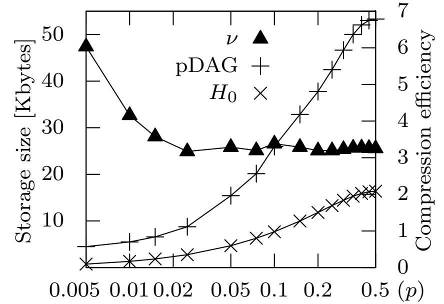


Figure 7: Size and compression efficiency ν over strings with Bernoulli distributed symbols as the function of parameter p.

the analysis in the string compression model: here, the FIB was generated as a complete binary trie with a string of 2^{17} symbols written on the leaves, again chosen by a Bernoulli distribution, and this was then compressed with trie-folding (see Fig. 7, with *XBW-l* omitted). The observations are similar, with compression efficiency closer to 3 now and the spike at low entropy more prominent.

5.3 Lookup Complexity

Finally, we tested IP lookup performance on real software and hardware prototypes. Our software implementations run inside the Linux kernel's IP forwarding engine. For this, we hijacked the kernel's network stack to send IP lookup requests to our custom kernel module, working from a serialized blob generated by the FIB encoders. Our *XBW-l* lookup engine uses a kernel port of the `RRR` bitstring index [42] and the Huffman-shaped `WaveletTree` [19] from `libcds` [38]. Trie-folding was coded in pure `C`. We used the standard trick to collapse the first $\lambda = 11$ levels of the prefix DAGs in the serialized format [60], as this greatly eases implementation and improves lookup time with practically zero effect on updates. We also experimented with the Linux-kernel's stock `fib_trie` data structure, an adaptive level- and path-compressed multibit trie-based FIB, as a reference implementation [41]. Last, we also realized the prefix DAG lookup algorithm in hardware, on a Xilinx Virtex-II Pro 50 FPGA with 4.5 MBytes of synchronous SRAM representing the state-of-the-art almost 10 years ago. The hardware implementation uses the same serialized prefix DAG format as the software code. All tests were run on the `taz` instance.

For the software benchmarks we used the standard Linux network micro-benchmark tool `kbench` [37], which calls the FIB lookup function in a tight loop and measures the execution time with nanosecond precision. We modified `kbench` to take IP addresses from a uniform distribution on $[0, 2^{32} - 1]$ or, alternatively, from a packet trace in the "CAIDA Anonymized Internet Traces 2012" data set [24]. The route cache was disabled. We also measured the rate of CPU cache misses by monitoring the `cache-misses` CPU performance counter with the `perf(1)` tool. For the hardware benchmark, we mirrored `kbench` functionality on the FPGA, calling the lookup logic repeatedly on a list of IP addresses statically stored in the SRAM and we measured the number of clock ticks needed to terminate the test cycle.

The results are given in Table 2. On the software side, the most important observations are as follows. The prefix DAG, taking only about 180 KBytes of memory, is most of the time accessed from the cache, while `fib_trie` occupies an impressive 26 MBytes and so it does not fit into fast memory. Thus, even though the number of memory accesses needed to execute an IP lookup is smaller with `fib_trie`, as most of these go to slower memory the prefix DAG supports about three times as many lookups per second. Accordingly, *not just that FIB space reduction does not ruin lookup performance, but it even improves it.* In other words, there is no space-time trade-off involved here. The address locality in real IP traces helps `fib_trie` performance to a great extent, as `fib_trie` can keep lookup paths to popular prefixes in cache. In contrast, the prefix DAG is pretty much insensitive to the distribution of lookup keys. Finally, we see that *XBW-l* is a distant third from the tested software lookup engines, suggesting that the constant in the lookup complexity is indeed prohibitive in practice and that our lookup code exercises some pretty pathologic code path in `libcds`.

The real potential of trie-folding is most apparent with our hardware implementation. The FPGA design executes a single IP lookup in just 7.1 clock cycles on average, thanks to that the prefix DAG fits nicely into the SRAM running synchronously with the logic. This is enough to roughly 7 million IP lookups per second even on our rather ancient FPGA board. On a modern FPGA or ASIC, however, with clock rates in the gigahertz range, our results indicate that prefix DAGs could be scaled to hundreds of millions of lookups per second at a terabit line speed.

We also measured packet throughput using the `udpflood` macro-benchmark tool [37]. This tool injects UDP packets into the kernel destined to a dummy network device, which makes it possible to run benchmarks circumventing network device drivers completely. The results were similar as above, with prefix DAGs supporting consistently 2–3 times larger throughput than `fib_trie`.

6. RELATED WORKS

In line with the unprecedented growth of the routed Internet and the emerging scalability concerns thereof [26,30,61], finding efficient FIB representations has been a heavily researched question in the past and, judging from the substantial body of recent work [22,31,53,60], still does not seem to have been solved completely.

Trie-based FIB schemes date back to the BSD kernel implementation of Patricia trees [45]. This representation consumes a massive 24 bytes per node, and a single IP lookup

Table 1: Results for *XBW-l* and trie-folding on *access*, *core*, and synthetic (*syn.*) FIBs: name, number of prefixes N and next-hops δ; Shannon-entropy of the next-hop distribution H_0; FIB information-theoretic limit I, entropy E, and *XBW-l* and prefix DAG size (pDAG, $\lambda = 11$) in KBytes; compression efficiency ν; and bits/prefix efficiency for *XBW-l* ($\eta_{XBW\text{-}l}$) and trie-folding (η_{pDAG}).

	FIB	N	δ	H_0	I	E	*XBW-l*	pDAG	ν	$\eta_{XBW\text{-}l}$	η_{pDAG}
access	`taz`	410,513	4	1.00	113	94	106	178	1.90	2.06	3.47
	`hbone`	410,454	195	2.00	356	213	230	396	1.85	4.47	7.71
	`access(d)`	444,513	28	1.06	236	149	168	370	2.47	3.02	6.65
	`access(v)`	2,986	3	1.22	4.1	3.6	4.1	7.5	2.09	11.07	20.23
	`mobile`	21,783	16	1.08	0.9	0.7	1.3	3.6	5.28	0.50	1.35
core	`as1221`	440,060	3	1.54	196	181	186	331	1.82	3.38	6.02
	`as4637`	219,581	3	1.12	79	67	74	129	1.91	2.71	4.69
	`as6447`	445,016	36	3.91	375	371	384	748	2.01	6.91	13.45
	`as6730`	437,378	186	2.98	421	293	309	545	1.85	5.65	9.96
syn.	`fib_600k`	600,000	5	1.06	309	261	296	462	1.77	3.95	6.16
	`fib_1m`	1,000,000	5	1.06	513	433	491	782	1.80	3.92	6.26

Table 2: Lookup benchmark with *XBW-l*, prefix DAGs, `fib_trie`, and the FPGA implementation on `taz`: size, average and maximum depth; and million lookup per second, lookup time in CPU cycles, and cache misses per packet over random IP addresses (rand.) and addresses taken from the trace [24] (trace).

		Linux			HW
		XBW-l	pDAG	`fib_trie`	`FPGA`
	size [Kbyte]	106	178	26,698	178
	average depth	–	3.7	2.42	–
	maximum depth	–	21	6	–
rand.	million lookup/sec	0.033	12.8	3.23	6.9
	CPU cycle/lookup	73940	194	771	7.1
	cache miss/packet	0.016	0.003	3.17	–
trace	million lookup/sec	0.037	13,8	5.68	6.9
	CPU cycle/lookup	67200	180	438	7.1
	cache miss/packet	0.016	0.003	0.29	–

might cost 32 random memory accesses. Storage space and search time can be saved on by expanding nodes' strides to obtain a multibit trie [5], see e.g., controlled prefix expansion [27,48], level- and path-compressed tries [41], Lulea [10], Tree Bitmaps [13] and successors [1, 47], etc. Another approach is to shrink the routing table itself, by cleverly relabeling the tree to contain the minimum number of entries (see ORTC and derivatives [12,53]). In our view, trie-folding is complementary to these schemes, as it can be used in combination with basically any trie-based FIB representation, realizing varying extents of storage space reduction.

Further FIB representations include hash-based schemes [1, 55], dynamic pipelining [23], CAMs [35], Bloom-filters [11], binary search trees and search algorithms [21,60], massively parallelized lookup engines [22, 60], FIB caching [31], and different combinations of these (see the text book [59]). None of these comes with information-theoretic space bounds. Although next-hop entropy itself appears in [53], but no analytical evaluation ensued. In contrary, *XBW-l* and trie-folding come with *theoretically proven* space limits, and thus *predicable memory footprint*. The latest reported FIB size bounds for >400K prefixes range from 780 KBytes (DXR, [60]) to 1.2 Mbytes (SMALTA, [53]). *XBW-l* improves this to just 100–300 Kbytes, which easily fits into today's SRAMs or can be realized right in chip logic with modern FPGAs.

Compressed data structures have been in the forefront of theoretical computer science research [9, 18, 19, 25, 34, 38, 39, 42, 62], ever since Jacobson in his seminal work [28] defined succinct encodings of trees that support navigational queries in optimal time within information-theoretically limited space. Jacobson's bitmap-based techniques later found important use in FIB aggregation [1, 13, 47]. With the extensive use of bitmaps, *XBW-l* can be seen as a radical rethinking of these schemes, inspired by the state-of-the-art in succinct and compressed data structures.

The basic idea of folding a labeled tree into a DAG is not particularly new; in fact, this is the basis of many tree compacting schemes [29], space-efficient ordered binary decision diagrams and deterministic acyclic finite state automaton [4], common subexpression elimination in optimizing compilers [7], and it has also been used in FIB aggregation [27, 46, 49] earlier. Perhaps the closest to trie-folding is Shape graphs [46], where common sub-trees, without regard to the labels, are merged into a DAG. However, this necessitates storing a giant hash for the next-hops, making updates expensive especially considering that the underlying trie is leaf-pushed. Trie-folding, in contrast, takes labels into account when merging and also allows cheap updates.

7. CONCLUSIONS

With the rapid growth of the Web, social networks, mobile computing, data centers, and the Internet routing ecosystem as a whole, the networking field is in a sore need of compact and efficient data representations. Today's networking practice, however, still relies on ad-hoc and piecemeal data structures for basically all storage sensitive and processing intensive applications, of which the case of IP FIBs is just one salient example.

Our main goal in this paper was to advocate compressed data structures to the networking community, pointing out that space reduction does not necessarily hurt performance. Just the contrary: the smaller the space the more data can be squeezed into fast memory, leading to faster processing. This lack of space-time trade-off is already exploited to a great extent in information retrieval systems, business analytics, computational biology, and computational geometry, and we believe that it is just too appealing not to be embraced in networking as well. This paper is intended as a first step in that direction, demonstrating the basic information-theoretic and algorithmic techniques needed to attain entropy bounds, on the simple but relevant example of IP FIBs. Our techniques could then prove instructive in designing compressed data structures for other large-scale data-intensive networking applications, like OpenFlow and MPLS label tables, Ethernet self learning MAC tables, BGP RIBs, access rules, log files, or peer-to-peer paths [33].

Accordingly, this paper can in no way be complete. For instance, we deliberately omitted IPv6 for brevity, even though storage burden for IPv6 is getting just as pressing as for IPv4 [47]. We see no reasons why our techniques could not be adapted to IPv6, but exploring this area in depth is for further study. Multibit prefix DAGs also offer an intriguing future research direction, for their potential to reduce storage space as well as improving lookup time from $O(W)$ to $O(\log W)$. On a more theoretical front, FIB entropy lends itself as a new tool in compact routing research, the study of the fundamental scalability of routing algorithms. We need to see why IP FIBs contain vast redundancy, track

down its origins and eliminate it, to enforce zero-order entropy bounds right at the level of the routing architecture. To what extent this argumentation can then be extended to higher-order entropy is, for the moment, unclear at best.

Acknowledgements

J.T. is with the MTA-Lendület Future Internet Research Group, and A. K. and Z. H. are with the MTA-BME Information Systems Research Group. The research was partially supported by High Speed Networks Laboratory (HSN Lab), J. T. was supported by the project TÁMOP - 4.2.2.B- 10/1–2010-0009, and G. R by the OTKA/PD-104939 grant. The authors wish to thank Bence Mihálka, Zoltán Csernátony, Gábor Barna, Lajos Rónyai, András Gulyás, Gábor Enyedi, András Császár, Gergely Pongrácz, Francisco Claude, and Sergey Gorinsky for their invaluable assistance.

8. REFERENCES

[1] M. Bando, Y.-L. Lin, and H. J. Chao. FlashTrie: beyond 100-Gb/s IP route lookup using hash-based prefix-compressed trie. *IEEE/ACM Trans. Netw.*, 20(4):1262–1275, 2012.

[2] R. Bolla and R. Bruschi. RFC 2544 performance evaluation and internal measurements for a Linux based open router. In *IEEE HPSR*, page 6, 2006.

[3] J. Bonwick and B. Moore. ZFS - the last word in file systems. Sun Microsystems, 2004.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[5] G. Cheung and S. McCanne. Optimal routing table design for IP address lookups under memory constraints. In *IEEE INFOCOM*, pages 1437–1444, 1999.

[6] J. Choi, J. H. Park, P. chun Cheng, D. Kim, and L. Zhang. Understanding BGP next-hop diversity. In *INFOCOM Workshops*, pages 846 –851, 2011.

[7] J. Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, 1970.

[8] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, 1991.

[9] E. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*, 18(2):113–139, 2000.

[10] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, pages 3–14, 1997.

[11] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using Bloom filters. In *ACM SIGCOMM*, pages 201–212, 2003.

[12] R. Draves, C. King, S. Venkatachary, and B. Zill. Constructing optimal IP routing tables. In *IEEE INFOCOM*, 1999.

[13] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.

[14] A. Elmokashfi, A. Kvalbein, and C. Dovrolis. BGP churn evolution: a perspective from the core. *IEEE/ACM Trans. Netw.*, 20(2):571–584, 2012.

[15] EZChip. NP-4: 100-Gigabit Network Processor for Carrier Ethernet Applications. `http://www.ezchip.com/Images/pdf/NP-4_Short_Brief_online.pdf`, 2011.

[16] K. Fall, G. Iannaccone, S. Ratnasamy, and P. B. Godfrey. Routing tables: Is smaller really much better? In *ACM HotNets-VIII*, 2009.

[17] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):1–33, 2009.

[18] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *IEEE FOCS*, pages 390–398, 2000.

[19] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.

[20] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving sub-second IGP convergence in large IP networks. *SIGCOMM Comput. Commun. Rev.*, 35(3):35–44, 2005.

[21] P. Gupta, B. Prabhakar, and S. P. Boyd. Near optimal routing lookups with bounded worst case performance. In *IEEE INFOCOM*, pages 1184–1192, 2000.

[22] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM*, pages 195–206, 2010.

[23] J. Hasan and T. N. Vijaykumar. Dynamic pipelining: making IP-lookup truly scalable. In *ACM SIGCOMM*, pages 205–216, 2005.

[24] P. Hick, kc claffy, and D. Andersen. CAIDA Anonymized Internet Traces. `http://www.caida.org/data/passive`.

[25] W.-K. Hon, R. Shah, and J. S. Vitter. Compression, indexing, and retrieval for massive string data. In *CPM*, pages 260–274, 2010.

[26] G. Huston. BGP routing table analysis reports. http://bgp.potaroo.net/.

[27] I. Ioannidis and A. Grama. Level compressed DAGs for lookup tables. *Comput. Netw.*, 49(2):147–160, 2005.

[28] G. Jacobson. Space-efficient static trees and graphs. In *IEEE FOCS*, pages 549–554, 1989.

[29] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 1(4):425–447, 1990.

[30] V. Khare, D. Jen, X. Zhao, Y. Liu, D. Massey, L. Wang, B. Zhang, and L. Zhang. Evolution towards global routing scalability. *IEEE JSAC*, 28(8):1363–1375, 2010.

[31] Y. Liu, S. O. Amin, and L. Wang. Efficient FIB caching using minimal non-overlapping prefixes. *SIGCOMM Comput. Commun. Rev.*, 43(1):14–21, Jan. 2012.

[32] LuceneTransform. Transparent compression for Apache Lucene. `http://code.google.com/p/lucenetransform`.

[33] H. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane Nano: path prediction for peer-to-peer applications. In *USENIX*, pages 137–152, 2009.

[34] V. Mäkinen and G. Navarro. Dynamic entropy compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4(3):32:1–32:38, 2008.

[35] A. McAuley and P. Francis. Fast routing table lookup using CAMs. In *IEEE INFOCOM*, pages 1382–1391, 1993.

[36] D. Meyer, L. Zhang, and K. Fall. Report from the IAB Workshop on Routing and Addressing. RFC 4984, 2007.

[37] D. S. Miller. net_test_tools. `https://kernel.googlesource.com/pub/scm/linux/kernel/git/davem/net_test_tools`.

[38] G. Navarro and F. Claude. libcds: Compact data structures library, 2004. `http://libcds.recoded.cl`.

[39] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[40] G. Navarro and J. Tarhio. LZgrep: a Boyer-Moore string matching tool for Ziv-Lempel compressed text. *Softw. Pract. Exper.*, 35(12):1107–1130, 2005.

[41] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE JSAC*, 17(6):1083 –1092, 1999.

[42] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM SODA*, pages 233–242, 2002.

[43] G. Rétvári, Z. Csernátony, A. Körösi, J. Tapolcai, A. Császár, G. Enyedi, and G. Pongrácz. Compressing IP forwarding tables for fun and profit. In *ACM HotNets-XI*, pages 1–6, 2012.

[44] J. Rossignac. Edgebreaker: Connectivity compression for

triangle meshes. *IEEE Trans. Visual Comput. Graphics*, 5:47–61, 1999.
[45] K. Sklower. A tree-based packet routing table for Berkeley UNIX. Technical Report, Berkeley, 1991.
[46] H. Song, M. S. Kodialam, F. Hao, and T. V. Lakshman. Scalable IP lookups using Shape Graphs. In *IEEE ICNP*, pages 73–82, 2009.
[47] H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster IP route lookup. In *IEEE ICNP*, pages 358–367, 2005.
[48] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *SIGMETRICS Perform. Eval. Rev.*, 26(1):1–10, 1998.
[49] S. Stergiou and J. Jain. Optimizing routing tables on systems-on-chip with content-addressable memories. In *System-on-Chip*, pages 1–6, 2008.
[50] H. Sun, O. Ozturk, and H. Ferhatosmanoglu. CoMRI: a compressed multi-resolution index structure for sequence similarity queries. In *IEEE CSB*, pages 553–, 2003.
[51] R. Teixeira, K. Marzullo, S. Savage, and G. M. Voelker. In search of path diversity in ISP networks. In *ACM IMC*, pages 313–318, 2003.
[52] P. M. Tolani and J. R. Haritsa. XGRIND: a query-friendly XML compressor. In *ICDE*, pages 225–234, 2002.
[53] Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis. SMALTA: practical and near-optimal FIB aggregation. In *ACM CoNEXT*, pages 1–12, 2011.
[54] S. Vigna and P. Boldi. MG4J: Managing Gigabytes for Java. `http://mg4j.dsi.unimi.it`, 2007.
[55] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *ACM SIGCOMM*, pages 25–36, 1997.
[56] WebGraph. A framework for graph compression. `http://webgraph.di.unimi.it`.
[57] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.
[58] J. Woods. PPP Deflate Protocol. RFC 1979, 1996.
[59] W. Wu. *Packet Forwarding Technologies*. Auerbach, 2008.
[60] M. Zec, L. Rizzo, and M. Mikuc. DXR: towards a billion routing lookups per second in software. *SIGCOMM Comput. Commun. Rev.*, 42(5):29–36, 2012.
[61] X. Zhao, D. J. Pacella, and J. Schiller. Routing scalability: an operator's view. *IEEE JSAC*, 28(8):1262–1270, 2010.
[62] N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.

Appendix

PROOF OF THEOREM 1. As $D(S)$ is derived from a complete binary tree the number of nodes V_D^j of $D(S)$ at level j is at most $|V_D^j| \leq 2^j$, and each node at level j corresponds to a 2^{W-j} long substring of S so $|V_D^j| \leq \delta^{2^{W-j}}$. Let κ denote the intersection of the two bounds $2^\kappa = \delta^{2^{W-\kappa}}$, which gives:

$$\kappa 2^\kappa = 2^W \log_2(\delta) = n \log_2(\delta) \ . \tag{3}$$

Set the leaf push barrier at $\lambda = \lfloor \kappa \rfloor = \lfloor \frac{1}{\ln 2} \mathcal{W}(n \log_2 \delta \ln 2) \rfloor$ where $\mathcal{W}()$ denotes the Lambert W-function. Above level $\lambda - 1$ we have $\sum_{j=0}^{\lambda-1} |V_D^j| = \sum_{j=0}^{\lambda-1} 2^j = 2^\lambda - 1 \leq 2^\kappa$ nodes; at level λ and $\lambda + 1$ each we have 2^κ nodes at maximum; at $\lambda + 2$ there are $|V_D^{\lambda+2}| \leq \delta^{2^{W-\lambda-2}} \leq \sqrt{2^\kappa}$ nodes; and finally below level $\lambda + 3$ we have an additional $\sqrt{2^\kappa}$ nodes at most as levels shrink as $|V_D^{j+1}| \leq \sqrt{|V_D^j|}$ downwards in $D(S)$. Finally, setting the pointer size at $\lceil \kappa \rceil$ bits and summing up the above yields that the size of $D(S)$ is at most $\left(1 + \frac{\log_2(\delta)}{\lceil \kappa \rceil} + 2\left(2 + \frac{2}{\sqrt{2^\kappa}}\right)\right) \lceil \kappa \rceil 2^\kappa = 5n \log_2(\delta) + o(n)$ bits, using the fact that $\lceil \kappa \rceil 2^\kappa = n \log_2(\delta) + o(n)$ by (3). □

PROOF OF THEOREM 2. We treat the problem as a coupon collector's problem on the sub-tries of $D(T)$ at level j. We are given a set of coupons C, each coupon representing a string of length 2^{W-j} on the alphabet Σ of size δ and entropy H_0, and we draw a coupon o with probability $p_o : o \in C$. Let $H_C = \sum_{o \in C} p_o \log_2 \frac{1}{p_o} = H_0 2^{W-j}$, and let V denote the set of coupons after $m = 2^j$ draws. Suppose $m \geq 3$.

LEMMA 6. $|E(V)| \leq \frac{m}{\log_2(m)} H_C + 3$.

PROOF. The probability of having coupon o in V is $\Pr(o \in V) = 1 - (1-p_o)^m$ and so $E(|V|) = \sum_{o \in C}(1-(1-p_o)^m)$. The right-hand-side of the statement is $\frac{m}{\log_2(m)} \sum_{o \in C} p_o \log_2 \frac{1}{p_o} + 3$. The claim holds if $\forall o \in C$:

$$p_o < \frac{1}{e} \Rightarrow 1 - (1-p_o)^m \leq \frac{m}{\log_2(m)} p_o \log_2 \frac{1}{p_o} \ . \tag{4}$$

First, assume $m \geq \frac{1}{p_o}$. As the right hand size is a monotone increasing function of m when $e \leq \frac{1}{p_o} \leq m$, we have $1 - (1-p_o)^m \leq 1 = \frac{1/p_o}{\log_2(1/p_o)} p_o \log_2 \frac{1}{p_o} \leq \frac{m}{\log_2(m)} p_o \log_2 \frac{1}{p_o}$. Otherwise, if $m < 1/p_o$ then let $x = \log_{1/p_o} m$. Note that $0 < x < 1$. After substituting $m = \frac{1}{p_o^x}$ we have

$$1 - (1-p_o)^{\frac{1}{p_o^x}} \leq \frac{1/p_o^x}{\log_2(1/p_o^x)} p_o \log_2 \left(\frac{1}{p_o}\right) = \frac{1}{x p_o^{x-1}} \ .$$

Reordering, taking the $p_o^{x-1} > 0$ exponent of both sides and using that $x < 1$ and so $\frac{1}{x} > 1$, we see that the above holds if $(1-p_o)^{1/p_o} \geq (1 - 1/p_o^{x-1})^{p_o^{x-1}}$. As $(1-p_o)^{1/p_o}$ is monotone decreasing the inequality holds if $p_o \leq 1/p_o^{x-1}$, but this is true because $p_o^x \leq 1$. This proves (4) under the assumption $p_o < \frac{1}{e}$. Note also that there are at most $3 > \frac{1}{e}$ coupons for which (4) cannot be applied. □

Using this Lemma, we have that the expected number of nodes at the j-th level of $D(S)$ is at most

$$E(|V_D^j|) \leq \frac{2^j}{\log_2(2^j)} H_0 2^{W-j} + 3 = \frac{H_0}{j} n + 3 \ . \tag{5}$$

We now have three bounds on the width of level j: $|V_D^j| \leq 2^j$, $E(|V_D^j|) \leq \frac{H_0}{j} n + 3$, and $|V_D^j| \leq \delta^{2^{W-j}}$. Let ξ denote the intersection of the former two and ζ that of the latter two. Easily, $\xi \geq W - \log_2 \frac{W}{H_0}$, $\zeta \leq W - \log_2(W - \log_2 W/H_0) + \log\log_2(\delta)$, $\xi + 1 \leq \zeta$, and $\xi \leq \kappa \leq \zeta$ where κ is as (3). We set the leaf push barrier to $\lambda = \lfloor \xi \rfloor$, which reproduces (2), and the pointer size is again $\lceil \kappa \rceil$ bits. There are at most 2^ξ nodes at the first $\lfloor \xi \rfloor - 1$ levels of $D(S)$, storing which needs $\lceil \kappa \rceil 2^\xi = \lceil \kappa \rceil (\frac{n H_0}{\xi} + 3) = n H_0 + o(n)$ bits by (5); below level $\lceil \zeta \rceil + 1$ there are $\sum_{j=\lceil \zeta \rceil + 1}^{W} |V_D^j| \leq \sum_{j=\lfloor \kappa \rfloor + 2}^{W} |V_D^j| \leq 2\sqrt{2^\kappa}$ nodes needing only $o(n)$ bits space as shown above; and finally the expected number of nodes at levels $\lfloor \xi \rfloor, \ldots, \lceil \zeta \rceil$ is $\sum_{j=\lfloor \xi \rfloor}^{\lceil \zeta \rceil} E(|V_D^j|) < (\lceil \zeta \rceil - \lfloor \xi \rfloor + 1) 2^\xi < (\zeta - \xi + 3) 2^\xi$, giving at most $2^\xi \left(3 + \log_2 \frac{W}{W - \log_2 W/H_0} + \log\log_2(\delta) + \log_2 1/H_0\right)$ nodes and $2\lceil \kappa \rceil$ times that many bits. The result now follows, as $2^\xi \lceil \kappa \rceil \log_2(\frac{W}{W - \log_2 W/H_0}) = o(n)$. □

A more detailed version of the proof of Theorem 2 is available online at `http://arxiv.org/abs/1305.5662`.