Compressing IP Forwarding Tables for Fun and Profit

Gábor Rétvári, Zoltán Csernátony, Attila Körösi, János Tapolcai
Budapest Univ. of Technology and Economics
Dept. of Telecomm. and Media Informatics

András Császár, Gábor Enyedi, Gergely Pongrácz
TrafficLab, Ericsson Research
Hungary

ABSTRACT

About what is the smallest size we can compress an IP Forwarding Information Base (FIB) down to, while still guaranteeing fast lookup? Is there some notion of FIB entropy that could serve as a compressibility metric? As an initial step in answering these questions, we present a FIB data structure, called Multibit Burrows-Wheeler transform (MBW), that is fundamentally pointerless, can be built in linear time, guarantees theoretically optimal longest prefix match, and compresses to higher-order entropy. Measurements on a Linux prototype provide a first glimpse of the applicability of MBW.

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

General Terms

Algorithms, Performance, Theory

1. INTRODUCTION

There are hardly any data structures in networking affected as compellingly by the growth of the Internet as the IP Forwarding Information Base (FIB). Stored in the line card or ASIC memory of routers, the FIB maintains an association from every routed IP destination address prefix to the corresponding next-hop, and it is queried on a packet-by-packet basis at line speed. 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.

As of 2012, there are more than 410,000 routed prefixes in the DFZ [1]. Correspondingly, FIBs tend to grow large both in size and management burden, presenting a crucial bottleneck in the data-plane performance of routers and forcing operators into rapid upgrade cycles [2]. As a quick reality check, the Linux kernel's `fib_trie` data structure [3], when filled with 400,000 prefixes, occupies more than 20 Mbytes of memory, takes several minutes to download to the forwarding plane, and is still heavily debated to scale to multi-gigabit speeds [4]. Commercial routers suffer similar troubles, aggravated by the fact that line card memory is much more difficult to upgrade than software routers.

Several recent studies have identified *FIB aggregation* as an effective way to reduce FIB size, thus extending the lifetime of legacy networking gear and mitigating the Internet routing scalability problem[1] temporarily [2, 5]. 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 [7]), use of hash-based schemes, path/level-compressed multibit tries [3], tree-bitmaps, etc., reduced FIB memory tax to just about 2-5 bytes/prefix [8, 9]. Meanwhile, lookup performance has also improved.

The evident questions "Is there an ultimate limit in FIB aggregation?" and "Can FIBs be reduced to fit in CPU cache entirely?" have been asked several times before [9–12]. In order to answer these questions, we need to go beyond conventional FIB aggregation and take a deliberate attempt at *encoding FIBs to information-theoretical entropy bounds, while still providing optimal worst-case lookup time.* We coined the term *FIB compression* to mark this ambitious undertaking.

Apart from being an intriguing problem in its own right, FIB compression could bring many additional benefits. A FIB compressed below 512 Kbytes would allow to at last truly test the hypothesis whether improved cache friendliness transforms into faster lookups,

[1]See [6] for an alternative perspective on whether FIB size poses a real scalability concern.

Addr/plen	label
-/0	2
0/1	3
00/2	3
001/3	2
01/2	2
011/3	1

(a)

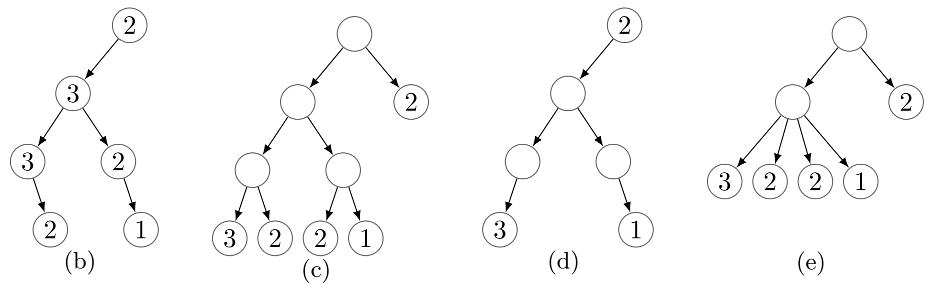


Figure 1: Different representations of an IP routing table: (a) tabular form (address in binary format, prefix length and next-hop address label), (b) prefix tree, (c) leaf-pushed binary trie, (d) ORTC-compressed prefix tree, and (e) stride-optimized multibit trie. Empty labels are omitted.

open the door for full-BGP routing to memory constrained virtual routers and embedded devices (think of doing full BGP with an OpenWrt router of <8MB memory), improve the control plane to data plane roundtrip for FIB resets (a true bottleneck in today's routers [13]), or could serve as a queryable FIB archive format for historic analysis. Or, if nothing else, it could fix a theoretically justified benchmark for FIB designers.

In this paper, we report on the preliminary results of a *systematic quest towards FIB compression*, in the course of which we identify and eliminate different sources of redundancy in FIBs. The resultant FIB encoder compresses a contemporary FIB to well below 512 Kbytes, fully preserves forwarding semantics, attains information-theoretical as well as k-th order entropy bounds and, at the same time, provides longest prefix match in optimal time. And, even though it was not our stated goal, we find in a rudimentary Linux-prototype that the compressed FIB is already applicable at modest data rates.

2. REDUNDANCY IN IP FIBS

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 so called *neighbor table*. This table maintains neighbor specific information, like next-hop IP address, aliases, ARP info, etc. Let the neighbors be identified by an integer *label* in $[1, K]$ and let N denote the total number of entries in the FIB. As an ordinary IP router does not need to maintain an adjacency with every other router in the Internet, we have $K \ll N$. Specifically, we shall assume that K is $O(\log(N))^2$ or $O(1)$. Finally, let W denote the address length (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 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 next-hop label 1. This is then used as an index into the neighbor table and the packet is forwarded on the line-card interface facing that neighbor. This tabular representation is not particularly efficient, as a single lookup operation requires looping through each entry, taking $O(N)$ time. The storage size is $O((\log(K)+W)N)$ bits.

Binary search trees (or *prefix trees*, or *tries* [7]), support lookups much more efficiently (see Fig. 1(b)). Each W bit of the address space corresponds to a level in the tree and lookup is guided by the bits in the destination address: if the next bit is 0 proceed to the left sub-tree, otherwise proceed to the right, and if the corresponding child is missing return the last label encountered along the way. Prefix trees can also contain unlabeled nodes. We allocate the special *empty label* 0 to mark such nodes, yielding the label space $\Sigma = [0, K]$ of cardinality $K + 1$. If a lookup returns the empty label, that means that the corresponding packet is to be dropped. Prefix trees generally improve lookup time from linear to $O(W)$, although memory size increases somewhat.

Our aim in this paper is to pinpoint the origins of redundancy in this simple prefix tree representation (if any), and systematically eliminate them. We shall say that *some information* in a FIB *is redundant*, *if removing it does not alter the forwarding association* maintained by the FIB in any ways, *assuming that each query is for complete W bit long addresses*.

2.1 Semantic Redundancy

Our first culprit is redundant labels in the tree. For example, the association $0/1 \rightarrow 3$ is superfluous, as the more specific entries $00/2 \rightarrow 3$ and $01/2 \rightarrow 2$ override it. Such redundancy, stemming from the very nature of longest prefix match, is called *semantic redundancy*.

A way to (partially) eliminate semantic redundancy is *leaf-pushing* [11]: first, in a preorder traversal labels are pushed from the parents towards the children, and then in a postorder traversal each parent with identically labeled leaves is substituted with a leaf with the children's label (see Fig. 1(c)). Leaf-pushing usually eliminates many redundant entries, reducing semantic redundancy. On the other hand, it also creates a so called *proper binary tree* with nice structure; for a leaf-

[2]Note that logarithms are of base 2.

pushed trie the invariant holds that "a node either has two children or it is labeled with nonzero label".

To eliminate all of semantic redundancy, we need to relabel the tree completely. The prefix tree in Fig. 1(d) has the exact same forwarding semantics as the one in Fig. 1(b), yet contains only 3 labeled nodes instead of 7. The ORTC algorithm of Draves *et al.* [12] yields such a representation in linear time, with provably minimal number of labeled nodes. Unfortunately, the resultant trees lack the nice structure of leaf-pushed tries.

2.2 Structural Redundancy

Binary trees are not necessarily ideal to implement FIBs, as they often contain vast numbers of interior nodes increasing storage size and introducing unnecessary intermediate steps in lookups. This adds *structural redundancy*, loosely defined as excess information needed to store the very tree structure.

The most prominent example of structural redundancy is the nodes at level 2 in Fig. 1(c); removing this level completely we obtain the so called (proper) *multibit trie* in Fig. 1(e). Multibit tries share the appealing properties of leaf-pushed tries, with the generalization that each interior node now has power of two children (this is called the *stride* of the node). To obtain the multibit structure yielding the fewest interior nodes, one can use the linear time *variable-stride optimization* algorithm in [11].

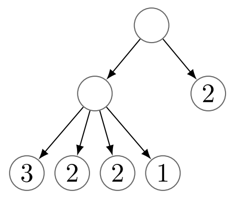
2.3 Information-theoretical Redundancy

The multibit trie in Fig. 1(e) still contains hidden redundancy. In particular, there are three nodes with label 2, but only one node with label 1 or 3. Thus, we could save space by representing label 2 on fewer bits, similarly to Huffman-coding for strings. Exploiting this *information-theoretical redundancy* promises with huge reductions in storage size and, as shall be seen, opens up the intriguing possibility of associating some notion of *entropy* with our FIB as well.

But exploiting *contextual information* one could compress FIBs even further. For instance, labels 1 and 3 only appear at level 3, while the empty label never appears at this level; thus, knowing some context, say, the level of some node, we can effectively predict its label. In the next section, we show that making some context available to the FIB encoder allows to strengthen our zero-order entropy bounds to arbitrary k-th order entropy while, strikingly, still maintaining full forwarding equivalence and optimal $O(W)$ time lookup.

3. MULTIBIT BURROWS-WHEELER

Our FIB encoder, called *Multibit Burrows-Wheeler* (MBW) for reasons to be made clear shortly, is essentially a mix of the succinct level-indexed binary trees of Jacobson [14], succinct k-ary trees, and, most importantly, the XBW transform due to Ferragina *et al.* [15], with some FIB-specific twists.



i	S_{last}	S_α	
1	1	0	level 0
2	0	0	level 1
3	1	2	
4	0	3	level 2
5	0	2	
6	0	2	
7	1	1	

Figure 2: Multibit trie and MBW transform.

We chose multibit-tries as the basis for the MBW transform. Even though it is not optimal in terms of semantic redundancy, we found that the nice structure simplifies the encoder substantially. Let T be a proper multibit trie, let n be some node in T, let l be a mapping $T \mapsto \Sigma$ specifying for a node n the corresponding label $l(n) \in \Sigma$, let L be the set of leaves, and let t denote the number of nodes in T.

A multibit trie T enjoys the following properties.

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

P2: $l(n) \neq 0 \Leftrightarrow n \in L$.

The main idea in MBW (on the traces of XBW) is to serialize T into a bitstring S_{last} encoding the structure and a string S_α 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 transform.

Given a multibit trie T, MBW first performs a breadth-first-search traversal of T, along which it sequentially fills up the strings S_{last} and S_α.

```
BFS-TRAVERSAL (node $n$, integer $i$)
    $S_\alpha[i] \leftarrow l(n)$
    if $n$ is the last child of its parent then
        $S_{\text{last}}[i] \leftarrow 1$ else $S_{\text{last}}[i] \leftarrow 0$
    $i \leftarrow i + 1$
```

We assume that the root is "last": $S_{\text{last}}[0] = 1$. Then, the MBW transform is essentially the tuple $\text{mbw}(T) = (S_{\text{last}}, S_\alpha)$. The following statement is now obvious.

CLAIM 1. *Given a multibit trie T on t nodes,* $\text{mbw}(T)$ *can be built in optimal $O(t)$ time.*

The transform $\text{mbw}(T)$ has some appealing properties (see Fig. 2). For instance, the children of some node, if exist, are stored on consecutive indices in S_{last} and S_α. In addition, nodes on the same level of T are also mapped to consecutive indices in $\text{mbw}(T)$.

An obvious next-step would be to apply some standard string compressor (like the venerable `gzip` tool) on S_{last} and S_α to obtain something of, supposedly, small size. However, we want our transform to retain fast navigability, that is, admit queries like "get all children of a node" *without first decompressing the string.* We use *string indexers* for this, which, in addition to lossless compression, also support certain navigational primitives in optimal $O(1)$ time in-place. Given a string $S[1,t]$ on alphabet Σ, $c \in \Sigma$, and $q \in [1,t]$, the usual primitives are as follows [14, 15]:

- access(S,q): return the symbol stored at position q in S,
- rank$_c(S,q)$: return the number of times symbol c occurs in the prefix $S[1,q]$, and
- select$_c(S,q)$: return the position of the q-th occurrence of symbol c in S.

Curiously, these simple primitives admit strikingly complex queries to be implemented and supported in optimal time. For example, the `get_children` operation, which returns the position of the first and last child of some interior node n mapped to index i in mbw(T), takes the following simple form.

```
get_children (integer $i$)
    $r \leftarrow \mathrm{rank}_0(S_\alpha, i)$
    first $\leftarrow \mathrm{select}_1(S_{\mathrm{last}}, r) + 1$
    last $\leftarrow \mathrm{select}_1(S_{\mathrm{last}}, r + 1)$
    return (first, last)
```

Here, r tells how many interior nodes, identified with the empty label 0, were found *before* n in the course of the bfs-traversal, `first` will get the index of the last child of the previous node plus 1, and `last` gets the label of n's last child. Using this primitive, IP lookup on mbw(T) is as follows.

```
lookup (address $a$)
    $p \leftarrow 0$, $i \leftarrow 0$
    while $p < W$ do
        if $\mathrm{access}(S,i) \neq 0$ then return $\mathrm{access}(S,i)$
        $(f,l) \leftarrow$ get_children($i$)
        $j \leftarrow$ get_bits($a$, $p$, $p + \log(l - f + 1) - 1$)
        $i \leftarrow f + j$, $p \leftarrow p + \log(l - f + 1)$
```

First, we ask if the current node n at position i is a leaf node. If it is, we return its label (recall **P2**), otherwise we ask for the children. Esaily, $l - f + 1$ gives the number of children of n, and the logarithm (by **P1**) tells how many bits to read from a, starting at position p, 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.

Claim 2. *MBW `lookup` terminates in $O(W)$ time.*

4. MEMORY SIZE BOUNDS

The first verifiable cornerstone of a data compression algorithm is whether it encodes to *information-theoretical minimum.* In our case, this corresponds to the minimum number of bits needed to differentiate between any two FIBs. As there are $F_t = \binom{2t+1}{t}/(2t+1)$ binary trees on t nodes (note that our trees are not binary and are of bounded height, but we disregard these nuisances here), and storing the label map requires an additional $t\lceil\log(K)\rceil$ bits, the information-theoretical minimum ends up at roughly $\log F_t + t\log(K) = 2t - \Theta(\log t) + t\log(K)$.

Curiously, even the trivial encoding of mbw(T) reproduces this bound. Store the bitstring S_{last} on t bits, store an additional bitstring for marking internal nodes again on t bits, and finally store S_α on $t\log(K)$ bits (for more details, see [15]).

Claim 3. *For mutlibit trie T,* mbw(T) *can be stored on optimal* $2t + t\log(K)$ *bits.*

As such, MBW qualifies as a *succinct* FIB data structure [14], similarly to, e.g., [8]. The real test is, however, whether we can go down below information-theoretical minimum and attain *entropy bounds*, for some suitable definition of FIB entropy.

That a prefix tree should have some sorts of entropy associated with it, and that this should be determined by the distribution of labels on the nodes, is not a completely far-fetched idea. Think of a FIB with each node labeled the same; here, the label distribution is deterministic and so its entropy is zero; this FIB can then be substituted with a single default route whose size is independent of the number of prefixes, and hence the corresponding FIB-entropy is also zero. The more random the label distribution the larger the entropy and worse the compressibility, or at least this is what we expect from our entropy notion. Interestingly, MBW exposes just the above intuitive notion of entropy, when S_α is encoded using e.g. generalized wavelet trees [15].

Claim 4. *For $K = O(\log(t))$,* mbw(T) *can be encoded on $tH_0(p_c) + t + o(t)$ bits, where $H_0(p_c)$ is the entropy of the label-distribution $p_c : c \in \Sigma$:*

$$H_0(p_c) = \sum_{c\in\Sigma} p_c \log\frac{1}{p_c}, \quad p_c = \frac{\sum_{n\in T} I(l(n) = c)}{t} \quad .$$

Higher order string compressors use the observation that elements of a string often depend on their neighbors. Let the *k-context* of a symbol c in a string S be defined as the k-long substring that precedes c. Then, the main postulate in string compression is that the larger the context (i.e., k), the better the prediction of c from its k-context. For a string S the k-th order entropy $H_k(S)$ is commonly used as a metric of compressibility of S, given that information on k-contexts is available to the encoder. Note that $H_k(S) \leq H_0(S)$. To actually attain k-th order entropy, string compressors usually apply the Burrows-Wheeler transform, a reversible permutation of the string so that symbols with similar context end up close to each other. This way, the transformed string compresses well, even with zero-order compressors.

Using the same argumentation as in [15], we can show that MBW is not just a zero-order FIB compressor but it can attain higher-order entropy as well. The idea is that the notion of k-context naturally generalizes to multibit tries: simply, *k-context of a node n* is the k-long upstream subpath from the parent of n to the root. For instance, the k-context of the leftmost leaf with label 3 in Fig. 2(a) is the string 00, which, for k large enough, essentially *corresponds the level of the node*. As MBW organizes nodes of similar 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). As for strings, we can assume that similarly labeled leaves in T reside at similar level, and the better the correspondence the more efficient the prediction. Currently it is entirely unclear to what extent such contextual dependency is present in IP routing tables, but our experiments provided some pointers that it actually is. Now, using a higher-order string indexer we obtain the following bound [15].

Claim 5. *Given a multibit trie $T : K = O(1)$, $\mathrm{mbw}(T)$ can be encoded on $tH_k(p_l) + O(t)$ bits.*

5. A LINUX-KERNEL PROTOTYPE

Finally, we set out to test MBW on real data. We coded up a quick prototype, where we do FIB construction in user space and the actual lookup is realized as a custom Linux kernel module.

Research on IP FIBs has for a long time been plagued by the unavailability of real data. Apart from the two IP core router FIBs we could get privately (`taz` and `hbone`), what is available publicly (`AS*`) are RIB dumps from BGP collectors, like RouteViews or looking glass servers. Unfortunately, these only very crudely model real FIBs, because BGP collectors run the 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.

We used a PC with a quad-core Intel Core i5 CPU at 2.50GHz, with 2x32 Kbyte L1 cache for each core and 256 Kbyte shared L2 cache. We built an initial prefix tree from the BGP dumps and first applied leaf-pushing to remove redundant entries. To remove structural redundancy as well, we ran a slightly modified version of variable-stride optimization [11], which simultaneously minimizes the number of internal nodes and the average depth and realizes various trade-offs between the two via a weighting parameter α. The intention is to decrease the number of levels in the trie, and hence the number queries to $\mathrm{mbw}(T)$ during IP lookup. Curiously, we also found that a certain over-compactification often reduces storage. We then applied the MBW transform and then the compressed FIBs were obtained using `libcds` [16]: S_{last} was compressed with `RRR` and the zero-order Huffman-shaped `WaveletTree` was used to compress S_α. The compressed strings were downloaded to the kernel, which ran a minimalistic port of `libcds` to do IP lookups right on the compressed form.

The plain storage size for some standard FIB instances is given in Table 1. The main observation is that on FIBs with only a few next-hops MBW compresses to about 2-3 bits per prefix, while in the rest of the cases about 6 bits/prefix is attained. Hence, FIBs of $>$400 thousand prefixes can easily be encoded into the 256-512 Kbyte caches common today. We found that this transforms into improved-cache friendliness: a process doing random IP lookups on the compressed `AS1221` FIB produced about $5 - 20\%$ cache misses in steady state (measured with the Linux `perf` tool), while for prefix trees this was an impressive $40 - 70\%$.

Additionally, MBW compresses below information-theoretic minimum *and* zero-order entropy bound, and the latter is substantially smaller than the former indicating that FIBs indeed contain exploitable entropy. We also experimented with calculating rough higher order entropy bounds by compressing S_α with the `ppmd` string compressor. This yielded smaller bounds than the other two, especially for FIBs with high K, suggesting that FIBs might contain additional contextual dependency as well. But to decide, we'll need real data.

Building and compressing the FIBs takes at maximum 300 ms and kernel download is in 10 millisecond range, so about $3-10$ complete rebuilds per second seem plausible. FIB insert and delete do not necessarily need complete rebuilds though, considering recent advances in *dynamic* entropy-compressed string indexes [17].

But the real test is under real traffic. We deployed a Linux router in a `VirtualBox` guest, the FIB came from the `AS1221` dataset, and the traffic was assembled from CAIDA's "Anonymized Internet Traces 2012", comprising 100K packets with mean packet size of 650 bytes. Before each round we flushed the route cache and then we used `tcpreplay` to replay the trace from the host to the guest at different data rates and we observed the packet loss. Note that packet loss is entirely due to the increased lookup latency of MBW (recall that MBW is lossless), and so it characterizes the price we pay for the constant term in MBW's $O(W)$ lookup complexity.

For the default setting of the Linux route cache, we have seen no observable data loss over MBW compared to the Linux FIB implementation `fib_trie` [3], while a conservative limit of 512 entries yielded the results in Fig. 3. The most important message seems to be that smaller average depth (through a proper setting α) transforms to faster routing (i.e., smaller lookup de-

Table 1: Number of prefixes (N) and labels (K), number of nodes/average depth in the binary tree (t_0/d_0) and the multibit trie obtained at the optimal α (t/d), and for the latter the information-theoretical lower bound (I), zero-order entropy bound (H_0), and size of mbw(T) (s) all in Kbytes, and bits/prefix efficiency η.

Name	N	K	t_0/d_0	t/d	I	H_0	s	η
taz	410K	3	298K/21.5	236K/5.3	118	90	80	1.6
hbone	410K	131	567K/21.7	440K/5.3	385	210	187	3.6
AS1221	379K	3	921K/22.6	344K/6.8	172	150	134	3.6
AS4637	219K	2	573K/22.5	164K/4.7	82	64	56	2.0
AS3257	385K	838	932K/22.6	507K/8.7	570	381	395	8.2
AS6447	400K	15	965K/22.6	573K/9.0	358	332	310	6.1
AS6730	388K	426	939K/22.6	502K/8.8	565	298	294	6.0

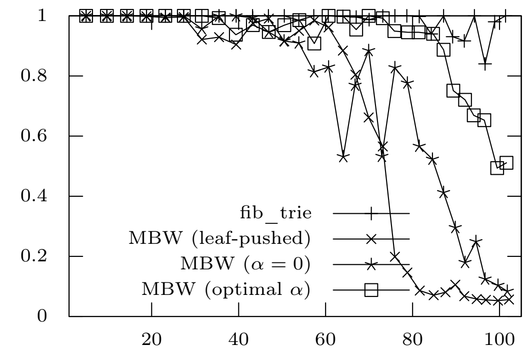


Figure 3: Fraction of packets the Linux implementation could process at different data rates [Mbit/sec].

lay), and that MBW supports about 80-100 Mbit/sec at at about 30K-50K lookups/second. This is about two orders of magnitude smaller than what is needed for IP routers, but still seems fair from a proof-of-concept prototype of an experimental data structure.

6. CONCLUSIONS

Our quest to FIB compression started with a crazy idea: "Could we compress IP FIBs to some sorts of entropy, with maintaining optimal lookup?" This paper is about taking that idea to the extreme.

It turned out that a suitable abstraction from string entropy to FIB entropy indeed exists, and this is via a simple MBW transform. Hence, FIB entropy provides us with solid information-theoretical guarantee that the storage size we achieved is indeed minimal, up to lower-order terms.

To actually encode to entropy, we eventually had to get rid of good-old trees and pointers. Pointers are "fat" in that they index the entire memory, and wasteful at smaller scales. The pointerless data structures we used, on the other hand, seem at least two orders of magnitude slower than needed. We are confident that, by optimizing the implementations and the compression algorithms for access time instead of mere memory footprint (and measuring on real hardware!), performance can be improved with at least one order of magnitude. Lookup is, after all, *theoretically* optimal $O(W)$. But the question still remains open, whether fully pointerless FIBs can be scaled to be *practically* relevant on multi-gigabit IP routers, with retaining entropy bounds and cache-friendly storage size.

The other issue is to thoroughly study the entropy present in IP FIBs. We need to see why it is there, track down its origins, and eliminate it. To what extent this argumentation can then be extended to higher-order entropy is, for the moment, unclear at best.

7. ACKNOWLEDGEMENTS

J. T. is the member of the MTA-BME Future Internet Research Group. The research was partially supported by High Speed Networks Laboratory (HSN Lab), the project TAMOP 421B-09/1/KMR-2010-0002, and the OTKA/PD-104939 grant.

8. REFERENCES

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

[2] Xiaoliang Zhao, Dante J. Pacella, and Jason Schiller. Routing scalability: an operator's view. *IEEE J.Sel. A. Commun.*, 28(8):1262–1270, October 2010.

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

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

[5] Varun Khare, Dan Jen, Xin Zhao, Yaoqing Liu, Dan Massey, Lan Wang, Beichuan Zhang, and Lixia Zhang. Evolution towards global routing scalability. *IEEE JSAC*, 28(8):1363–1375, October 2010.

[6] Kevin Fall, Gianluca Iannaccone, Sylvia Ratnasamy, and P. Brighten Godfrey. Routing tables: Is smaller really much better? In *ACM HotNets-VIII*, 2009.

[7] Keith Sklower. A tree-based packet routing table for Berkeley UNIX. Technical Report, Berkeley, 1991.

[8] Wencheng Lu and S. Sahni. Succinct representation of static packet forwarding tables. In *ICN*, page 78, 2007.

[9] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, pages 3–14, 1997.

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

[11] V. Srinivasan and George Varghese. Faster IP lookups using controlled prefix expansion. *SIGMETRICS Perform. Eval. Rev.*, 26(1):1–10, 1998.

[12] Richard Draves, Christopher King, Srinivasan Venkatachary, and Brian Zill. Constructing optimal IP routing tables. In *INFOCOM*, March 1999.

[13] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second IGP convergence in large IP networks. *SIGCOMM Comput. Commun. Rev.*, 35(3):35–44, 2005.

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

[15] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, 2009.

[16] Gonzalo Navarro and Francisco Claude. libcds: Compact data structures library, 2004. `http://libcds.recoded.cl`.

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