# Towards a Billion Routing Lookups per Second in Software

Marko Zec
University of Zagreb, Croatia
zec@fer.hr

Luigi Rizzo
Università di Pisa, Italy
rizzo@iet.unipi.it

Miljenko Mikuc
University of Zagreb, Croatia
miljenko.mikuc@fer.hr

## ABSTRACT

Can a software routing implementation compete in a field generally reserved for specialized lookup hardware? This paper presents DXR, a lookup scheme based on transforming large routing tables into compact lookup structures which easily fit into cache hierarchies of modern CPUs.

Our transform distills a real-world BGP snapshot with 417,000 IPv4 prefixes and 213 distinct next hops into a structure consuming only 782 Kbytes, less than 2 bytes per prefix. Experiments show that the corresponding lookup algorithm scales linearly with the number of CPU cores: running on a commodity 8-core CPU it yields average throughput of 840 million lookups per second for uniformly random IPv4 keys.

We prototyped the algorithm inside the FreeBSD kernel, so that it can be used with standard APIs and routing daemons such as Quagga or XORP, and tested for correctness by comparing lookup results with the traditional BSD radix tree implementation.

## Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking; C.4 [**Performance of Systems**]:

## General Terms

Algorithms, Performance, Experimentation

## Keywords

Packet Lookup and Classification, Software Routers

## 1. INTRODUCTION

Determining each packet's next hop by finding the longest matching prefix in a forwarding table is the most fundamental operation which every Internet router has to perform. Lookups have to be fast: for example, a single 100 GB/s Ethernet port may receive up to 144 million packets per second, and the search database in today's Internet routers may contain up to around 420,000 IPv4 prefixes, with no signs of slowdown in growth of number of prefixes being announced to global BGP tables. Furthermore, route lookup is only one of many operations that a (software) router must perform, so the loookup cost must be a small fraction of the packet interarrival time.

Doing routing lookups in software has somewhat fallen out of focus of the research community in the past decade, as the performance of general-purpose CPUs was not on par with quickly increasing packet rates. Faced with other bottlenecks inherent to software routers, such as limited speeds of early peripheral buses, both the research community and the industry moved to devising specialized hardware routing lookup schemes. While early hardware routing lookup proposals were centered around TCAMs or fast SRAMs, the rumor has it that most router vendors today use ASICs coupled with lots of reduced latency DRAM chips for doing pipelined routing lookups in high-speed line cards.

The landscape has changed, however. First, the performance potential of general-purpose CPUs has increased significantly over the past decade, thanks to major improvements in instruction-level parallelism, shorter execution pipelines, better branch predictors, larger CPU caches, and higher system and memory bus bandwidths; only the DRAM latency remains unchanged at roughly 50 ns. The silicon industry is embracing parallelism: 4 or 8 core general-purpose CPUs are a commodity today, with 16 and 32 core systems expected to become mainstream soon.

Secondly, the increasing interest for virtualized systems and for software defined networks calls for a communication infrastructure that is more flexible and less tied to the hardware. Software packet processing can address this requirement, and the forementioned performance improvements in general purpose hardware are very timely in this respect.

Third, a number of recent works [6, 9, 10] have proposed and demonstrated fast frameworks for software packet processing with very little per-packet overhead, making the case for revisiting the problem of software route lookups.

**Our contribution:** In this paper we address the problem of next-hop lookups for IPv4 packets, and propose a technique which runs efficiently in software on modern general-purpose systems (multi core, fast caches, high latency memory). We describe DXR, a routing lookup scheme which aims at achieving high speeds by exploiting CPU cache locality, even when operating with large sets of network prefixes. Small memory footprint permits our lookup structures to reside entirely in caches even with full sized BGP tables. This largely eliminates expensive DRAM accesses during the lookup process, and makes the algorithm scale well across multiple execution cores, exceeding 800 million lookups per second on an 8-core system for random queries, and reaching significantly higher speeds for repetitive queries.

Parts of DXR are similar to other proposed techniques, which is not surprising given that the minuscule time budget for each lookup requires simple solutions. Our smaller data structures, however, present a significant improvement with respect to other software proposals.

The rest of the paper is structured as follows: Section 2 describes previous work. Section 3 describes our lookup scheme, motivates design choices, and discusses implementation tradeoffs. Section 4 contains a detailed performance evaluation under different operating conditions.

## 2. RELATED WORK

IP lookup algorithms have been well studied in the past, first for software-based solutions, and eventually focusing on designs that could be implemented in hardware to overcome the perceived (or actual) mismatch between network and CPU speeds. Due to space limitations we only summarize the main results here.

Ruiz-Sanchez et al. [11], and Waldvogel et al [16] give useful surveys of software solutions up to 2001 (which covers most of the research on software lookups). Traditional solutions involve tries [12], optimized to reduce the number of search steps by compressing long paths (Level-Compressed tries, [8]), or using n-ary branching (Multibit Tries, [14]). Given the small and fixed problem size, some ad-hoc solutions have been proposed that expand the root into a $2^k$ array of pointers to subtree, as used by DIR-24-8 [5], Lampson-Varghese [7], and also in this paper. Prefixes can be transformed into address ranges or intervals, which reduces the lookup to a binary search into an array of ranges [7]. Similarly exploiting the problem size, the Lulea scheme [3] partitions the trie in three levels (using 16, 8, 8 bits) and then uses a compact representation of the pointers.

As an alternative, Waldvogel et al [16] propose the use of separate hash tables for each prefix length, starting the search from the most specific prefix and then moving up. This scheme is elegant but not particularly fast compared to other solutions for IPv4.

Caching recent look-up results using on-chip memory is discussed for instance in [2] and [13]. [2] in 1999 achieved around 88 million lookups per second with host address caching on 500 MHz DEC Alpha with 1 Mbyte of L2 cache (updates were not discussed). The approach presented in [13] relies on temporal locality in the lookups. However, such locality is today rare in the core of the network.

Especially important in [11] is the comparison of actual run times of multiple algorithms, which permits ranking them irrespective of absolute performance. The peak performance reported in the literature for such software solutions ranges between 2 and 5 Million lookups per second (Mlps) on 1999 machines [11], and 3 to 20 Mps on 2006 hardware [4].

Scaling these figures to modern hardware is not trivial, because the performance is dominated by memory access latencies. In fact, all the rest being equal, performance may vary by an order of magnitude or more depending on routing table size and request distributions. This also means that the memory footprint of a lookup scheme has a strong impact on its feasibility, especially as the number of prefixes grows (going from approx. 38 K prefixes in 1997 to the current 420 K prefixes in a full BGP table). In this respect, existing schemes tend to have quite large memory footprints, from the 24 bytes per prefix of the Lampson-Varghese scheme [7] to the 4.5 bytes/prefix of the Lulea [3].

The problem size can be reduced by doing routing table aggregations. SMALTA [15] shows a practical, near-optimal FIB aggregation scheme that shrinks forwarding table size without modifying routing semantics or the external behavior of routers, and without requiring changes to FIB lookup algorithms and associated hardware and software. The claimed storage reduction is by at least 50%.

Due to the general inability of doing packet processing at line rate in software, the past decade has seen a shift of interest to hardware-related solutions for routing lookups. As mentioned in the previous section, however, the combination of faster processing nodes, and an increased interest in virtualization, makes software IP lookups relevant again.

## 3. DISTILLING ROUTING TABLES

Our lookup algorithm and data structures are based on the idea that a routing table, essentially a database containing a large number of network prefixes with different prefix lengths, can be projected onto a set of contiguous and non-overlapping address ranges covering the entire address span of a network protocol. The lookup procedure becomes simple: the address range containing the key can be found through binary search.

The concept of constructing lookup schemes based on address ranges (or intervals) is not new [7]; the novelty in our scheme is a careful encoding of the information, so that address ranges and associated information consume a small amount of memory, and are organized in a way which inherently exploits cache hierarchies of modern CPUs to achieve high lookup speeds and parallelism of execution on multiple processor cores.

Note that our primary goal was not to implement a universal routing lookup scheme: data structures and algorithms which we describe in this paper have been optimized exclusively for the IPv4 protocol.

### 3.1 Building the search data structure

A network prefix commonly refers to a tuple {network address, prefix length}. Prefix length is the number of leftmost bits of a network address which are matched against the key; the remaining bits are ignored. Classless Interdomain Routing principle (IETF RFC 4632) mandates that among all the matching prefixes found in a database for a given key, the one with the longest prefix must be selected.

Consider a sample routing database specified in a canonical {prefix, next hop} notation:

```
   IPv4 prefix   next hop
1: 0.0.0.0/0     A
2: 1.0.0.0/8     B
3: 1.2.0.0/16    C
4: 1.2.3.0/24    D
5: 1.2.4.5/32    C
```

Building of the search data structure begins by expanding all prefixes from the database into address ranges, and taking into account that more specific prefixes take precedence over less specific ones, this results in a sorted sequence of non-overlapping address ranges:

```
   IPv4 address interval         next hop  (prefix)
1: [0.0.0.0 .. 0.255.255.255]    A         (1)
2: [1.0.0.0 .. 1.1.255.255]      B         (2)
3: [1.2.0.0 .. 1.2.2.255]        C         (3)
4: [1.2.3.0 .. 1.2.3.255]        D         (4)
5: [1.2.4.0 .. 1.2.4.4]          C         (3)
6: [1.2.4.5 .. 1.2.4.5]          C         (5)
7: [1.2.4.6 .. 1.2.255.255]      C         (3)
8: [1.3.0.0 .. 1.255.255.255]    B         (2)
9: [2.0.0.0 .. 255.255.255.255]  A         (1)
```

Neighboring address ranges that resolve to the same next hop are then merged. In the above example, ranges 5, 6 and 7 all resolve to next hop "C", and the table reduces to:

| | IPv4 address interval | next hop | (prefix) |
|---|---|---|---|
| 1: | [0.0.0.0 .. 0.255.255.255] | A | (1) |
| 2: | [1.0.0.0 .. 1.1.255.255] | B | (2) |
| 3: | [1.2.0.0 .. 1.2.2.255] | C | (3) |
| 4: | [1.2.3.0 .. 1.2.3.255] | D | (4) |
| 5: | [1.2.4.0 .. 1.2.255.255] | C | (3,5) |
| 6: | [1.3.0.0 .. 1.255.255.255] | B | (2) |
| 7: | [2.0.0.0 .. 255.255.255.255] | A | (1) |

Next, we can trim information that is redundant or not useful for lookup purposes. We only need the start of an interval (the end is derived from the next entry); the reference to the original prefix is not needed; and the next hop can be encoded as a small (e.g. 16-bit) index in an external next hop descriptor table. This reduces the size of each entry to only 6 bytes. Thus, the structure derived from the above example becomes:

| | interval base | next hop |
|---|---|---|
| 1: | 0.0.0.0 | A |
| 2: | 1.0.0.0 | B |
| 3: | 1.2.0.0 | C |
| 4: | 1.2.3.0 | D |
| 5: | 1.2.4.0 | C |
| 6: | 1.3.0.0 | B |
| 7: | 2.0.0.0 | A |

It can be shown that for any given routing table containing P prefixes, the resulting address range table will contain no more than N = 2P-1 non-overlapping ranges. Provided that such a table is kept sorted, it can be searched in logarithmic time in the number of address range entries. With global BGP routing table sizes approaching 500,000 prefixes, and with a pessimistic assumption that the prefix distribution does not permit for address range aggregation, the address range table would contain no more than $10^6$ elements. At 6 bytes each, this is still too much space, and the number of search steps (20 for $10^6$ elements) is prohibitively large compared to our target search times.

The next construction step is then to shorten the search by splitting the entire range in $2^k$ *chunks*, and using the initial $k$ bits of the address to reach, through a *lookup table*, the *range table entries* corresponding to the chunk. Figure 1 shows the arrangement.

## 3.2 Data structures

The concept of indexing the lookup tables with a relatively large portion of the IPv4 key was inspired by [5], although we use fewer bits and thus smaller lookup tables, aiming at fitting the whole table as high as possible in the CPU cache hierarchy.

The small key size (32 bit) makes this approach particularly effective, because the initial table in Figure 1 does not consume an exceeding amount of space, and permits further optimizations in time or space, as discussed below.

**Lookup Table Entries:** Each entry in the lookup table must contain the offset of the corresponding chunk in the range table, and the number of entries in the chunk. However, if a chunk only contains one entry (i.e. a single next hop), the lookup table can point directly to the *next hop table*, allowing the completion of the lookup with a single L2 cache access.

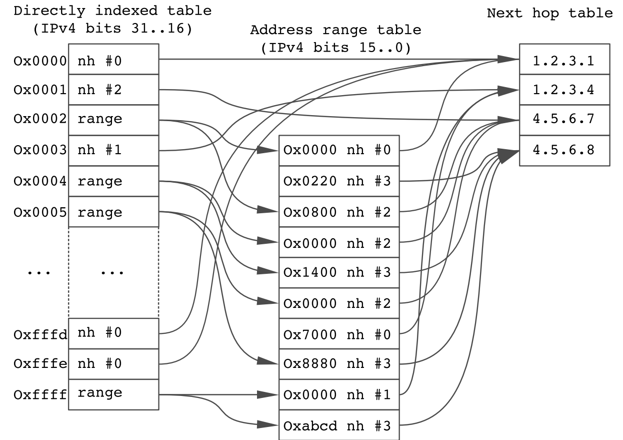


**Figure 1: The lookup table on the left, the range table in the center, the next hop table on the right.**

32 bits suffice for the entries in the lookup table. We use 19 bits as an index into the range table (or into the prefix table), and 12 bits for the size of each chunk; one of these 12-bit configurations is reserved to indicate that this entry points to the prefix table. The remaining bit is used to indicate the format of the range table entries, as discussed below.

This arrangement works for up to $2^{19}$ address ranges after aggregation, and up to 4095 entries per chunk. These numbers should provide ample room for future growth. Also consider that the decision on how to split the bits can be changed at runtime when rebuilding the tables, and it is trivial to recover extra bits e.g. by artificially extending chunks so that they have a multiple of 2, 4, 8 entries, with negligible memory overhead (in fact, chunks with only one address cannot exist, as they are represented by a pointer to the next hop table).

**Range Table Entries:** The addresses in the range table entries only need to store the remaining $32 - k$ bits, further reducing the memory footprint. Thus, if we choose $k \geq 16$ bits for the lookup index, and assuming each next hop can be encoded with 16 bits, 4 bytes suffice for these "long" entries. For example, the chunk covering the range 1.2.0.0/16 can be encoded as:

| | interval base | next hop |
|---|---|---|
| 1: | 0.0 | C |
| 2: | 3.0 | D |
| 3: | 4.0 | C |

We introduced a further optimization which is especially effective for large BGP views, where the vast majority of prefixes are /24 or less specific, and the number of distinct next hops is typically small. If all entries in a chunk contain /24 or less specific ranges, and next hops that can be encoded in 8 bit, we use a "short" format with only 16 bits per entry (the least significant 8 bits do not need to be stored). Therefore, the short format for the 1.2.0.0/16 chunk is:

| | interval base | next hop |
|---|---|---|
| 1: | 0 | C |
| 2: | 3 | D |
| 3: | 4 | C |

Table 1: Sizes of the DXR data structures for several different IPv4 routing table snapshots. The time to rebuild the lookup structures from scratch is also given, in milliseconds.

| | | | D16R scheme | | | | | D18R scheme | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Table snapshot | IPv4 prefixes | Next hops | Total bytes | Full chunks | Short ranges | Long ranges | Build time | Total bytes | Full chunks | Short ranges | Long ranges | Build time |
| static | 4 | 2 | 262248 | 2 | 0 | 8 | 0.01 | 1048676 | 2 | 0 | 7 | 0.03 |
| LINX | 417523 | 213 | 800672 | 12935 | 200436 | 33130 | 96.81 | 1548700 | 25494 | 211724 | 17885 | 220.52 |
| U. Oregon | 421059 | 38 | 827204 | 13627 | 220042 | 31010 | 98.91 | 1574872 | 27251 | 233618 | 14531 | 230.99 |
| PAIX | 412568 | 80 | 765752 | 13082 | 215432 | 17700 | 96.28 | 1522992 | 24801 | 217078 | 9579 | 221.73 |

As mentioned, one bit in the lookup table entry is used to indicate whether a chunk is stored in long or short format.

## 3.3 Updating

Given that our lookup structures store only the minimum of the information necessary for resolving next hop lookups, a separate database which stores information on all the prefixes is required for rebuilding the lookup structures. The radix tree already available in FreeBSD kernel is perfectly suitable for that purpose, although it should be noted that our lookup structures can be derived from any other routing database format.

When a prefix covering a single address range chunk is added or removed from the primary database, our current implementation replaces the entire chunk and rebuilds it from scratch. The process of rebuilding the chunk begins by finding the best matching route for the first IPv4 address belonging to the chunk, translating it to an range table entry, and storing it on a heap. The algorithm then continues to search the primary database for the next longest-matching prefix until it falls out of the scope of the current chunk. If the heap contains only a single element when the process ends, the next hop can be encoded in lookup table, and the heap may be released.

As more prefixes are found, if they point to the same next hop as the descriptor currently on the top of the address range heap, they are simply skipped over, until a prefix pointing to a different next hop is encountered. This allows for very simple yet surprisingly efficient aggregation of routing information, and is the key factor which contributes to the small footprint of the lookup structures.

Updates covering multiple chunks (prefix lengths $< k$) can be processed in parallel, since each chunk can be treated as an independent unit of work. We have not implemented this feature (we service all update requests in a single thread), but that approach could be used to improve update speed where update latencies would be particularly important.

Another technique for reducing the time spent on processing updates is coalescing multiple add/remove requests into a single update; we have accomplished this by delaying the processing of updates for several milliseconds after they have been received.

## 3.4 Lookup algorithm

The lookup procedure is completely straightforward. We first use the $k$ leftmost bits of the key as an index in the lookup table. From there we know whether the entry points to the next hop (in which case we are done), or it points to a chunk (short or long, depending on the format bit). In this case we use the (offset, length) information to perform a binary search on the entries of the range table covering our chunk.

Since range table entries are small (2 or 4 bytes), as the binary search proceeds, chances are that the remaining entries to be looked up have already migrated from L2 to the L1 CPU cache (today's cache line sizes being 64 bytes long). This inherently further speeds up the lookup process.

# 4. PERFORMANCE EVALUATION

Our primary test vectors were three IPv4 snapshots from routeviews.org BGP routers, each with slightly different distribution and number of prefixes and next hops (Table 1). Here we present the results obtained using the LINX snapshot, selected because it is the most challenging for our scheme: it contains the highest number of next hops, and an unusually high (for a BGP router) number of prefixes more specific than /24, see Figure 2.

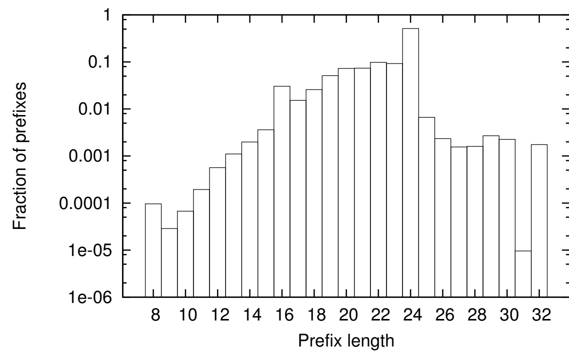


**Figure 2: Distribution of prefix lengths for April 2012 linx.routeviews.org BGP snapshot**

All tests were performed on an AMD FX-8150 machine with 4 Gbytes of RAM. The CPU has 8 independent execution cores; each pair of cores share a private 2 Mbyte L2 cache block, for a total of 8 Mbytes of L2 cache. All eight cores share a single 8 Mbytes block of L3 cache. While the CPU supports automatic clock frequency scaling up to 4.2 GHz, we fixed the clock to 3.6 GHz during our tests. According to manufacturer specifications, all cores may run continuously at full load at that frequency level with the chip remaining within its thermal design limits.

Lookup performance was measured using synthetic streams of uniformly random IPv4 keys, excluding multicast and reserved address ranges. To remove the overhead of random number generation and its impact on CPU caches, the keys were precomputed and stored in a flat 1 Gbyte array, where they were fetched from within the timing threads running on each core.

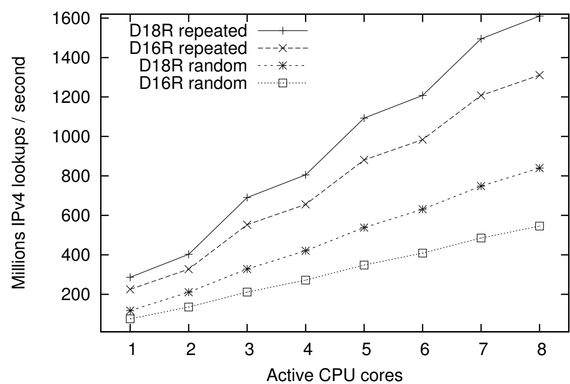


**Figure 3: Average IPv4 lookup performance. The top two curves are for repeated lookups (mimicking a more realistic traffic pattern) of the same targets, the bottom curves are for random address lookups. D18R is slightly better than D16R due to a larger fraction of hits in the main table. The staircase in the top curves is due to interference in accessing the shared L2 cache between core pairs.**

We evaluated performance of two different configurations of our lookup scheme. In the D16R scheme, the first 16 bits are resolved via the lookup table; the D18R scheme uses 18 bits to index the lookup table, trading a potential reduction in the length of the binary search with an increased memory footprint of the lookup table.

## 4.1 DXR performance

Figure 3 shows the average lookup throughput as a function of the number of active execution threads. The pattern of requests has a significant affect on cache effectiveness. With random search keys (a worst-case situation) D18R goes from 117 million lookups per second (Mlps) on a single core, to 840 Mlps with all 8 cores active. Performance with repeated lookups for the same keys (in a sense a best-case option, trying to mimic long lived flows) is almost twice as fast, with 285 Mlps on one core, and 1600 Mlps on 8 cores. On the particular system we are using, interference on the use of L2 caches shared between core pairs produces the staircase effect in the graph.

The cost of a lookup depends on the number of search steps: matches in the lookup table are very fast, whereas moving to the binary search phase makes things slower. This is the reason why the D16R scheme is slower than D18R: as shown by Table 2, D18R has a much higher hit rate in the lookup table, which saves an expensive binary search.

Figure 4 indicates how the lookup time changes depending on the number of steps in the binary search. A large number of steps also means a larger chunk to search, which in turn means a higher probability of a cache miss. This results in a slightly superlinear dependency on the number of steps, especially for random keys.

## 4.2 Comparison with other schemes

From [11] we can derive the *relative* performance of a number of proposed schemes (the absolute numbers refer to 1999 hardware). To relate these numbers with today's systems, we measured the performance of one of those algorithms – the BSD radix tree – under the same conditions and hardware used to test DXR. The results are shown in Figure 5. The BSD code is about 40 times slower than DXR on 1 core, and collapses as the number of cores increases. Considering that [11] shows a factor of 5 between the BSD code and the best of the other schemes, we believe that DXR is very competitive in terms of performance compared to other proposals for software lookups.

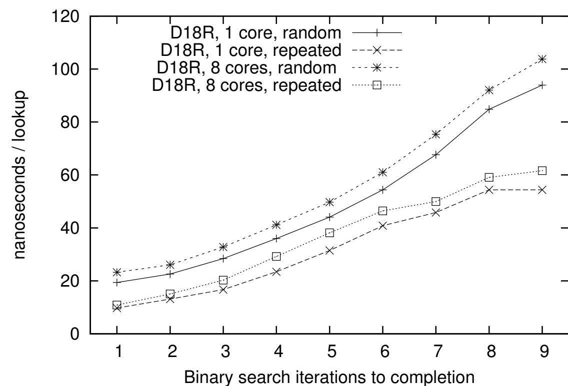


**Figure 4: Average lookup time for D18R depending on the number $n$ of search steps. The use of multiple core adds about 15% overhead, on individual lookups. The request pattern (random vs. repeated keys) almost doubles the lookup time. We only report the D18R numbers as there is no practical difference from the D16R case.**

| | Number of binary search steps | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7+ |
| D18R | 86.4 | 3.55 | 5.15 | 3.02 | 1.47 | 0.40 | 0.04 | $< .003$ |
| D16R | 75.7 | 4.00 | 6.58 | 5.46 | 4.30 | 2.68 | 1.06 | $< .22$ |

**Table 2: Percentage of addresses that require exactly $n$ iterations in the binary search ($n = 0$ means a direct match in the lookup table). D18R has a 10% higher hit rate in the lookup table, but at the price of almost twice the memory (see Table 1).**

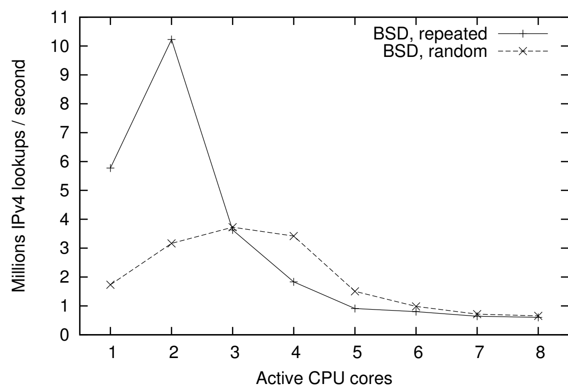


**Figure 5: Average IPv4 lookup performance for the BSD radix tree, in the same configurations used to test DXR.**

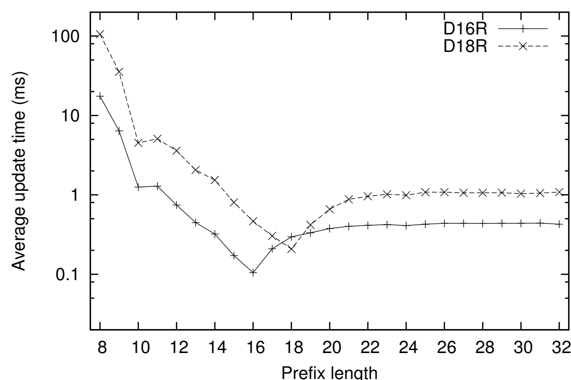


**Figure 6: Average update time per prefix length**

## 4.3 Table updates

As a final experiment, we evaluated the time to update the table when one prefix changes. Timescales are much longer than for lookups, as it is often the case with schemes optimized for lookups rather than updates. Figure 6 shows that, as expected, short prefixes are more time consuming as they require rebuilding a large number of chunks, whereas long prefixes only affect a few entries in the range table.

# 5. CONCLUSION

We have presented DXR, a scheme for doing IPv4 lookups in software at very high speed. Depending on the configuration and query pattern, DXR achieves between 50 and 250 million lookups per second on 1 core, and scales almost linearly with the number of cores, exceeding one billion lookups per second range when running on a commodity 8-core CPU. This makes the algorithm a practical solution for software routing in the 100 Gbit/s range of aggregate traffic, and is an appropriate and timely match with some of recent work [9] on software packet processing. Much of the performance of DXR comes from the simplicity of the algorithm and a very lean memory usage, which permits the working set to fit easily in the cache of modern processors.

Our implementation [1] can be used as a drop-in replacement for the FreeBSD routing lookup, and it is easily portable to other systems or environments, including dedicated appliances.

Because of its nature, DXR pays a toll in terms of update costs, but updating is infrequent, and can be done in parallel with lookups. Our implementation already offsets some of the per-prefix updating cost by batching multiple prefixes in a single update, and should the circumstances require, can rebuild from scratch all lookup structures corresponding to a full sized BGP feed in less than 100 ms.

# 6. REFERENCES


[1] DXR: Direct/Range Routing Lookups home page. http://www.nxlab.fer.hr/dxr/.

[2] T. cker Chiueh and P. Pradhan. High performance ip routing table lookup using cpu caching. In *INFOCOM*, pages 1421–1428, 1999.

[3] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *SIGCOMM Comput. Commun. Rev.*, 27(4):3–14, Oct. 1997.

[4] J. Fu, O. Hagsand, and G. Karlsson. Performance evaluation and cache behavior of lc-trie for ip-address lookup. In *High Performance Switching and Routing, 2006 Workshop on*, page 7 pp., 0-0 2006.

[5] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM*, pages 1240–1247, 1998.

[6] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 195–206. ACM, 2010.

[7] B. Lampson, V. Srinivasan, and G. Varghese. Ip lookups using multiway and multicolumn search. In *IEEE/ACM Transactions on Networking*, pages 324–334, 1998.

[8] S. Nilsson and G. Karlsson. Ip-address lookup using lc-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, 1999.

[9] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Usenix ATC 2012*. Usenix, 2012.

[10] L. Rizzo. Revisiting network i/o apis: the netmap framework. *Communications of the ACM*, 55(3):45–51, 2012.

[11] M. Ruiz-sanchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *IEEE Network*, 15:8–23, 2001.

[12] K. Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, pages 93–104, 1991.

[13] H. Song, F. Hao, M. S. Kodialam, and T. V. Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *INFOCOM*, pages 2518–2526, 2009.

[14] V. Srinivasan and G. Varghese. Faster ip lookups using controlled prefix expansion. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, pages 1–10, New York, NY, USA, 1998. ACM.

[15] 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 *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 29:1–29:12, New York, NY, USA, 2011. ACM.

[16] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed prefix matching. *ACM Trans. Comput. Syst.*, 19(4):440–482, Nov. 2001.