Concise Lookup Tables for IPv4 and IPv6 Longest Prefix Matching in Scalable Routers

Fong Pong, *Senior Member, IEEE*, and Nian-Feng Tzeng, *Fellow, IEEE*

***Abstract*—We present a distinct longest prefix matching (LPM) lookup scheme able to achieve exceedingly concise lookup tables (CoLT), suitable for scalable routers. Based on unified hash tables for handling both IPv4 and IPv6 simultaneously, CoLT excels over previous mechanisms in: 1) lower on-chip storage for lookup tables; 2) simpler table formats to enjoy richer prefix aggregation and easier implementation; and 3) most importantly, deemed the only design able to accommodate both IPv4 and IPv6 addresses uniformly and effectively. As its hash tables permit multiple possible buckets to hold each prefix (following a migration rule to avoid false positives altogether), CoLT exhibits the best memory efficiency and can launch parallel search over tables during every LPM lookup, involving fewer cycles per lookup when on-chip memory is used to implement hash tables. With 16 (or 32) on-chip SRAM blocks clocked at 500 MHz (achievable in today's 65-nm technology), it takes 2 (or 1.6) cycles on average to complete a lookup, yielding 250 (or 310+) millions of packets per second (MPPS) mean throughput. Being hash-oriented, CoLT well supports incremental table updates, besides its high table utilization and lookup throughput.**

***Index Terms*—Border gateway routers, hashing functions, IPv4 and IPv6 addressing, longest prefix matching (LPM), next-hop addresses, prefix expansion and collapsing, routing tables, table load balancing.**

I. Introduction

IP ADDRESS lookups in routers follow longest prefix matching (LPM) over their Border Gateway Protocol (BGP) tables. The address lookups can easily become performance bottlenecks as table sizes or link data rates increase, necessitating scalable LPM designs for future routers. Given BGP tables in core routers have expanded at the pace of 16%–18% annually for the past three years [3], [20] and 40-Gb/s line cards are currently available, scalable and rapid LPM able to support effective incremental updates is most desired. Expected 100 Gigabit Ethernet products will only exacerbate demands on improving on-chip SRAM efficiency to attain needed LPM throughput.

Currently, IPv4 prefixes dominate the BGP tables of Internet core routers, with up to 340 K+ prefixes in a table. There are relatively fewer IPv6 prefixes (varying from 16 to 64 bits [1], [6]). However, it is envisaged to have large-scale deployment of IPv6 in the future, with possibly hundreds of thousands of IPv6 prefixes in a core router. IPv4 and IPv6 are thus likely to coexist for years to come. While ternary content addressable memory (TCAM) is fast and adopted commercially for IPv4 LPM [17], it is expensive to scale and ineffective in handling incremental table updates unless special mechanisms are incorporated [29]. With long prefix length, IPv6 renders TCAM especially unattractive due to a lofty transistor count (of 16) per TCAM bit.

Various LPM techniques with different data structures and search methods exist, commonly aiming to either reduce the number of memory accesses during each lookup [7], [9], [14], [25] or lower the memory requirement (so as to fit the routing table into fast on-chip SRAM) [16]. In general, they are based on either trie structures [7], [9], [14], [25] or prefix hashing/filtering [5], [10], [16], [28]. Trie-based algorithms may adopt compressed data structures and/or techniques (like leaf pushing [7]) to save memory. Unfortunately, prefix table updates are likely to invoke considerable trie rebuilding under such algorithms. Lately, shape similarities among different parts of the trie structure have been exploited [24] to reduce memory usage markedly, albeit to complicated prefix insertion upon incremental table updates. A recent study on parallel IP lookups using multiple SRAM-based pipelines was carried out [11], where a lookup throughput of 7.8 packets per cycle (PPC) time was achieved with eight pipelines.

Bloom filters [4] and their variants have been proposed for IP lookups [8], [22], [23]. Parallel hashing for multiple Bloom filters realized by on-chip memory was considered [8], with one filter assigned for one prefix length. As the prefix length distribution is rather dynamic, a design with distributed and load-balanced Bloom filters (DLB-BF) was introduced later [23]. However, DLB-BF calls for specialized hardware support and is subject to memory inefficiency concerns resulting from its many small point-like SRAM modules.

With O(1) complexity for query and maintenance operations, hash tables are popularly employed to accomplish efficient networking functions, including IP lookups [5], [12], [13], [30]. Based on multiple hash functions, the d-left hashing scheme [5] bounds the maximum number of prefixes mapped into one hash table. It permits all hashes and memory accesses to be carried out in parallel for improved lookup performance. Later, Peacock hashing aims to lower the on-chip memory requirement by storing the main hash table off-chip, while maintaining a hierarchy of relatively small on-chip backup tables to serve as collision buffers [13]. Its throughput, however, is governed by off-chip memory characteristics and structure (such as the latency and the numbers of read ports and banks). Recently, a

Manuscript received October 12, 2010; revised February 11, 2011; accepted August 11, 2011; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor T. Wolf. Date of publication October 06, 2011; date of current version June 12, 2012.

F. Pong is with the Broadcom Corporation, Santa Clara, CA 95054 USA (e-mail: fpong@broadcom.com).

N.-F. Tzeng is with the Center for Advanced Computer Studies, University of Louisiana, Lafayette, LA 70504 USA (e-mail: tzeng@cacs.louisiana.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2011.2167158

hash-based IP lookup scheme using Bloom and fingerprint filters has been treated [30] to achieve O(1) pipelined throughput of table query and maintenance operations, boosting memory efficiency.

This paper pursues a prefix hash-based LPM scheme capable of concise lookup table (CoLT) construction for scalable routers. Central to our CoLT design is on-chip hash tables indexed using transformed prefixes [16], with a route prefix enjoying multiple candidate table entries for load balancing to achieve better table storage efficiency than any earlier LPM design. Those candidate entries are identified by the prefix and its round-down versions for possibly accommodating the prefix. Correct LPM can be ensured if the prefix is migrated (i.e., stored in an alternative table entry indexed by one of its round-down versions, as detailed in Section III-B) by a migration rule. CoLT targets both IPv4 and IPv6 addressing with high memory efficiency. It is the first unified LPM design tailored for situations when both IPv4 and IPv6 coexist, without overprovisioning table width to hold IPv4 prefixes simply because longer IPv6 prefixes are present at the same time. This is achieved nicely by using a separate hash table to keep possible IPv6 prefix portions beyond bit 32, in addition to the first hash table for all IPv4 prefixes and for the first 32-b portions of all IPv6 prefixes. Given that more deployment of IPv6 is expected going forward, the coexistence of both IPv4 and IPv6 will continue for years, making this unified CoLT design particularly attractive for swift IP lookups.

CoLT is shown to arrive at LPM throughput exceeding 310+ millions of packets per second (MPPS) if SRAM operates at 500 MHz (common to 65-nm technology [23]). An earlier trie-based design with eight SRAM pipelines [11] is claimed to reach 7.8 PPC [11],[1] but the design cannot handle incremental updates efficiently (unlike CoLT). A recent scheme combines the hash operation and the multibit-trie compressed structure to achieve scalable LPM lookups [32]. It is evaluated using one FPGA (Xilinx Virtex-4) board plus four SDRAM chips, incurring a hardware accelerator.

Key Contributions: CoLT requires substantially less storage than earlier LPM schemes [5], [7], [31] due to its support of the following:

- prefix migration among hash buckets, letting a route prefix installed in one of multiple candidate buckets to alleviate adverse hash collisions;
- rich prefix aggregation and a simple lookup table format;
- most importantly, unified data structures for both IPv4 and IPv6 to accommodate prefixes in a memory-efficient way seamlessly.

The simple CoLT lookup table with prefix aggregation (e.g., P_3 and P_5 aggregated into one entry in Set 1) and prefix migration (i.e., P_8 migrated from Set b to Set 0) was demonstrated in Fig. 3. This rich aggregation lowers the lookup table size, and prefix migration improves load balancing on table entries to reduce hash collisions drastically (as can be found in Table I, where the overflow probability is reduced from 17.59% down to 1.6% under the BGP Table of rrc00 and the set capacity of 4). With drastic reduction in the memory requirement, CoLT lets major lookup data structures fit in on-chip SRAM for high lookup throughput. The reduction amount is expected to grow with larger BGP tables and/or more IPv6 prefixes involved going forward.

[1]For comparison, the metric of PPC is used since it is *independent* of the SRAM clock rate, which is determined by many complex factors, including sense amplifiers. Per our experience, SRAM can get a realistic speed up to some 550 MHz under a typical 65-nm LP library (or 650 MHz under a faster 65-nm G process with larger area and much higher power and leakage, e.g., see [33]). It may not reach 1.33 GHz assumed in [11].

II. Background on Hashing-Based LPM

A. Prefix Hashing

Hashing prefixes to a flat data structure can achieve fast lookups, particularly if the hash table is made concise enough to fit in on-chip SRAM. A hash function usually takes bit strings of fixed length to produce values over a predetermined range. To mitigate hash collisions, the d-left scheme uses multiple hash functions to index a hash table that is split into disjoint parts which together hold the prefixes, with one function for indexing one part so that a prefix is put in the indexed bucket with the lightest load [5], [27]. Similarly, multiple candidate locations in a table for a prefix can be determined by multiple hash functions, with the prefix put in the favored location (e.g., the least loaded one) among those candidates [2].

To deal with prefixes of variable lengths, an early solution employed multiple (say, α) hash tables, one for each possible prefix length, to hold a BGP table [28]. It incurred a high cost for those α hash tables. Later, the use of a counting Bloom filter was pursued [22] (as an approximation filter) to support exact matches using multiple hash functions keyed by prefixes obtained from *controlled prefix expansion* (CPE) [26]. CPE transforms prefixes of variable lengths to ones with predetermined lengths by expanding a given prefix to one with the next longer, predetermined length. Separately, a hash-based design for LPM, called Chisel [10], was considered later. Chisel achieves near-collision-free hashing (with a probability arbitrarily close to 1) and handles prefixes with variable lengths by collapsing those with length x ($>l$, a permissible length) into prefixes of length l, before performing "leaf pushing" to expand every such collapsed prefix to multiple prefixes of length l' (being the next permissible length longer than x).

B. Prefix Transformation and Table Consolidation

Prefix transformation was first introduced in a superior storage-efficiency (SUSE) design [16]. Under prefix transformation, a prefix P of length w (expressed by P/w) is treated as a *polynomial*, $\mathrm{p}(x)$, of degree $w-1$, i.e., $\mathrm{p}(x) = c_{w-1}x^{w-1} + \cdots + c_1x^1 + c_0$ defined over a Galois Field GF(2), so that its coefficients are either 0 or 1. The algebra under GF(2) is modulo 2, namely, the exclusive-or (XOR) operators. Given a *primitive* generator $g(x)$, we have $\mathrm{p}(x) = q(x) \cdot g(x) + r(x)$, where $q(x)$ is the quotient and $r(x)$ is the remainder. This *transformed prefix representation* (TPR) permits us to characteristically differentiate polynomial $\mathrm{p}(x)$ using $q(x)$ and $r(x)$. In other words, two distinct polynomials $\mathrm{p}_1(x) = q_1(x) \cdot g(x) + r_1(x)$ and $\mathrm{p}_2(x) = q_2(x) \cdot g(x) + r_2(x)$ can be differentiated by their unique pairs of quotients and remainders, i.e., $\{q_1(x), r_1(x)\} \neq \{q_2(x), r_2(x)\}$ iff $\mathrm{p}_1(x) \neq \mathrm{p}_2(x)$ [16].

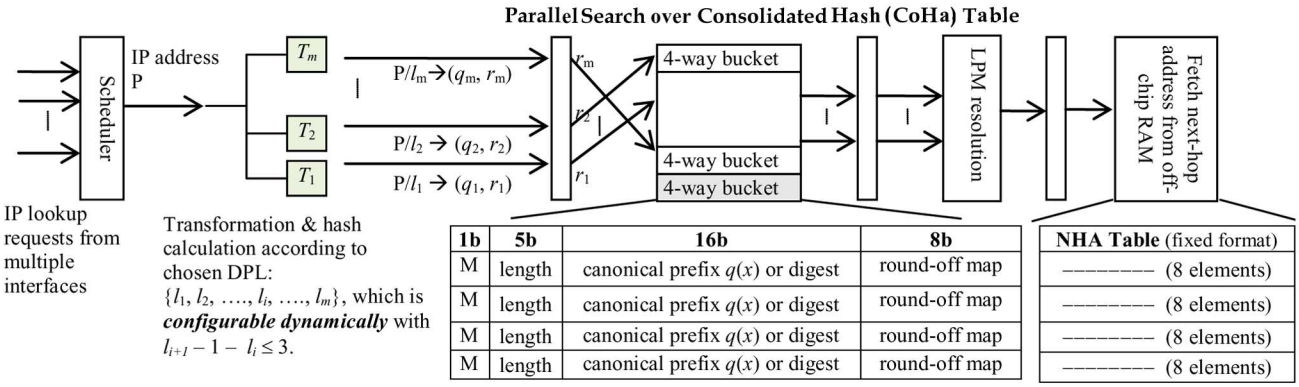

Fig. 1. CoLT design with a generator polynomial of degree =16 under IPv4.

Given a hash table composed of $2^{|g(x)|}$ buckets, it is sufficient to keep only the quotient $q(x)$ in the hash table, indexed by the remainder $r(x)$ for storage savings, where $|g(x)|$ is the degree of generator $g(x)$. With $|g(x)| = 16$ in Fig. 1, for example, TPR lets a prefix P/24 (or P/30) yield an 8-b (or a 14-b) quotient and a 16-b remainder. A hash table comprising 2^{16} sets can then be indexed by the remainder, with a corresponding quotient (instead of the whole prefix) kept in an entry of the indexed set. Storing transformed representations $q(x)$ of prefixes (instead of prefixes themselves) reduces the width of table entries, lowering the overall table storage requirement.

A single set-associative hash table was employed previously by SUSE to enhance storage efficiency and cut down the number of hash probes (and tables) for LPM lookups [16]. Prefixes in the routing table are rounded down to *designated prefix lengths* (DPL) before employed as keys for the set-associative hash function. Specifically, any prefix P/w with $l_i \le w < l_{i+1}$ under a designated prefix length set, DPL: $\{l_1, l_2, \ldots l_i, \ldots l_m\}$, where l_i denotes a prefix length (also referred to as a tread), is rounded down to P/l_i before its hash value is calculated [16] according to TPR described above.

III. Concise Lookup Table Design

This section introduces a novel hash table-based design for scalable routers under IPv4 addressing, with its extension to deal with IPv6 addressing treated in Section V. The CoLT design enjoys fewer hash overflows and achieves better hash storage efficiency than its earlier SUSE counterpart [16] due to the existence of multiple hash buckets for each prefix and rich prefix aggregation (as stated in key contributions of Section I). Being the very first to employ unified hash tables for both IPv4 and IPv6 addressing, CoLT exhibits better memory efficiency than providing separate hash tables respectively for IPv4 and IPv6, as described in Section V.

A. CoHa Table and LPM Lookups

As depicted in Fig. 1, one *co*nsolidated *ha*sh (CoHa) table is adopted under CoLT to hold all prefixes of possible lengths. To accommodate a routing table with up to 340 K+ prefixes (in large core routers) now, the hash table is assumed to comprise 64 K ($=2^{16}$) sets (or interchangeably, buckets) for easy explanation, with each set having four entries to tolerate hash collisions. Totally, the CoHa table has 256 K entries. An entry

Add_Route_Prefix(u32_t P, int *w*)::
Find l_i in DPL: {l_1, l_2, … l_i, … l_m} such that $l_i \le w < l_{i+1}$.
Calculate *q*(*x*), *r*(*x*) on P/32, comprising P/l_i *concatenated with* (32 - l_i) 0's.
Find a free entry *e* in bucket r(*x*) or an existing entry *e* with same canonical prefix *q*(*x*) {
Set Length field to properly keep track of length l_i.
Keep *q*(*x*) in e.
Set the k^{th} bit of the round-off map, for all *k* equal to the values of those (*w* - l_i) bits concatenated with $v = (l_{i+1} - w - 1)$ bits of 2^v binary combinations. Next, program the off-chip NHA table entry accordingly. (If a bit was set by a prior longer prefix, keep the content for the longer prefix.}

LPM(u32_t P)::
For all treads l_i in DPL: {l_1, l_2, … l_i, … l_m} {
Calculate *q*(*x*), *r*(*x*) on P/32, comprising P/l_i *concatenated with* (32-l_i) *0's*.
For each valid entry *e* in bucket *r*(*x*) {
Find *e*.Length = l_i, and check if *q*(*x*) == *e.q*(*x*).
Assume stride $s = (l_{i+1} - l_i - 1)$, for all round-off length *y* ranging from *s* down to 0, extract *y* round-off bits from P, concatenated with (*s* – *y*) 0's, and check if the k^{th} bit of the round-off bit map is set, where *k* equals the value of the composed round-off bits. When all checks pass, a route exists in bucket *r*(*x*), entry *e*, with its prefix length being (*e*.Length + round-off length *y*.) } }
Select LPM among all matched prefixes.

Fig. 2. Pseudocodes for prefix additions and LPM with respect to prefix P under CoLT.

of the CoHa table consists of four fields: 1) a 1-b field to record if the entry holds a canonical $q(x)$ after prefix transformation (for $\mathrm{M} = 0$) or a signature after prefix migration (detailed in Section III-B); 2) a 5-b indictor to specify the tread length of the prefixes (ranging from P/8 to P/32) stored in the entry; 3) a 16-b canonical $q(x)$ or a signature; 4) an 8-b map to record path aggregation based on the values of round-off bits. Note that a prefix is viewed as "migrated" when stored in an alternative (i.e., a colonial) table entry.

A new prefix is inserted in the CoHa table, after the prefix P/w with $l_i \le w < l_{i+1}$ is rounded down to the nearest, shorter tread P/l_i ($\in$ DPL) and then corresponding $q(x)$ and $r(x)$ of a P/32, comprising P/l_i concatenated with $(32 - l_i)$ 0's, are calculated, as listed in the "Add Route" pseudocode of Fig. 2.[2] Each table entry holds a "(tread) length indicator" for w plus

[2]Henceforth, for simplicity with respect to the context of hashing, we will use the shorthand notation of P/w to denote that the input to a hash function, P/32, is derived by concatenating P/w with $(32 - w)$ 0's.

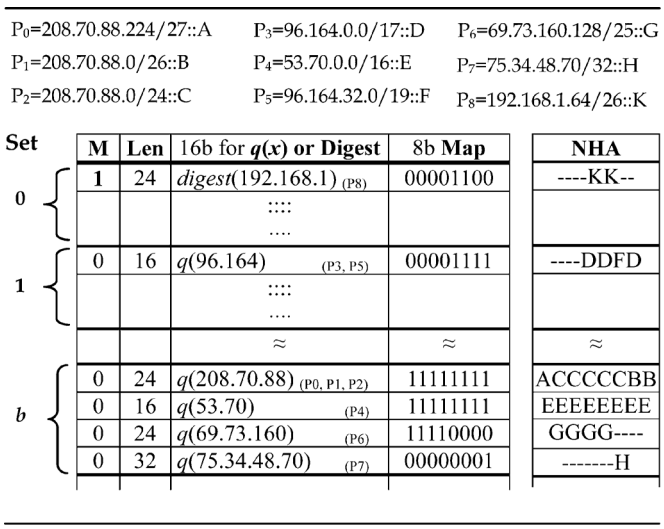

Fig. 3. Example CoHa table and NHA contents after admitting nine given prefixes, with P_8 migrated from Set b to its colony Set 0.

its rounded off $(w - l_i)$ bits, encoded by a round-off bitmap of $2^{(w-li)}$ bits that enables route prefix aggregation to lower storage. Clearly, the procedure for adding a prefix is straightforward, provided that the bucket indexed by $r(x)$ still has a free entry. When a route is withdrawn, its associated entry in the hash table is identified in the same way as an insertion; the identified entry then sets its availability bit.

Once a new entry is added to the hash table, the next-hop data of the entry is then registered in next-hop storage (off-chip RAM, as indicated in Fig. 1), which is disassociated from the hash table (following several earlier LPM designs [10], [22], [23]). Under CoLT, *all prefixes* that lead to the *same rounded-down prefix* can be stored in the same hash table entry inside the bucket indexed by $r(x)$ obtained from the rounded-down prefix, attaining exceptional prefix aggregation for concise lookup tables. Each CoHa table entry is associated with one NHA map, which signifies one $(l_{i+1} - l_i - 1)$-level binary trie whose leaves record NHA data of all prefixes aggregated in the CoHa table entry, under the designated prefix length set of DPL: $\{l_1, l_2, \ldots l_i, \ldots l_m\}$, with $l_i < l_{i+1}$. NHA map storage can be off-chip commodity SRAM devices, unlike the CoHa table implemented by on-chip SRAM for high lookup throughput. In the example design of Fig. 1, each NHA entry denotes a three-level binary trie, involving the fixed format of $\mathrm{e}_7\mathrm{e}_6\mathrm{e}_5\mathrm{e}_4\mathrm{e}_3\mathrm{e}_2\mathrm{e}_1\mathrm{e}_0$, where each e_i (for $7 \leq i \leq 0$) specifies one NHA. The initial value of the binary trie is "$--------$," representing the default NHA. Once a prefix with its NHA being "χ" is rounded down with the round-off bits of "10" before aggregated into a CoHa table entry, its corresponding NHA map is set to "$--\chi\chi----$." For the example of Fig. 3, seven of nine prefixes map to the same four-way bucket (i.e., Set b) in the CoHa table, following TPR under $\mathrm{DPL} = \{8, 12, 16, 20, 24, 28, 32\}$. Among those nine prefixes, $\mathrm{P}_0/27, \mathrm{P}_1/26$, and $\mathrm{P}_2/24$ are aggregated in one CoHa table entry of the bucket, with its 16-b field storing the quotient of $\mathrm{P}(208.70.88) \div \mathrm{g}(x)$. P_4, P_6, and P_7 take other entries, and $\mathrm{P}_8/26$ would have to require a separate entry in the same bucket, causing an overflow. If $\mathrm{P}_8/26$ is allowed to round down to $\mathrm{P}_8/\{20, 16, 12, 8\}$, which may index alternative buckets (e.g., Set 0) for the prefix, an overflow is avoided. In this case, a *digest* (rather than $q(x)$) of P_8 is stored in the 16-b field of the CoHa entry, signifying the result of *migration* (when a prefix is held in an alternative set indexed by one of its round-down prefixes}, as detailed in Section III-B). Meanwhile, $\mathrm{P}_3/17$ and $\mathrm{P}_5/19$ are aggregated in one entry of Set 1, as illustrated in Fig. 3. This is a distinct advantage in attained prefix aggregation over the earlier SUSE design [16]. Those aggregated prefixes stored in one table entry are used to establish their corresponding next-hop address (NHA) map, which contains a fixed format to permit direct accesses to NHA information (without search) after an LPM lookup match. With effective aggregation under CoLT, the total number of hash table entries can be smaller than the number of prefixes in the BGT table while achieving an extremely low spillover probability, as will be demonstrated in Section IV.

The purpose of migration is to avoid situations that otherwise could overflow indexed buckets, by allowing multiple candidate buckets per prefix to achieve far better table utilization. It is not meant for balancing the load over hash buckets since storing a given prefix in any one of the candidate buckets has no effect on lookup performance or accuracy. For CoLT with $|\mathrm{DPL}| = m$, the lookup of each arrival packet requires to index m hash table entries (using its destination IP address plus all round-down versions of the address) before choosing the one with the longest length among those indexed entries, constituting the LPM resolution and involving the operational time complexity of O(1). Search over those indexed entries in the CoHa table can be done *in parallel* if multiple memory blocks are employed to construct the table. Inherent to the nature of hash tables, the design supports incremental table updates effectively, well amenable to future routers whose table dynamics are expected to rise continuously. Note that prefixes of different lengths after being rounded down may turn into the same key to the hash function. If a new prefix is hashed to candidate sets that are all full, an overflow happens, and the prefix is then stored in a separate overflow set (like the victim cache) or TCAM.

B. Improving Hash Table Storage Efficiency

As depicted in Fig. 1, the CoHa table holds the transformed representations $q(x)$ of prefixes to reduce the total table size. The CoHa table can be made more effective by permitting more than one candidate bucket for a given rounded-down prefix. It is achieved by employing a distinguished "*transitive property*" pertaining to prefixes—namely, for any prefix P/w, P/t is a prefix of P/w for all $t < w$, arriving at CoLT. Due to the transitive property of prefixes and the fact that the LPM search algorithm of Fig. 2 looks for all tread lengths, a route prefix $\mathrm{P}/w, l_i \leq w < l_{i+1}$, can be stored in *any one of the* i *candidate sets* indexed by round-down P (i.e., $\mathrm{P}/\{l_1, l_2, \ldots l_i\}$ under DPL: $\{l_1, l_2, \ldots l_i, \ldots l_m\}$). We call the additional buckets identified by P/t, for all $t < l_i$, the "*colonies*" (or colonial buckets, colonial sets) for P/w. With colonies likely to exist in the CoHa table for a prefix, the likelihood of the prefix unable to find an available entry drops dramatically, yielding very high table storage efficiency.

In Fig. 3, prefix $\mathrm{P}_8/26 = 192.168.1.64/26$ is initially rounded down to $\mathrm{P}_8/24 = 192.168.1.0/24$ before mapped to Set b, causing an overflow. However, it is subsequently migrated to Set 0, as shown in the figure. In fact, prefix P_8 can be installed in any one of the hash buckets indexed by Hash $(192.168.1.0/\{24, 20, 16, 12, 8\})$, under DPL of $\{8, 12, 16, 20, 24, 28, 32\}$. Due to the nature of LPM and P/8 being a prefix of P/12 (which in turn is a prefix of P/16 and so on), the parallel search algorithm of Fig. 2 performed with respect to all DPL tread lengths *never* reports a *false negative* (i.e., mismatch), provided that P_8 is installed in one of its colonies.

Lemma 1: Given any prefix of P/w, denoted by P/t for $t < w$, storing P/w in any one of the hash buckets indexed by Hash (P/32 | for P equal to P/t *concatenated* with $(32-t)0'$s) will not cause false negatives on P/w, provided that all those indexed buckets are searched during lookups.

Lemma 2: In general, the suffix appended to P/t (to form the $\mathrm{P}/32$ hash key) in Lemma 1 can be any constant string C, as long as the same suffix C is used consistently during adding routes and LPM lookups.

This way increases the capacity of a bucket and the degree of tolerating hash collisions from 4 to $i \times 4$, making the CoLT behave like an $i \times 4$ set-associative design under ideal conditions to enjoy exceeding storage efficiency. This property also suits for LPM well because most prefixes in a routing table are of length 24 or larger.

To take advantage of transitive prefixes, we need to handle the transformed canonical prefix $q(x)$ carefully. Because an installed route prefix P is identified by its $q(x)$ and $r(x)$, but the hash index $r(x)$ is not explicitly stored in the CoHa table, P can no longer be accurately located by $q(x)$ and $r(x)$ when P moves to its colonies (say, from the bucket indexed by $r(x)$ to that by $r'(x)$). Fortunately, this issue can be dealt with using the *digest* of P obtained according to the remainder under the target colony of P, plus $q(x)$ and $r(x)$, as follows. Consider the prefix $\mathrm{P}/24 = 192.168.1.0/24$, which can be expressed by

$$\mathrm{P}_{24}(x) = q_{24}(x) \cdot g(x) + r_{24}(x)$$

under the (primitive) generator polynomial $g(x)$ of $x^{16} + x^{12} + x^3 + x^2 + 1$. Since the polynomial context is understood, we shall drop the variable x from the preceding expression henceforth for clarity. As demonstrated by Fig. 4, when the bucket indexed by r_{24} is full, CoLT checks if the bucket for P/16 (following the transitive property for P/24) has room to host P/24, utilizing

$$\mathrm{P}_{16} = q_{16} \cdot g + r_{16}.$$

If q_{24} is stored in the bucket indexed by r_{16}, $\langle q_{24}, r_{16} \rangle$ no longer correctly identify the original prefix P/24 during lookups. Mathematically, we want a digest stored in the bucket indexed by r_{16} instead, such that $\mathrm{digest} \cdot g + r_{16} = q_{24} \cdot g + r_{24}$, and the digest stored therein indicates P/24. The above equation means $\mathrm{digest} = q_{24} + r_{24} + r_{16}$, if it is divided by g and r_{16} is moved to the right-hand side (under modulo 2 operations). As a result, after migrating P/24 to the bucket indexed by r_{16}, the M (migration) bit of the table entry is turned on, with the length field set to 24, as shown in Figs. 1 and 4. It signifies that the entry

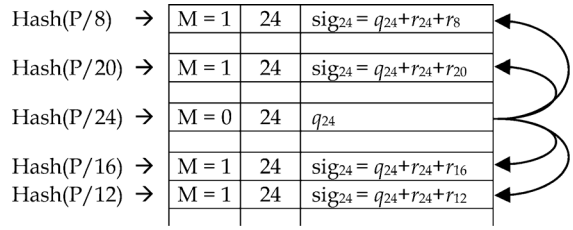

Fig. 4. Example of LPM with respect to migrated prefixes in CoHa table.

keeps the digest of a P/24 prefix as a result of migration. For the migrated entry to match with P/24 during lookups, every destination IP address and all its round-down prefixes following the DPL treads need to calculate their $q(x)$'s and $r(x)$'s for search. As an example, IP address 192.168.1.100 leads to search using $\langle q'_{24}, r'_{24} \rangle$ and $\langle q'_{16}, r'_{16} \rangle$. Thus, inspecting the bucket indexed by r'_{16} discovers that a migrated P/24 entry exists. It calls for validating if $(q'_{24} + r'_{24} + r'_{16} + \text{stored digest for } \mathrm{P}/24) = 0$, involving simple XOR (addition) operations under GF(2). Since every lookup calculates $q(x)$ and $r(x)$ for all DPL treads (see LPM in Fig. 2), q'_{24}, r'_{24}, and r'_{16} are thus available for fast validation. Clearly, if $(q'_{24} + r'_{24} + r'_{16} + \text{stored digest})$ is 0, we have $\langle q'_{24}, r'_{24} \rangle$ equal to $\langle q_{24}, r_{24} \rangle$, a lookup match with P/24.[3]

Clearly, CoLT performance is dictated by the chosen DPL set, calling for simple examination of the routing tables to guide DPL selection. It may follow a simple heuristic aiming to ensure that distinct prefixes between two consecutive treads are abundant. For example, with the current tread of $l_{i+1} = 28$, the next tread l_i is selected from candidates in $\{27, 26, 25, \ldots\}$ such that the number of unique prefixes (after rounded down to l_i) is maximal. The heuristic works as long as the hashing function gives reasonably uniform distributions on unique keys, a prerequisite for the hashing function.

C. Avoiding Ambiguity

The method detailed above is effective in balancing load distributions across the CoHa table by moving prefixes across hash buckets, with their digests kept in target entries. However, ambiguity could result, as discussed next.

Consider a route prefix of P_1/m. Without load balancing via migration, P_1/m $(= q_1(x) \cdot g(x) + r_1(x))$ is stored (with its q_1 kept) in the bucket indexed by r_1. If P_1/m is subsequently moved to a colony bucket r_t (determined by a shorter prefix P_1/t, for $t \in \mathrm{DPL}$ and $t < m$), with the digest of $\mathrm{digest}_1 = q_1 + r_1 + r_t$ stored in Bucket r_t. Our concern here is whether the lookup of an arrival IP address may *falsely* match Prefix P_1/m stored in Buckets r_t. To facilitate subsequent discussion, let us assume that P_1 is a 32-b route. The bucket indexed by r_t can be any colony indexed by a shorter prefix $\mathrm{P}_1/\{28, 24, 20, 16, 12, 8\}$. The ambiguity problem is explained as follows.

1) $\mathrm{P}_1/32$ $(= q_1 \cdot g + r_1)$. For a shorter thread $t \in \{28, 24, 20, 16, 12, 8\}$, Colony r_t is obtained by finding the remainder $\mathrm{rem}(\mathcal{L}(\mathrm{P}_1)/t)$, where $\mathcal{L}(\mathrm{P}_1)$ refers to the leading t-bit prefix. Let $\mathcal{T}(\mathrm{P}_1)$ denote the trailing

[3]When the CoHa table is sized properly to involve $2^{|g(x)|}$ buckets, there is no alias to r_{16}, the index for the colony. Hence, the digest may comprise solely $q(x)$ and $r(x)$ of the original prefix P/24.

$(32-t)$-bit suffix of P_1. The polynomial representation of P_1 can thus be expressed by $\mathcal{L}(\mathrm{P}_1)x^{(32-t)} + \mathcal{T}(\mathrm{P}_1)$.

2) Given an input IP address $\mathrm{P}_v/32$ $(=q_v \cdot g + r_v)$, the lookup algorithm searches for buckets indexed by all treads, as stated in Fig. 2.
3) If any ambiguity occurs, $\mathrm{rem}(\mathcal{L}(\mathrm{P}_v)/t)$ must result in the same index r_t and passed validation of the stored signature for P_1, signifying $(q_1 + r_1 + q_v + r_v)$ equal to 0.

To ensure that the digest is sufficient in *uniquely* characterizing a moved prefix, the next rule is followed upon prefix migration, with its proof given in the Appendix.

Migration Rule: Given $l, t \in \mathrm{DPL}$ and a $g(x)$ of degree $k(= |g(x)|)$, a prefix P/m, rounded down to the nearest tread P/l for $l < m$, is allowed to migrate to its colony indexed by round-down prefix P/t, $t < l$, provided that $(l - t) \leq k$.

The impact of the prefix migration rule on table storage efficiency is generally very light. Given $\mathrm{DPL} = \{32, 28, 24, 20, 16, 12, 8\}$ and a degree-16 $g(x)$, for example, the rule means that prefixes P/32 may not migrate to sets indexed by P/12 and P/8, but migration opportunities are still abundant making use of $\mathrm{tread} = 28, 24, 20$, and 16. Similarly, P/28 only lose one candidate set indexed by P/8. This negligible impact due to the migration rule was confirmed by our study using the real-world routing tables. Furthermore, by design, more treads can be selected carefully and added to DPL such that the opportunity for migration grows.

D. Next-Hop Accesses

Unlike the CoHa table (which is on the critical path dictating the LPM lookup throughput), next-hop storage is fetched by hit information via indexing directly without search. The current CoLT architecture keeps a simple design where the access to off-chip NHA storage is *targeted*, involving always one RAM access per lookup and thus rendering the off-chip NHA storage size not essential. We place little emphasis on engineering the NHA part for perfection in this paper, as common to recent LPM designs [10], [22], [23]. Nevertheless, it is easy to devise a simple solution, such as one with base pointers aided by bitmaps (or offset counters), similar to Lulea [7] and others [31], for packing NHAs into a compact data structure, should NHA storage conservation be necessary.

Given the LPM rate of hundreds of MPPS is desired, CoLT can employ *multiple* banks of such today's commodity memory as Cypress's QDR®II+ SRAMs (clocked up to 550 MHz [33]) to hold NHA information. With such a QDR®II+ SRAM, four data beats are pumped to its output ports at both the rising and the falling edges of each clock in two consecutive cycles, signifying that one request can be accepted by the SRAM per two cycles. As a result, NHA storage implemented by such SRAM devices supports 275 M accesses/second per device. Given CoLT makes targeted accesses to NHA (without search), two such SRAM devices permit up to 550 M accesses per second, keeping pace with the CoHa table's lookup rate.

IV. Empirical Evaluation

Empirical evaluation of the CoLT architecture and comparison to other methods are made by using real (IPv4) routing tables collected for public use [3], [20]. Due to space limitations, we demonstrate results for only a few large routing tables. However, the outcomes of all tables available in [3] and [20] dated between 2002 and 2011 follow the same trends, consistent with those illustrated in this section. CoLT achieves a balanced hashing load distribution, high lookup performance, and exceeding memory efficiency due to small storage for concise lookup tables with simple and unified formats.

TABLE I
Load Distributions for Routing Tables Under CoLT With 64 K × 4 Entries (Amounting to 7.5 Mb), Given $\mathrm{DPL} = \{8, 12, 13, 16, 20, 22, 25, 29\}$ and Φ = Number of Prefixes. Results With Migration Are Listed in Next Shaded Columns

Load	For all $\Phi \geq$ 340K prefixes (in January 2011)					
	rrc00 (Φ = 348,866)		rrc01 (Φ = 341,163)		rrc14 (Φ = 342,929)	
0	4.00%	2.33%	4.65%	2.89%	4.47%	2.73%
1	14.10%	7.92%	15.14%	9.16%	14.88%	8.81%
2	23.07%	13.43%	23.58%	14.60%	23.46%	14.40%
3	23.85%	41.7%	23.63%	42.00%	23.67%	42.03%
4	17.41%	33.02%	16.77%	30.05%	16.95%	30.66%
5	10.34%	1.6%	9.65%	1.30%	9.91%	1.37%
6	4.73%		4.40%		4.44%	
7	1.74%		1.52%		1.56%	
8	0.57%		0.48%		0.49%	
≥ 9	0.21%		0.18%		0.17%	
Utilization		72.9%		71.1%		71.5%

A. Hash Load Distribution Aided by Migration

Table I displays the load distribution outcomes of the CoHa table under BGP tables of rrc00, rrc01, and rrc14, each with more than 340 K prefixes when obtained in January 2011. The results are shown to demonstrate the effectiveness of CoLT's migration for handling hash overflows under large BGP tables. The column of *load* refers to the number of taken entries (l) of a bucket in the CoHa table. Two sets of results, in terms of the percentage of buckets (out of 64 K total buckets) each with l occupied entries, are listed for every prefix table, one for the basic hash scheme and the other involving hash entry migration. Clearly, the load distributions can be quite unbalanced without migration. With transitive prefix migration to move entries from congested buckets to lightly loaded ones, CoLT arrives at a balanced hash table with utilization as high as 72.9%. Totally, there are about 1000 overflows (for each table), able to be overcome by a spillover TCAM (as adopted by earlier designs [10], [16], [23]).

DPL Selection to Better Load Distribution: It is expected that more even hashing distribution and fewer overflows can be achieved by selecting more suitable DPL treads. Consider $\mathrm{DPL}_\alpha = \{8, 9, 13, 17, 21, 25, 29\}$, which is a revision of the DPL used to produce the results of Table I. As DPL_α contains one fewer tread and has wider stides, two conflicting effects arise: more aggregation due to wider strides and less migration due to fewer colonies for a given prefix. More aggregation reduces the CoHa table load (to accommodate more prefixes without overflows), whereas less migration hurts load balancing (to cause overflows sooner). With DPL_α and the same

TABLE II
COLT LOOKUP PERFORMANCE (IN CYCLES) UNDER RRC00

Memory # M	Single-Port SRAMs		Dual-Port SRAMs	
	Average	Worst	Average	Worst
8	2.5	8	1.5	4
16	2.0	7	1.2	4
32	1.6	6	1.1	3
64	1.4	6	1.0	3

table size as specified in Fig. 1, for example, the CoHa table has lower utilization of 51% under rrc00.

Based on prefix transitivity, CoLT permits supplementary *treads* to be added to the DPL set freely for more prefix migration opportunities, provided that buckets indexed by the added threads are also searched during LPM lookup (see Lemma 1). This way of activating treads adaptively is deemed as a *unique feature* of CoLT, able to cut down overflows considerably. As an example, adding an extra tread of 16 to DPL_2 lowers the number of overflow down to 1, under the largest BGP table, rrc00, at the expense of eight hash probes per LPM lookup (in contrast to seven probes under DPL_α). All results reported in the rest of this article are based on $\mathrm{DPL}_\alpha + \{16\}$ for greater prefix migration during table installation.

B. Lookup Performance

Our measurements of lookup performance are done by simulating the CoLT design under *realistic* hardware configurations. Specifically, the CoLT table is assumed to be realized by M on-chip SRAM blocks, which allow parallel issues of eight (i.e., $|\mathrm{DPL}|$) hash probes. Furthermore, a memory block was assumed to be 120 bits ($=4 \times 30$ b, see Fig. 1) wide so that all four elements in a bucket can be reported at once. Each LPM lookup thus involves an access of $|\mathrm{DPL}| \times 120$ bits totally from the CoHa table. Because packet traces for those these prefix tables are not available to the public, we conducted evaluation by randomly choosing an address and/or a prefix from the prefix table to generate lookup addresses. When the tests were generated from a prefix table, e.g., selecting prefix P/16 (denoted by a1.a2.0.0) from the table, test IP addresses were obtained by replacing don't-care bits (i.e., 0.0 in P/16) with arbitrary bit strings a3.a4 to yield a1.a2.a3.a4. Thus, for each given prefix in the studied routing tables, a number of IP lookup addresses would find their LPM results to be the given prefix. Totally, hundreds of millions of lookup packets were generated for evaluation.

Table II depicts lookup performance by CoLT under rrc00 for different configurations, where the hash buckets are interleaved in the M memory blocks, i.e., Bucket b is in the (b/M)th line of the (b modulo M) block. Results for single-port and dual-port SRAM are listed separately. For single-port SRAMs, conflicting accesses to the same block must be serialized, with the worst case taking eight cycles to fetch all needed contents. On the other hand, accesses to different modules are served concurrently. As the number of SRAM blocks increases, the average number of cycles per LPM lookup dwindles. On average, it takes 2 (or 1.6) cycles to complete a lookup under 16 (or 32) blocks. With today's technology, on-chip SRAM can clock at 500 MHz (as stated in [23] and well adopted in industry), yielding a lookup rate of 250 (or 310+) MPPS, adequate to support 16×10-G links practically. In the worst case, CoLT takes 7 (or 6) cycles for a lookup under 16 (or 32) blocks, exhibiting a lookup rate of 71+ (or 83+) MPPS.

Note that the foregoing analysis is solely from the perspective of serializing LPM requests. In practice, requests can be scheduled from multiple interfaces to realize the maximum lookup bandwidth offered by the exemplary memory system. Given a scheduler to harmonize multiple requests in every cycle, the design with M memory blocks can serve requests from multiple interfaces simultaneously. For example, CoLT with eight interfaces and with 16 SRAM blocks of total 7.5 Mb may deliver one lookup per cycle on average, which means a sustainable 500-MPPS rate for an ideal implementation.

Table II also shows lookup performance for dual-port SRAMs to reflect the impact of memory collisions. While dual-port SRAMs may tolerate single structural collisions to reach higher lookup throughput, they can be unfortunately twice as large as their single-port counterparts. Hence, their deployment should be carefully evaluated.

C. Comparison and Discussion

This section compares CoLT to three other best representative methods under the largest BGP table, rrc00.

1) Result Comparison: d-left hashing [5] employs CPE [26], which expands prefix P/w to the next *longer* tread P/l_{i+1} (with $l_{i+1} \in \mathrm{DPL}$ and $l_{i+1} > w$). Under CPE, every tread length is associated with a hash table of *multiple segments*, each fetched using a separate hash function [5]. To add a route prefix, d-left+CPE starts with finding a free entry in the first segment; upon collisions, it percolates the prefix to the next segments in order. The prefix is installed in a free bucket of the smallest numbered segment. We implemented d-left+CPE hashing for evaluation, with the following details: 1) $\mathrm{DPL} = \{12, 16, 18, 20, 24, 28, 32\}$ is used, involving eight hash tables, which are four-way set-associative; 2) a spillover TCAM with the capacity of one hundred entries exists to keep overflows is assumed; 3) the hash table H_l for prefixes (after CPE) of length equal to l is sized by a dilation factor $\boldsymbol{\rho}$, i.e., H_l has ($\rho\times$ number of prefixes P/l) entries; and 4) each hash table is evenly divided into d segments, governed by d independent hash functions. It is found from our implemented evaluation that at least three hash functions are needed to contain overflows, calling for a total of 24 ($=8 \times 3$) memory blocks to permit full parallel hash probes simultaneously. As a result, d-left+CPE hashing takes 12.8 Mb of memory to realize its hash lookup tables, representing some 70% larger storage than provisioned for CoLT (7.5 Mb).

CoLT is also compared to the Lulea tries [7] and the Tree Bitmap scheme [31] with compressed trie data structures, which need to pack data objects (e.g., trie/tree nodes) in *consecutive* memory locations for lowering memory overhead caused by pointers. With such a rigid data structure at the cost of difficulty in supporting incremental updates, nevertheless the Lulea trie (or Tree Bitmap) still takes 13.8 Mb (or 10.7 Mb) to accommodate the rrc00 table (with their NHA storage also excluded for fair comparison). Under CoLT, the CoHa lookup table is

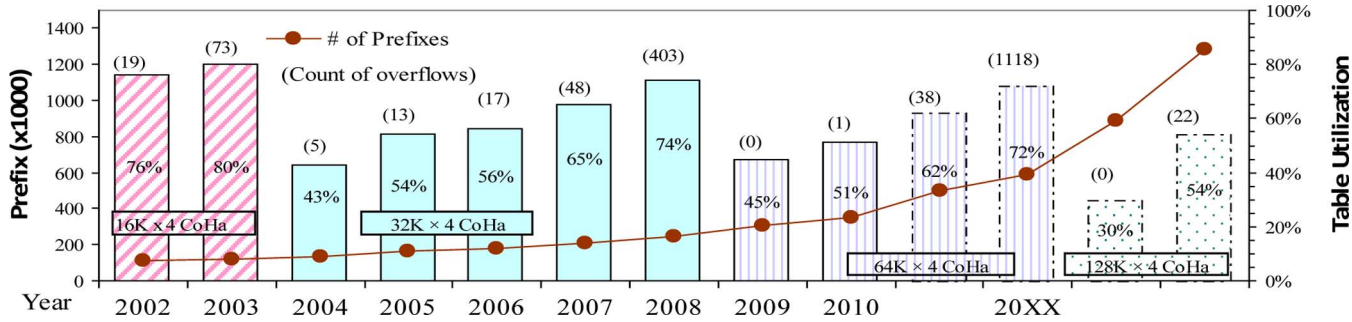

Fig. 5. Growth in prefix tables and corresponding CoHa hash table size to contain overflows.

provisioned with 7.5 Mb on-chip SRAM (with 3.6 Mb actually taken). Hence, CoLT calls for much smaller storage for its lookup table than its Lulea trie and Tree Bitmap counterparts.

2) Further Discussion: From an engineering perspective, CoLT is easier to be made effective by changing the DPL set adaptively for different prefix distributions while keeping the same uniform CoHa table structure as demonstrated in Fig. 1. Treads can also be turned on dynamically (to act like adding them to the DPL set) for better load balancing. Conversely, d-left hashing cannot be made as uniform and storage-efficient as CoLT across various routing tables.

For a small single-port SRAM, the overhead area of its control logics, address decoders, and sense amplifiers dominates (taking up 85% for a 64 × 64-b SRAM) even under 65-nm technology, with the overhead area to rise further under 40 nm and/or multiport memory modules. Therefore, the design with a simple and regular structure comprising a small number of large SRAM modules (like CoLT) is far more *area-efficient* than its d-left hashing counterpart, which involves many small SRAM units of different sizes. The simple CoHa table structure also leads to fast and straightforward memory management, unlike complex management expected for the Lulea and the Tree Bitmap schemes. Being hash-based to incur O(1) time for adding or deleting a route, CoLT well supports incremental route updates, in sharp contrast to any trie-oriented scheme.

D. Scalability

The number of hash probes per lookup under CoLT is a constant, equal to the number of DPL treads and able to be completed in no more than 2.0 cycle on an average for $M \geq 16$ (as shown in Table II), independent of the lookup table size. This fact benefits performance scalability of CoLT, ensuring high performance as BGP tables grow.

To gain insight into CoLT's resilience to overflows, we examined all BGP tables with respect to their growth patterns over years, discovering that all tables exhibit similar trends and statistics. Hence, the outcomes for Table rrc00 from the year 2002 to the present are depicted in Fig. 5 for simplicity. CoLT under a small hash table with 16 K × 4 entries (taking 1.725 Mb) starts to experience growing overflows (as indicated within parentheses) near 2004, when table utilization of 80% is observed. By doubling the hash table capacity to 32 K × 4 entries (involving 3.75 Mb), CoLT works well until 2008, when the hash table is 74% utilized and the number of overflows rises sharply. A larger table with 64 K × 4 entries (7.5 Mb) is required to cope

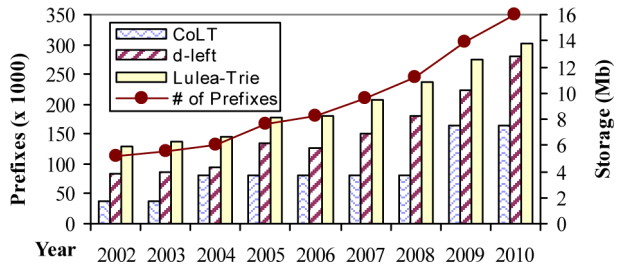

Fig. 6. Scaling for different LPM methods over years.

with growing prefix tables henceforth, bringing table utilization down to roughly 50% even under the largest BGP table, rrc00, with some 340 K prefixes (in the beginning of year 2011).

We conjecture a consistent trend by the past results and by the outcomes of large tables composed by blending together chronological data. As depicted in Fig. 5, a 64 K × 4 CoHa table can handle up to some 590 K routes (under Year 20XX) before the number of overflows become large. Experimentally (with data not shown), we may dramatically cut down the number of overflows and achieve better hashing load distributions with one more DPL tread for enriching migration; more parallel table probes per lookup will be involved as a result. It meets our expectation. Alternatively, with the table capacity doubled to 128 K × 4 entries, no overflow occurs until there are about 1 M routes (denoted by the second rightmost dashed rectangle in Fig. 5. If the number of routes grows to 1.3 M, a few overflows exist, with table utilization approaching 54%.

Fig. 6 shows how different LPM methods scale with growth in routing tables from 2002 to 2011, reflecting storage scalability. For CoLT, the lookup table grows its size at a slower pace than other methods, able to hold up until its utilization reaches some 74% and signifying its better storage scalability.

V. IPv6 Support

A. Lookup Table Construction for IPv4 and IPv6

According to IPv6 address allocation and assignment policy [1], [6], IPv6 prefix length varies from 16 to 64 bits. While it is possible to directly expand our CoLT design to accommodate IPv6 prefixes by widening the CoHa table entries (to hold a larger length field and a longer quotient/digest field; see Fig. 1), such a direct expansion, however, is unattractive because the storage footprint thus grows significantly and the table entries are mostly underutilized since IPv4 addresses

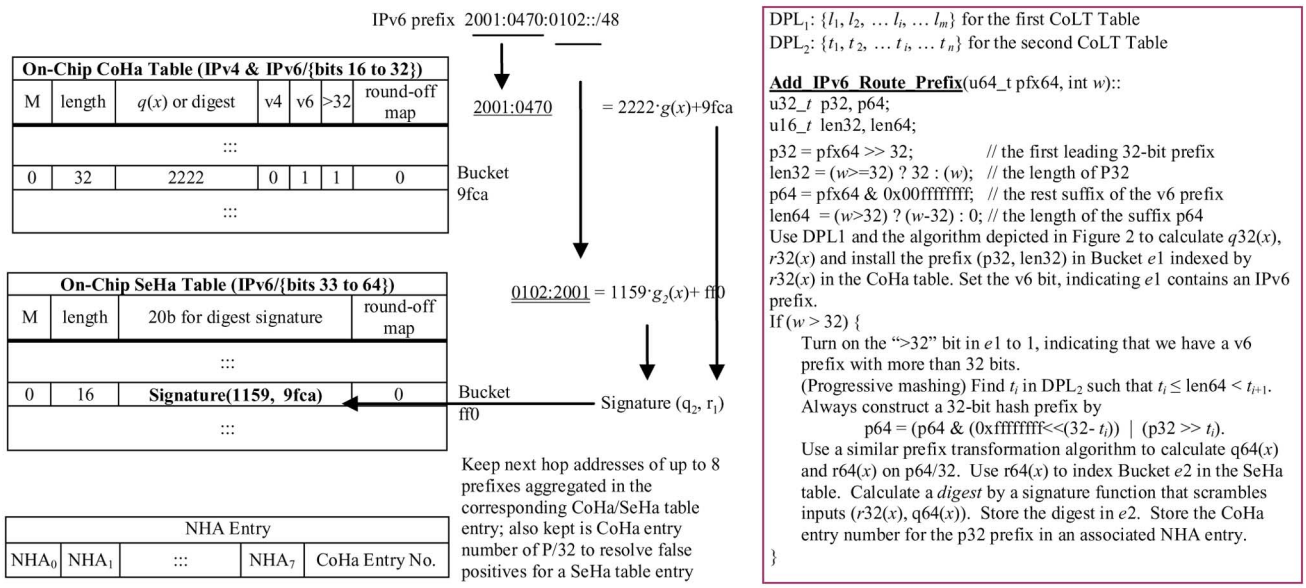

Fig. 7. Algorithm and an example for installing an IPv6 prefix of 2001:0470:0102::/48 under CoLT.

dominate. Furthermore, a larger number of DPL treads are needed in order to keep the strides between consecutive treads reasonably small. Otherwise, the round-off bitmap field can become very long. This calls for higher SRAM bandwidth to support correspondingly larger parallel fetches to the CoHa table.

To accomplish an efficient CoLT design, we employ a separate hash table to hold those IPv6 prefixes with more than 32 b. This *se*cond *ha*sh table (referred to as the SeHa table hereafter) has much fewer entries (than the CoHa table), as it is meant for holding rear portions (after bit 32) of long IPv6 entries only. Given DPL of IPv6 being $\{16, 20, 24, 28, 32, 33, 37, 41, 45, 49, 53, 57, 61\}$, for example, our CoLT produces $\mathrm{DPL}_1 = \{16, 20, 24, 28, 32\}$ for the CoHa table and $\mathrm{DPL}_2 = \{1, 5, 9, 13, 17, 21, 25, 29\}$ for the SeHa table, where 1 (or 5) in DPL_2 is due to 33 (or 37–32).

Our design is best explained by the example shown in Fig. 7. In addition to keeping IPv4 prefixes, the CoHa table also holds IPv6 prefixes, with table entries installed according to DPL_1. Each table entry is augmented by three bits: 1) a v4-bit to indicate that the entry keeps an IPv4 prefix; 2) a v6-bit for IPv6 prefixes; and 3) a ">32" bit to signify that the entry keeps an IPv6 prefix longer than 32 b. Both v4 and v6 indicators are set at the same time, when an IPv6 and an IPv4 routes share a common prefix. IPv6 prefixes longer than 32 b are considered to comprise two parts: a 32-b IPv6 leading prefix stored in the CoHa table, and the remaining suffix stored in the SeHa table. As illustrated in Fig. 7, to install a 48-b IPv6 prefix of 2001:0470:0102::/48, the first 32-b prefix 2001:0470 is stored in an entry e1 in the CoHa table according to the algorithm outlined in Fig. 2. The prefix (polynomial) 2001:0470 is divided by generator $g(x) = x^{16} + x^{12} + x^3 + x^2 + 1$ to produce a quotient, 0x2222, and a remainder, 0x9fca. Following the CoLT algorithm, the quotient (i.e., 0x2222) is installed in the hash bucket 0x9fca, with its fields "v6" and ">32" set to specify that it contains an IPv6 route longer than 32 b.

1) Suitable Keys to SeHa Table by Progressive Mashing: For explanation purposes, let us assume a SeHa table of 4 K sets (with 4 K × 4 entries totally), adequate to accommodate thousands of IPv6 prefixes existing in any core router currently. Such a table requires a generator polynomial $g_2(x)$ with $\mathrm{degree} = 12$, say an example CRC-12 with polynomial code 0x1069. To install the remaining 16-b suffix 0102 in the SeHa table smartly, we employ a "*progressive mashing*" process to get a suitable key with 32 b always to the hash function, formed by taking $(32-k)$ least-significant bits from the leading 32-b prefix 2001:0470, appended to the k bits from the suffix 0102, where k belongs to DPL_2 and equals 16 to yield 0102:2001 in this particular example. A suitable key so obtained is dubbed a progressive mashing (PM) key. In general, if a suffix contains u bits, with $u \notin \mathrm{DPL}_2$, the suffix is rounded down to length v, such that $v\ (< u)$ is the largest value in DPL_2; this follows the same way as rounding down IPv4 prefixes for hashing to CoHa buckets. The progressive mashing process aims to deal with cases where suffixes are short or contain mostly 0's, producing PM keys to the SeHa table. Without progressive mashing, short IPv6 suffixes as hash keys will have the form of $0 \cdot g_2(x) + \mathrm{suffix}(x)$, which causes a poor hashing distribution in the SeHa table and always-0 quotients, undesirable situations to be avoided. PM keys also provide far better opportunities for entry migration in the SeHa table and ensure the high-order bits of v6 prefixes to maintain the same transitive property so that both IPv4 and IPv6 can be treated uniformly by one hardware.

Note that a 32-b key with respect to degree-12 $g_2(x)$ produces a 20-b quotient used to compute a digest signature for storing in the SeHa bucket indexed by its remainder, as demonstrated in Fig. 7.

2) SeHa Table Construction and Use of Digest Signatures: Like installing prefixes into the CoHa table, long IPv6 prefixes are installed into the SeHa table buckets keyed by the remainders of TPR expression $\mathrm{p}(x) = q(x) \cdot g_2(x) + r(x)$, as stated in Section II-B, where $\mathrm{p}(x)$ refers to 32-b PM keys obtained by the

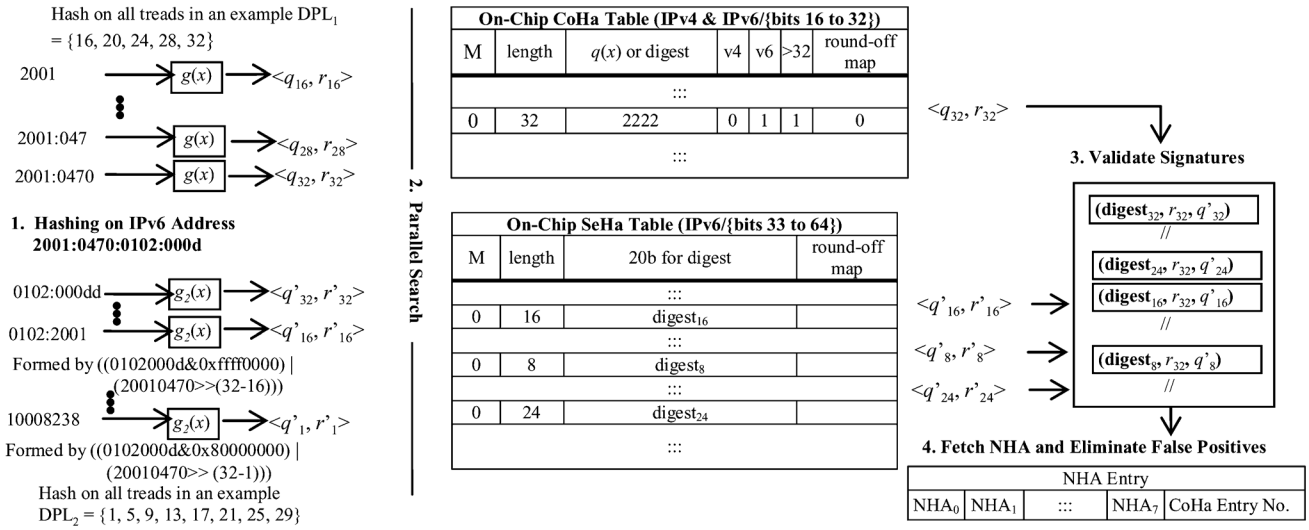

Fig. 8. Example of LPM to installed IPv6 prefix of 2001:0470:0102::/48 under CoLT.

progressive mashing process. To have high storage-efficiency, $\mathrm{p}(x)$ can be stored in any one of the m candidate sets indexed by $\mathrm{p}(x)$ and by its round-down prefixes (i.e., $\mathrm{p}(x)/\{l_1, l_2, \ldots l_m\}$ under $\mathrm{DPL}_2 = \{l_1, l_2, \ldots l_m\}$). Those candidate sets provide multiple choices for each suffix as before.

Continuing on the example, the obtained PM key of 0102:2001 yields a quotient 0x1159 and a remainder 0xff0 by generator $g_2(x)$. Rather than keeping the quotient of 0x1159 (say, in the bucket e2 indexed by 0xff0), we store a *digest* obtained by the XOR operation of the first index 0x9fca and the quotient 0x1159 derived from the key of 0102:2001. This digest acts effectively as approximate filtering to validate a "*legitimate binding*" of matched entries (e1, e2) in the CoHa and the SeHa tables during LPM search.

Next-hop information is stored in off-chip RAM, as illustrated in Fig. 7. Every NHA entry contains eight fields for keeping eight next-hop data, enabling direct accesses indexed by the round-off bitmap kept in the CoHa table (or the SeHa table for any IPv6 prefix with more than 32 b, denoted by $\mathrm{P}/ > 32$). Also, one extra field is provisioned to hold information about the CoHa entry that keeps the leading 32 b of $\mathrm{P}/ > 32$ to avoid false positive lookups. This entry information essentially comprises the bucket index r_1 (i.e., 0x9fca for the example of Fig. 7) and a 2-b indictor for one of the four-way entries in the bucket. Matching r_1 and the exact CoHa entry number for associated P/32 ensures to detect *all possible* false positives.

B. LPM Procedure for IPv6 Addressing

During IPv6 LPM, both tables are searched in parallel by the basic algorithm depicted in Fig. 2, with the additional requirement of composing the hash keys for the SeHa table by the above progressive mashing process. If the search result of the CoHa table does not result in a match to any IPv6 P/32 entry with the ">32" bit set, there is no need to further probe the SeHa table. Otherwise, longer than 32-b IPv6 routes potentially exist, and the search continues.

To best illustrate this, let us consider an example of LPM search for IPv6 address 2001:0470:0102:000d in Fig. 8, where the hash tables are assumed to contain the prefix route 2001:0470:0102::/48 installed in Fig. 7. Hashing of transformed prefixes to all treads in the DPL is performed. Both CoHa and SeHa tables are accessed in parallel (or in two stages of a pipelined design). In this example, the CoHa table reports a match to a P/32 entry with its "v6" and ">32" bits set, indicating that a longer-than-P/32 IPv6 route may exist. Hence, the SeHa table search result is taken into consideration.

1) Use of Scrambled Signature for Approximate Filtering: As each long IPv6 prefix has multiple candidate sets to select in the SeHa table (characterized by DPL_2) during table construction, LPM search for a given IPv6 address must examine all table sets (i.e., m for $|\mathrm{DPL}_2| = m$) identified using the IPv6 address and DPL_2. This search may report multiple candidates. It is desirable to rapidly filter out mismatches. Recall the example of Fig. 7. For route 2001:0470:0102::/48, the leading 32-b prefix 2001:0470 and the remaining suffix 0102 are stored respectively in Bucket 0x9fca and Bucket 0xff0 of the two tables. During search, not only both entries must report matches, but also an assertion must exist: "the prefix stored in Bucket 0x9fca of the CoHa table is the leading P/32 prefix of an IPv6 route whose remaining suffix is stored in Bucket 0xff0 of the SeHa table." A straightforward method for assertion is to store the first index 0x9fca together with the suffix 0102 in the same SeHa table entry. However, such a method incurs additional $|g(x)|$ bits per SeHa table entry.

To arrive at a concise lookup table, aforementioned assertion is embedded inside the SeHa table (in the form of digest signatures) without an extra field, subject to a negligible probability of false positive errors. Specifically, a scrambled digest is calculated by a signature function on the index of 0x9fca for the CoHa table and the quotient of 0x1159 obtained from the PM key for the SeHa table, as signature(0x1159, 0x9fca) in Fig. 7. Upon LPM lookups, the same process is taken to calculate and validate the digest signature in Fig. 8.

While false positives may be practically rare, CoLT detects and eliminates any false positive by keeping the CoHa entry number of the leading P/32 prefixes in off-chip NHA storage, as shown in the last step of LPM search in Fig. 8. Note that one field for the CoHa entry number is adequate for all prefixes aggregated to the same SeHa table entry corresponding to the NHA storage entry. This serves as the final defense in preventing wrong lookup results.

C. Analysis, Evaluation, and Comparison

If the probability of errors is denoted by f and scrambled digests have a length of k bits, it is assumed permissible with $f \approx 2^{-k} \ll 1.0$, where k is shorter than $|g(x)| + |q(x)|$, the total bits needed to keep the index of the CoHa table plus the quotient of a PM key for the SeHa table. Intuitively, the length of the scrambled digest (i.e., k) should increase to lower the allowable fraction of errors (i.e., f). The expected false positive rate f is $(1/2)^k$ when validating digests upon lookups since every bit has a probability of 0.5 to be different. If k rises to $|g(x)| + |q(x)|$ bits, all information is explicit, and no errors will exist. In this paper, the field for scrambled digests has 20 b, exhibiting a mean false positive rate of 9.5E–7 in matching two 20-b numbers.

The concept of representing information by a short digest with a small allowable fraction of errors was first introduced in [4] and was later enhanced and deployed to construct a fast, small, and effective lookup table for TCP connections by scrambling the digest [15]. Basically, a scrambling function is similar to a hash function, which comprises repeated "*combine-and-mix*" steps of inputs and internal states [18], [19], [21]. To have better scrambling results, the progressive mashing process is designed to ensure that always-0 quotients do not happen for IPv6 suffixes stored in the SeHa table. While it is desirable for a scrambling function to take several runs in shuffling and combining bits [18], [19], [21], a simple scrambling function adopted here (as in [15]) works just fine; in our study, we have observed only one false positive for all BGP tables examined under tens of millions of lookup requests.

We followed the same procedure stated in Section IV-C to evaluate CoLT performance under large BGP datasets, which were synthesized from three original BGP tables based on their prefix length distributions: rrc00, rrc01, and rrc14, each of which has about 8000 IPv6 prefixes with the rest being IPv4 prefixes. Collected results of the SeHa table with 16 K $\times$ 4 entries under different numbers of IPv6 prefixes are listed in Table III. With multiple hashing bucket candidates and the aid of migration under CoLT, the SeHa table is found to experience very few overflows (just like the CoHa table). Using tens of millions of IPv6 LPM requests produced randomly and purposely according to the BGP tables' route prefixes, we observed just one single false positive case (i.e., one SeHa entry falsely matches another input address), which was eliminated after fetching the indexed NHA entry (as shown in Step 4 in Fig. 8). This evaluation confirms that the proposed CoLT design offers a new solution for extremely rapid IP lookups in future backbone routers where IPv4 and IPv6 addresses coexist, due to its high memory efficiency in handling both address types via a unified fashion.

TABLE III
LOAD DISTRIBUTIONS AND FALSE POSITIVES UNDER SEHA WITH 16 K $\times$ 4 ENTRIES, GIVEN $\mathrm{DPL}_2 = \{1, 5, 9, 13, 17, 21, 25, 29\}$ (RESULTS WITH MIGRATION LISTED IN NEXT SHADED COLUMNS)

Load	50K IPv6 Prefixes		100K IPv6 Prefixes		160K IPv6 Prefixes	
0	74.6%	2.33%	61.8%	42.2%	51.9%	26%
1	16.2%	7.92%	22.5%	33.0%	26.2%	30.1%
2	3.9%	13.43%	2.2%	13.1%	8.9%	19.8%
3	1.6%	41.7%	1.2%	10.2%	3.6%	19.2%
4	1.3%	33.02%	0.7%	1.4%	1.7%	4.4%
5	1.1%	1.6%	0.9%	0.2%	0.9%	0.5%
≥6	1.3%		10.7%		6.8%	
Flase-Positives	0		0		1	

VI. CONCLUSION

A hash table-based scheme capable of constructing concise lookup tables (CoLT) for fast IPv4 and IPv6 LPM lookups in scalable routers has been introduced and evaluated. CoLT resorts to simple, unified hash tables indexed by the remainders of round-down prefixes to lower the memory requirement, stores "transitive signatures" of transformed representations, and allows for rich prefix aggregation in CoLT tables built over on-chip SRAM. For an IPv4 packet, its lookup involves search over the first hash table only. An IPv6 packet calls for search across both hash tables simultaneously, with the first 32 b of its address used to search over the first hash table. Handling both IPv4 and IPv6 effectively, CoLT achieves exceptionally high SRAM storage efficiency, resulting from its provision of multiple candidate sets in hash tables for a given prefix to improve load balancing and thus lower the number of set overflows. The prefix can be kept in those candidate entries, following a migration rule to ensure correct LPM without false positives. Our evaluation results based on real routing tables (available to the public [3], [20]) reveal that CoLT leads to noticeable storage reduction when compared to other schemes, making it possible to fit a large BGP table in on-chip SRAM for rapid lookups. Being hash-oriented, CoLT naturally supports incremental table updates effectively, able to reach lookup throughput over 310 MPPS practically when its constituent SRAM modules operate at 500 MHz.

APPENDIX
PROOF OF THE MIGRATION RULE

Consider previous $\mathrm{P}_1/32$ and the example design of Section III-A, where $g(x)$ is a degree-16 primitive polynomial and the CoHa table has 2^{16} buckets. Since $g(x)$ is a *prime*, using the total of 2^{32} possible P/32 prefixes for indexing, each of the 2^{16} buckets will be hit precisely 2^{16} times. The first 2^{16} (i.e., $0, 1, \ldots, 2^{16} - 1$) polynomials are mapped one-to-one to the 2^{16} buckets, with their quotients all equal to 0. The next 2^{16} numbers are also mapped one-to-one to those buckets, with their quotients being 1. Similarly, any subsequent group i of 2^{16} numbers is mapped one-to-one to the buckets, with their quotients all equal to i. When $\mathrm{P}_1/32$ is moved to its colony r_t,

there will be a group υ of totally 2^{16} $\mathrm{P}_\upsilon/32$'s able to identify the same bucket r_t via $\mathrm{rem}(\mathrm{P}/32)$, where P is composed generally by concatenating $\mathcal{L}(\mathrm{P}_\upsilon)/t$ with a (32-t)-bit constant suffix C. For notational simplicity, we use $\mathrm{rem}(\mathcal{L}(\mathrm{P}_\upsilon)/t)$ to denote the above process.

According to the stated migration rule, $\mathrm{P}_1/32$ can be migrated to the bucket identified by tread t in DPL, with 16 (or $32-|g(x)|) \leq t < 32$. The cases of $t = 16, t > 16$, and $t < 16$ are treated in sequence.

Case of $t = 16$*:*

a) $\mathrm{P}_1/32 = q_1 \cdot g + r_1$, which is migrated. After migration, the digest comprising $(q_1 + r_1)$ is stored in the bucket indexed by $r_t = \mathrm{rem}(\mathcal{L}(\mathrm{P}_1)/16)$.

b) The group υ with a total of 2^{16} $\mathrm{P}_\upsilon/32$ prefixes will index to the same colony bucket r_t via $\mathrm{rem}(\mathcal{L}(\mathrm{P}_\upsilon)/16)$. When taking any prefix P_υ in υ as its input, the lookup algorithm (outlined in Fig. 2) inspects all buckets indexed by $\mathrm{rem}(\mathrm{P}_\upsilon/\mathrm{DPL})$. If false positives arise, Bucket r_t must be matched by $\mathrm{rem}(\mathcal{L}(\mathrm{P}_\upsilon)/16)$. Due to the *primitivity* of $g(x)$, all P_υ's in υ (including P_1 itself) must share the same common leading prefix of $\mathcal{L}(\mathrm{P})/16$ (as otherwise, they will not index to the same bucket r_t through their leading prefix $\mathcal{L}(\mathrm{P})/16$). More precisely, all those 32-b $\mathrm{P}_\upsilon/32$ prefixes in Group υ can be expressed in a polynomial form

$$(\text{same leading})\ \mathcal{L}(\mathrm{P})/16 \cdot x^{16} + (\text{trailing suffix})\ \mathcal{T}(\mathrm{P})/16.$$

c) Next, to obtain q_υ and r_υ in order to validate $(q_1 + r_1 + q_\upsilon + r_\upsilon)$ for the stored digest in r_t for $\mathrm{P}_1/32$, any prefix $\mathrm{P}_\upsilon\,|\,32$ in Group υ is divided by a degree-16 $g(x)$, giving rise to $\mathrm{P}_\upsilon/32 = q_\upsilon \cdot g + r_\upsilon$. For easy comprehension, let us assume that the simple Linear Feedback Shift Register (LFSR) is used for the binary division. The quotient is shifted out on the left of the LSFR, and the prefix input is fed to LSFR on the right. After processing the common leading prefix $\mathcal{L}(\mathrm{P})/16$, all prefix $\mathrm{P}_\upsilon\,|\,32'$s in Group υ produce exactly identical *intermediate quotients and reminders*, $q'(x)$ and $r'(x)$. ($q'(x)$ equals 0 in this case because $|g(x)| = 16$. Nevertheless, generality of the following proof holds.) The division continues by taking in the trailing 16-b $\mathcal{T}(\mathrm{P})/16'$s of all $\mathrm{P}_\upsilon/32'$s to arrive at precisely 2^{16} different reminders (with numerical numbers varying from 0 to $2^{16} - 1$, as a result of the property of primitive polynomial $g(x)$), and meanwhile to get the same collection of quotient q_υ's (equal to binary strings concatenating $q'(x)$ and $r'(x) \cdot x^{16} \div g(x)$). For the given q_υ, XORing it with 2^{16} different $r(x)$'s produces 2^{16} unique signatures, thereby incurring *no false positive*. In other words, q_1 always equals q_υ for any $\mathrm{P}_\upsilon/32$ in Group υ, while r_1 is equal to r_υ only for $\mathrm{P}_\upsilon/32 = \mathrm{P}_1/32$, implying $r_\upsilon \neq r_1$ for any $\mathrm{P}_\upsilon \neq \mathrm{P}_1$. Hence, the validation equation of $(q_1 + r_1 + q_\upsilon + r_\upsilon == 0)$ suffices to eliminate ambiguity, avoiding false positives.

Case of $t > 16$*:* For a longer tread $t > 16$ (say, $t = 17$ without loss of generality), Group υ will involve exactly 2^{16} $\mathrm{P}_\upsilon/32$ prefixes that produce the same index r_t through $\mathrm{rem}(\mathcal{L}(\mathrm{P}_\upsilon)/17)$ and $\mathrm{rem}(\mathcal{L}(\mathrm{P}_1)/17)$, according to the following arguments.

a) $\mathrm{P}_1/32 = q_1 \cdot g + r_1$, which is migrated. After migration, the digest comprising $(q_1 + r_1)$ is held in the bucket indexed by $r_t = \mathrm{rem}(\mathcal{L}(\mathrm{P}_1)/17)$.

b) The total 2^{16} $\mathrm{P}_\upsilon/32$ prefixes (in Group υ) that produce the same index of r_t via $\mathrm{rem}(\mathcal{L}(\mathrm{P}_\upsilon)/17)$ can be expressed by

$$(\text{same leading})\ \mathcal{L}(\mathrm{P_a})/17 \cdot x^{15} + (\text{trailing suffix})\ \mathcal{T}(\mathrm{P})/15$$

and

$$(\text{same leading})\ \mathcal{L}(\mathrm{P_b})/17 \cdot x^{15} + (\text{trailing suffix})\ \mathcal{T}(\mathrm{P})/15$$

where $\mathrm{P_a}$ is $(q = 0 \cdot g + r)$ and $\mathrm{P_b}$ equals $(q = 1 \cdot g + r)$, which denote leading 17 b of all prefixes in Group υ. This is because $g(x)$ is a degree-16 prime, letting every bucket mapped to exactly twice via $\mathrm{rem}(\mathcal{L}(\mathrm{P}_\upsilon)/17)$ for all possible 2^{17} combinations of 17-b strings.

c) Based on b) above, we observe that $\mathrm{P}_\upsilon/32 = q_\upsilon \cdot g + r_\upsilon$ yields exactly 2^{15} unique r_υ due to the 15-b trailing suffix $\mathcal{T}(\mathrm{P}_\upsilon)/15$ under a degree-16 $g(x)$, given that the starting state is r (same for 2^{15} all suffix combination strings) after $\mathcal{L}(\mathrm{P}_\upsilon)/17$. Furthermore, there are two important properties involved in the division of $\mathcal{T}(\mathrm{P})/15$ by a degree-16 primitive $g(x)$. First, shifting the 15 bits of $\mathcal{T}(\mathrm{P})/15$ into LFSR (which contains the initial state r) moves the least significant bit of r to the most significant position, whose value is either 0 or 1. Hence, 2^{15} unique r_υ's can be denoted by $\{\text{constant bit}, 2^{15}$ binary combinations$\}$. Second, there are two and only two final quotients, $\{q^{15} = 0/1, (r)x^{15} \div g(x)\}$, for all 2^{15} unique r_υ's. By starting with the two states, $(q = 0, r)$ and $(q = 1, r)$, based on b) above and taking in the 15-b suffix $\mathcal{T}(\mathrm{P})/15$, the two quotients are 16-b binary strings whose most significant bit is 0 or 1, concatenated by a 15-b field produced by $(r)x^{15} \div g(x)$. Hence, when the quotient of q_υ $(=\{q^{15} = 0, (r)x^{15} \div g(x)\}$ or $\{q^{15} = 1, (r)x^{15} \div g(x))$ is added (over GF(2)) to those 2^{15} unique r_υ of the form $\{\text{constant bit}, 2^{15}\text{binary combinations}\}$, we have 2^{16} unique signatures (which equal r_υ for $q_\upsilon^{15} = 0$ and equal r_υ with the most significant bit toggled for $q_\upsilon^{15} = 1$). As a result, the signatures so derived by $(q_1 + r_1)$ can distinguish exactly 2^{16} $\mathrm{P}_\upsilon/32$ prefixes in Group υ, avoiding any false positive. The same proof steps above can be repeated and hold true for all $t > 17$.

Case of $t < 16$*:* This case could lead to ambiguity under $|g(x)| = 16$, which implies $32 - |g(x)| > t$. Given $t = 8$, for example, there are totally 2^{24} P/32 prefixes mapped to the bucket indexed by P/t (=8). This calls for a way to produce at least 2^{24} unique digests in order to support prefix migration, with false positives completely evaded. When the CoHa table has a field with only 16 b for signatures (as depicted in Fig. 1), false positives may occur after prefix migration. The proof is thus completed. ■

REFERENCES

[1] "ARIN number resource policy manual (NRPM)," American Registry for Internet Numbers, Chantilly, VA, Sep. 2009 [Online]. Available: https://www.arin.net/policy/nrpm.html#ipv6

[2] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," in *Proc. 26th ACM Symp. Theory Comput.*, May 1994, pp. 593–602.
[3] "BGP table data," Jan. 2011 [Online]. Available: http://bgp.potaroo.net/indx-bgp.html
[4] B. H. Bloom, "Space/time trade-offs in Hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[5] A. Broder and M. Mitzenmacher, "Using multiple Hash functions to improve IP lookups," in *Proc. 20th IEEE INFOCOM*, Apr. 2001, pp. 1454–1463.
[6] B. Carr *et al.*, "IPv6 address allocation and assignment policy," Sep. 2009 [Online]. Available: http://www.ripe.net/ripe/docs/ipv6policy.html
[7] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, Sep. 1997, pp. 3–14.
[8] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proc. ACM SIGCOMM*, Aug. 2003, pp. 201–212.
[9] W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on longest-matching prefixes," *IEEE/ACM Trans. Netw.*, vol. 4, no. 1, pp. 86–97, Feb. 1996.
[10] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakrodhar, "Chisel: A storage-efficient, collision-free Hash-based network processing architecture," in *Proc. 33rd ISCA*, May 2006, pp. 203–215.
[11] W. Jiang and V. Prasanna, "Parallel IP lookup using multiple SRAM-based pipelines," in *Proc. IEEE IPDPS*, Apr. 2008, pp. 1–14.
[12] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with choices," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 218–231, Feb. 2008.
[13] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *Proc. 27th IEEE INFOCOM*, Apr. 2008, pp. 101–105.
[14] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1083–1092, Jun. 1999.
[15] F. Pong, "Fast and robust TCP session lookup by digest hash," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst.*, Dec. 2006, pp. 507–514.
[16] F. Pong and N.-F. Tzeng, "SUSE: Superior storage-efficiency for routing tables through prefix transformation and aggregation," *IEEE/ACM Trans. Netw.*, vol. 18, no. 1, pp. 81–94, Feb. 2010.
[17] V. C. Ravikumar and R. N. Mahapatra, "TCAM architecture for IP lookup using prefix properties," *IEEE Micro*, vol. 24, no. 2, pp. 60–69, Mar./Apr. 2004.
[18] R. Rivest, "The MD5 message-digest algorithm," Internet Engineering Task Force, RFC 1321, Apr. 1992.
[19] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 26, no. 1, pp. 120–126, Feb. 1978.
[20] "Routing information service," RIPE, Amsterdam, The Netherlands [Online]. Available: http://www.ripe.net/
[21] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. New York: Wiley, 1996.
[22] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash table lookup using extended bloom filter: An aid to network processing," in *Proc. ACM SIGCOMM*, Aug. 2005, pp. 181–192.
[23] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 lookups using distributed and load balanced bloom filters for 100 Gbps core router line cards," in *Proc. 28th IEEE INFOCOM*, Apr. 2009, pp. 2518–2526.
[24] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, "Scalable IP lookups using shape graphs," in *Pro. 17th IEEE ICNP*, Oct. 2009, pp. 73–82.
[25] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," in *Proc. ACM SIGMETRICS*, Jun. 1998, pp. 1–11.
[26] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," *Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, Feb. 1999.
[27] B. Vocking, "How asymmetry helps load balancing," in *Proc. 40th IEEE Symp. Found. Comput. Science*, 1999, pp. 131–141.
[28] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *Proce. ACM SIGCOMM*, Oct. 1997, pp. 25–36.
[29] G. Wang and N.-F. Tzeng, "TCAM-based forwarding engine with minimum independent prefix set (MIPS) for fast updating," in *Proc. IEEE ICC*, Jun. 2006, pp. 103–109.
[30] H. Yu, R. Mahapatra, and L. Bhuyan, "A Hash-based scalable IP lookup using bloom and fingerprint filters," in *Proc. 17th IEEE ICNP*, Oct. 2009, pp. 264–273.
[31] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: Hardware/software IP lookup with incremental updates," *Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, Apr. 2004.
[32] M. Bando and H. J. Chao, "FlashTrie: Hash-based prefix-compressed trie for IP route lookup beyond 100 Gbps," in *Proc. 29th IEEE INFOCOM*, Mar. 2010, pp. 1–9.
[33] "QDR-II+ SRAM overview," Cypress Semiconductor Corporation, San Jose, CA [Online]. Available: http://www.cypress.com/?id=108

Fong Pong (M'92–SM'10) received the M.S. and Ph.D. degrees in computer engineering from the University of Southern California, Los Angeles, in 1991 and 1995, respectively.

He is with Broadcom Corporation, Santa Clara, CA, where he has developed multicore SoCs, a GPON device, and network processors and conducted research on algorithmic solutions for packet classification, filtering, and QoS. Before Broadcom, he was with several startups; HP Labs, Palo Alto, CA; and Sun Microsystems, Mountain View, CA, where he developed blade servers, network and storage protocols offload products, and multiprocessor systems. He has received 38 patents.

Dr. Pong has served the National Science Foundation, the IETF RDMA Consortium, and program committees for several conferences.

Nian-Feng Tzeng (M'86–SM'92–F'10) received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1986.

He has been with Center for Advanced Computer Studies, University of Louisiana at Lafayette, since 1987. His current research interest is in the areas of computer communications and networks and parallel and distributed computer systems.

Prof. Tzeng was on the Editorial Board of the IEEE TRANSACTIONS ON COMPUTERS from 1994 to 1998 and the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS from 1998 to 2001. He was the Chair of the Technical Committee on Distributed Processing, the IEEE Computer Society, from 1999 to 2002. He was the recipient of the Ooutstanding Paper Award of the 10th International Conference on Distributed Computing Systems, May 1990, and received the University Foundation Distinguished Professor Award in 1997.