SUSE: Superior Storage-Efficiency for Routing Tables Through Prefix Transformation and Aggregation

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

***Abstract*—A novel storage design for IP routing table construction is introduced on the basis of a single set-associative hash table to support fast longest prefix matching (LPM). The proposed design involves two key techniques to lower table storage required drastically: 1) storing transformed prefix representations; and 2) accommodating multiple prefixes per table entry via prefix aggregation, achieving superior storage-efficiency (SUSE). With each prefix $(\mathrm{p}(x))$ maneuvered as a polynomial, $\mathrm{p}(x) = \mathrm{q}(x) \times \mathrm{g}(x) + \mathrm{r}(x)$ based on a divisor $\mathrm{g}(x)$, SUSE keeps only $\mathrm{q}(x)$ rather than full and long $\mathrm{p}(x)$ in an $\mathrm{r}(x)$-indexed table with $2^{\mathrm{degree}(\mathrm{g}(x))}$ entries, because $\mathrm{q}(x)$ and $\mathrm{r}(x)$ uniquely identify $\mathrm{p}(x)$. Additionally, using $\mathrm{r}(x)$ as the hash index exhibits better distribution than do original prefixes, reducing hash collisions, which can be tolerated further by the set-associative design. Given a set of chosen prefix lengths (called "treads"), all prefixes are rounded down to nearest treads under SUSE before hashed to the table using their transformed representations so that prefix aggregation opportunities abound in hash entries. SUSE yields significant table storage reduction and enjoys fast lookups and speedy incremental updates not possible for a typical trie-based design, with the worst-case lookup time shown upper-bounded theoretically by the number of treads (ζ) but found experimentally to be 4 memory accesses when ζ equals 8. SUSE makes it possible to fit a large routing table with 256 K (or even 1 M) prefixes in on-chip SRAM by today's ASIC technology. It solves both the memory- and the bandwidth-intensive problems faced by IP routing.**

***Index Terms*—Hash tables, linear feedback shift registers, longest prefix matching, prefix aggregation, prefix transformation, routing tables, table storage, tries.**

I. Introduction

FAST IP address lookups have become indispensable for all routers, whether core or edge ones, as the line rates reach the OC-192 or even OC-768 speeds for many connections, resulting from widespread deployment of 10 G Ethernet switches. Meanwhile, the number of prefixes in core routers has experienced explosive growth recently, with large BGP (Border Gateway Protocol) routing tables seen rapid surges in their prefix numbers continuously. Earlier solutions for IP address lookups were mostly trie-based through software execution to match an IP address progressively a few bits at a time against prefixes stored in a tree-like data structure [4], [5], [11], [12], [17], [19], to support longest prefix matching (LPM), which chooses the longest prefix among those which match the given IP address. While they usually have lower hardware costs, trie-based solutions are no longer attractive due to large latencies. On the other hand, hash tables are deemed ideal for fast IP lookups because of their constant-time search latencies, provided that care is taken to address potential collisions inherent to the hash table (where multiple prefixes map to the same hash index) [10]. Composed of regular SRAM, they are relatively inexpensive than ternary content addressable memories (TCAM), consuming less power and exhibiting much higher density.

However, the application of hash tables to LPM is not straightforward because the number of IP address bits to use for hash calculation is not known *a prior* due to the variable lengths of prefixes used in building hash table databases for lookups. Therefore, hash tables must deal with two challenges when applied for LPM, namely, (1) prefixes of variable lengths and (2) address collisions associated with hashing. Techniques have been considered [10], [22], [24] for handling these challenges. An earlier design uses one hash table for each prefix length [22], employing 25 tables in total for prefix length ranging from 8 to 32 under IPv4 addressing. If those 25 tables are examined in sequence or following a binary search [22] for a given IP address, it takes many memory accesses, thus exhibiting a long latency. On the other hand, if parallel search for prefixes of all lengths is to be performed, a large memory system which supports 25 parallel accesses is required. Later solutions intend to lower the degree of search parallelism by employing Bloom filters or their variations [10], [23], coupled with hash tables. They keep the salient features of hash-based architectures, such as low lookup latencies and efficient updates. A recent solution, referred to as *Chisel* [10], lowers table sizes and decouples next-hop addresses (NHA) from prefixes themselves so that all the involved hash tables, other than the NHA table, may fit in on-chip SRAM to realize fast lookups. A smaller hash table naturally is less expensive and consumes less power. Such a solution also permits to accommodate a larger routing table on-chip to yield better scalability. Separately, another approach by partitioning a BGP routing table has been considered recently to reduce total storage required for holding prefixes in the table [14].

This paper proposes a novel storage design for routing tables, realized by prefix transformation and aggregation to achieve superior storage-efficiency (SUSE). The proposed design

Manuscript received August 30, 2007; revised December 19, 2008; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Buddhikot. First published August 18, 2009; current version published February 18, 2010. An earlier version of this work was presented at the 32nd IEEE Conference on Local Computer Networks (LCN 2007), October 2007.

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).

Digital Object Identifier 10.1109/TNET.2009.2022085

1063-6692/$26.00 © 2009 IEEE

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

makes use of a *single set-associative hash table* to support fast LPM for all lengths. It employs two key techniques for lowering table storage required drastically: 1) storing transformed prefix representations; and 2) accommodating multiple prefixes per table entry via prefix aggregation. Significant storage reduction makes it possible to fit a large routing table with 256 K (or even 1 M) prefixes in on-chip SRAM by today's ASIC technology. This solves both the memory- and the bandwidth-intensive problems faced by IP routing. Simulation results on real routing tables show that SUSE leads to more than 55% storage reduction (from some 23 Mb SRAM for Chisel down to 10 Mb under an example routing table with 170 k+ prefixes), when compared with any known hash table-based design or any trie-based design (except for Lulea-Trie [12]). In addition, SUSE enjoys fast lookups and rapid incremental updates, with the lookup latency bounded theoretically by ζ SRAM accesses (where ζ is the number of designated prefix lengths chosen for collapsing prefixes to, as will be detailed in Section III), if a small TCAM is used to hold spillover prefixes. In practice, the worst-case lookup time is found experimentally to be 4 memory accesses, when ζ equals 8. SUSE involves one single hash table in SRAM, as opposed to $3 * \zeta$ tables of various sizes in SRAM plus one result table in DRAM for Chisel, which also needs at least 3 times as many hash functions.

After pertinent background and related work are provided in Section II, this article introduces our proposed design in Section III, including prefix transformation and a fast mechanism for this transformation, which constitute the first superiorly storage-efficient technique for routing tables. Section IV deals with mechanisms for prefix aggregation, which is the second key technique for SUSE realization. A skewed method for memory module layout is stated in Section V to avoid access conflicts during parallel memory search for fast lookups under SUSE. SUSE is evaluated in Section VI, with its performance results obtained using real-world routing tables unveiled therein. Size and power considerations for the hardware implementation of SUSE are detailed in Section VII.

II. Pertinent Background and Related Work

Techniques for IP lookups fall into two categories, depending on what type of memory is employed to hold prefixes constituting lookup databases: TCAM- and RAM-oriented ones. Different TCAM-oriented techniques have been investigated [6], [7], [9], [18], [21] lately, but they are usually more expensive than their RAM-oriented counterparts and consume far more power (which is proportional to the size). As a result, we deemphasize TCAM-based solutions and focus on the RAM-oriented techniques, which can be grouped as trie-based and hash-based ones.

A. Trie-Based LPM

A trie is an ordered tree data structure used to succinctly store a set of "strings," which are the IP route addresses in this context. Each path from the root to a leaf in a trie corresponds to one string in the represented set, and hence, the trie nodes denote the prefixes of the strings.

1) Path-Compressed DP Tries: In its simplest form, a binary trie treats an IP address comprising a sequence of bits. One bit information is inspected at each level. A bit of value 0 (or 1) directs branching to the left (or right) subtree. Unfortunately, binary tries often yield space-inefficient structures where long one-child branches waste storage and cause long search times. *Path compression* is the traditional solution to this problem, resulting in the Patricia tree [5].

In a path-compressed trie, storage is saved and the search time is reduced by collapsing one-child branches. This is done by associating internal nodes with an integer value, which indicates the next bit position to be inspected. For example, let's consider a long one-child branch leading to route 101000*, and it shares the same prefix 101 with another route 1011*, with their difference starting at the fourth bit. By specifying that the fourth bit is the next bit to be inspected at an internal marked node which branches to leaf nodes 101$\underline{0}$00* and 101$\underline{1}$*, the long one-child path is eliminated. When performing search, a descent in the trie is performed according to the address bits as usual. However, only the bit positions specified by the traversed internal nodes are inspected, with other bits skipped. When a leaf is encountered, a comparison with the actual prefix is performed, calling for the need to store the actual prefixes in the leaves.

The *dynamic prefix trie* (DP-trie) [20] is another well-known method in this category. It differs from the Patricia tree more or less in the data structure used to represent the trie. The idea of path-compression by skipping insignificant bits applies to both path-compressed tries and DP-tries.

2) Multibit LC and Lulea Tries: While path-compressed binary tries may be improved by eliminating one-child branches, the gain could be moderate on an average. Because one bit is inspected at a time, the worst case requires 32 memory accesses for IPv4 addressing. To speed up the lookup operations, alternative solutions by checking multiple bits at a time have been proposed [4], as common practice in the industry. For example, a routing table for IPv4 addresses may comprise 5 levels following the (6, 6, 6, 6, 8) inspection pattern. The first level of 64 entries is indexed by the first 6 bits of an IPv4 address. The result is either an entry of forwarding information, or a pointer to a second level. The same apparatus is applied to the second, the third, the fourth, and the fifth levels, with their branching degrees of $2^6, 2^6, 2^6$, and 2^8, respectively. This leads to 64 copies of 64-ary tables in the 2nd level, 64×64 copies of 64-ary tables in the third level, and so on, until the last stage, where the longest IP prefixes (of 32 bits) reside. As it inspects multiple bits per cycle, this solution yields a mutibit trie, which renders a nice pipeline design, allowing "one lookup per cycle." It lets one new lookup be initiated every cycle, with the lookup latency bounded deterministically by 5 cycles. Unfortunately, this design tends to waste a lot of memory space since m-ary tables are often sparsely filled.

To overcome the problem, researchers have proposed novel data structures to represent the multibit trie, with compression techniques applied to keep the trie small. The Level-Compressed (LC)-trie [17] and the Lulea-trie [12] are two representative methods. To form an LC-trie, each "complete" binary subtree of d levels (which should have 2^{d} children) is replaced by a single 2^{d}-ary node. Recursively, this compression process may continue as long as complete binary subtrees are found and replaced by single multiway nodes. The LC-trie places its

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

internal nodes in consecutive memory locations following a succinct, compact representation [17].

Under the Lulea trie [12], a given set of prefixes are first transformed into "disjoint" prefixes which do not overlap (i.e., no route prefix is itself a prefix of others), and then the trie is made complete by *leaf pushing* [19]. Therefore, a Lulea trie is a complete trie, with IP route prefixes all kept at the leaves reflecting disjoint intervals of addresses. Search in such a trie is transformed to a more general interval set membership problem. All route prefixes (and information pertinent to the route lookup resolution) are stored in a memory array of consecutive locations. The Lulea algorithm then applies an efficient encoding scheme to map the corresponding intervals into a concise "code word array." When search is performed, the IP address is used as an index to a "map-table" for retrieving interval information and then forwarding data [12].

While the LC and the Lulea methods require less memory than a tree-like counterpart, techniques such as compression and leaf pushing make incremental updates very difficult. Most importantly, the linear memory array representations of the tries leave holes upon route deletion and are difficult to accommodate new routes. They are not appealing practically, since whole trie rebuilding is needed in those situations.

B. Hash- and Bloom Filter-Based LPM

In contrast to trie-based methods, hash-based techniques utilize a flat data structure to hold prefixes, leading to fast lookups, particularly if hash tables are made compact enough to fit in on-chip SRAM. A hash function typically takes bit strings of a fixed length as its inputs to produce values over a predetermined range. It does not work for bit strings of variable lengths and often exhibits collisions. Unfortunately, prefixes in a routing/forwarding table are of variable lengths, calling for solutions to deal with these problems. An early solution employed multiple hash tables, one for each possible prefix length, to hold the prefixes of a routing/forwarding table [22], with markers included in many prefix table entries to guide binary search over those hash tables. The early solution thus requires accesses to as many as α (> 1) hash tables for a given IP address during binary search, exhibiting a long search time, in addition to a high cost associated with those α hash tables (where α is the number of allowable prefix lengths). No discussion or treatment about hashing collisions was provided in [22]. Later, the use of a counting Bloom filter was pursued [23] to support exact matches using multiple hash functions for each item to calculate values, among which one is chosen to point to the likely bucket (stored in off-chip SDRAM) [24].

1) The Counting Bloom Filter (CBF): The Counting Bloom Filter (CBF) is an extension to the Bloom Filter (BF) [29], where a vector of m counters is used to efficiently track a set (say, N) of n elements. The counter vector is programmed as follows. For each element x in N, k hash functions are computed on x to produce k values ranging from 0 to $(m-1)$, and the values are used to access the counter vector. Each accessed entry increases its counter by one. In this way, a membership query follows the same calculation of k hash values. If all the entries corresponding to the k hash values have non-zero counts, the lookup key is potentially a member of the set, N.

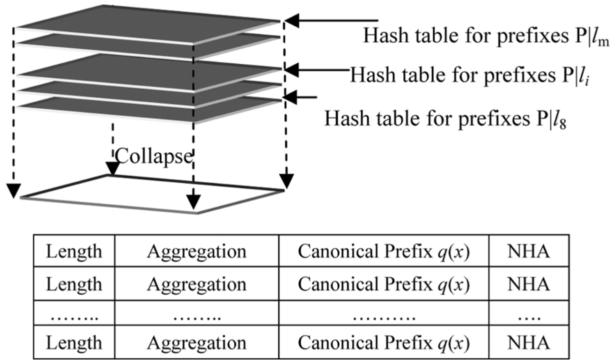
It is well-know that BF may report *false positives*, but never *positive negatives*. A false positive happens when BF reports a match for an element not in N. To achieve a sufficiently small probability of false positives, BF must use many hash functions and have a much larger size than the number of elements in N, n [3]. BF is found optimal under $k = \frac{m}{n}\ln(2)$, where m is its size and k is the number of hash functions. The ratio of (m/n) is usually considered as the average number of slots (among m) consumed by a single member in N. In the optimal case, a linear increase in (m/n) decreases the probability of false positives exponentially.

The basic architecture of using CBFs for LPM is illustrated in [25], where there is one CBF for each prefix length. All CBFs are kept on-chip while the much larger routing table database is stored in off-chip memory. Since the hash functions do not support wildcard bits and the use of one CBF for each prefix length can be too expensive, prefixes of variable lengths are often dealt with via *controlled prefix expansion* (CPE) [19] to arrive at predetermined lengths, such as 12, 16, 24, and 32 bits. Each prefix in the route table database, if needed, is expanded to the next (predetermined) length. All expanded IP prefixes are treated as separate items programmed into the CBFs and inserted into the hash table.

Unfortunately, CBFs tell only the presence or absence of items of interest in the hash table, but they do not reveal their whereabouts in the table. As a result, revised designs were proposed later [24], where the calculated k hash values are employed to index a set of k counters which are associated with k buckets for the hash table. On insertion, a new item is added to one hash bucket chosen (out of the k buckets) based on heuristic metrics, such as the bucket with least-load, i.e., one with the smallest counter value. Because multiple items can be stored in one bucket associated with a counter when their respective hash results yield that same counter ID, those colliding items in the bucket are chained [24]. The search time tends to be long, not only because it involves slow off-chip SDRAM accesses but also because there could be many such accesses (in sequence to traverse those chained items held therein). Although nearly collision-free hashing was shown to be achievable [24], it requires many hash functions (e.g., $k = 10$) and Bloom filters (which are much larger than the number of items in set N).

2) Collision-Free Bloom Filters: Recently, a hash-based design with guaranteed deterministic lookup rates has been investigated for LPM, called *Chisel* [10]. Built upon a Bloomier filter (which is an extension to Bloom filters [3]), Chisel achieves nearly collision-free hashing (i.e., with a probability arbitrarily close to 1) by utilizing an index table with m entries, satisfying $m \geq k \cdot n$, if there are n prefixes in total and k hash functions employed in the design. When k equals 3, the index table size is at least 3 times the number of prefixes. To set up the index table, the k hash values for all elements in the set N are calculated. For element x, its calculated hash values, which are also indices to the index table, define a neighborhood $HN(x)$. Subsequently, all elements x_i in N are *ordered* by pushed into a stack. When an element x_j is popped from the stack, a hash value $\tau(x_j)$ in $NH(x_j)$ is selected, based on one criterion—for all x_i present before x_j in the stack, $\tau(x_j)$ cannot be a member

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.



Length	Aggregation	Canonical Prefix $q(x)$	NHA
Length	Aggregation	Canonical Prefix $q(x)$	NHA
……..	……..	……….	….
Length	Aggregation	Canonical Prefix $q(x)$	NHA

Fig. 1. Lump multiple hash tables into one set-associative table.

of the neighborhood $NH(x_i)$ of x_i. When this criterion is met, the value stored in entry $\tau(x_j)$ is: $\mathrm{p}(x_j)$^value$(NH(x_j) - \tau(x_j))$, where $\mathrm{p}(x_j)$ is a pointer or address to a separate filter table (which keeps the actual N elements), specifying the entry holding item x_j. During lookups, pointer $\mathrm{p}(x_j)$ is easily recovered by XORed values in the neighborhood $NH(x_j)$ of x_j, that is, $\mathrm{p}(x_j) =$ ^value(NH(x_j)). Clearly, if $\tau(x_j)$ is in the neighborhood $NH(x_i)$ of some prior element $x_i (< x_j)$ in the stack, we cannot recover $\mathrm{p}(x_i)$ when a lookup on x_i is performed because values of $NH(x_i)$ are altered due to $\tau(x_j)$. If the above process does not converge, a separate TCAM will keep those colliding prefixes. It is conjectured that a small spillover TCAM (with some 16 to 32 entries) suffices [10].

To deal with variable lengths of prefixes, Chisel collapses prefixes with length x ($> l$, a permissible length) into prefixes of length l, and then performs "leaf pushing" to expand every such collapsed prefix to multiple prefixes of length l' (being the next permissible length and $> x$). All original prefixes collapsed to ones with length l, plus all expanded prefixes through leaf pushing to reach length l', constitute one *subcell*, which includes an index table, a filter table, and a bit-vector table (all kept in on-chip SRAM), plus a result table (held in off-chip DRAM) [10]. As each expanded prefix through leaf pushing takes a bit-vector entry and an NHA (next-hop address) entry, SRAM and DRAM storage needed for Chisel is enlarged considerably. If ζ distinct prefix lengths are chosen as permissible ones for prefix collapsing use, Chisel then requires $(3 \times \zeta)$ on-chip tables of various sizes, involving many hash functions $(\geq 3 \times \zeta)$. During a lookup, all the ζ subcells are searched in parallel using the given IP address, with the results sent to a priority encoder to get the LPM result. Given the number of hash functions involved in one Bloomier filter [3] equal to k (≥ 3), the worst-case lookup time for Chisel is given by $(k+1)$ SRAM accesses plus one DRAM access, with k (or 1) due to accesses to the index table (or to the filter table and the bit-vector table concurrently), assuming that TCAM and the index table are fetched simultaneously.

As will be seen later, our proposed SUSE not only reduces the SRAM storage requirement by a factor of 2 and employs fewer hash functions (only ζ in total) but also enjoys a smaller average-case lookup time (of 1.07 SRAM accesses). In addition, SUSE involves just one single (set-associative) table in SRAM, as opposed to $3 \times \zeta$ tables of various sizes in SRAM plus one result table in DRAM for Chisel, which requires separate access logics and control circuitry to gain parallelism. Its worst-case lookup time is bounded by ζ when a small TCAM is employed to hold overflow prefixes.

III. Proposed Design

A. Lump Hash Table

In order to arrive at a desirable hash-table-based solution, we apply prefix transformation, realized by choosing a *designated prefix length set*, DPL: $\{l_1, l_2, \ldots l_i, \ldots l_m\}$, where l_i denotes a prefix length, such that for any prefix P of length w (expressed by P$|w$) with $l_i \leq w < l_{i+1}$, the prefix is rounded down to P$|l_i$. A naïve design assigns one hash table to hold all prefixes P$|w$ $(l_i \leq w < l_{i+1})$, under the chosen DPL, with each table entry holding one prefix P$|w$ together with its rounded down $(w - l_i)$ bits. During lookups, parallel search for prefixes of m different lengths under DPL, $\{l_1, l_2, \ldots l_i, \ldots l_m\}$, is performed. This prefix transformation lowers the number of hash tables down to m (from 25 under IPv4 addressing). Nonetheless, those hash tables tend to waste many entries, exhibiting poor table utilization, since most prefixes in a routing table are of length between 16 and 24, making tables for prefixes P$|w$, with $16 \leq w \leq 24$, heavily populated, whereas others are likely underutilized.

We thus lump the individual tables together to arrive at an efficient design, as illustrated in Fig. 1. Each entry in the resulting hash table uses an indictor to specify the length of its stored prefix. It also contains an aggregation field for keeping the $(w - l_i)$ round-off bits (described above). The aggregation field enables deep prefix compression, as will be elaborated in Section IV, to reduce the hash table size. The canonical prefix field holds the arithmetically transformed representation of a prefix; transformed representations can be shorter than their corresponding prefixes and yet suffice to distinguish prefixes. The use of transformed representations reduces the width of each table entry. Details about the transformed representations of prefixes will be provided in Section III-B. Each entry has a field for NHA, which keeps a pointer to another table where outgoing port numbers and port configurations are stored.

Because a perfect hash function is impossible, collisions do occur. We employ a simple set-associative design for the resulting hash table to accommodate collisions, namely, those prefixes mapped to the same table index. Colliding prefixes may be combined when their NHA's are identical in order to keep the associative degree low in practice, as will be illustrated by our simulation results in Section VI.

B. Transformed Prefix Representations

The transformed prefix representations (TPR) are obtained by considering each prefix P$|w$ as a *polynomial*, $\mathrm{p}(x)$, of degree $w - 1$, i.e., $\mathrm{p}(x) = c_{w-1}x^{w-1} + \ldots + c_1\mathrm{x}^1 + c_0$. Such a polynomial is defined over a Galois Field GF(2), so that its coefficients are either 0 or 1. The algebra used for GF(2) is modulo 2, namely, the *XOR* (exclusive-or) operators. Given a generator polynomial $\mathrm{g}(x)$, we have $\mathrm{p}(x) = \mathrm{q}(x) \cdot \mathrm{g}(x) + \mathrm{r}(x)$, where $\mathrm{q}(x)$ is the quotient and $\mathrm{r}(x)$ is the remainder. Note that

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

the above polynomial arithmetic is the same as that for CRC calculation. This TPR permits us to characteristically differentiate polynomial $\mathrm{p}(x)$ using $\mathrm{q}(x)$ and $\mathrm{r}(x)$.

Theorem 1: Given a generator polynomial $\mathrm{g}(x)$, two distinct polynomials $\mathrm{p}_1(x) = \mathrm{q}_1(x) \cdot \mathrm{g}(x) + \mathrm{r}_1(x)$ and $\mathrm{p}_2(x) = \mathrm{q}_2(x) \cdot \mathrm{g}(x) + \mathrm{r}_2(x)$ can be differentiated by their unique pairs of quotients and remainders, i.e., $\langle \mathrm{q}_1(x), \mathrm{r}_1(x) \rangle \neq \langle \mathrm{q}_2(x), \mathrm{r}_2(x) \rangle$.

Proof: According to $\mathrm{p}_1(x)+\mathrm{p}_2(x) = \mathrm{q}_1(x) \cdot \mathrm{g}(x)+\mathrm{r}_1(x)+\mathrm{q}_2(x) \cdot \mathrm{g}(x)+\mathrm{r}_2(x) = (\mathrm{q}_1(x)+\mathrm{q}_2(x)) \cdot \mathrm{g}(x)+(\mathrm{r}_1(x)+\mathrm{r}_2(x))$, we have $\mathrm{p}_1(x) + \mathrm{p}_2(x) = 0$, if $\mathrm{q}_1(x) = \mathrm{q}_2(x)$ and $\mathrm{r}_1(x) = \mathrm{r}_2(x)$, since the addition is a XOR operation. This means that $\mathrm{p}_1(x)$ must equal $\mathrm{p}_2(x)$, contradicting to the fact of $\mathrm{p}_1(x) \neq \mathrm{p}_2(x)$. ■

Based on Theorem 1, it is clear that we may keep only the quotient $\mathrm{q}(x)$ in the hash table, indexed by the remainder $\mathrm{r}(x)$. The hash table consists of $2^{|\mathrm{g}(x)|}$ sets, each with α entries, where $|\mathrm{g}(x)|$ is the degree of generator polynomial $\mathrm{g}(x)$ and α is the associative degree. Consider $\mathrm{g}(x)$ being CRC-16 expressed by $\mathrm{x}^{16} + \mathrm{x}^8 + \mathrm{x}^6 + \mathrm{x}^5 + \mathrm{x}^4 + \mathrm{x}^2 + 1$, for example, a prefix $\mathrm{P}|24$ will give rise to an 8-bit quotient and a 16-bit remainder (i.e., of order 15). A hash table with 2^{16} sets, indexed by the remainder, can then be used, with only the quotient kept in an entry of the indexed set to differentiate the prefix from others. In general, this works as long as the hash table has 2^r sets when the remainder is of r bits, without aliasing concerns. This way leads to significant savings in storage when compared with keeping full prefixes. Given a degree-16 $\mathrm{g}(x)$, for example, it takes only 16 bits (or 8 bits) to keep $\mathrm{q}(x)$ for a prefix $\mathrm{P}|32$ (or $\mathrm{P}|24$). For a prefix with length less than 16, it does not even need to store any quotient (which is always 0).

While it is not uncommon in the design of hash tables to use the l least significant bits of a key to address a table with 2^l entries and to keep the remaining bits in the addressed entry. Such a design, however, often results in poor distributions of hash keys. On the other hand, remainders calculated based on a *primitive* generator polynomial were shown to yield good hash distributions [16]. A primitive polynomial $\mathrm{g}(x)$ refers to one which cannot be factored and its every possible remainder is a remainder of some power of x. We are to employ primitive generator polynomials for obtaining remainders (i.e., $\mathrm{r}(x)$) and quotients (i.e., $\mathrm{q}(x)$) of prefixes and IP addresses in our hash table design.

C. Calculation of $r(x)$ and $q(x)$

Calculating $\mathrm{q}(x)$ and $\mathrm{r}(x)$ can be achieved by a Linear Feedback Shift Register (LFSR). Fig. 2 illustrates the LFSR for $\mathrm{g}(x) = x^{16} + x^8 + x^6 + x^5 + x^4 + x^2 + 1$ used in this paper. An LFSR performs binary divisions by shift and exclusive-or operations. In every cycle, it feeds the linear system with one bit of the prefix as its input. At the end of a division, the LFSR contains its remainder.

This LFSR circuitry is relatively simple, but it takes time, namely, 32 cycles for a 32-bit prefix. A quick way of calculating remainders is to leverage on a remainder table. Since a primitive generator polynomial is such that its every remainder is a remainder of a term of some power of x, we can precalculate the remainders of terms x^n, $16 \leq n \leq 32$. A simple fact about polynomials is that the remainder of a sum of polynomials is the sum of their respective remainders, namely, $\mathrm{Rem}(\mathrm{a}(x) + \mathrm{b}(x)) = \mathrm{Rem}(\mathrm{a}(x)) + \mathrm{Rem}(\mathrm{b}(x))$. Thus, one may calculate the remainder of a prefix by considering its every bit individually. For a prefix of length l, $\mathrm{P}|l$, denoted by $\psi(x) = p_{l-1}x^{l-1} + p_{l-2}x^{l-2} + \ldots + p_j x^j + \ldots + p_0 x^0$, one first gets the remainder for each x^j whose coefficient p_j is nonzero and then performs XOR on those remainders. Hence, $\mathrm{r}(x)$ can be computed by hardware in one cycle with today's ASIC technology. After $\mathrm{r}(x)$ is obtained, one can start accessing the hash table while hiding the latency of calculating $\mathrm{q}(x)$. First of all, $\mathrm{q}(x)$ is always 0 for any prefix of length less than or equal to 16, because $\mathrm{g}(x)$ is of degree 16 in this work. For a prefix with its length greater than 16, one can obtain its $\mathrm{q}(x)$ by the following algorithm for binary divisions (identical to CRC calculation).

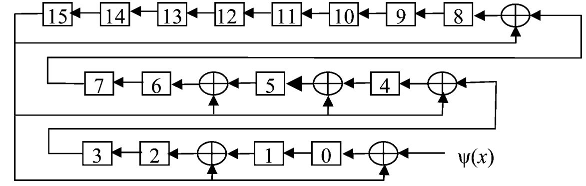


Fig. 2. The LFSR for $\mathrm{g}(x) = x^{16} + x^8 + x^6 + x^5 + x^4 + x^2 + 1$.

TABLE I
SEQUENCE OF EVENTS DURING A BINARY DIVISION

Step	q(x)	p(x)		g(x)
0		1001110	⊕	1011
1	1	~~0~~010110		1011
2	10	~~00~~10110	⊕	1011
3	101	~~000~~0000		1011
4	1010	~~0000~~000		1011

- Given an initial dividend polynomial $\psi(x)$ (represented by its coefficient set $= \{\mathrm{p}_{\mathrm{m}-1}, \mathrm{p}_{\mathrm{m}-2}, \ldots, \mathrm{p}_0\}$) and a divider $\mathrm{g}(x)$ (represented by its coefficient set $= \{\mathrm{g}_{\mathrm{n}-1}, \mathrm{g}_{\mathrm{n}-2}, \ldots \mathrm{g}_0\}$), there are two possibilities at each computation stage:
  1) If the leftmost bit of $\psi(x)$ is 0, $\psi(x)$ is shifted to the left by one position. The quotient $\mathrm{q}(x)$ is shifted to the left, appended by a value of 0.
  2) If the left-most bit of $\psi(x)$ is 1, an XOR is performed between $\psi(x)$ and $\mathrm{q}(x)$. Then, $\psi(x)$ is shifted to the left by one position. The quotient $\mathrm{q}(x)$ is shifted to the left, appended by a value of 1.

The sequence of events in Table I illustrates an example of $(x^6 + x^3 + x^2 + x)/(x^3 + x + 1)$.

Clearly, the above scheme gives rise to one resulting bit per cycle. To obtain $\mathrm{q}(x)$ for a prefix of 32 bits, it takes 16 cycles over a degree-16 $\mathrm{g}(x)$. Although one may hide the latency by overlapping accesses to the hash table, it is desirable to speed up the operation. This can be achieved by a modulo-2 arithmetic, as follows:

$$\Phi(i+1) = \mathrm{F} \cdot \Phi(i) \oplus \mathrm{G} \cdot \Phi(i)_0 \qquad (1)$$

where $\Phi(i)$ is the ith state of the linear system for $\psi(x)$, namely, the current dividend after the ith subtraction and $\Phi(i)_0$ denotes the one-bit shift-in serial input of current $\psi(x)$, which is the first element of matrix $\Phi(i)$ and also the resulting bit

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

for $\mathrm{q}(x)$ after this step. Additionally, G is an (m by 1) matrix $[\mathrm{g}_{n-2}, \ldots \mathrm{g}_0, 0, 0 \ldots 0]^T$, where the first $(n-1)$ elements are from $\mathrm{g}(x)$, with g_{n-1} unused. As shown in the above example, the left-most resulting bit after each subtraction is discarded due to the shift operation. Finally, F is an (m by m) matrix of

$$F = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \ldots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \ldots & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & \ldots & 0 & 0 \end{bmatrix}.$$

Note that Equation (1) simply implements the algorithm for a binary division. If $\Phi(i)_0$ equals 0, only the shift operation is performed on $\Phi(i)$ (by $\mathrm{F} \cdot \Phi(i)$). If $\Phi(i)_0$ is 1, an XOR and a shift operations are performed over the two components. In a generic form, the solution for Equation (1) is given by

$$\Phi(i) = \mathrm{F}^{(i)} \cdot \Phi(0) \oplus [\mathrm{F}^{i-1} \cdot \mathrm{G}, \ldots \mathrm{F} \cdot \mathrm{G}, \mathrm{G}] \cdot [\Phi(i-1)_0, \ldots \Phi(0)_0]^{\mathrm{T}}. \quad (2)$$

The preceding equation demonstrates the ith state of the dividend related to the initial state and the inputs. Hence, one can accomplish parallel calculation on $\mathrm{q}(x)$ by expanding Equation (2) for the first w bits of $\psi(x)$. Then, the next batch of w bits can be calculated by replacing $\Phi(0)$ in the expression with $\Phi(w)$. This way accelerates the calculation of $\mathrm{q}(x)$ by w times. We believe $2 \leq w \leq 4$ is reasonable, in light of the additional hardware cost. In the worst case, dividing a 32-bit prefix $\psi(x)$ by a degree-16 $\mathrm{g}(x)$ takes 8 (or 4) cycles for $w = 2$ (or 4).

IV. Prefix Aggregation

For table storage reduction and speedy IP address lookups, our SUSE design has chosen a DPL (designated prefix length) set, based on which prefixes are rounded off before being stored in the hash table, as mentioned earlier. Each prefix length in the DPL set is referred to as a *tread* and the distance between two consecutive treads is called the *stride*. As the length of a given prefix (say, $\psi(x)$) is known, our design calculates $\mathrm{r}(x)$ and $\mathrm{q}(x)$ of $\psi(x)$ before storing it in a hash table entry indexed by $\mathrm{r}(x)$ during hash table creation and updating, and this is achieved with respect to just a single tread (determined by the length of $\psi(x)$). To arrive at a compact hash table, SUSE follows prefix aggregation to keep multiple prefixes in a table entry. This aggregation can be carried out in two ways: round-off aggregation and bit-mapping aggregation, where the former targets prefixes with the same NHA (next-hop address) and the latter deals with more efficient aggregation via bit mapping. An important concern in aggregation is to preserve complete route information even after route updates and withdrawals, since an aggregated entry corresponds to multiple prefixes and its deletion removes all its represented prefixes. If one prefix (say, $\mathrm{P}|l$ which was aggregated with another prefix, say, $\mathrm{P}'|l'$, to share a table entry) is withdrawn, Prefix $\mathrm{P}'|l'$ should remain in the table after the withdrawal of $\mathrm{P}|l$, to ensure proper LPM lookups for incoming IP addresses. For example, two prefixes 101* and 10* with the same NHA can be aggregated for a $\mathrm{tread} = 2$, represented by the single prefix of 10* (with different round-off bits). When route 10* is deleted, Prefix 101*should be preserved in the routing table. A design whose route information is never lost during route updates and withdrawals is referred to as *lossless*. Unlike many earlier optimization techniques which could lose route information upon route deletion [13], SUSE is always lossless, existing in 3 different versions governed by aggregation schemes detailed in sequence.

A. Round-Off Aggregation (ROA)

This aggregation can be made very aggressively by putting many prefixes, which belong to the same tread and share the same NHA, into one hash entry, with appropriate fields employed to record their corresponding round-off bits. However, this aggressive approach complicates hash table maintenance logics greatly, since a single prefix withdrawal could result in multiple round-off fields being created to ensure the lossless property. If Prefixes 1001<u>00</u>*, 1001<u>01</u>*, 1001<u>10</u>*, and 1001<u>11</u>* are allowed to share one hash entry under $\mathrm{tread} = 4$, for example, the round-off bits of the four prefixes are denoted by $**$, occupying a single field. When a prefix, say 100101*, is deleted, the original aggregated entry has to take up two round-off fields, namely, 00 and 1*, for indicating the remaining three prefixes after deletion.

Our design chooses to adopt limited round-off aggregation for hardware simplicity and a fast operation during prefix deletion, letting only two prefixes aggregated in each entry. As a result, a single don't care field (to denote if the last round-off bit is don't care) is needed in the hash table design under this limited aggregation, as shown in Fig. 3. This aggregation scheme exhibits less than satisfactory utilization in hash table storage, as will be demonstrated in Section VI-A by results obtained with respect to real-world prefix tables of various sizes. It gives rise to SUSE with round-off aggregation, denoted by $\mathrm{SUSE_{ROA}}$.

B. Lossless Aggregation Bit-Maps

Improved utilization in table storage can be achieved using bit-mapping aggregation. Two bit-mapping designs which enable deep prefix aggregation with slightly more storage than simple ROA, are presented next.

1) Full-Fledged Bit Maps (FFBM): The first scheme employs a full-fledged bit-map (FFBM) to record round-off bits, yield $\mathrm{SUSE_{FFBM}}$. Because any prefix P of length w with $l_i \leq w < l_{i+1}$ is <u>rounded down</u> to $\mathrm{P}|l_i$, $(w - l_i)$ round-off bits mean that at most $2^{(w-l_i)}$ prefixes of length w can have the same hash index. Instead of using $2^{(w-l_i)}$ full-size entries, a vector of $2^{(w-l_i)}$ bits can accurately record the $(w - l_i)$ round-off bits; this way lowers needed storage dramatically. In total, all prefixes $\mathrm{P}|w$, where $l_i \leq w < l_{i+1}$, can be remembered by a full map of $(\sum_{k=0}^{l_{i+1}-l_i-1} 2^k)$ bits when all the prefixes have the same NHA. It represents the greatest aggregation achievable by this FFBM scheme.

Fig. 4 demonstrates the layout of prefixes with different lengths in a hash table entry. The first column lists the chosen prefix lengths in DPL (encoded as shown in the second column). In this design, only the lengths in DPL need to be encoded because the bit-maps of the third column freely indicate the

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

Tread	5-bit indicator	3-bit round-off / 1 don't-care aggregation bit/*m*-bit q(*x*)			8-bit NHA
P|32	00100	16-bit q(*x*)			NHA
P|28	P|{31..28}:11000..11011	round-off	*d*	12-bit q(*x*)	NHA
P|24	P|{27..24}:11100..11111	round-off	*d*	8-bit q(*x*)	NHA
P|20	P|{23..20}:01000..01011	round-off	*d*	4-bit q(*x*)	NHA
P|16	P|{19..16}:01100..01111	round-off	*d*		NHA
P|12	P|{15..12}:10000..10011	round-off	*d*		NHA
P|8	P|{11..8}:10100..10111	round-off	*d*		NHA
	← Total 29 bits per entry →				

Fig. 3. Per-entry format in simple round-off scheme.

Tread	4-bit indicator	15b bit-maps				13-bit q(*x*)/NHA	8-bit NHA
P|29	0001	8	4	2	1	13-bit q(*x*)	NHA
P|25	0010	8	4	2	1	9-bit q(*x*)	NHA
P|24	1101/1110/1111	8-bit q(*x*)	8-bit q(*x*)			8-bit NHA	NHA
P|20	0011	8	4	2	1	4-bit q(*x*)	NHA
P|16	0100	8	4	2	1		NHA
P|12	0101	8	4	2	1		NHA
P|8	0110	8	4	2	1		NHA
	← Total 40 bits per entry →						

Fig. 4. Per-entry format in full-fledged bit-maps scheme.

number of round-off bits. For example, to search for a prefix $\mathrm{P}|28$, the hash index is calculated based on $\mathrm{P}|25$. While a matching $\mathrm{q}(x)$ must be found, the 8-bit bit map is also examined for the three round-off bits.

As shown by Figs. 3 and 4, the FFBM scheme requires a longer entry (of 40 bits) than ROA (of 29 bits). With additional storage, FFBM can achieve a greater amount of aggregation, as to be shown in Section VI-A. The longest entry is determined by the bit maps and the length of $\mathrm{q}(x)$ for $\mathrm{P}|29$. To keep the bit maps reasonably short, a small stride of 3 is chosen. This leaves $\mathrm{P}|24$ to be specially handled. Our idea is to pack two $\mathrm{P}|24$ prefixes in one entry so that better storage utilization can be achieved. The indicator with a value of 1101 or 1110 (or 1111) means that the first or the second subentry holds (or both subentries hold) a $\mathrm{P}|24$ prefix. Nevertheless, unused space for treads below 24 could waste sizeable storage, as depicted in Fig. 4. To make effective use in space, we introduce the Controlled Bit-Maps scheme next.

2) Controlled Bit-Maps (CBM): FFBM's effectiveness relies on a single hypothesis—a large number of prefixes between treads can be aggregated. This happens only to those prefixes which have the same NHA. Prefixes with the same leading bits are found more commonly to share a few next-hop addresses (instead of just one NHA). For example, a group of $\mathrm{P}|19$ subnets shares $\mathrm{NHA_A}$ and another group of $\mathrm{P}|19$ subnets uses $\mathrm{NHA_B}$. Although both groups may be under the same $\mathrm{P}|16$ network, FFBM's effectiveness is compromised due to their different NHAs. In such cases, packing and fitting as many prefixes as possible in one table entry is a more effective solution.

Fig. 5 illustrates the formation of a table entry under controlled bit-maps (CBM). In essence, every entry contains two fields, each comprising a bit-map, a $\mathrm{q}(x)$, and one NHA. Each of the two fields intends to hold one aggregated prefix, except for prefixes of length greater than 24 (where each table entry then holds just one aggregated prefix). The second column of Fig. 5 contains the length indicators (but for $\mathrm{P}|20$ and $\mathrm{P}|21$, we leverage on two unused bits in the bit-map to encode them as shown). Since each field has its own length indicator, there is no restriction on the prefixes stored in an entry. Prefixes of different lengths can be mixed in the same entry as long as space permitted. The bit-map serves the same purpose as that in the FFBM scheme, but it works only for the *single designated* prefix length specified by the indicator. As the length indicator now specifies the actual prefix length (rather than the tread as in FFBM), we aggregate only prefixes of the same length under CBM, leading to $\mathrm{SUSE_{CBM}}$.

For example, two $\mathrm{P}|18$ prefixes with the same NHA can be aggregated. They are rounded down to tread $\mathrm{P}|16$ for hash calculation (but the length indicator will specify 18). The two round-off bits will be captured by the bit-map as usual. In this case, only the last 4 bits of the map are used. When a new $\mathrm{P}|19$ prefix is to be added, a new entry will be allocated. Although a controlled prefix expansion [19] may be applied to expand a $\mathrm{P}|18$ prefix into two $\mathrm{P}|19$ prefixes so that all these prefixes can be aggregated aggressively into one $\mathrm{P}|19$ entry, our CBM chooses not to, because otherwise, a prefix withdrawal may result in creating an extra table entry. This choice keeps a simple *lossless* design for CBM, facilitating fast route updates and withdrawals.

V. Skewed Memory Layout for Mapping Multiple Hash Tables

Section III-A has explained that SUSE collapses multiple hash tables [8] into a single one. A method is then demonstrated to transform any prefix into a polynomial representation for easy calculating its quotient $\mathrm{q}(x)$ and remainder $\mathrm{r}(x)$ over a divisor $\mathrm{g}(x)$. It is followed by the selection of treads used to search the hash table during lookups when IP addresses arrive. Each table search involves memory accesses. Naturally, all memory

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

Tread	Length Indicator			8 bits bit-map/q(x)	16 bits field: bit-map/q(x)/NHA		8-bit NHA
P|29	0	0000	P|{32..29}	8b bit-map	13b q(x)		NHA
P|25	0	0000	P|{28..25}	8b bit-map		9b q(x)	NHA
P|16/12/8	1	P|{8..19}	P|{8..19}	8b bit-map	8b bit-map	NHA	NHA
P|20	1	P|{20,21}	P|{20,21}	2b | 00/01 | 4b q(x)	2b | 00/01 | 4b q(x)	NHA	NHA
P|22	1	P|{22,23}	P|{22,23}	2b | 6b q(x)	2b | 6b q(x)	NHA	NHA
P|24	1	P|24:1111	P|24:1111	8b q(x)	8b q(x)	NHA	NHA
	← Total 41 bits →						

Encoding: P|{32..29}: 0001..0100, P|{28..25}: 0101..1000, P|{8..19}: 0000..1011, P|{20,21}: 1100, P|{22,23}: 1101..1110

Fig. 5. Per-entry format in controlled bit-maps (CBM) scheme.

accesses should take place in parallel to minimize the search time, calling for independently accessed memory modules. To keep hardware complexity low, the number of memory modules is assumed to be a power of 2 (for example, 8 modules). Our design requires a hash table with a total of 2^{16} sets, because the generator $\mathrm{g}(x)$ has a degree of 16. For 8 modules to constitute this hash table, each module then has 2^{13} sets. Each prefix, $\psi(x)$, belongs to one of the treads in set {29, 25, 24, 22, 20, 16, 12, 8} given in Section IV-B2 based on its length, with corresponding $\mathrm{r}(x)$ and $\mathrm{q}(x)$ calculated.

During the table lookup, an IP address is searched for *all the legitimate treads* (i.e., those in the set {29, 25, 24, 22, 20, 16, 12, 8}) to realize LPM (longest prefix matching). Eight sets of hardware logics are equipped for computing $\mathrm{r}(x)$ and $\mathrm{q}(x)$ simultaneously. To support fast lookups, it is desirable to launch hash table accesses in parallel, guided by the eight computed $\mathrm{r}(x)$'s. To this end, the sets pointed by those eight $\mathrm{r}(x)$'s must fall in different memory modules. If multiple $\mathrm{r}(x)$'s point to a memory module, accesses to the module experience a conflict, which requires corresponding memory accesses issued in order. Take an IP address of 192.128.x.y as an example. The most significant 16 bits of the address are 11000000 10000000. By the arithmetic of Section III-C, we arrive at $\mathrm{r}(x)$'s for treads 8, 12, and 16, as follows:

$$\begin{aligned}\mathrm{r}_8(x) &= T(\mathbf{11}000000) = \mathrm{Rem}(x^7) + \mathrm{Rem}(x^6)\\ &= 0000000011000000\\ \mathrm{r}_{12}(x) &= T(\mathbf{11}0000000\mathbf{1}000)\\ &= \mathrm{Rem}(x^{11}) + \mathrm{Rem}(x^{10}) + \mathrm{Rem}(x^3)\\ &= 0000110000001000\\ \mathrm{r}_{16}(x) &= T(\mathbf{11}0000000\mathbf{1}0000000)\\ &= \mathrm{Rem}(x^{15}) + \mathrm{Rem}(x^{14}) + \mathrm{Rem}(x^7)\\ &= 1100000010000000\end{aligned}$$

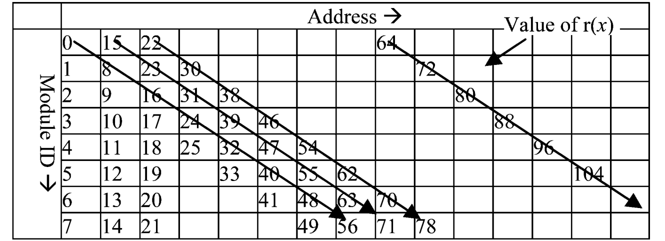


Fig. 6. Map $\mathrm{r}(x)$ to memory array with $\mathrm{skew\ degree} = 1$.

If a straightforward mapping is adopted to modulo the number of memory modules, all three indexes will point to memory module 0. This causes access conflicts, leading to undesired serial launches of the three search operations to the same memory module. To overcome this problem, a skew factor is employed when collapsing multiple hash tables into one. Fig. 6 depicts an example of distributing $\mathrm{r}(x)$ to 8 memory modules, with a skew degree of one. Mathematically, address decoding can be expressed by the equation at the bottom of the page.

Because the number of memory modules is assumed to be a power of 2, it only takes simple operations to compute the address and the module ID of each computed $\mathrm{r}(x)$. This way uses distinct skews for different treads to reduce the likelihood of memory conflicts. Search by $\mathrm{r}_8(x), \mathrm{r}_{12}(x)$, and $\mathrm{r}_{16}(x)$ in the above example will be directed respectively to modules 0, 1, and 2, if skew degrees of 0, 1, and 3 are used accordingly.

VI. Evaluation and Results

A. Prefix Distribution in Hash Table

To examine effectiveness in the storage utilization of SUSE under round-off aggregation (denoted by $\mathrm{SUSE_{ROA}}$) and under bit-mapping aggregations (denoted by $\mathrm{SUSE_{FFBM}}$ and

$$\mathrm{Address} = \mathrm{r}(x)/8$$

and

$$\mathrm{Module\ ID} = \begin{cases} r(x) \bmod 8, & \text{when skew} = 0\\ (r(x)/8 + r(x) \bmod 8 + \text{skew} - 1) \bmod 8, & \text{when skew} > 0.\end{cases}$$

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

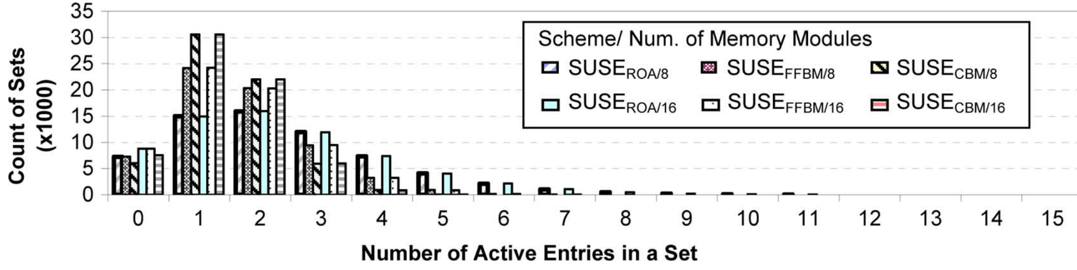


Fig. 7. Distribution of prefixes in AS12654 BGP table under different aggregation schemes.

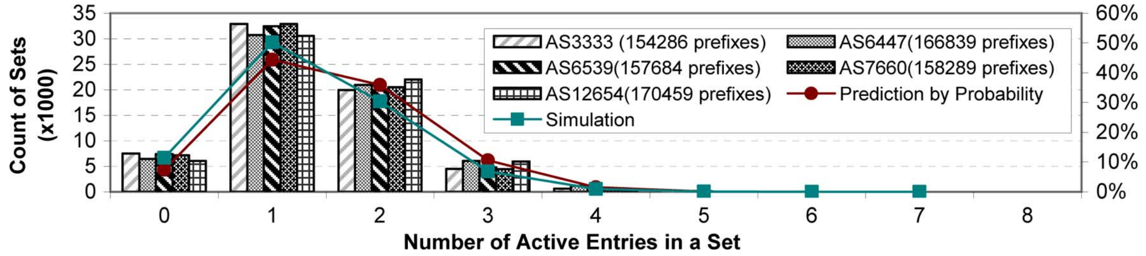


Fig. 8. Prefix distribution for five other routing tables under $\mathrm{SUSE_{CBM}}$.

$\mathrm{SUSE_{CBM}}$) as described in Section IV, real-world prefix tables obtained from [2] are employed to build the hash table. Fig. 7 demonstrates the prefixes distribution under the three design schemes with the number of memory modules equal to 8 and 16, when AS12654 BGP routing table is used for table construction. $\mathrm{SUSE_{ROA}}$ is observed to have a relatively limited success on aggregating prefixes. One don't-care bit in each table entry groups at most two prefixes only. Under $\mathrm{SUSE_{ROA}}$, up to 15 prefixes (aggregated or not) can fall into one hash table set in the worst case. It means that a 4-way set-associative table needs a method to keep 12 overflow prefixes. $\mathrm{SUSE_{FFBM}}$ and $\mathrm{SUSE_{CBM}}$ achieve much higher degrees of prefix aggregation than $\mathrm{SUSE_{ROA}}$. They result in at most 7 and 6 prefixes, respectively, hashing into one table entry, no matter whether the number of memory modules is 8 or 16.

Overall, a 4-way set-associative table is adequate, especially for $\mathrm{SUSE_{CBM}}$. $\mathrm{SUSE_{FFBM}}$ is particularly effective when prefixes share the same common "leading" prefixes. For example, all prefixes of length 16, 17, 18, and 19 which share the same 16 leading bits (namely, for all $\mathrm{B_1.B_2}$.*.*) can be represented together in *one* entry if they have the same next-hop address. However, it is more common that prefixes with the same leading prefix share a few next-hop addresses. Specifically, in both $\mathrm{SUSE_{FFBM}}$ and $\mathrm{SUSE_{CBM}}$, $\mathrm{P|19}$ is rounded down to $\mathrm{P|16}$ for calculating the hash index. Out of the 8 combination values of the three round-off bits $\{\mathrm{b_{17}, b_{18}, b_{19}}\}$, some may have the same next-hop address, $\mathrm{NHA_A}$, and others have a different next-hop address, $\mathrm{NHA_B}$. Consequently, $\mathrm{SUSE_{CBM}}$ which packs two prefix groups in one entry leads to better space utilization than its $\mathrm{SUSE_{FFBM}}$ counterpart. Because $\mathrm{SUSE_{CBM}}$ exhibits the best overall benefit, we henceforth focus merely on $\mathrm{SUSE_{CBM}}$.

Similar results have been obtained for other BGP tables, which all contain more than 150 K prefixes each (including AS3333, AS6447, AS6539, and AS7660). As can be seen in Fig. 8, a negligibly small number of sets receive more than 4 prefixes for all the routing tables examined under $\mathrm{SUSE_{CBM}}$. $\mathrm{SUSE_{CBM}}$ is also effective in grouping prefixes, with 90% of the sets involving no more than two active entries.

From a theoretic analysis perspective, the probability distribution could be approximated by a Bernoulli process assuming a uniform hash distribution. Thus the probability of hashing a prefix $\psi(x)$ to an entry in a table with m entries is $1/m$. The probability for k prefixes hashing to an entry equals $\binom{n}{k} \bullet (1/m)^k \bullet (1-1/m)^{n-k}$. For a 4-way set-associative hash table with 64 K sets and each table entry comprising two fields (Fig. 5), prediction on the utilization of a set by the simple Bernoulli model is illustrated in Fig. 8. The results are fairly close to the simulation outcomes, though the latter shows that the hash table is less loaded than prediction. This minor discrepancy is because the simple Bernoulli model does not include the benefit of prefix aggregation (Section IV-B2). Overall, 4-way $\mathrm{SUSE_{CBM}}$ is shown to be pleasantly effective.

B. Impact on Prefix Distribution by Number of Treads

The number of memory modules has little impact on the hash distribution or collisions of prefixes (as depicted in Fig. 7 for the situations of 8 and 16 memory modules), nor collapsing multiple prefix tables intensifies the problem. However, it is found that the selection of treads (namely, legitimate prefix lengths) has a more pronounced impact than other factors.

Three representative cases, in which the number of treads (denoted by ζ) equals 25, 8, and 5, are examined, with their outcomes demonstrated in Fig. 9, where the percentage values of active entries in a set > 4 are listed explicitly. Calculating hash indexes based on the GF(2) arithmetic stated in Section III-B is believed to yield good distributions, as unfolded by prior research [16]. The case with 25 treads, one for each prefix length ($\mathrm{P|8}$ to $\mathrm{P|32}$), reflects the best possible prefix distribution. It

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

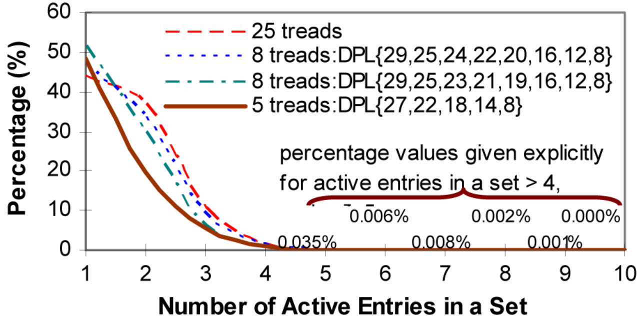


Fig. 9. Distribution of prefixes under different ζ.

offers an indication of how well prefixes scatter across the collapsed hash table. Fig. 9 depicts that most sets in this case have small numbers of active entries.

While a finer stride is expected to yield a better hash distribution, it also comes with an increased cost for the memory system since it then has to support more parallel accesses. Furthermore, table storage also increases to accommodate the bit-maps, whereas no bit-map is needed for $\zeta = 25$. The proposed approach detailed in Section IV-B2 achieves a good compromise, by using merely 8 treads for hashing. $\zeta = 8$ (requiring the support of 8 parallel accesses) arrives at a cost-effective memory system design, taking about 20% more storage than the case under $\zeta = 25$.

When ζ drops further, prefix distribution results deteriorate. As shown by Fig. 9, collisions start sprouting markedly when ζ equals 5, although the percentage of overflowing a 4-way set is considerably low. Many overflow sets are due to the following reason. Consider a number of prefixes between $\mathrm{P}|16$ and $\mathrm{P}|21$ that have the same leading prefix $\mathrm{P}|16$. When length 16 is chosen for hashing, all those prefixes collide because of their identical leading prefix. In this case, the 5 round-off bits can result in up to 32 prefixes falling into one table entry. Thus, it complies with the general expectation—the more round-off bits, the higher chance of hash collisions. Additionally, dropping ζ from 8 to 5 increases the storage by a whopping 50% because of the longer bit map.

To further assist in understanding treads selection and to prove the effectiveness of $\mathrm{SUSE_{CBM}}$ at $\zeta = 8$, we simulate another DPL: $\{29, 25, 23, 21, 19, 16, 12, 8\}$. In this run, we adopt a small stride of 2 for prefixes of lengths between 19 and 24, which are known to contain most of prefixes in existing routing tables. As illustrated in Fig. 9, the result is consistent with the previous findings. Most sets include a few active entries. Consequently, the following general guidance is reached. For the set of most common prefixes between $\mathrm{P}|24$ and $\mathrm{P}|19$, we prefer a design with multiple prefixes *packed* in one table entry. Those prefixes tend to be *disjoint* in that they occasionally share the same leading prefixes. As a result, we rely on a good hash distribution achieved by the hash function outlined in Section III-B to spread the prefixes throughout the table. On the other hand, those shorter prefixes mean larger network routes, thereby providing abundant opportunities for aggregation, which favors $\mathrm{SUSE_{CBM}}$.

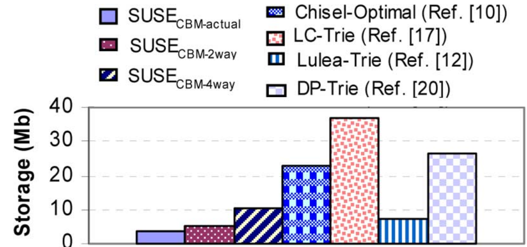


Fig. 10. Storage requirement for AS12654 BGP routing table (with 170 K+ prefixes).

C. Storage Requirements

Fig. 10 depicts storage usage for two families of designs: one based on hash tables and the other on tries. All results are for the AS12654 BGP routing table (with 170 K+ prefixes). For $\mathrm{SUSE_{CBM}}$, three numbers are reported: (a) storage used to actually keep active prefixes, (b) storage provision for a 2-way hash table, and (c) storage provision for a 4-way table. The ratio between the first number to the second (or third) number is referred to as *table space utilization*.

First of all, $\mathrm{SUSE_{CBM}}$ consumes 3.8 Mb, as depicted by the leftmost bar in Fig. 10. This is very small when compared with other schemes. The second (or third) bar shows that the storage size for a 2-way (or 4-way) table is 5 Mb (or 10 Mb) SRAM. Interestingly, according to the results of Figs. 7 and 8, 90% of the sets have no more than two active entries. We thus conjecture that a 2-way table with the aid of CBM aggregation and a simple overflow scheme under SUSE can be very effective for routing/forwarding tables with 200 K+ prefixes. Since the number of overflow prefixes under a 4-way design for DPL:$\{29, 25, 23, 21, 19, 16, 12, 8\}$ is found to range from 11 (for AS7660) to 109 (for AS6447), an alternative design may use a 4-way table (comprising 10 Mb SRAM) with a small spillover TCAM (composed of 256 entries or less) to achieve a deterministic lookup latency.

Fig. 10 shows the results for LC-Trie [17], Lulea-Trie [12], and DP-Trie [20]. In the family of trie-based schemes, Lulea normally requires the smallest amount of space. In our evaluation, Lulea-Trie takes up 7.45 Mb, close to that of $\mathrm{SUSE_{CBM}}$. However, $\mathrm{SUSE_{CBM}}$ enjoys easier and faster incremental updates than Lulea-Trie. LC-Trie and DP-Trie consume 37.14 and 26.3 Mb, respectively. Proposed $\mathrm{SUSE_{CBM}}$ thus demonstrates at least 85% storage reduction in comparison to LC-Trie and DP-Trie.

SUSE is also compared with the Chisel design [10]. We simulated Chisel by a set of proven universal hash functions [31], [32], which produce good distributions of hash values. Recall the discussion of Section II-B2. Given n prefixes, k hash functions (satisfying $m \geq k \cdot n$) are used to generate k hash values to index a table with m entries. According to [31], a universal hash function is achieved by treating a prefix of l bits as a series of disjoint k words $\{w_{k-1}, \ldots, w_0\}$, where each word consists of l/k bits. k random values are then selected from the value range of 0 to $m-1$. A hash function is defined as $h = (w_{k-1} * d_{k-1})^\wedge(w_{k-2} * d_{k-2})^\wedge \ldots ^\wedge (w_0 * d_0)$. While a more generic form of the universal hash function is to narrow

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

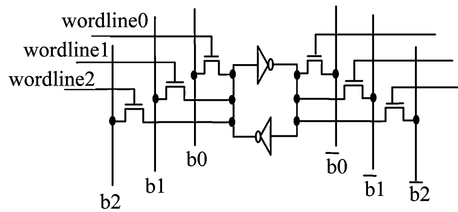


Fig. 11. A multiport SRAM cell.

each word to one bit (at the cost of many more hash functions), we do not find any significant difference in the simulation results. Therefore, we present one set of data only.

The m/n ratio is another important factor for the Chisel-like design. The larger the ratio is, the easier the counting Bloom filters converge and the lower the probability of hash collisions, naturally taking more memory. By adopting the parameters suggested in [10], we set the m/n ratio of 3 and the $\text{stride} = 4$ such that there are totally 5 subcells. Fig. 10 illustrates that the optimal configuration for Chisel takes about 23 Mb SRAM, as opposed to 10 Mb for our 4-way SUSE_{CBM},[1] leading to a storage reduction of 56%. The simulated configuration is optimal because we measure the storage required to keep the route prefixes of AS12654 exactly. For instance, if a subcell has n prefixes, the filter table is sized with precisely n entries to keep those prefixes with the designated tread lengths, without provision for expansion (understood to be impractical in real field deployment). Nevertheless, this optimal configuration serves as the comparison baseline. Results depicted in Fig. 10 clearly demonstrate the superior storage-efficiency of SUSE_{CBM}.

Another factor to be considered is the SRAM bit cell size, in addition to the number of bits. Solutions such as Chisel require calculation of k hash functions and k accesses to the Bloom filter in parallel. This calls for k-port SRAM memory. Fig. 11 shows a multiport memory using the simple 5 T SRAM cell. A k-port cell needs k word lines and k bit lines. If the number of ports grows, the cell becomes more wire-limited, especially when the feature size shrinks.

Generally speaking, multiport SRAM is not space-efficient. While a smaller feature size improves device performance, the impact of miniaturization has been less positive for interconnects [26]–[28]. A smaller cross section, wire pitch, a longer wire, and larger loading all increase the resistance and the capacitance of the device, consequently elevating the signal propagation (RC) delay. Thus, the bit cell size is rather determined by the wiring requirements [26]. To the best of our knowledge, there are no publicly known area results for multiport SRAM cells, especially when exceeding two ports. To gain the first-order idea, we compare a single-port SRAM module to its dual-port counterpart using the TSMC 65-nm library by Broadcom's memory compiler. It is revealed that the dual-port SRAM almost doubles the area. Therefore, the chip area size for Bloom filter-based solutions like Chisel is worse than what is suggested by simply counting the number of bits in Fig. 10. We further contemplate that a Bloom filter with k hash functions will post a challenge to practical implementation of space-efficient circuit.

[1]Results for the Chisel turn out to be difficult to generalize. By changing the strides, the results fluctuate dramatically. Nevertheless, we had simulated strides ranging from 1 to 6 (or equivalently, 12, 8, 5, and 4 subcells). SUSE_{CBM} consistently shows 56% to 83% storage reduction when compared with its Chisel counterpart.

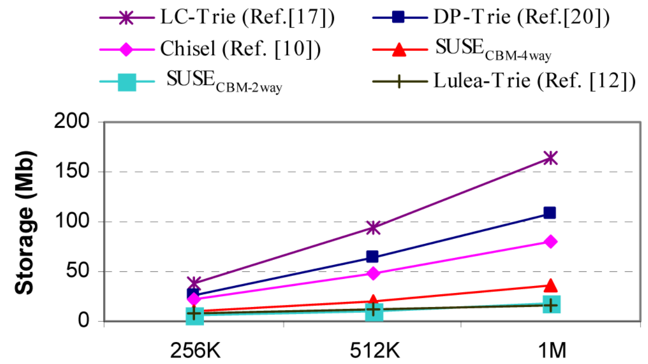


Fig. 12. Storage space versus routing table size.

An alternative which uses k single-port memory modules comes with two issues. First, the worst case may cause all k accesses to the same memory module. This is the same structural conflicting issue discussed in Section V; it forces serial accesses to the same module. The Bloom filter itself posts a danger of being the bottleneck in this case. Second, as we have discussed in Section II-B2, all original prefixes collapsed to a predetermined length l form a *subcell* in Chisel [10], where the subcell assimilates to our DPL treads. If each subcell requires k memory modules for the index table and two modules respectively for the filter table and for the vector table, the design will comprise tens of memory blocks, and worst yet, they all have different sizes. The difficulty in such a design is that the physical placement of those point-like blocks becomes absolutely critical; otherwise, the majority of wires turn to be global, presenting a challenging physical design problem to the overall wiring plan.

D. Scalability

Proposed SUSE_{CBM} is scalable. When provisioning for more prefixes, a higher degree $\text{g}(x)$ will naturally be employed to produce a longer index $\text{r}(x)$ necessary for a larger hash table. Subsequently, the length of $\text{q}(x)$ shrinks. Considering a design for one million prefixes, as an instance, a 4-way hash table of one million entries comprises 2^{18} sets. A $\text{g}(x)$ with degree-18 produces $\text{q}(x)$ with 2 bits shorter than does a $\text{g}(x)$ of degree-16. Due to this interesting property, $\text{q}(x)$ eventually occupies a small portion of total storage. Naturally, this scheme is toward using prefixes as the addresses of memory modules which constitute the routing table [15].

Because a real routing table with 512 K (or 1 million) prefixes does not exist yet, we put together two synthetic tables from those real AS tables. The two synthesized prefixes tables have 486 435 and 888 335 prefixes, respectively. The hash table is configured accordingly with 512 K and 1024 K entries. Under a 4-way set-associative configuration, the storage amount for SUSE_{CBM} is depicted in Fig. 12 for comparison with other methods. All results are obtained by simulation.

SUSE_{CBM} is clearly superior, as its storage growth is steadily linear. While Lulea-Trie has the lowest storage requirement, it employs several techniques to lower storage, including leaf

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

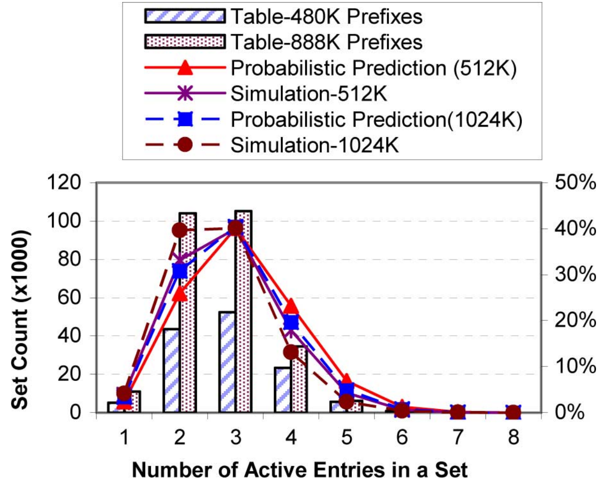


Fig. 13. Prefix distribution for large routing tables.

pushing, making a complete trie, the use of codeword encoding, and packing pointers in a sequential array. These techniques render incremental updates for Lulea-Trie very difficult, if not impossible. By contrast, incremental updates for $\mathrm{SUSE_{CBM}}$ are straightforward. It should be noted that $\mathrm{SUSE_{CBM}}$ spends almost 40% of its storage on the next-hop address. Strictly speaking, NHA is not a part of the lookup apparatus. Thus, $\mathrm{SUSE_{CBM}}$ has room for improvement.

Fig. 13 demonstrates the prefix distribution in $\mathrm{SUSE_{CBM}}$ for the two synthesized routing/forwarding tables. Once again, because of the prefix aggregation benefit, the simulations show a less loaded table than what the simple Bernoulli model predicts. The results further prove that $\mathrm{SUSE_{CBM}}$ is effective, since a vast majority of sets contain no more than two active entries. It is very rare to have more than 4 entries in a set to cause collisions. As a result, a 4-way design with some simple strategy for handling set overflow suffices. The active entries nevertheless comprise only some 45% of total storage, presenting room for improvement in storage utilization.

The proposed architecture is suitable for IPv6 addressing as well by employing more treads (probably 16), with larger gaps between treads. The generator polynomial $g(x)$ used can have a degree higher than 16.

E. Performance

1) Constant-Time O(1) Operations: $\mathrm{SUSE_{CBM}}$ exhibits constant-time operations for lookups and incremental updates. Fig. 14 illustrates the steps for a lookup. Algorithmically, searching different prefix lengths incurs no dependent memory accesses. However, structural dependency may arise due to conflicts in memory accesses. If memory conflicts occur, a penalty exists for sequencing accesses in order. Because the number of treads used for hash calculation is 8, the pathological case happens when all 8 memory accesses must be performed in order. Of course, it is only a theoretic upper bound on the IP lookup time, assuming that a small spillover TCAM is accessed simultaneously.

Incremental updates are based on the lookup operation. To announce a new route or withdraw an existing one, the same hashing steps are performed to find the appropriate table entry,

Lookup(Prefix):

For each length in the DPL set: for example, {P|8, P|12, P|16, P|20, P|22, P|24, P|25, P|29}
Calculate q(x) and r(x) by division using g(x) (Sections III.B and III.C).
Remember the round-off bits (Section IV).
Compute the memory addresses and the module ID (Section V).
Memory Scheduler dispatches accesses in batches to memory modules avoiding conflicts.
Check returned table entries for the chosen prefix lengths, matched q(x) and round-off bits.
Report the LPM result.

Fig. 14. Pseudo code for the lookup operation.

involving constant time operations as well. This is another advantage over its trie-based counterparts, besides a gain in storage reduction.

2) Average Number of Memory Accesses: To gain insight into memory conflicts, we employed synthetic lookup traces for our studies due to the lack of real ones. The input lookup file was created as follows. A prefix was chosen randomly from the routing table to generate lookup IP addresses. Given a prefix with length 16 (say, $\mathrm{B_1.B_2}$.0.0), for example, IP lookup addresses were produced by replacing 0.0 with random bit strings $\mathrm{B_3.B_4}$ to yield $\mathrm{B_1.B_2.B_3.B_4}$. Thus, for a chosen prefix in AS1256 routing table, a needed number of IP addresses were generated, with their LPM (longest prefix matching) being the chosen prefix.

A parallel probe to different sets in distinct modules is counted as one memory access. Collisions to the same module result in a serial order of accesses. All prefixes in one set are fetched simultaneously. Given each entry with 41 b, a 4-way set design requires 164 b datapath per module. This can be easily accomplished by today's technology (Section VII). In a run of millions of lookups in our simulation for a 4-way CBM design, per lookup operation on an average logged 1.07 memory accesses. The worst case took 4 memory accesses with a standard deviation of 0.25, according to our extensive simulation study. By fitting the routing table into on-chip SRAMs, SUSE can easily achieve over 100 M lookups per sec.

We further simulated a design using the Counting Bloom Filter (CBF). The results in Fig. 15 reveal that the occurrence of false positives dwindles when more hash functions are deployed. In order to lower false positives to a content level, it often requires more than 4 or 5 hash functions. With adequate CBFs, the number of hash probes approaches one per lookup. This is consistent with the past results [25].

As discussed in Introduction, CBF may be a good data structure for the membership query, but it does not manifest where the interested objects are stored in a hash table nor does it suggest how hash collisions should be handled. That said, to arrive at a collision-free hash function for routing tables is a classic issue. Assuming a straightforward open hash design with the number of its buckets being twice the number of prefixes (N) and collisions resolved by linked lists, we obtained the average number of memory accesses per probe as demonstrated in Fig. 15. As stated above, a query to the routing table is more targeted and selective for CBF with more hash functions. According to our simulation, the mean linked-list length is 2, signifying that the average number of (off-chip)

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

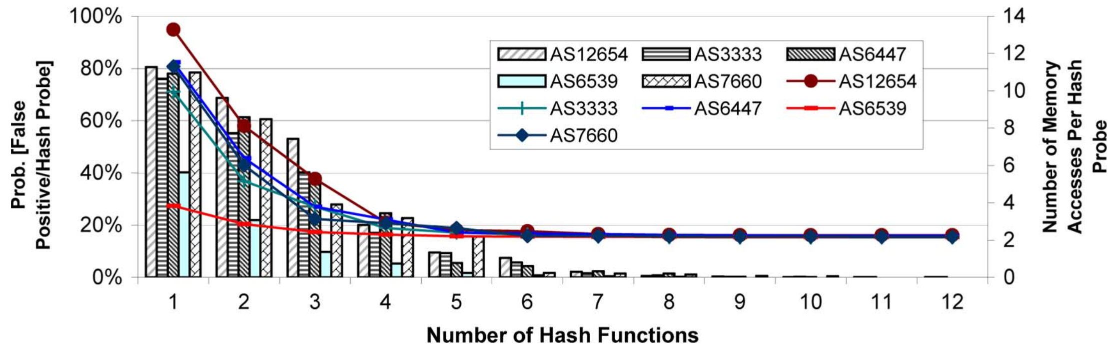


Fig. 15. Probability of false positives per hash probe and number of probes per lookup under CBF.

memory accesses per lookup approaches 2 asymptotically, versus an average of 1.07 (on-chip) accesses under $\mathrm{SUSE_{CBM}}$.

It is understood that the open hash table used in our simulations may not be optimized, and nearly collision-free hashing was shown to be achievable earlier [24], where it called for many hash functions (e.g., $k = 10$) and large Bloom filters (i.e., with their sizes equal to k times the number of items in set N). The methods considered in [24] also required complex algorithms to maneuver incremental updates such that the shortest search path to any route prefix appeared as the outcome for the next lookup. On the contrary, incremental updates are always straightforward for $\mathrm{SUSE_{CBM}}$, deemed as its another clear advantage.

VII. Size and Power Considerations for Hardware Implementation

A highly integrated chip design for SUSE is demonstrated in Fig. 16. As $\mathrm{SUSE_{CBM}}$ requires memory in the mega-byte range only, a whole routing table can be fit in on-chip SRAM blocks easily. For today's 90 nm ASIC technology, a 2048×41-bit building memory array takes an area of 122 114 um^2, as reported by Broadcom's memory compiler. For a provision of 256 K-prefix capacity, the memory block is roughly 16 mm^2. Power-wise, a building memory array has active and leakage power around 20 uW/MHz and 15 mW, respectively. Thus, assuming a 500 MHz clock, total dissipation for both static and dynamic power will be less than 2 W. If the more advanced 65-nm technology is used, the building block area drops to 76 174 um^2, and the memory block becomes $\sim$9.7 mm^2, consuming about 1.15 W at 500 MHz. It is worthwhile mentioning that these numbers include NHA information storage. For a design with NHA kept in off-chip memory, the on-chip memory size will then be cut down by 40%.

For being power-conscious, $\mathrm{SUSE_{CBM}}$ is compared with TCAM-based solutions. While a TCAM device allows parallel search of all entries, it has lower density than SRAM of an equal capacity. Importantly, power dissipation is often prohibitively high for TCAM. A conventional 4.5 Mb TCAM device consumes 7 W, assuming no power management, a supply voltage of 1.5 V, and operating frequency at 143 MHz [9]. An earlier study on general power-consumption for TCAM [1] reveals that it takes about 1.7 W per Mbits at 100 MHz frequency for the active power, equal to about 8.7 times that for SRAM of the same size.

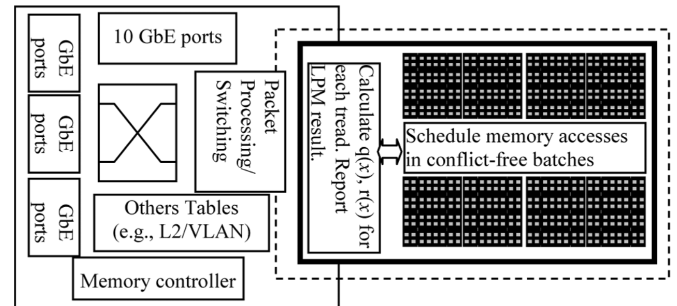


Fig. 16. Highly integrated switch chip with 8 on-chip memory modules.

To overcome high power dissipation, recent advances in TCAM technology break the memory into banks. Dynamic power-savings is achieved by activating selected banks only. However, one must know exactly the appropriate banks to activate [7] during accesses, e.g., following the methods introduced earlier [7] to partition and to distribute prefixes into TCAM banks. Specifically, the earlier methods involve two lookup stages, with the first stage being a trie-based structure or the selection of prefix bits and the second stage searching for the TCAM bank(s) indicated by the results of the first stage [7]. Therefore, the two-stage design incurs extra time overhead. In addition, TCAM entries have to be sorted for LPM. While a recent article [30] provided one possible solution for quickening updates to TCAM-based tables, incremental updates typically involve tricky operations in such tables and may require reallocation of TCAM banks due to overflows.

VIII. Conclusion

IP address lookups call for longest prefix matching (LPM) among those prefixes in a routing/forwarding table. There are four major parameters of interest in search for an ideal LPM solution, including small table storage, a low lookup latency, easy route updates, and low power dissipation. Motivated by inefficiency of existing solutions, we have proposed a design which excels in all four aspects for routing/forwarding tables, arriving at superior storage-efficiency, dubbed SUSE. With novel prefix transformation and controlled bit-maps (CBM) aggregation techniques, $\mathrm{SUSE_{CBM}}$ is shown by extensive simulation based on real prefix tables to save up to 85% of SRAM storage, when compared with prior designs. This significant storage reduction makes it possible to fit a large BGP table

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.

in on-chip SRAM. In contrast to off-chip RAM hampered by chip-crossing latency and humble bandwidth due to budget pin counts, on-chip SRAM can be accessed fast and be banked to realize parallel accesses. Thus, SUSE enjoys rapid lookups in support of LPM. Being hash table-based, it also empowers much speedier route updates than trie-based counterparts. In addition, on-chip SRAM has higher density and lower power dissipation than TCAM, carrying a far smaller price tag.

REFERENCES

[1] B. Agrawal and T. Sherwood, "Modeling TCAM power for next generation network devices," in *Proc. IEEE Int. Symp. Perf. Anal. Syst. Softw.*, Mar. 2006, pp. 120–129.

[2] "BGP table data," 2005 [Online]. Available: http://bgp.potaroo.net/index-bgp.html

[3] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: An efficient data structure for static support lookup tables," in *Proc. 15th Symp. Discrete Algor.*, Jan. 2004, pp. 30–39.

[4] B. Lampson *et al.*, "IP lookups using multiway and multicolumn search," *IEEE/ACM Trans. Netw.*, vol. 7, no. 3, pp. 324–334, Jun. 1999.

[5] D. R. Morrison, "PATRICIA—Practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.

[6] D. Shah and P. Gupta, "Fast updating algorithms for TCAMs," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, Jan./Feb. 2001.

[7] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-efficient TCAMs for forwarding engines," in *Proc. IEEE INFOCOM*, 2003, pp. 42–52.

[8] H. Lim, J.-H. Seo, and Y.-J. Jung, "High speed IP address lookup architecture using hashing," *IEEE Commun. Lett.*, vol. 7, no. 10, pp. 502–504, Oct. 2003.

[9] H. Noda *et al.*, "A cost-efficient high-performance dynamic TCAM with pipelined hierarchical searching and shift redundancy architecture," *IEEE J. Solid-State Circuits*, vol. 40, no. 1, pp. 245–253, Jan. 2005.

[10] J. Hasan *et al.*, "Chisel: A storage-efficient, collision-free hash-based network processing architecture," in *Proc. 33rd Int. Symp. Comput. Archit.*, May 2006, pp. 203–215.

[11] K. Sklower, "A tree-based routing table for Berkeley Unix," Univ. Calif., Berkeley, Tech. Rep., 1993.

[12] M. Degermark *et al.*, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, Sep. 1997, pp. 3–14.

[13] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Netw.*, vol. 15, no. 2, pp. 8–23, Mar./Apr. 2001.

[14] N.-F. Tzeng, "Routing table partitioning for speedy packet lookups in scalable routers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pp. 481–494, May 2006.

[15] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM*, Apr. 1998, vol. 3, pp. 1240–1247.

[16] R. Jain, "A comparison of hashing schemes for address lookup in computer networks," *IEEE Trans. Commun.*, vol. 40, no. 10, pp. 1570–1573, Oct. 1992.

[17] 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.

[18] 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.

[19] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," *Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, Feb. 1999.

[20] 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.

[21] Z. Wang, H. Che, M. Kumar, and S. Das, "CoPTUA: Consistent policy table update algorithm for TCAM without table lock," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1602–1628, Dec. 2004.

[22] M. Waldvogel *et al.*, "Scalable high speed IP routing lookups," in *Proc. ACM 2005 Conf. Appl., Technol., Arch., and Protocols for Comput. Commun. (SIGCOMM'97)*, Oct. 1997, pp. 25–36.

[23] L. Fan *et al.*, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[24] H. Song *et al.*, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *Proc. ACM 2005 Conf. Appl., Technol., Arch., and Protocols for Comput. Commun. (SIGCOMM'05)*, Aug. 2005, pp. 181–192.

[25] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *Proc. ACM 2003 Conf. Appl., Technol., Arch., and Protocols for Comput. Commun. (SIGCOMM'03)*, Aug. 2003, pp. 201–212.

[26] R. W. Keyes, "The wire-limited logic chip," *IEEE J. Solid-State Circuits*, vol. SC-17, no. 6, pp. 1232–1333, Dec. 1982.

[27] K. C. Saraswat and F. Mohammadi, "Effect of interconnection scaling on time delay of VLSI circuits," *IEEE Trans. Electron Devices*, vol. ED-29, no. 4, pp. 645–650, 1982.

[28] M. T. Bohr, "Interconnect scaling—The real limiter to high performance ULSI," in *Proc. Int. Electron Devices Meet. Tech. Dig.*, Dec. 1995, pp. 241–244.

[29] B. Bloom, "Space-time tradeoffs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[30] G. Wang and N.-F. Tzeng, "TCAM-based forwarding engine with minum independent prefix set (MIPS) for fast updating," in *Proc. 2006 IEEE ICC*, Jun. 2006, pp. 103–109.

[31] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proc. 9th ACM Symp. Theory Comput.*, May 1977, pp. 106–112.

[32] M. V. Ramakrishna and G. A. Portice, "Perfect hashing functions for hardware applications," in *Proc. 7th Int. Conf. Data Eng.*, Apr. 1991, pp. 464–470.

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

At present, he is with Broadcom Corporation, Santa Clara, CA, where he has developed the award-winning BCM1280/BCM1480 multicore SoCs used in many products in the telecomm, datacomm, and storage industry. His current project is a GPON device aiming to stream multimedia data at the gigabit speed via different interfaces, including GPON, ETH, Wireless, MoCA, and/or xDSL. Before Broadcom, he was with several startups, HP Labs, and Sun Microsystems, where he developed blade servers, network and storage protocols offload products, and multiprocessor systems. He has received 25 patents.

Dr. Pong is a Member of the IEEE Computer Society since 1992. He has also served on the National Science Foundation, the IETF RDMA Consortium, and program committees for several conferences.

Nian-Feng Tzeng (M'86–SM'92) 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, the University of Louisiana at Lafayette, since 1987. His current research interest is in the areas of computer communications and networks, high-performance computer systems, and parallel and distributed processing.

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

Authorized licensed use limited to: University of Louisiana at Lafayette. Downloaded on March 04,2010 at 11:31:21 EST from IEEE Xplore. Restrictions apply.