J. Parallel Distrib. Comput. 71 (2011) 1–15

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Advanced hashing schemes for packet forwarding using set associative memory architectures

Michel Hanna *, Socrates Demetriades, Sangyeun Cho, Rami Melhem

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA

ARTICLE INFO

Article history:
Received 22 December 2009
Received in revised form
12 October 2010
Accepted 12 October 2010
Available online 16 October 2010

Keywords:
Hardware hashing
Set associative memory
IP lookup
Packet forwarding
Hash schemes

ABSTRACT

Building a high performance IP packet forwarding (PF) engine remains a challenge due to increasingly stringent throughput requirements and the growing size of IP forwarding tables. The router has to match the incoming packet's IP address against all entries in the forwarding table. The matching process has to be done at increasingly higher wire speed; hence, scalability and low power consumption are critical for PF engines.

Various hash table based schemes have been considered for use in PF engines. Set associative memory can be used for hardware implementations of hash tables with the property that each bucket of a hash table can be searched in a single memory cycle. However, the classic hashing downsides, such as collisions and worst case memory access time have to be dealt with. While open addressing hash tables, in general, provide good average case search performance, their memory utilization and worst case performance can degrade quickly due to collisions (that lead to bucket overflows).

The two standard solutions to the overflow problem are either to use predefined probing (e.g., linear or quadratic probing) or to use multiple hash functions. This work presents two new simple hash schemes that extend both aforementioned solutions to tackle the overflow problem efficiently. The first scheme is a hash probing scheme that is called Content-based HAsh Probing (CHAP). As the name suggests, CHAP, based on the content of the hash table, avoids the classical side effects of predefined hash probing methods (i.e., primary and secondary clustering phenomena) and at the same time reduces the overflow. The second scheme, called Progressive Hashing (PH), is a general multiple hash scheme that reduces the overflow as well. The basic idea of PH is to split the prefixes into groups where each group is assigned one hash function, then reuse some hash functions in a progressive fashion to reduce the overflow. Both schemes are amenable to high-performance hardware implementations with low overflow and constant worst-case memory access time. We show by experimenting with real IP lookup tables and synthetic traces that both schemes outperform other existing hashing schemes.

1. Introduction

High speed routers require wire speed packet forwarding while the sizes of the IP tables across core routers are increasing at a very high rate [16]. IP address lookup has been a significant bottleneck for core routers. The advancement of optical networks made the situation even worse with link rates already beyond 40 Gbps. It is predicted that in the near future "Terabit" link rates will be available at affordable prices [43,17].

IP lookup proceeds as follows: the destination address of every incoming packet is matched against a large forwarding database (i.e., routing table) to determine the packet's next hop on its way to the final destination. An entry in the forwarding table (called a *prefix*) is a binary string of a certain length (*prefix length*), followed by don't care bits. The adoption of Classless Inter-Domain Routing resulted in the need for *longest prefix match* (LPM) in case of multiple matchings [34].

Existing IP forwarding engines are categorized into two main groups: hardware based and software based. The hardware based schemes are generally constrained by the size and power consumption of the engine. The software based schemes are mainly constrained by the throughput, measured as the number of lookups per second. Trie-based solutions are among the earliest IP lookup schemes [38,9,11,27]. The main idea is to build a trie (a tree-like structure) from the IP table. The problem with these schemes is the low throughput [2].

Recently, hash-based IP lookup techniques gained a lot of momentum. Hash tables come in two flavors: open addressing hash and closed addressing hash (or *chaining*) [4,23,39,5]. The hash table in a closed addressing hash has a fixed height (number of buckets), and each bucket is an unbounded linked list. During the lookup

* Corresponding author.
E-mail addresses: mhanna@cs.pitt.edu (M. Hanna), socrates@cs.pitt.edu (S. Demetriades), cho@cs.pitt.edu (S. Cho), melhem@cs.pitt.edu (R. Melhem).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.10.006

process, a specific row index is generated by the hash function and the row is searched to find the target key.

In open addressing, the hash table has a fixed height and a fixed bucket width (number of elements per bucket). Open addressing hash has a simpler table structure than closed addressing hash and is more amenable to hardware implementations. However, the issues of overflow and overflow handling have to be dealt with. Normally, the overflow is handled by means of probing or by using multiple hash functions [8].

The hardware schemes use special hardware such as Ternary Content Addressable Memory (TCAM) [24,28] to increase the lookup throughput. Unfortunately, the TCAM approach has its own set of limitations: high power consumption, poor scalability, and low bit density [26,30,37,47].

In this paper, we assume open addressing hash schemes for which a number of efficient hardware prototype implementations have been proposed recently [7,20,46]. In these implementations, the hash table is stored in a set associative memory where each set stores all the elements in a bucket and the buckets are indexed through the hash function.

Our goal is to fit an entire IP lookup table in a single fixed size hash table by using simple and efficient hash functions that could be easily implemented in hardware. The main challenge is to achieve maximum space utilization and minimum overflow. In addition, we want to keep both insertion/deletion into/from the table simple and straightforward. This work makes the following contributions to the area of open addressing hash:

- The introduction of the new concept of *content-based hash probing* (CHAP) which tackles the overflow more effectively than other existing probing techniques.
- The application of content-based probing to multiple hash function schemes.
- The introduction of the *progressive hashing* (PH) scheme for better space utilization and overflow reduction.
- The use of content-based probing and progressive hashing together to implement an efficient hardware-based IP lookup engine.

The rest of this paper is organized as follows. In Section 2 we give a brief background on open addressing hashing and the use of hashing in the presence of wildcards that play a major role in packet forwarding tables. Furthermore, we describe an example of the state-of-the-art set associative memory architecture in Section 2 as well. In Section 3 we describe CHAP, our first scheme. We discuss our second scheme Progressive Hashing in Section 4. Section 5 shows the experimental results of each scheme alone and the results of combining the two schemes. Finally, we give both the conclusions and future work in Section 7.

2. Background

2.1. General open addressing hash

Searchable data items, or records, contain two fields: key and data. Given a search key, k, the goal of searching is to find a record associated with k in the database. Hash achieves fast searching by providing a simple arithmetic function $h(\cdot)$ (hash function) on k so that the location of the associated record is directly determined. The memory containing the database can be viewed as a two-dimensional memory array of N rows with L records per row.

It is possible that two distinct keys $k_i \neq k_j$ hash to the same value: $h(k_i) = h(k_j)$. Such an occurrence is called a *collision*. A worst-case (pathological) which restrict the effectiveness of hashing is when all the keys are mapped to the same row. There are two solutions to solve the problem of collision in this case: (1) Make the row large enough to hold all the possible colliding prefixes at the cost of a large amount of wasted memory. The statistics in Section 2.2 shows that this is going to be fairly inefficient. (2) Control the row size and handle the overflow prefixes in a different way such as "probing" which we describe next.

When there are too many ($\geq L$) colliding records, some of these records must be placed elsewhere in the table by finding, or *probing*, an empty space in a bucket. For example in linear probing the probing sequence used to insert an element into a hash table is given as follows:

$$(h(k))\mathrm{mod}(N), (h(k)+\beta_0)\mathrm{mod}(N), \ldots, (h(k)+\beta_{m-1})\mathrm{mod}(N) \quad (1)$$

where each β_i is a constant, and m is the maximum number of probes. Linear probing is simple, but often suffers from what is called "primary key clustering" [8]. Another type of probing is called quadratic probing where we use a quadratic equation to determine the next bucket to be probed. The quadratic probing sequence used to insert an element into a hash table is generated by the following equation:

$$h(k,i) = (h'(k) + c_1 \times i + c_2 \times i^2)\mathrm{mod}(N), \quad i = 0, 1, \ldots m-1 \quad (2)$$

where $h'(\cdot)$ is called the auxiliary hash function and both c_1 and c_2 are constants. Quadratic probing suffers from another type of clustering which is called "secondary key clustering" [8].

Instead of probing, we can apply a second hash function to find an empty bucket, which is known as *double hashing* [8]. In general, the use of $H \geq 2$ hash functions is shown to be better in reducing the overflow than probing [1]. In this case (which we will refer to as *multiple hashing* in the rest of this paper) the probing sequence of inserting a key into the hash table is given as follows:

$$h_0(k), h_1(k), \ldots, h_{H-1}(k) \quad (3)$$

where H is the maximum number of hash functions. Most work that is done in the multiple hashing area is for closed addressing hash [1,44]. Note that using a different hash table for each hash function in Eq. (3) is a valid design option; however, using different hash tables leads to more overflow and hence results in poor space utilization [12,20,43]. Here is how we argue that using a single hash table is better than using multiple hash tables.

Proof. Consider the case where we have two identical hash tables, **A** and **B**, of size ($N \times L$), where N = number of rows and L = row width, and the case where we have an equivalent single hash table, **C**, of size ($N \times 2L$). Assume that "a" elements are mapped to row i of table A and "b" elements are mapped to row i of table B, then "c" $= (a + b)$ elements are mapped to row, i, of table C. The overflow is calculated for tables A, B and C, respectively, as follows

$$overflow_A = \max\{0, (a-L)\}$$
$$overflow_B = \max\{0, (b-L)\}$$
$$overflow_C = \max\{0, (c-2\times L)\}.$$

It is straightforward to show that if ($a > L$ and $b > L$) or ($a < L$ and $b < L$), then: $overflow_C = (overflow_A + overflow_B)$. if one of a or b is larger than L and the other is smaller than L then $overflow_C < (overflow_A + overflow_B)$. Specifically, if ($a = L + x$) and ($b = L - y$) for some integers $x, y > 0$, then ($overflow_A + overflow_B = x$) while ($overflow_C = 0$) or ($x - y$) when ($y < x$ or $y > x$), respectively. Thus, having more than one hash table case has more overflow than having a single hash table. □

To achieve high space utilization (the ratio between the required memory to store the database and the capacity of the actual RAM used) we apply multiple hash functions on a single hash table. Specifically, a key is inserted in the hash table using any of the H hash functions in Eq. (3).

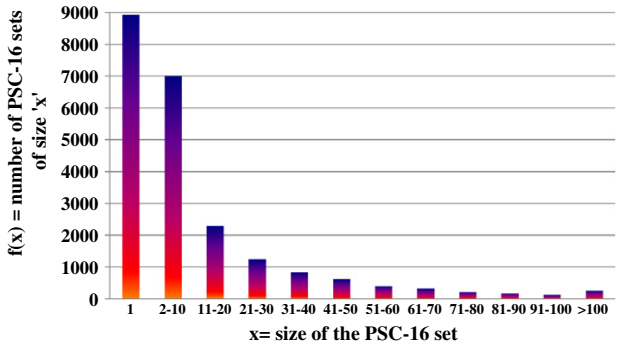


Fig. 1. Histogram of the prefixes sharing the first 16 bits.

Given a database of M records and an N-bucket hash table, the average number of hash table accesses to find a record is heavily affected by the choice of $h(\cdot)$, L (the number of slots per bucket), and α, or the *load factor*, defined as $M/(N \times L)$. With a smaller α, the average number of hash table accesses can be made smaller, however at the expense of more unused memory space, which leads to increasing both the power consumption and access time latency [2].

2.2. Hashing in the presence of wildcards

Applying hash functions in packet forwarding is very challenging due to the fact that wildcard, or don't care bits are heavily present in the IP lookup tables. Hashing with wildcards requires one of two solutions: restricted hashing or grouped hashing [14].

In restricted hashing, **RH**, the hash functions are restricted to use only the non-wildcard bits of the keys. For example, prefixes can be either expanded [41] to increase the number of non-wildcard bits or only a specific prefix length, that rarely includes wildcards, is used for hashing. In the latter case, the shorter prefixes are kept in a small fast memory [13,14]. The hashing scheme that we use in our first scheme, CHAP, restricts the hash functions to use only 16 bits to generate the hash indices, which will lead to a lot of collisions. Fig. 1 shows a histogram of how many prefixes share the most significant 16 bits for 15 real IP tables.[1] The number of prefixes that are shorter than 16 is less than 2% of the lookup table population and these are ignored in the figure.

To explain the figure, we define PSC-16 (Prefix Set with Common 16 bits) as a set that contains prefixes from the same IP forwarding table having an identical 16 first bits (sharing a common 16-bit prefix). The size of a PSC-16 set is the number of prefixes in that set. We then define $f(x)$ as the number of PSC-16 sets of size x averaged over the 15 tables and plot $f(x)$ for different ranges of x. For example, the point $(1, 8920)$ in the figure indicates that there are, on average, 8920 PSC-16 sets of size 1. In other words, there are, on average, 8920 unique prefixes per table. The next point, (2–10, 7000), indicates that there are on average 7000 PSC-16 sets per table, each containing between 2 and 10 prefixes that share the first 16 bits. The last point, $(>100, 247)$, is an aggregation of the PSC-16 sets that contain more than 100 prefixes.

The maximum size of a PSC-16 set is 1552 (table rrc04) with an average of 532 over the 15 tables. That is, on average, there may be as many as 532 prefixes per table having an identical first 16 bits. These prefixes will definitely collide if a single hash function is used. Using multiple hash functions alleviates that problem.

In grouped hashing, **GH**, prefixes are grouped based on their lengths, then different hash functions are applied to each group. For example, the 32 bit IPv4 wide address space can be split into 5 groups as follows:

[1] These tables' statistics are given in Table 1.

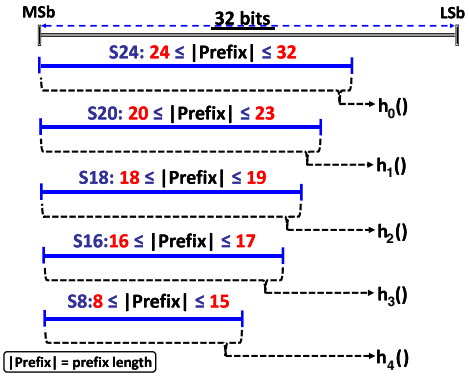


Fig. 2. Splitting the hashing space into groups. We represent the 32-bit address space with bold line and *MSb* and *LSb* stand for most significant bit and least significant bit, respectively.

- Group $S24$ that contains prefixes with at least 24 specific (non-wildcard) bits.
- Group $S20$ which contains prefixes of length between 20 and 23 bits.
- Group $S18$ which contains prefixes of length 18 and 19 bits.
- Group $S16$ which contains prefixes of length 16 and 17 bits.
- Group $S8$ which contains prefixes of length between 8 and 15 bits.

Then, each group is associated with a different hash function. For example, a hash function $h_0()$ that uses 24 bits can be associated with group $S24$, $h_1()$ that uses 20 bits can be associated with group $S20$, . . ., and $h_4()$ that uses 8 bits can be associated to group $S8$. This scheme is similar to the one used in [20]. Fig. 2(a) shows the five groups and their associated hash functions. The prefixes that are less than 8 bits long, which are fewer than 0.1% of the lookup table, are stored in a special buffer which we call "overflow_buffer" that is searched after failing the search of the main hash table. As a common practice, researchers proposed to use a small TCAM chip to store the overflow [3]. Based on the results we got in Section 5 we suggest to add a TCAM of size 8 to 20 kB to the main hash table to work as an overflow_buffer depending on which scheme is used.

Grouping the prefixes based on their lengths is not new. In fact, the IPStash architecture [20,21] utilizes a special case of the grouped hashing, where the number of groups is only three and the groups are called classes. Grouped hashing is used in Section 4 to derive the progressive hashing (PH) scheme, which is our second proposed scheme.

2.3. Set associative memory architecture overview

There are a few set associative memory architectures that are devised to work for IP forwarding [7,20,21,46]. In this section we use CA-RAM (Content Addressable-Random Access Memory) as a representative of set associative memory architectures proposed for IP lookup because of its flexibility [7,13,14]. We consider IP-Stash [20,21] architecture as the second best representative in this case. Both CA-RAM and IPStash are similar as they are set associative memory architectures. The main difference between the two architectures is that IPStash is a cache memory architecture; but CA-RAM is a more flexible and general memory architecture that is proven to work for other applications in addition to IP forwarding [7,14].

CA-RAM is a specialized, yet generic memory structure that is proposed to accelerate search operations. The basic idea of CA-RAM is simple; it implements the well-known hashing technique in hardware. It uses a conventional high-density memory (i.e., SRAM or DRAM) and a number of small match logic blocks to provide parallel search capability. Records are pre-classified and stored in

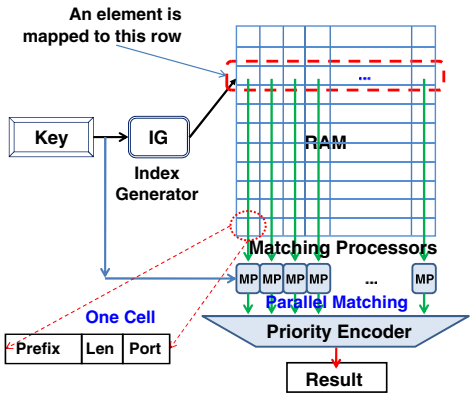


Fig. 3. The CA-RAM as an example of set associative memory architectures.

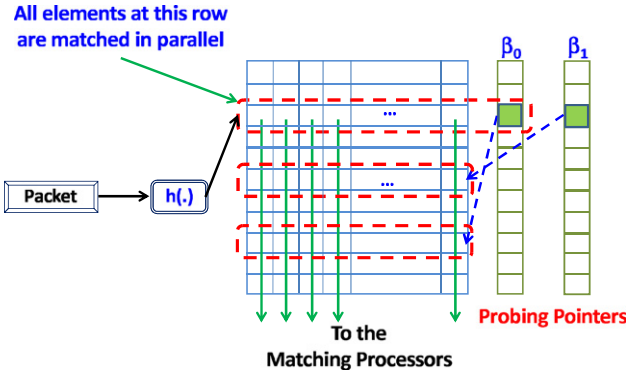


Fig. 4. The CHAP basic concept.

memory so that given a search key, access can be made accurately on the memory row having the target record. Each match logic block then extracts a record key from the fetched memory row, usually holding multiple candidate keys, and determines if the record key under consideration is matched with the given search key.

CA-RAM provides a row-wise search capability comparable to TCAM. More importantly, the bit-density of CA-RAM is much higher than that of TCAM, up to nearly five times higher if DRAM is used in the CA-RAM implementation [7]. A CA-RAM takes, as an input, a search key and outputs the result of a lookup. Its main components are: an index generator, a memory array (SRAM or DRAM), and match processors, as shown in Fig. 3.

The task of the index generator is to create an index from a key input. The actual function of the index generator will highly depend on the target application. In many applications, index generation is as simple as bit selection, incurring very little additional logic or delay. In other cases, simple arithmetic functions, such as addition or subtraction, may be necessary. Depending on the application requirements, a small degree of programmability in index generation can be implemented using a set of simple shift functions and multiplexers.

A row may be divided into entries of the form shown at the left corner of Fig. 3 where a CA-RAM entry (cell) stores a prefix, its length and the port number. Alternatively, two bits can be used to store a ternary digit to represent 0, 1 and don't care, rather than binary (like in TCAM arrays except that the comparison hardware in this case is shared among all the rows in the memory array). Optionally, each row can be augmented with an auxiliary field to provide information on the status of the associated bucket (e.g., how many keys are stored in this row). We use the auxiliary field in our two hashing schemes.

Once the index is generated from the input key, the memory array is accessed and L candidate keys are fetched simultaneously. The match processors then compare the candidate keys with the search key. A large area saving in CA-RAM comes from decoupling memory cells and match logic. Unlike conventional CAM where each individual row in the memory array is coupled with its own match logic, CA-RAM separates the dense memory array from the common match logic (i.e., match processors) completely. Since the match processors are simple and lightweight, the overall area cost of CA-RAM will be close to that of the memory array used. At the same time, by performing a number of candidate key matching operations in parallel, low-latency, constant-time search performance is achieved.

CA-RAM was compared against TCAM in terms of performance, power and area (cost). The result obtained in [7] shows that CA-RAM is over 26 times more power-efficient than the 16T SRAM-based TCAM [24], and over 7 times than the 6T dynamic TCAM [28]. The CA-RAM cell size is over 12× smaller than a 16T SRAM-based TCAM cell, and 4.8× smaller than a state-of-the-art 6T dynamic TCAM cell. Overall, CA-RAM is performance-competitive with TCAM, in terms of both search latency and bandwidth. The detailed area and power issues are addressed in [7].

3. Content-based hash probing

As we mentioned in the last section, a CA-RAM row stores the elements of a bucket and is accessed in one memory cycle. Because the CA-RAM architecture is very flexible, we may keep some bits at the end of each row for auxiliary data; this allows for more efficient probing schemes with multiple hash functions. In this section we first present the basic content-based hash probing scheme, **CHAP(1,m)**, which is a natural evolution of the linear probing scheme described by Eq. (1). We then extend this scheme to H hash functions, which we call **CHAP(H,m)**.

In open addressing hash, some rows may incur overflow while others have unoccupied space. While linear probing uses predetermined offsets to solve that problem as specified by Eq. (1), CHAP uses the same probing sequence, but with the constants $\beta_0, \beta_1, \ldots, \beta_m$ determined dynamically for each value of $h(k)$, depending on the distribution of the data stored in a particular hash table. Specifically, the probing sequence to insert a key "k" is:

$$h(k), \beta_0[h(k)], \beta_1[h(k)], \ldots, \beta_{m-1}[h(k)]. \quad (4)$$

This means that for each row we associate a group of m pointers to be used if overflow occurs to point to other rows that have space. We call these pointers "probing pointers" and the overall scheme is called CHAP(1,m) since it has one hash function and m probing pointers per row.

Fig. 4 shows the basic idea of CHAP when $m = 2$. In order to match the overflow excess keys to specific rows, we need to collect all the overflow elements across all the rows. We achieve this by counting the excess elements per row and finding for each row i two rows in which these overflow elements can fit. These two rows' indices are recorded in $\beta_0[i]$ and $\beta_1[i]$.

Assume that we are searching for a key k. If the hash function points to row $i = h(k)$ and it turns out that the input key k is not in this row, we check to see if the probing pointers at row i are defined or not. If defined, this means that there are other elements that belong to row i but reside in either row $\beta_0[i]$ or in row $\beta_1[i]$ and these elements might contain k. Consequently, rows $\beta_0[i]$ and $\beta_1[i]$ are accessed in subsequent memory cycles to find the matching key.

The content-based probing can also be applied to the multiple hashing scheme. Specifically, we refer to CHAP with H hash functions and m probing pointers by CHAP(H,m). For example, in **CHAP(H,H)** we have H hash functions and $m = H$ probing pointers. In this case, the probing sequence for inserting a key, k, can be defined by:

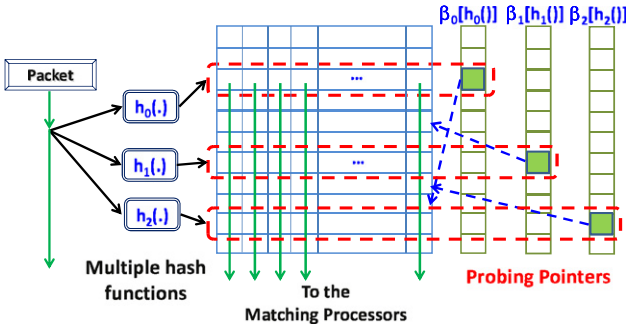


Fig. 5. The CHAP (3,3).

$$h_0(k), h_1(k), \ldots, h_{H-1}(k), \beta_0[h_0(k)], \beta_1[h_1(k)], \ldots, \beta_{m-1}[h_{H-1}(k)]. \quad (5)$$

In essence, we dedicate to each hash function a pointer per row. An example is shown in Fig. 5 for CHAP(3,3). In the example, this key will have six different buckets to which it can be allocated: $h_0(k)$, $h_1(k)$,$h_2(k)$, $\beta_0[h_0(k)]$, $\beta_1[h_1(k)]$ and $\beta_2[h_2(k)]$ in the given order, where $\beta_i[h_i(\cdot)]$ is the probing pointer of hash function $h_i(\cdot)$.

There are different ways to organize CHAP(H,m) when $m \neq H$ depending on whether or not the probing pointers are shared among the hash functions in a given row. In the example described above for CHAP(H,H), we assume that one probing pointer is associated with each hash function. Another organization could share probing pointers among hash functions. Yet a third organization could assign multiple pointers for each hash function, which is the only possible organization for CHAP(1,m), when $m > 1$. In the rest of this paper, we limit our discussion to CHAP(1,m) and CHAP(H,H) with one pointer for each hash function and with the probing order given by Eqs. (4) and (5) for the two organizations, respectively.

3.1. The CHAP(H,H) scheme

In this section we describe how to establish an IP lookup engine using CHAP(H,H). We present the setup algorithm that sets the probing pointers and maps actual IP prefixes into the CHAP hash table. With minor modifications, this algorithm can apply to the case of CHAP(1,m).

Before we describe the CHAP setup algorithm, we note that on average 98% of IP prefixes are 16 bits or longer [16]. In CHAP, we use restricted hashing (RH) where we restrict the hash functions to use only the most significant 16 bits. This means that prefixes shorter than 16 bits are not included in the hash table and they are stored as overflow in a separate memory (overflow_buffer). The overflow_buffer is used also to store the prefixes that cannot fit in the table during the setup algorithm as described in Section 3.2. In addition, the overflow_buffer is searched after a lookup failure in the main hash table [47,3,13,14].

3.2. The CHAP setup algorithm

Algorithm 1 lays out the setup phase of CHAP. In that algorithm, $j = 0, \ldots, M-1$ is used to index the prefixes, where M is the total number of prefixes in an IP routing table. The goal is to map this table into a hash table with $2^R = N$ rows, where R is the number of bits used to index the hash table. We use i as an index for hash functions and H as the maximum number of hash functions. An array of counters, $HC[\ ]$ of size N, is used to count the number of elements that will be mapped to each row of the hash table. We define a two dimensional array of counters $OC[\ ][\ ]$ of size $N \times H$ to count the overflow elements for each hash function per row. The maximum value of a single counter in this array is equal to λ, where $\lambda \leq L$, and L is the number of prefixes per row. This bound comes from the fact that a hole, or an empty space in any row of the hash table, can never exceed L. The CHAP setup phase determines if the configuration parameters of the hash table is valid or not. In other words, do the parameters L, H, λ and N result in a mapping of the M prefixes into a single hash table with acceptable overflow or not?

Algorithm 1 CHAP(H,H) Setup Algorithm.

```
1:  CHAP_Setup(IP forwarding table)
2:  Sort the IP prefixes from longest to shortest
3:  initialize the arrays HC[ ] and OC[ ][ ] to zeros
4:  table_overflow = number of prefixes shorter than 16 bits
5:  for(j = 0; j < M; j++) {
6:      inserted = false
7:      for(i = 0; i < H; i++) {
8:          r_i = h_i(k_j) }
9:      for (i = 0; i < H AND inserted == false; i++) {
10:         if(HC[r_i] < L), then {
11:             HC[r_i]++
12:             inserted = true }
13:     }
14:     for (i = 0; i < H AND inserted == false; i++) {
15:         if(OC[r_i][i] < λ), then {
16:             OC[r_i][i]++
17:             inserted = true }
18:     }
19:     table_overflow++
20: }
```

Algorithm 1 calculates the number of prefixes to be assigned to each row. By "assigned" we mean not only the prefixes that are hashed to this row, but also the overflow prefixes that are supposed to be in this row but will reside in other rows that are pointed to by this row's probing pointers. It starts by sorting prefixes from long to short, then initializing the two arrays HC and OC to zeros, while the table_overflow counter is initialized to the number of prefixes that are less than 16 bits long (lines 1–2). Sorting the prefixes helps to stop at the first matching prefix as will be proved in Section 3.3. The set of hash values $\{r_0, \ldots, r_{H-1}\}$ for each prefix is calculated (lines 6–7). Then, the algorithm updates the counter HC as follows: if there is a spot for the current prefix in HC then the algorithm will move on to the next prefix (lines 8–11), if not, it increments the corresponding OC counter (lines 12–15).

When Algorithm 1 exits, table_overflow will include the number of prefixes that could not fit in either HC or OC (lines 16–17) in addition to the number of prefixes that are shorter than 16 bits long. If that number is not acceptable, then the algorithm can be repeated with more hash functions, that is with a new $H' = H + 1$. In that setting, the acceptability of the overflow depends on the capacity of the overflow_buffer. The progressive hashing scheme discussed in Section 4 may be applied in conjunction with CHAP to further reduce the overflow.

3.2.1. The mapping of IP prefixes in CHAP

The last step in CHAP is to allocate the elements into the hash table using hash functions and probing pointers. Before moving to the actual mapping of the prefixes, we need to assign values to the probing pointer's array. This is done by running the best fit algorithm [33]. The algorithm starts by finding the largest counter value from the OC array, say $OC[T][I]$, and the smallest counter value from HC, say $HC[J]$. We say that row 'J' has a *hole* of size "$L - HC[J]$". Then the Ith probing pointer of row T is assigned the value of J, the row having the largest hole provided that the hole size is larger than $OC[T][I]$. This process is repeated iteratively.

Clearly, the best fit algorithm may not find a hole for each overflow counter, which means that some keys will not be able to

fit in the hash table. The number of these keys are added to table_overflow, and again, if the resulting overflow is not acceptable, then Algorithm 1 has to be re-executed with a larger value of H. After setting the probing pointers, the prefixes are mapped to the hash table.

3.3. Search in CHAP

As discussed in Section 2.3, a read operation fetches a full row (bucket) from the hash table into a buffer and uses a set of comparators to determine, in parallel, the longest prefix match among the elements in that bucket. A complete search might need to search more than one bucket. Hence, a metric that will be used to measure the efficiency of the search in CHAP is the Average Memory Access Time, **AMAT**, which is simply the average number of rows accessed for successful search.

The CHAP search algorithm, Algorithm 2, is straightforward. We call our main hash table "**H_Table[ ][ ]**", of size $N \times L$ where N and L are the number of rows and the row capacity respectively. Each element in H_Table[ ][ ] consists of the actual prefix, H_Table[ ][ ].key, and the prefix length, H_Table[ ][ ].len which is used to determine the LPM. Given a packet P, we calculate the row address $r_i(P) = h_i(P)$ and $r_{i+H} = \beta_i[h_i(P)]$, where $i = 0, \ldots, H-1$ (lines 1–3).

For each row of the $2H$ rows, we match the packet against all the prefixes in this row in parallel and if we hit at this row, we return the port number associated with the matched prefix (lines 4–6). If we do not find a match in these rows, we simply search the overflow_buffer (line 8).

Algorithm 2 The CHAP Search Algorithm.

```
Search_Hash_Table(Packet P) {
1:    for(i = 0 ; i < H ; i++) {
2:        r_i = h_i(P)
3:        r_{i+H} = β_i[h_i(P)] }
4:    for(i = 0 ; i < 2H ; i++) {
5:        if(P matches H_Table[r_i][j].key),
          then { /* done in parallel for all values of j */
6:                return H_Table[r_i][j].port }
7:    }
8:    search the overflow_buffer }
```

To be able to stop at the first matching prefix during search in the CHAP's search algorithm, Algorithm 2, we store the prefixes according to their length from the longest to the shortest [13]. In addition to sorting the prefixes during the insertion, we have to maintain what is called the "hash order" during both insertion and search phases. The hash order is merely the order of applying the hash functions in addition to the order of accessing the probing pointers. Theorem 1 proves that these two conditions are enough to find the LPM first.

Theorem 1. *In CHAP, the first matching prefix is the LPM if:*

1. *The prefixes are inserted from the longest to the shortest.*
2. *The search's hash order, which includes both the order of accessing the probing pointers and the order of applying the hash functions, is the same as the insertion's hash order.*

Proof. In a restricted multi hashing scheme all the H hash functions are applied to all keys. Let us assume that we have M keys to be hashed and that they are sorted according to their length from the longest to the shortest. Also, assume that the hash order during the insertion is as follows: $r_0(k_m), \ldots, r_{2H-1}(k_m)$, where $r_i(k_m) = h_i(k_m)$ for $i = 0, \ldots, H-1$ and $r_i(k_m) = \beta_i[h_i(k_m)]$ for $i = H, \ldots, 2H-1$. In addition, assume that there exists a packet P_X that matches two prefixes k_X and k_Y and that k_X is longer than k_Y. This means that k_X is mapped to the hash table before k_Y.

Without losing the generality, assume that $r_t(k_X) = r_t(k_Y) = r_t$. We can see that it is impossible for k_Y to find a space in row r_t if k_X could not find a space. This means that if k_X is stored in row $r_i(k_X) = r_X$ and if k_Y is stored in row $r_j(k_Y) = r_Y$, then $i < j$. Hence, while searching for a match for P_X as follows: $r_0(P_X), \ldots, r_{2H-1}(P_X)$, we will match k_X at row r_X before matching k_Y at row r_Y. □

Note that if both prefixes k_X and k_Y in Theorem 1 are mapped to the same row, the matching processors will determine the LPM in this case.

3.4. Incremental updates in CHAP

An important issue in the IP forwarding engine is the incremental updates of the prefix database. The number of prefixes included in a routing table grows with time [16,43]. The updates consist of two basic operations, *Insert/Update* and *Delete* a prefix. In CHAP the delete operation is straightforward. For any prefix deletion operation we find the prefix first, then we delete it by storing all zeros and then decrement the row counter HC which is used to keep track of the rows' populations. Deleting a prefix from any row does not require shifting since the matching processors will ignore the deleted prefix spot as it will contain all zeros.

The basic idea of the insert/update operation, which is detailed in Algorithm 3, is to find the appropriate row that the new prefix should fit in, taking into account the LPM feature. In other words, we need to find where the new prefix should be stored according to its length to achieve LPM. If it is found that the prefix already exists in the CHAP table, the existing entry will be updated.

Algorithm 3 CHAP Insert Update Algorithm.

```
0: define r_i as an integer array of 2 × H elements
1:CHAP_Insert_Update (prefix k_n) {
2:for(i = 0; i < H; i++) {
3:     r_i = h_i(k_n)
4:     r_{i+H} = β_i[h_i(k_n)] }
5:By searching the rows r_0, ··· r_{2H-1}, find: {
6:     k_l = longest prefix matching k_n and l = index of row
containing k_l
7:     k_s = shortest prefix matching k_n and s = index of row
containing k_s }
8:if(k_l is not defined AND k_s is not defined ), then {
9:    return(Insert_in_Rows(k_n, 0 ,(2H − 1))) }
     /* insert k_n in any of rows r_0, . . . , r_{2H-1} */
10:else if ((|k_n| == |k_l|) OR (|k_n| == |k_s|)), then {
11:      Replace k_l or k_s with k_n /*an update operation*/
12:      return (true)}
13:else if(|k_n| > |k_l|), then { return(Insert_in_Rows(k_n, 0, l))}
14:else if(|k_n| < |k_s|), then { return(Insert_in_Rows(k_n, s, (2H −
1))) }
15:else, return(Insert_in_Rows(k_n, l, s))
}

16:Insert_in_Rows (prefix k_x, a, b) {
/* insert k_x in any of the rows r_a, r_{a+1} · · · , r_b */
17:for(i = a; i <= b; i++) {
18:    if(HC[r_i] < L), then {
19:         insert k_x in r_i and HC[r_i]++
20:         return (true)}
}
21:return (false) }
```

Algorithm 3 consists of two boolean functions, **CHAP_Insert_Update()** and **Insert_in_Rows()**. The first function, CHAP_Insert_Update(), determines the appropriate rows to insert the new prefix

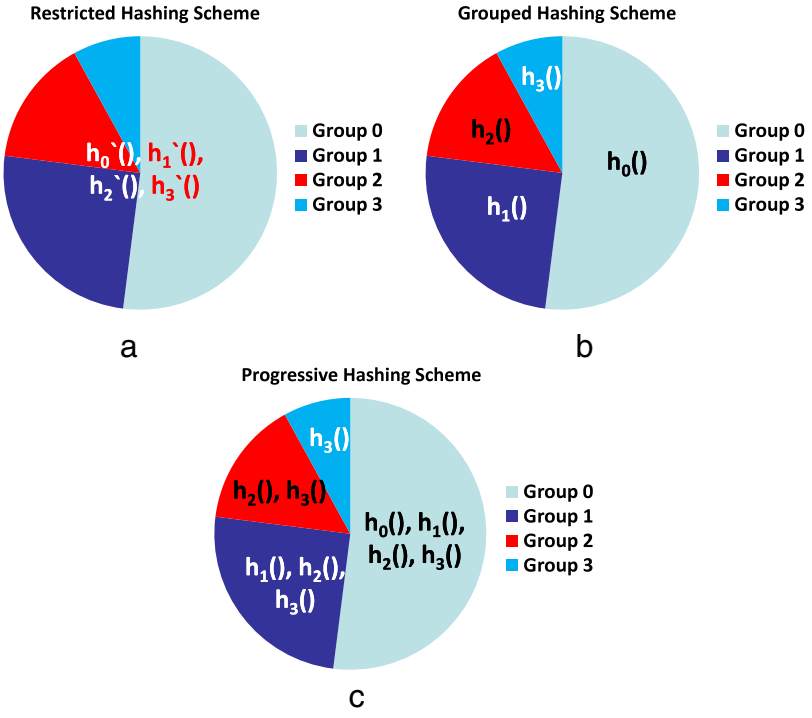


Fig. 6. The evolution of the PH scheme.

k_n (lines 16–21). The second function is where the actual insertion is made, as it take a prefix k_x then it tries to insert it in a series of rows starting from row index a all the way to row index b.

In the first function, the row array r_i, of size $2H$, is used to store the computed values of the hash functions of k_n and the corresponding probing pointers (lines 2–4). Note that r_i is a global variable because it is accessed by the second function, Insert_in_Rows(). For each row r_i we match k_n against all the prefixes in this row and extract both the longest prefix, k_l, and the shortest prefix, k_s, that match k_n (lines 5–7). We record the rows indices l and s that include k_l and k_s if such matchings are found. Depending on the length of k_n relative to the length of both k_l and k_s, we try to insert k_n in one of the $2H$ rows. This is done through an *if-else* construct (lines 8–15). The first case is when neither k_l nor k_s are defined (i.e., no matching), thus we can insert k_n into any row (lines 8–9). The second case, which is route update [16], is when k_n is equal either k_l or k_s in which case we replace either k_l or k_s with k_n (lines 10–12). The third case is if $|k_n|$[2] is larger than $|k_l|$, then we try to insert k_n into one of the buckets $\{r_0, \ldots, r_l\}$ if there is a space (line 13). In the next case we check to see if $|k_n| < |k_s|$ is true, then we try to insert k_n in a row among $\{r_s, \ldots, r_{2H-1}\}$ (line 14). Finally, if $|k_s| < |k_n| < |k_l|$, then we try to put k_n in any row between r_l and r_s (line 15).

In any case, the functions terminate successfully if we are able to insert k_n. Otherwise, we try to either insert k_n into the overflow_buffer, or use a backtracking scheme like "Cuckoo hashing" [29] to replace an existing prefix, say k_y, from the hash table by k_n, then try to recursively reinsert k_y back to the hash table [12].

The implementation of the incremental updates algorithm is done in the control plane (which contains a network processor to preform the necessary compactions). We propose that the actual updates are issued as a special case during the idle time of the CA-RAM.

[2] Throughout this paper, we use the notation $|k|$ to represent the length of prefix 'k'.

4. The progressive hashing scheme

In this section, we propose the Progressive Hashing scheme (PH) as another effective mechanism for reducing collisions (hence overflow) for open-addressing hash systems. As we mentioned in Section 2.1, using multiple hash functions is efficient in reducing collisions. In Section 2.1 we described the two multiple hashing schemes for dealing with don't care bits, which are abstracted in Fig. 6(a) and (b) where the hashing space is represented as a circle. In the restricted hashing scheme (Fig. 6(a)) the hash functions $h'_0(), \ldots, h'_3()$ are applied to all the keys in the hashing space. On the other hand, in the grouped hashing (Fig. 6(b)) we split the hashing space into groups and a single hash function is associated with each group. In Fig. 6(b), functions $h_0(), \ldots, h_3()$ are associated with groups $0, \ldots, 3$ respectively.

In this section we group the prefixes based on their lengths (i.e., use grouped hashing or GH). Consequently, groups with a longer prefix length can use the hash functions of other groups that have shorter prefix lengths. For example, in Fig. 2, group $S24$ can use the hash functions of groups $S20$ and $S16$. Motivated by this observation, we propose to apply the hash functions in a progressive manner as illustrated in Fig. 6(c) to give some keys more chances to be mapped to the hash table thus reducing the overflow.

The effectiveness of progressive hashing depends mainly on how we select the groups and their associated hash functions. One important aspect during the grouping of the keys is to maintain "hashing specificity hierarchy", where "hash function specificity" is defined as follows:

Definition 1. A hash function $h_i(\cdot)$ is said to be more specific than another hash function $h_j(\cdot)$ if any bit used in $h_j(\cdot)$ is also used in $h_i(\cdot)$.

For example, in Fig. 2, the hash function $h_0(\cdot)$ is more specific than $h_1(\cdot)$, $h_2(\cdot)$, $h_3(\cdot)$ and than $h_4(\cdot)$. Fig. 7 demonstrate the PH scheme applied to the same groups of Fig. 2. As an example, group $S24$, which is assigned to hash function $h_0(\cdot)$, can use the less

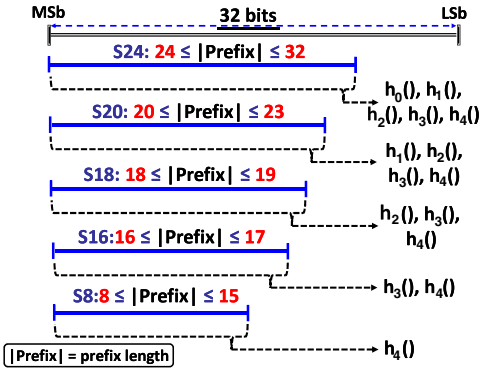


Fig. 7. Applying the PH scheme.

specific hash functions of groups $S20, S18, S16$ and $S8$ as illustrated in Fig. 7.

In the next two sections we show the PH setup and search algorithms.

4.1. The PH setup algorithm

In this section we introduce the PH setup algorithm, Algorithm 4. Before dividing the prefixes into groups, we sort the prefixes from longest to shortest and insert them in that order. In Algorithm 4, $j = 0, \ldots, M-1$ is used to index the keys, where M is the total number of prefixes in an IP routing table. The goal is to map the prefixes into a hash table, H_Table[ ][ ] of size $N \times L$, where $L =$ maximum bucket size, $N = 2^R$ maximum number of rows and R is the maximum number of bits used to index the hash table. Each entry in H_Table[ ][ ] contains the field "*key*" which consists of the actual prefix, the prefix length (or mask), the prefix port number and the hash function field (lines 9–12). The hash function index is used to store the index of the hash function that is used to store the prefix. In the next section, Section 4.2, we show the importance of this field. H is the maximum number of hash functions and an array of counters, $HC[]$ of size N, is used to count the number of elements that are mapped to each row of the hash table. A counter, *table_overflow*, that records the number of overflow elements is initialized by the number of prefixes that are shorter than 8 bits long. Group number 'i' is represented by $\mathfrak{G}_i$.

Algorithm 4 The PH Setup Algorithm.

```
PH_Setup(IP forwarding table){
1: Sort the IP prefixes from longest to shortest and define the groups
2: Initialize HC[] array to zeros and table_overflow = number of prefixes shorter than 8 bits
3: for(j = 0; j < M; j++){
4:     inserted = false
5:     for(i = 0 ; i < H ; i++){
6:         if (k_j ∈ G_i), then {
7:             r_i = h_i(k_j)
8:             if(HC[r_i] < L), then {
9:                 H_Table[r_i][HC[r_i]].key = k_j
10:                H_Table[r_i][HC[r_i]].len = |k_j|
11:                H_Table[r_i][HC[r_i]].port = k_j port number
12:                H_Table[r_i][HC[r_i]].h = i
13:                HC[r_i]++, inserted = true }
14:            }
15:    }
16:    if(inserted == false), then
17:        Store k_j in overflow_buffer, table_overflow++ } }
```

Algorithm 4 attempts to allocate k_j, (line 6) in the hash table, if the attempt is not successful, it stores the key in the overflow_buffer that is searched after the main hash table. Note that we apply the hash functions according to their specificity starting from the most specific to the least specific during the insertion.

4.2. Searching in PH

In this section we show how to find the LPM in the PH scheme. The goal for any given packet is to find its longest prefix matching. But since we might find multiple matches, we want to guarantee that the first prefix that matches a packet is its LPM. Unfortunately, Theorem 1 cannot be used for PH as some prefixes have a different insertion's hash order than their search's hash order. For example, if a packet P matches two prefixes $k_X \in (S18)$ and $k_Y \in (S16)$ in Fig. 7(a), then k_X is the LPM of P. Assume that during the prefixes mapping, both prefixes are stored in two different rows as follows: $h_2(k_X) = r_X$ and $h_3(k_Y) = r_Y$. During the search for P we try all the five hash functions $r_0 = h_0(P), \ldots, r_4 = h_4(P)$. Assume that one of the hash functions that was not used to store either k_X or k_Y generates the row r_Y when it is applied to P, i.e., $r_0 = r_Y$ or $r_1 = r_Y$. This means that we search r_Y before r_X, thus, we report k_Y as the LPM instead of k_X, which is wrong.

To solve this problem, the hash function that was used to insert k_Y has to be checked. In this case it turns out that k_Y was stored using $h_3()$ and not $h_0()$. Hence, k_Y has to be skipped as a matching because there is a better matching, which is k_X in this case. This is why we store the hash function index in the PH setup algorithm, Algorithm 4, (line 12).

Algorithm 5 The PH Search Algorithm.

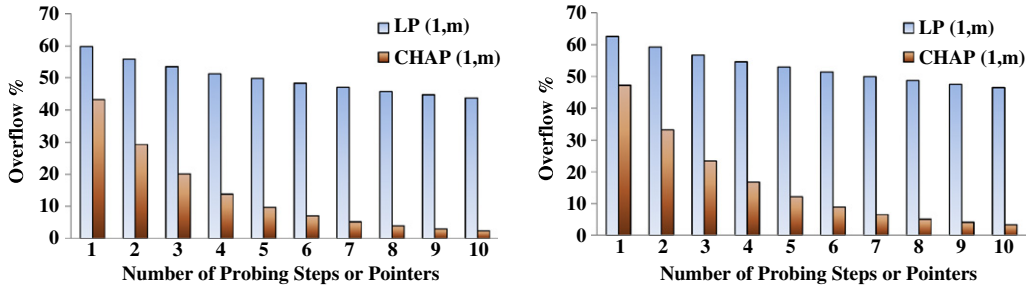
```
Search_Hash_Table(Packet P){
1:   for(i = 0 ; i < H ; i++){
2:     r_i = h_i(P)
3:     if((P matches H_Table[r_i][j].key) AND (i == H_Table[r_i][j].h))
,then /* done in parallel for all values of j */
4:       return H_Table[r_i][j].port
5:     }
6:     search the overflow_buffer }
```

The PH search algorithm is given in Algorithm 5. It works as follows: for each packet P that arrives at the packet processing unit, we calculate the row index addresses $r_0 = h_0(P), \ldots, r_{H-1} = h_{H-1}(P)$ (lines 1 and 2). For each row r_i we match P against all the elements in that row in parallel in a single clock cycle using the matching processors (line 4). The matching processors return the LPM in the bucket if and only if the stored hash function index ".h" is identical to the hash function index that is used to lookup the prefix during the search (line 4). If we do not find any match, then we search the overflow_buffer (line 6).

4.3. Incremental updates in PH

Deleting a certain prefix is straightforward in PH. It involves locating this prefix, deleting it by storing all zeros in its place and adjusting the corresponding HC row counter. The insert/update operation for the PH scheme is similar to that of the CHAP scheme that is given in Algorithm 3 except that the rows r_i are not defined for $i = H, \ldots, 2H-1$. Also, we use only the hash functions that are applicable to the prefix being inserted. Specifically, we replace lines 1–3 in Algorithm 3 with the following lines:

```
2: for(i = 0 ; i < H ; i++){
3:     if (k_n ∈ G_i), then
4:         r_i = h_i(k_n) }
```



(a) $\{L = 200, N = 1024\}$. (b) $\{L = 100, N = 2048\}$.

Fig. 8. Overflow of CHAP(1,m) vs. Linear Probing(1,m) for table rrc07.

Table 1
The statistics of the IP lookup tables on January 31st 2009.

IP table name	Size	% Short prefixes
rrc00 (Amsterdam)	292,717	0.78
rrc01 (London)	276,224	0.82
rrc02 (Paris)	272,743	0.79
rrc03 (Amsterdam)	283,147	0.80
rrc04 (Geneva)	283,075	0.81
rrc05 (Vienna)	301,383	0.77
rrc06 (Otemachi)	277,555	0.81
rrc07 (Stockholm)	274,479	0.83
rrc10 (Milan)	276,912	0.82
rrc11 (New York)	275,903	0.82
rrc12 (Frankfurt)	277,132	0.82
rrc13 (Moscow)	280,961	0.81
rrc14 (Palo Alto)	274,824	0.82
rrc15 (Sao Paulo)	275,828	0.82
rrc16 (Miami)	280,744	0.81
Average	280,242	0.81

After that, we decide the bucket that should store the new prefix, k_n, as we did in Algorithm 3. To summarize, Algorithm 3 can be used as an insert/update algorithm for PH except for the aforementioned modifications.

5. Evaluation

We used C++ to build our own simulation environment. This environment allows us to choose and arrange different types of hash functions. The hash functions used in our experiments are from three different hashing families: bit-selecting, CRC-based, and H_3 [32] hashing families. Those families have the advantage of being simple and fast enough to be easily realized in hardware. Note that in all our experiments, we choose by trial and error the hash functions that give the lowest overflow percentages for our schemes as well as the rival schemes we are comparing against.

For the evaluation, we collected 15 tables from the Border Gateway Protocol (BGP) Internet core routers of the routing information service project [35] on January 31st 2009. Table 1 lists the 15 routing tables, their sizes, and the percentage of "short prefixes", that are shorter than 16 bits long. To measure the average search time, we generate uniformly distributed synthetic traces using the same tables.

We define a "configuration" by specifying both $N =$ the number of rows, and $L =$ the number of entries per row. The performance of the CHAP and PH schemes, in terms of both overflow percentage and AMAT, depends on the number of hash functions, H, and on the load factor (space utilization) $\alpha = M/(N \times L)$ where M is the database size and $(N \times L)$ is the hash table size. For a given α, the hashing overflow depends on the aspect ratio of the memory N/L.

In Section 5.1 we evaluate CHAP and in Section 5.2 we evaluate PH. Finally, Section 5.4 evaluates the combined scheme of both PH and CHAP.

5.1. The evaluation of content-based hash probing

For a given hardware implementation N and L are fixed and the performance of CHAP depends on two important parameters, namely the maximum overflow value of the OC counters, λ, and the number of hash functions used, H, which is also the number of probing pointers per row in CHAP(H,H). Intuitively, if λ is small, then the setup algorithm (Algorithm 1) may not be able to eliminate the overflow. On the other hand, if λ is large, then Algorithm 1 may terminate with every OC having a value smaller than λ, but the best fit algorithm may not find holes that are large enough in the table to accommodate the values of the OC, thus increasing the overall overflow of the hash table. In Section 5.1.2, we study the sensitivity of CHAP(H,H) against λ.

5.1.1. The advantages of content-based hash probing

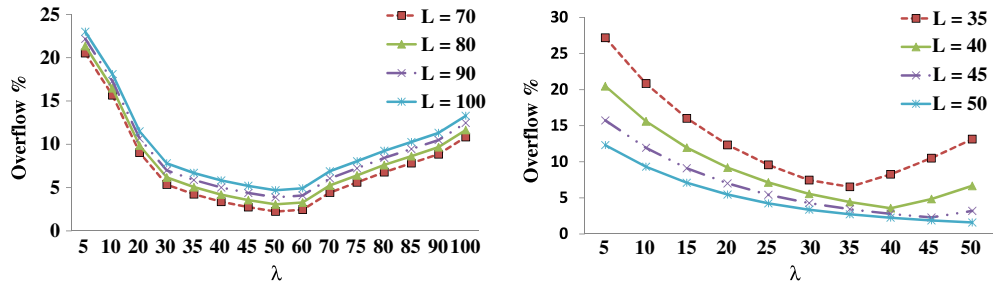
In order to show the advantage of content-based probing over linear probing, we compare the overflow generated by both CHAP (1,m) and linear probing (that has the same number of probing steps) when mapping routing tables to hash tables with specific configurations (that is with specific L and N). We use the table "rrc07" and two different configurations: $\{L = 200, N = 1024\}$ and $\{L = 100, N = 2048\}$. We tried many different configurations and they all led to results similar to those shown in Fig. 8. In addition, these two configurations have a high average load factor $\alpha =$ 98.5% for the "rrc07" table, which articulates the strength of CHAP.

Fig. 8 shows that for the same number of probing steps, overflow in CHAP(1,m) is less than that in linear probing. In fact, CHAP achieves 72.4% more overflow reduction than linear probing on average. Moreover, we can see that the longer the probing sequence, the more effective is CHAP in eliminating overflow compared to linear probing. The main reason behind this is that CHAP is addressing the overflow problem directly by choosing empty, or partially empty, buckets to reallocate the overflow elements. This is in contrast to linear probing which blindly tries to put the overflow elements in the nearest available bucket which may not be found within m probes.

5.1.2. Sensitivity analysis of CHAP (H,H)

In this section we study the effect of varying λ in the CHAP setup algorithm (Section 3.2). We only report the results for table "rrc07" since all other tables give similar results. We show the results for two different groups of configurations where each group has 4 configurations. In one group we use $N = 2^{12} = 4096$ rows, and in the other $N = 2^{13} = 8192$ rows. In these groups we use $H = 3$.

Fig. 9(a) and (b) show the values of overflow versus λ for the two groups. For Fig. 9(a), we set $L = 70$, 80, 90 and 100 entry per row for $N = 4096$ rows, which results in $\alpha =$ 94.9%, 83.1%, 73.8% and 66.5% respectively. As for Fig. 9(b), we set $L =$ 35, 40, 45 and 50 entry per row for $N = 8192$ rows, which results in the same loading factors. Note that $\lambda \in [0, L]$.



(a) $N = 4096$. (b) $N = 8192$.

Fig. 9. The overflow vs. λ.

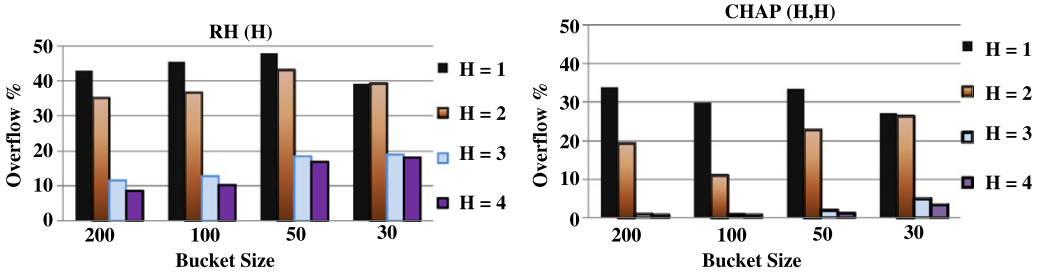


Fig. 10. Average overflow of 4 bucket widths for RH(H) and CHAP(H,H) for table rrc05.

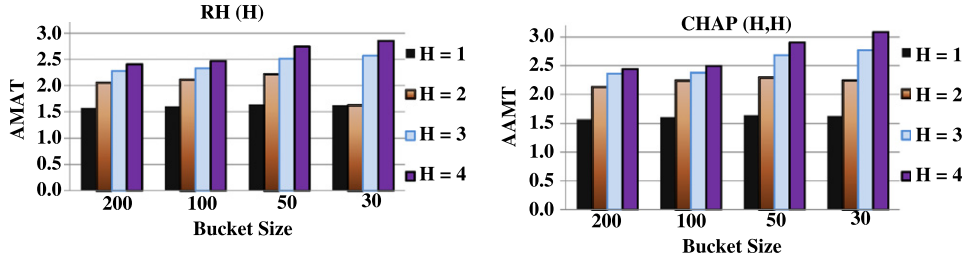


Fig. 11. Average AMAT of 4 bucket widths for RH(H) and CHAP(H,H) for table rrc05.

As the figures indicate, the overflow starts at some non-zero value and then decreases in the range $0 < \lambda < \frac{L}{2}$. At $\lambda = \frac{L}{2}$ the overflow is almost zero in Fig. 9(a), which is an indication that the average hole size in the hash table is equal to $\frac{L}{2}$. For larger values of λ, the maximum hole size becomes smaller than λ and thus we are unable to insert all the elements that were counted by *OC* into the hash table. This increases the overflow. In Fig. 9(b) we notice that the overflow is smaller when $\lambda = L$. This happens because the bucket (row) size is small and we have a large number of buckets and a lot of them are almost empty. This is expected since there is low entropy (randomness) among the prefixes in the lookup tables, which leads to a lot of empty spaces in the hash table. In the following section we use $\lambda = \frac{L}{2}$ for bucket sizes larger than 50 and $\lambda = L$ for smaller bucket sizes.

5.1.3. CHAP(H,H) versus restricted hashing(H)

In this section we compare the CHAP(H,H) scheme against the restricted hashing scheme (or **RH(H)**), where H is the number of hash functions used. We compare the two schemes in terms of the AMAT (Average Memory Access Time) and the overflow. In this experiment we use the routing table "rrc05" since it has the largest number of entries among other tables.

In Fig. 10 we show the average values of overflow for different number of hash functions (between 1 and 4) and for four different configurations. We set $N = 2048$, 4096, 8192 and 16,384 where $L = 200$, 100, 50 and 30, respectively for CHAP. On the other hand, we set the same N values for RH(H) but with $L = 201$, 102, 52 and 32 to compensate for the overhead of storing the CHAP probing pointers and row counters (which are around 5 bytes). From this point on in this work, we compensate the RH, GH and PH schemes with more keys per row when compared against the CHAP scheme. It is obvious from Fig. 10 that CHAP(H,H) has much less overflow than multiple hashing for the same number of hash functions. On average CHAP(H,H) overflow is 48.7% lower than RH(H) over all four bucket sizes.

The results shown in Fig. 11 indicate that the average AMAT over the four bucket sizes for RH(H) is 2.16, while it is 2.28 for CHAP(H,H) which is only 5% higher than RH. We note here that both CHAP and RH have a separate memory (overflow_buffer) to accommodate the overflow prefixes which is searched after the main hash table (CA-RAM) is searched. Therefore, the worst case search time for CHAP(H,H) and RH(H) are $2H + 1$ and $H + 1$, respectively. Although the difference between the two schemes seems large in terms of the worst case memory access time (WMAT), we have to take into consideration that at $H = 3$ the overflow of CHAP is almost zero (less than 1%) for the bucket sizes of $L = 200$, 100 and 50, while it is less than 5% for $L = 30$. Thus adding more hash functions only makes the average memory access time worse. A classical trade-off between the overflow and the AMAT can be seen in Figs. 10 and 11. However, a better understanding of the trade-off that CHAP and RH present can be obtained by comparing CHAP(H,H) with RH(2H) since both have $2H$ as the maximum number of table accesses (i.e., WMAT).

5.1.4. CHAP(H,H) versus restricted hashing(2H)

In order to show that CHAP(H,H) can achieve both low overflow and AMAT compared to RH(2H), we plot in Fig. 12(a) the overflow

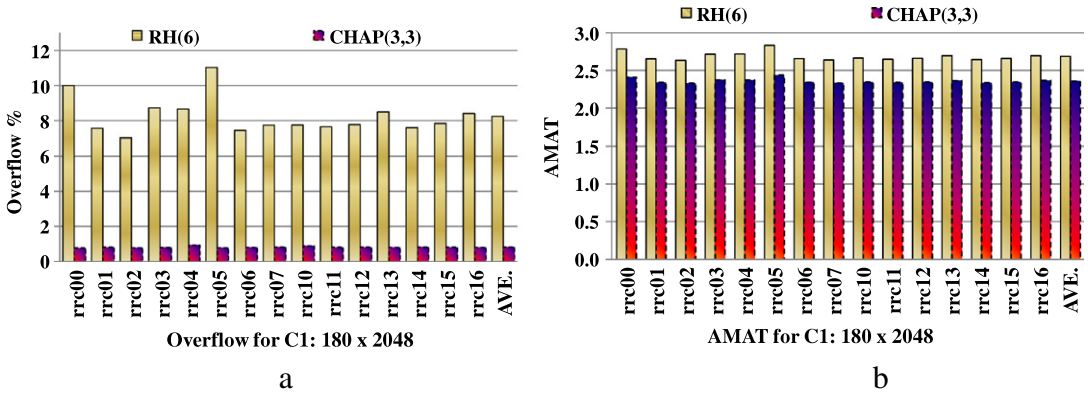


Fig. 12. (a) Average overflow and (b) AMAT for CHAP(3,3) vs. RH(6) for 15 lookup tables for C1: $\{L = 180, N = 2048\}$.

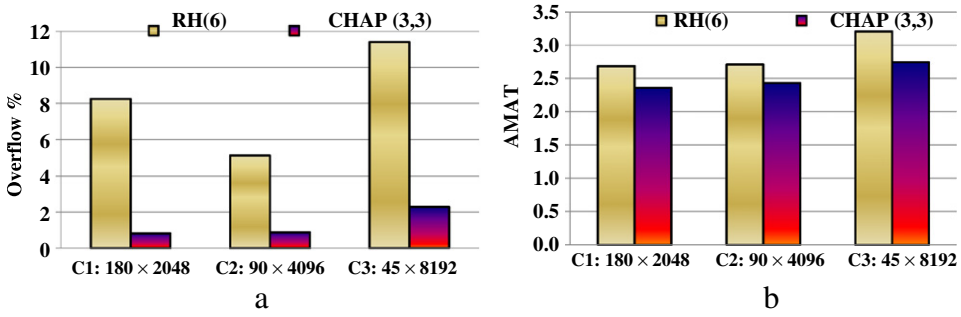


Fig. 13. (a) Average overflow and (b) Average AMAT for CHAP(3,3) vs. RH(6) for 3 configurations.

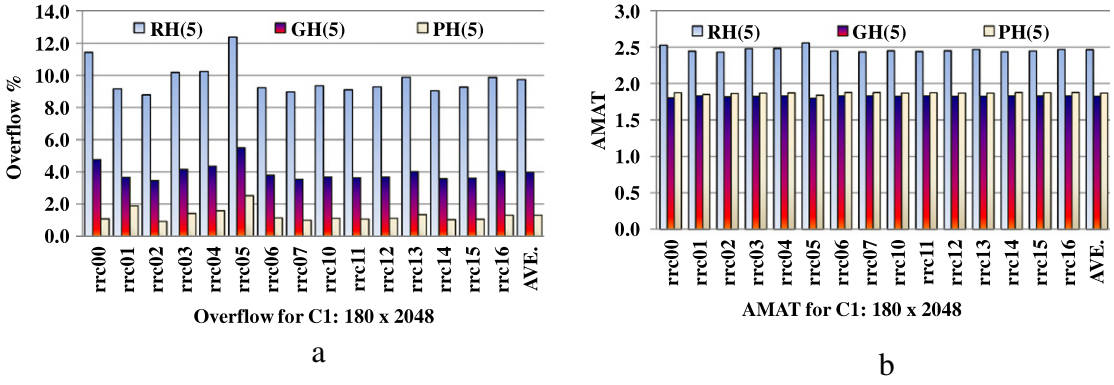


Fig. 14. (a) Average overflow (b) AMAT of RH(5) vs. GH(5) vs. PH(5) for 15 lookup tables for C1 : {180 × 2048}.

and (b) the average memory access time of both schemes for one configuration C1 : $\{L = 180, N = 2048\}$. We increased the bucket size by one prefix for RH(2H) to compensate for the CHAP overhead as we discussed previously. For this experiment we map each of the 15 IP tables into a fixed hash table of 368,640 entries.

Note that we represent the average of all files as another independent point that is called AVE in Fig. 12. As we can see, CHAP(3,3) is better than RH(6) for all files in terms of both the AMAT and the overflow. In fact CHAP(3,3) reduces the overflow by 90% and at the same time improves the AMAT by 12.2% for this configuration.

To evaluate CHAP performance under other configurations, we use three different configurations: C1 : $\{L = 180, N = 2048\}$, C2 : $\{L = 90, N = 4096\}$ and C3 : $\{L = 45, N = 8192\}$ in Fig. 13(a) and (b). Again, we set $L = 181$, 92 and 47 for RH(6). Fig. 13(a) shows the average overflow over all 15 lookup tables for RH(6) and CHAP(3,3). These three configurations have the same average load factor of 76.0% which is considerably high. Fig. 13(b) shows the AMAT of the same three configurations for RH(6) and CHAP(3,3).

For the three configurations, CHAP(3,3) reduces the overflow by 90%, 82.9% and 79.7% respectively over RH(6). At the same time, CHAP(3,3) improves the AMAT over RH(6) by 12.2%, 10.2% and 14.4% respectively.

5.2. The evaluation of progressive hashing

In this section we compare PH against grouped hashing (GH) and restricted hashing (RH) each using 5 hash functions. For RH(5), all 5 hash functions use the most significant 16 bits and are applied to all the prefixes in the lookup tables. Those prefixes that are less than 16 bits long are inserted in the overflow_buffer. On the other hand, we split the 32 bits IPv4 address space according to Fig. 2 for both GH(5) and PH(5).

Fig. 14(a) shows the overflow percentage, which is the ratio of the overflow to the total number of prefixes of a routing table. We show results for all 15 routing tables for one configuration C1 : $\{L = 180, N = 2048\}$, for the three schemes: RH, GH and PH. On average, PH reduces the overflow by 86.5% compared to RH and by 66.9% compared to GH. At the same time, the AMAT (Fig. 14(b))

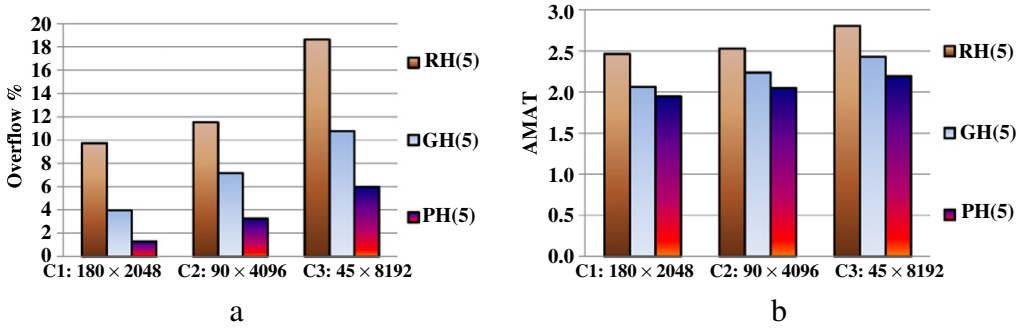


Fig. 15. (a) Average overflow and (b) Average AMAT for of RH(5) vs. GH(5) vs. PH(5) for 3 configurations.

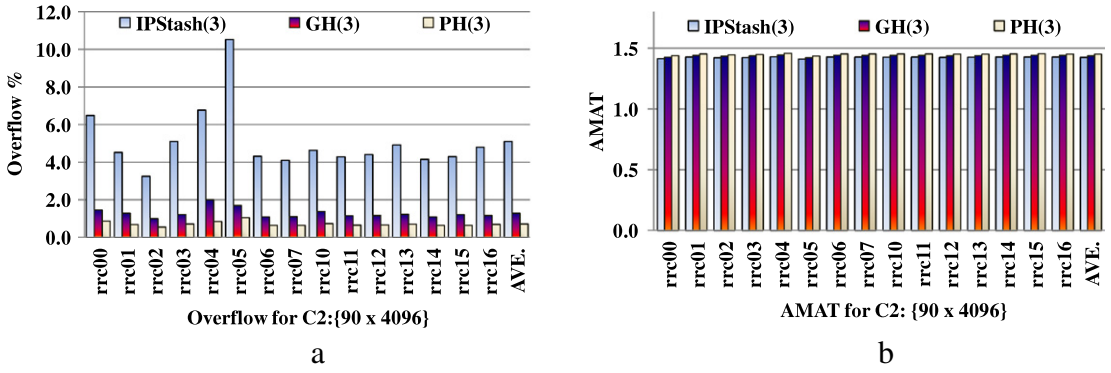


Fig. 16. (a) Average overflow and (b) Average AMAT of IPStash(3) vs. GH(3) vs. PH(3) for C2 : {90 × 4096}.

of PH is improved by 22.0% over RH and 3.4% over GH. Note that the overflow prefixes are added to the overflow_buffer that is searched after exhausting all possible buckets in the main hash table.

To show that PH is robust under other configurations, we use the same three configurations that we used before: C1 : $\{L = 180, N = 2048\}$, C2 : $\{L = 90, N = 4096\}$ and C3 : $\{L = 45, N = 8192\}$ in Fig. 15(a) and (b). Fig. 15(a) shows the average overflow over all the 15 lookup tables for the three schemes RH, GH and PH. Fig. 15(b) shows the AMAT for the same configurations and schemes.

PH has the lowest overflow percentage among the three schemes. The average (over the three configurations) overflow reduction percentages of PH is 55.2% compared to GH, while it is 75.3% compared to RH. PH improves the AMAT by 20.5% and 7.9% compared to RH and GH respectively.

5.3. Progressive hashing v.s. IPStash

In this section we evaluate our progressive hashing scheme against IPStash [21] which is another open address hashing scheme. In Section 2.3 we mentioned that IPStash architecture is similar to the CA-RAM architecture and that IPStash uses a special form of the grouped hashing scheme as it classifies the prefixes into only three groups according to their lengths and uses only 12 bits for hash table indexing. The main difference is that IPStash use the controlled prefix expansion [41] (CPE) to expand prefixes of length from 8 to 15 bits to at least 16 bits then choose any 12 bits to index the hash table [21], while GH does not utilize any prefix expansion.

We implemented the IPStash scheme and compared it against the GH and our PH schemes. Since IPStash uses only 12 bits for indexing, we have to use configuration C2 : {90 × 4096}. Fig. 16(a) shows that IPStash has an average overflow of 5% while GH and PH reduce the overflow by 75% and 85%. The high overflow percentage is directly due to the CPE technique used by IPStash. The AMAT is depicted in Fig. 16(b). The AMAT of the three schemes is almost identical. Note that in case we could not find the prefix in the main hash table we search the overflow_buffer thus we have a maximum of 4 memory accesses as a worst case.

5.4. Applying content-based hash probing to progressive hashing

In this section we combine both proposed schemes PH and CHAP into a third scheme that we call **PH_CHAP(H,H)**. As described before, CHAP is using restricted hashing scheme where the hash functions uses only the most significant 16 bits of all the prefixes. However, we use PH instead of RH to get better performance than both schemes.

In Fig. 17(a) we show the average overflow of CHAP(5,5), PH(5) and PH_CHAP(5,5) for the same three configurations we used in Sections 5.1 and 5.2. The largest average overflow belongs to PH(5) with 3.38% and the lowest average overflow is 0.3% for PH_CHAP(5,5) over the 3 configurations. The first two configurations, C1 and C2, have zero overflow for PH_CHAP(5,5) with a reduction of 100%. For the third configuration, C3, PH_CHAP(5,5) reduced the overflow by 86.9% over PH(5) and by 23.6% over CHAP(5,5).

At the same time, we note that PH_CHAP(5,5) has a lower AMAT (Fig. 17(b)) than CHAP(5,5) and PH(5) with an average of 19.7% improvement and 2.3%. The improvement in the AMAT comes from the fact that the PH_CHAP(5,5) uses PH to reduce the overflow in addition to the probing pointers, in contrast to CHAP(5,5) that relies only on its probing pointers to reduce the overflow since CHAP uses hash functions that are restricted to use only 16 bits, thus, increasing its AMAT. This is why the largest average AMAT over the 3 configurations belongs to CHAP(5,5) though it has the second lowest overflow percentage among the three schemes.

5.5. Memory overhead of CHAP and PH

In this section we estimate the memory overhead of the PH(5), CHAP(5,5) and PH_CHAP(5,5) packet forwarding schemes. Note

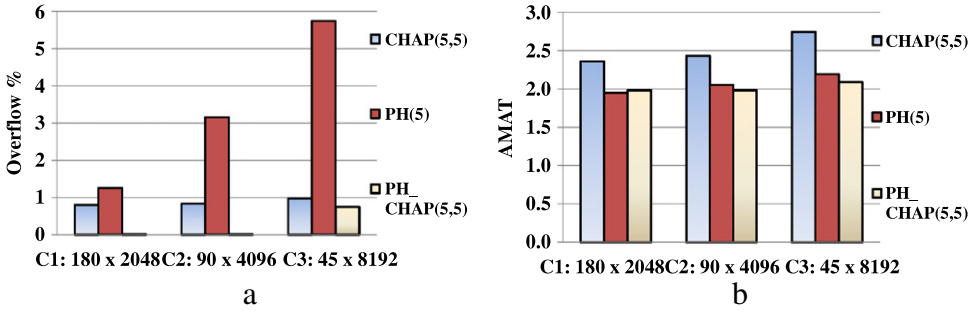


Fig. 17. (a) Average overflow and (b) Average AMAT of CHAP(5,5) vs. PH(5) vs. PH_CHAP(5,5) for 3 configurations.

that this estimation does not include a 8 to 20 kB of TCAM that is used to accommodate the overflow_buffer. We estimate the memory requirements for one configuration C1 : $\{L = 180, N = 2048\}$ as an example for the three schemes. In general, C1 requires $180 \times 2048 = 368{,}640$ or 370 K entry, where an entry is represented by 4 bytes prefix plus 5 bits prefix length. 1 byte per row is used for the row counter.

For PH(5), we need 3 bits per entry for the hash function index. Thus, the total memory requirement for PH(5) for the configuration C1 is ∼2.12 MB. For CHAP(5,5) we add 5 pointers per row where each pointer is represented by 12 bits. Note that we will use the auxiliary field of the CA-RAM that is at the end of each row, as we mentioned in Section 2.3, to store the pointers and the row counters. Thus, the total memory requirement for CHAP(5,5) is ∼2.00 MB for the configuration C1. For the same C1 configuration, the combined PH_CHAP(5,5) scheme needs ∼2.13 MB.

Finally, we estimate the actual size of the over_flow buffer. The maximum overflow, belonging to CHAP for the rrc04 file, equals 0.93% + 0.81% (short prefixes) = 4926 prefixes. Each prefix takes 4 bytes to be stored in a TCAM; thus the maximum overflow_buffer size is 19.7 kB ≃ 20 kB. The maximum overflow_buffer size for the PH_CHAP(5,5) hybrid scheme is ≃8 kB for configuration C3 : $\{45 \times 8192\}$, while it is zero for both C1 and C2.

5.6. Performance estimation of CHAP and PH

We estimate both the average and the minimum throughput of our hybrid scheme "PH_CHAP(H,H)" using two different SRAM memory architectures. Since our scheme depends on a set associative RAM, we conservatively assume that the clock rate is halved. This will take care of the delay for the hash functions computation and the matching logic. The first memory architecture that we use is the 500 MHz QDR III SRAM [31]. This is the latest architecture among the famous Quad Data Rate (QDR) SRAM family which is to be commercially available soon. The QDR is an SRAM that can transfer up to four words of data in each clock cycle [31]. The other memory architecture is a state-of-the-art CMOS technology SRAM memory design [45] which reports an experimental single chip of 36.375 MB that runs at 4.0 GHz.

Assuming an AMAT of ∼2.0 (according to Fig. 15(b)), if the clock rates are 250 MHz and 2.0 GHz for the two architectures, then we have a forwarding throughput of 125 mega packets per seconds and 1.0 giga packets per seconds. In other words, 40 Gbps and 320 Gbps for the minimum packet size of 40 bytes.

Beside the AMAT we use another throughput metric which is the Worst case Memory Access Time (WMAT). The WMAT of the hybrid PH_CHAP(5,5) scheme is $10 + 1 = 11$ ($2 \times H +$ searching the overflow_buffer). Thus, the worst case throughput is slightly less than 20% of the AMAT throughput. Note that all these estimated rates are for a single CA-RAM chip. In order to increase both throughputs (AMAT-based and WMAT-based) we can use multiple CA-RAM chips per line card as described in [19,18]. In addition, a typical router has multiple line cards [6]. The aggregate throughput of a router is calculated as the sum of throughputs of the line cards.

5.6.1. Performance estimation using CACTI

In addition to the above estimations, we use the standard "CACTI" (version 5.3) cache simulator [42] to estimate the throughput in addition to the area and the power requirements of our PH-CHAP(5,5) scheme. We assume a total memory of 2.25 MB (actually CACTI takes only the total amount of memory as a power of 2) and that the row width is 512 bytes. Since we propose to build our schemes as high-performance hardware ASIC chip; we use the High Performance International Technology Roadmap for Semiconductors SRAM 45 nm technology model (ITRS-HP). CACTI gives us a 2.12 ns access time with a 0.77 ns cycle time. In other words, CACTI estimates that this chip has 3 pipeline stages each runs on 1.3 GHz frequency, which translates to a throughput of 1.3 giga packets per second or 416 Gbps for the minimum packet size of 40 bytes.

In addition to the time and frequency information, CACTI provides area and power information. The total dynamic power consumed per read port (we assumed only one read port and one write port) is 4.9 W at the maximum frequency, while the total dynamic energy per read port is 3.8 nJ. The total area, excluding the matching processors, is estimated to be 33.5 mm^2. The authors in [7] estimated that the matching processors overhead is less than 10% of the RAM space, thus the total area comes to 36.9 mm^2.

6. Related work

The IP lookup is a mature and well-researched problem. However, it remains a challenging problem in the networking area. Various techniques have been proposed to achieve high-performance IP lookup. In this section we discuss some of these techniques.

As we mentioned in Section 1, there are two families of schemes for the IP lookup problem: software-based and hardware-based. Most software solutions are based on tries. A trie is a tree-based data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching [43]. Many schemes are devised in this area [38,9,11,27]. The main advantages of these solutions are that they provide straightforward time and space bounds but with low throughput [2]. In addition, most of these early works are built on compressing the trie which saves the space hence the power [38,9,27], but on the other hand makes the incremental updates very hard. Newer versions of trie-based solutions that are based on pipelining have been recently proposed [10,2,25,19,18,17,22,36]. These solutions are elegant pipeline hardware architectures that use smart caching techniques as well as multiple pipelines [19,18,17,22] to boost the throughput. The average number of pipeline stages varies between 20 and 25 stages, where each stage consumes between 20 and 40 kB.

Another software-based family of schemes that are used to solve the IP lookup is hash-based. Hashing is a well known software technique that can be adapted in hardware [8,32]. Most of the work done in this area is adapting closed addressing hashing [8]. Since

chaining is used to solve the collision problem, an important design goal in this case is to minimize efficiently the worst-case length of the linked lists and to balance the bucket population by using Bloom filter [4] data structures as done in [23,39,15,40]. A Bloom filter is an efficient data structure for membership queries with tunable false positive errors [4]. Hence, the problem with these kind of solutions is to control the false positive probability. M. Mitzenmacher et al., proposed another elegant family of IP lookup tables that uses multiple hash tables where each table has its own hash function [5,23]. The original seminal scheme is called "d-left" scheme which is very efficient in reducing the hash collisions. The authors also introduced a full analysis to this scheme. The only problem with using multiple tables is memory fragmentation.

Using Ternary Content Addressable Memories (TCAM) is the defacto standard in the industry [43]. A TCAM cell is capable of storing 3 logic states (0, 1 and wildcard). Upon receiving a packet, the TCAM searches the entire chip in parallel for the LPM [26]. Thus providing the right answer in a single memory cycle. This high degree of parallelism comes at the cost of storage density, access time, and power consumption [39,7]. Moreover, most commodity TCAMs run at low speed compared to SRAM memory [17,7]. Many researchers proposed optimizations to the TCAM architecture trying to accommodate these disadvantages [26,30,37,47].

Our work is unique in the sense that we use an open addressing hash-based architecture. Hence, it is not appropriate to compare our schemes against trie-based schemes or other closed addressing hash-based schemes. We only compare with them indirectly with respect to memory size (hence power) and throughput.

7. Conclusions and future work

In this paper we describe and study two different hash-based schemes for IP forwarding: Content-based HAsh Probing (CHAP) and Progressive Hashing (PH). The schemes solve the overflow problem by utilizing content-based probing and multiple hash functions, respectively, and have small average memory access times. We also illustrate that both schemes can be realized in hardware by taking advantage of set associative memory architectures. In this work we use simple hash functions that can be easily realized in hardware. We provide setup and incremental update algorithms for both schemes.

Simulation results show that CHAP is superior compared to linear probing in terms of overflow elimination. CHAP achieves 71.61% more overflow reduction than linear probing on average. The results also show that CHAP improves the average memory access time over the restricted multiple hash function scheme while reducing the overflow.

While we introduce PH as a new open addressing hash-based packet processing scheme, it can also work for closed addressing hash systems. PH is effective in reducing the overflow on average by 55.2% compared to grouped hashing (IPStash [20]), while it is 75.3% compared to restricted hashing. We show that our schemes can have an average forwarding speed of 320 Gbps if using future SRAM technology and 40 Gbps with the standard QDR III SRAM. Both our schemes use less than 2.5 MB of RAM. The CACTI memory simulator shows that such architecture could achieve a throughput of 416 Gbps while maintaining a moderate area and power requirements.

Our future work includes applying both schemes to other packet processing applications such as Packet Classification (PC). In addition, we plan to introduce optimizations to reduce the worst case memory access time of both CHAP and PH schemes. We want to study a fully synthesized PH_CHAP(H,H) packet forwarding engine.

Finally, the authors recognize that the schemes presented are general and can be applied to many other applications that use hashing. In this paper we showed that both schemes give good results for the IP application. We will consider other applications as well in the future work.

References

[1] Y. Azar, A. Broder, A. Karlin, E. Upfal, Balanced allocations, SIAM J. Comput. 29 (1) (2000) 180–200.
[2] F. Baboescu, D.M. Tullse, G. Rosu, S. Singh, A tree based router search engine architecture with single port memories, Sigarch Comput. Archit. News 33 (2) (2005) 123–133.
[3] A. Basu, G. Narlikar, Fast incremental updates for pipelined forwarding engines, IEEE Infocom (July) (2003) 64–74.
[4] B. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[5] A. Broder, M. Mitzenmacher, Using multiple hash functions to improve IP lookups, IEEE Infocom (April) (2001) 1454–1463.
[6] H.J. Chao, B. Liu, High Performance Switches and Routers, 1st ed., Wiley-IEEE Press, 2007.
[7] S. Cho, J. Martin, M. Hammoud, R. Melhem, CA-RAM: a high-performance memory substrate for search-intensive applications, IEEE ISPASS (April) (2007) 230–241.
[8] T. Cormen, C. Leiserson, R. Rivest, C. Stien, Introduction to Algorithms, McGraw Hill, 2003.
[9] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, ACM Sigcomm (September) (1997) 3–14.
[10] L. Devroye, Efficient construction of multibit tries for IP lookup, IEEE/ACM Trans. on Net. (TON) 11 (August) (2003) 650–662.
[11] P. Gupta, S. Lin, N. Mckeown, Routing lookups in hardware at memory access speeds, IEEE Infocom (April) (1998) 1240–1247.
[12] M. Hanna, S. Demetriades, S. Cho, R. Melhem, An efficient hardware-based multi-hash scheme for high speed IP lookup, IEEE HOTi (August) (2008) 103–110.
[13] M. Hanna, S. Demetriades, S. Cho, R. Melhem, CHAP: enabling efficient hardware-based multiple hash schemes for IP lookup, IFIP Netw. (May) (2009) 756–769.
[14] M. Hanna, S. Demetriades, S. Cho, R. Melhem, Progressive hashing for packet processing using set associative memory, IEEE/ACM ANCS (October) (2009) 103–112.
[15] J. Hasan, S. Cadambi, V. Jakkula, S. Chakradhar, Chisel: a storage-efficient, collision-free hash-based network processing architecture, IEEE ISCA (May) (2006) 203–215.
[16] G. Huston, Analyzing the internet bgp routing table, The Internet Protocol J. 4 (1) (2001).
[17] W. Jiang, V. Prasanna, A Memory-Balanced Linear Pipeline Architecture for Trie-Based IP Lookup, 2007, pp. 83–90.
[18] W. Jiang, V. Prasanna, Multi-Terabit IP Lookup Using Parallel Bidirectional Pipelines, 2008, pp. 241–250.
[19] W. Jiang, V. Prasanna, Reducing dynamic power dissipation in pipelined forwarding engines, IEEE ICCD (October) (2009) 144–149.
[20] S. Kaxiras, G. Keramidas, IPSTASH: a power-efficient memory architecture for IP-lookup, IEEE Micro (November) (2003) 361–373.
[21] S. Kaxiras, G. Keramidas, IPSTASH: a set-associative memory approach for efficient IP-lookup, IEEE Infocom (March) (2005) 992–1001.
[22] K. Kim, S. Sahni, Efficient construction of pipelined multibit-trie router-tables, IEEE Trans. Comput. 56 (1) (2007) 32–43.
[23] A. Kirsch, M. Mitzenmacher, Simple summaries for hashing with multiple choices, IEEE/ACM Trans. Netw. 16 (1) (2008) 218–231.
[24] H. Noda, K. Inoue, H.J. Mattauschtt, T. Koide, K. Arimoto, A cost-efficient dynamic Ternary CAM in 130 nm cmos technology with planar complementary capacitors and tsr architecture, IEEE Symp. VLSI Circuits (June) (2003) 83–84.
[25] S. Kumar, M. Becchi, P. Crowley, J. Turner, CAMP: fast and efficient IP lookup architecture, ACM/IEEE ANCS (October) (2006) 51–60.
[26] K. Lakshminarayanan, A. Rangarajan, S. Venkatachary, Algorithms for advanced packet classification with Ternary CAMs, ACM Sigcomm (May) (2005) 193–204.
[27] S. Nilsson, G. Karlsson, IP-Address Lookup Using LC-Tries, vol. 17, 1999, pp. 1083–1092.
[28] H. Noda, et al., A cost-efficient high-performance dynamic TCAM with pipelined hierarchical searching and shift redundancy architecture, IEEE J. Solid-State Circuits 40 (1) (2005) 245–253.
[29] R. Pagh, F. Rodler, Cuckoo hashing, in: Lec. Notes in Comp. Sci. (LNCS), vol. 2161, 2001, pp. 121–133.
[30] R. Panigrahy, S. Sharma, Reducing TCAM power consumption and increasing throughput, HOTi'02 (August) (2002) 107–112.
[31] M. Pearson, Qdrtmiii: Next generation SRAM for networking. http://www.qdrconsortium.org/.
[32] M. Ramakrishna, E. Fu, E. Bahcekapili, Efficient hardware hashing functions for high performance computers, IEEE Trans. Comput. 46 (12) (1997) 1378–1381.
[33] B. Randell, A note on storage fragmentation and program segmentation, Commun. ACM 12 (July) (1969) 365–372.
[34] Y. Rekhter, T. Li, An Architecure for IP address allocation with CIDR, RFC 1518 (1) (1993) 1–27.
[35] RIS. Routing information service. http://www.ripe.net/ris/, December 2009.
[36] S. Sahni, H. Lu, Dynamic tree bitmap for IP lookup and update, IEEE ICN (April) (2007) 79.

[37] D. Shah, P. Gupta, Fast updating algorithms for TCAMs, IEEE Micro Mag. 21 (1) (2001) 36–47.
[38] K. Sklower, A tree-based packet routing table for berkeley UNIX, in: Winter Usenix Conference, 1991, pp. 93–99.
[39] H. Song, S. Dharmapurikar, J. Turner, J. Lockwood, Fast hash table lookup using extended Bloom filter: an aid to network processing, ACM Sigcomm (August) (2005) 181–192.
[40] H. Song, F. Hao, M. Kodialam, T. Lakshman, IPV6 Lookups Using Distributed and Load Balanced Bloom Filters for 100 Gbps Core Router Line Cards, 2009, pp. 2518–2526.
[41] V. Srinivasan, G. Varghese, Fast address lookups using controlled prefix expansion, ACM Trans. Comput. Syst. 17 (1) (1999) 1–40.
[42] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, N.P. Jouppi, Cacti 5.1: An integrated cache timing, power, and area model. Technical report, HP Labs, April 2008.
[43] G. Varghese, Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices, 1 edition, Morgan Kaufmann, 2005.
[44] B. Vocking, How asymmetry helps load balancing, ACM J. 50 (4) (2003) 568–589.
[45] Y. Wang, U. Bhattacharya, F. Hamzaoglu, P. Kolar, Y. Ng, L. Wei, Y. Zhang, K. Zhang, M. Bohr, A 4.0 ghz 291 mb voltage-scalable SRAM design in 32 nm high-*k* metal-gate cmos with integrated power management, IEEE ISSCC (February) (2009) 456–457.
[46] S. Yun, Hardware-based IP lookup using *n*-way set associative memory and lpm comparator, in: Lec. Notes in Comp. Sci. (LNCS), vol. 4017, 2006, pp. 406–414.
[47] F. Zane, G. Narlikar, A. Basu, Coolcams: power-efficient TCAMs for forwarding engines, IEEE Infocom (April) (2003) 42–52.

Michel Hanna is a Ph.D. student in the Computer Engineering program, Computer Science department at the University of Pittsburgh. Michel is working under the supervision of Professor Rami Melhem and Professor Sangyeun Cho both are with the computer science department at the University of Pittsburgh. In November 2009 he defended his Master's degree in Computer Engineering from the University of Pittsburgh. He also received his Electrical Engineering Master's degree from Cairo University, Egypt in November 2004 after he got his Bachelor's degree in Electrical Engineering from Cairo University at Fayoum, Egypt in May 1999. Michel's research interests are: high-performance IP routers and switches packet processing engines, computer network security and high-performance computing.

Socrates Demetriades is a Ph.D. student in the Computer Science department at the University of Pittsburgh. His advisor is Prof. Sangyeun Cho. He joined the computer science department in August 2006 after receiving his Computer Engineering Diploma from the Polytechnic school, University of Patras, Cyprus in May 2006. Socrates' current research interests are in Computer Architecture.

Sangyeun Cho received his B.S. in Computer Engineering from Seoul National University, Seoul, in 1994, and his Ph.D. in Computer Science from the University of Minnesota, Minneapolis, in 2002. From 1999 to 2004, he worked for Samsung Semiconductor, where he designed several generations of the CalmRISC^{TM} embedded processor core and their cache memories. His research focus is in the area of computer architecture, microprocessor design, and system-on-a-chip (SOC). Dr. Cho joined the Department of Computer Science at the University of Pittsburgh in fall 2004.

Rami Melhem has received the following degrees: B.S. (Electrical Engineering, 1976) from Cairo University; B.S. (Mathematics, 1978) from Ein Shams University, Cairo; MA (Mathematics, 1981), M.S. (Computer Science, 1981), and Ph.D. (Computer Science, 1983) from the University of Pittsburgh. He was Assistant Professor in the Department of Computer Science at Purdue University 1984–87 (on leave 1985–87), and Visiting Professor in the Department of Mathematics at the University of Pittsburgh 1985–86. Since 1986 he has been on the faculty of the Department of Computer Science at the University of Pittsburgh. He has published numerous papers in the areas of systolic architectures, parallel computing, fault-tolerant computing, and optical interconnection networks. He served on program committees for several conferences and is on the Editorial Board of IEEE Transactions on Computers. He is a member of the IEEE Computer Society, the Association for Computing Machinery, and the International Society for Optical Engineering. His research interests include: parallel and distributed high-performance computing, fault-tolerant computing, multiprocessor interconnection networks, real-time systems and optical computing.