CHAP: Enabling Efficient Hardware-Based Multiple Hash Schemes for IP Lookup

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

Dept. of Computer Science, Univ. of Pittsburgh,
Pittsburgh, PA 15260, USA
{mhanna,socrates,cho,melhem}@cs.pitt.edu

Abstract. Building a high performance IP lookup engine remains a challenge due to increasingly stringent throughput requirements and the growing size of IP tables. An emerging approach for IP lookup is the use of set associative memory architecture, which is basically a hardware implementation of an open addressing hash table with the property that each row of the hash table can be searched in one memory cycle. 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 bucket overflows. This paper presents a new simple hash probing scheme called CHAP (Content-based HAsh Probing) that tackles the hash overflow problem. In CHAP, the probing is based on the content of the hash table, thus avoiding the classical side effects of probing. We show through experimenting with real IP tables how CHAP can effectively deal with the overflow.

Keywords: IP lookup, hardware multiple hashing, content-based probing.

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 [1]. 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. Some predict that in the near future,“Terabit” link rates will be available with affordable prices [2, 3].

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) [4].

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. Recently, using hash techniques for IP lookup gained a lot of momentum. Hash tables come in two flavors: open addressing hash and closed addressing hash (or *chaining*). The hash table in closed addressing hash has a fixed height (number of buckets), and each bucket is an infinite size linked list. During the lookup process, a specific row index is generated by the hash function and the row is searched to find the target key. An important design goal in this case is to minimize the worst-case length of the linked lists and to balance the bucket population by using Bloom filters-like data structures [5,6,7].

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 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 [8].

The hardware schemes use special hardware such as Ternary Content Addressable Memory (TCAM) to increase the lookup throughput. Unfortunately, the TCAM approach has its own set of limitations: high power consumption, poor scalability, and low bit density. Moreover, most commodity TCAMs run at low speed compared to SRAM memory [3]. Hence many researchers proposed optimizations to the TCAM architecture [9,10,11,12].

In this paper, we assume open addressing hash schemes for which a number of efficient hardware prototype implementations have been proposed recently [13,14]. 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 with no/acceptable overflow and with good space utilization. In addition, we want to keep both insertion/deletion into/from the table simple and straightforward. This paper makes the following contributions to the area of open addressing hash in general:

– The introduction of the new concept of content-based hash probing which more effectively tackles the overflow than other existing probing techniques.
– The application of content-based probing to multiple hash function schemes.
– The use of content-based probing and multiple hashing, together, to implement an efficient hardware-based IP lookup engine.

The rest of the paper is organized as follows. In Section 2 we briefly summarize the state-of-the-art hardware-based hash techniques. In Section 3 we describe CHAP, our main scheme. Section 4 discusses the setup procedure of CHAP and how search is done. We will then discuss the incremental updates in Section 5. Section 6 shows experimental results. Finally, we give conclusions and future work in Section 7.

2 Background

2.1 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 *collision*. When there are too many ($\geq L$) colliding records, some of those 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:

$$h(k), h(k) + \beta_0, h(k) + \beta_1, \cdots, h(k) + \beta_{m-1} \tag{1}$$

where each β_i is a constant, and m is the maximum number of probes. Linear probing is simple, but often suffers from *primary key clustering* [8].

Instead of probing, one 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 eliminating hash overflow than probing [15]. 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:

$$h_0(k), h_1(k), \cdots, h_{H-1}(k) \tag{2}$$

where H is the maximum number of hash functions. Most work that is done in the multiple hashing domain is for closed addressing hash as in [15, 16].

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.

2.2 Set Associative Memory Architecture Overview

We will use the CA-RAM (Content Addressable-Random Access Memory) as a representative of a number of set associative memory architectures proposed for IP lookup [13, 14]. A CA-RAM takes as an input a search key and outputs

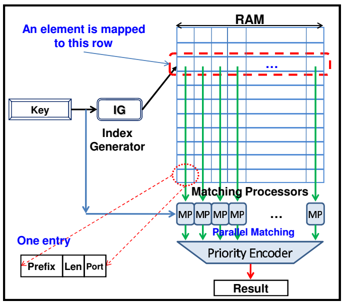

Fig. 1. The basic CA-RAM Architecture

the result of a lookup. Its main components are: an index generator, a memory array (SRAM or DRAM), and match processors, as shown in Figure 1.

Given a key, the index generator uses a hash function to create an index which is used to access a row of the memory array. All the keys stored in that row are fetched simultaneously and the match processors compare the row of keys with the search key in parallel, resulting in constant-time matching. Note that the format of each memory row is flexible and the matching processors are programmable.

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 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 Equation (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 space. While linear probing uses predetermined offsets to solve that problem as specified by Equation (1), CHAP uses the same probing sequence, but with the constants $\beta_0, \beta_1, \cdots, \beta_m$ 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:

$$h(k), \beta_0[h(k)], \beta_1[h(k)], \cdots, \beta_{m-1}[h(k)] \quad (3)$$

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 empty spaces. We call those pointers "probing pointers" and the overall scheme is called **CHAP(1,m)** since it has only one hash function and m probing pointers per row.

Figure 2 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 $\beta_1[i]$.

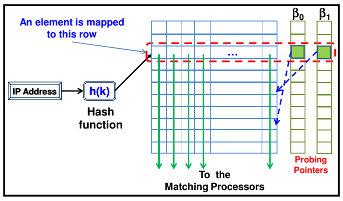

Fig. 2. The CHAP basic idea

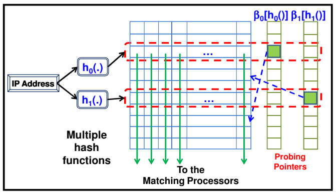

Fig. 3. The CHAP(2,2)

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 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:

$$h_0(k), h_1(k), \cdots, h_{H-1}(k), \beta_0[h_0(k)], \beta_1[h_1(k)], \cdots, \beta_{m-1}[h_{H-1}(k)] \qquad (4)$$

In essence, we dedicate to each hash function a pointer per row. An example is shown in Figure 3 for a two hash functions CHAP scheme where a key is mapped to two different buckets. In the example, this key will have four different buckets to which it can be allocated: $h_0(k)$, $h_1(k)$, $\beta_0[h_0(k)]$ and $\beta_1[h_1(k)]$ 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 is to share probing pointers among hash functions. Yet a third organization is to 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 will limit our discussion to CHAP(1,m) and CHAP(H,H) with one pointer for each hash function and with the row order given by Equation (3). We defer the investigation of the other organization to future work.

4 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 [1]. In this work, we will use only the most significant 16 bits in our hash functions. Prefixes shorter than 16 bits (short prefixes) are not included in the hash table. A separate small TCAM can be used to store those prefixes. This small TCAM is to be searched in parallel with the main hash table on every lookup, which is a common practice [12, 17].

4.1 The Setup Algorithm

Algorithm 1 lays out the setup phase of CHAP. In that algorithm, $j = 0, \cdots, M-1$ is used to index the prefixes, where M is the total number of prefixes in an IP routing

Algorithm 1. CHAP(H,H) Setup Algorithm

```
1:  Sort the IP prefixes from long to short and initialize the arrays $HC[N]$ & $OC[N][H]$ to zeros
2:  table_overflow = 0
3:  for($j = 0; j < M; j++$)
4:      finished = $false$
5:      for($i = 0; i < H; i++$)
6:          $r_i = h_i(k_j)$
7:      for ($i = 0$; $i < H$ AND finished == $false$; $i++$)
8:          if($HC[r_i] < L$), then
9:              $HC[r_i]++$
10:             finished = $true$
11:     for ($i = 0$; $i < H$ AND finished == $false$; $i++$)
12:         if($OC[r_i][i] < \lambda$), then
13:             $OC[r_i][i]++$
14:             finished = $true$
15:     if(finished == $false$), then table_overflow++
```

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[row index], 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[row index][hash function index] 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.

CHAP setup phase determines if the configuration parameters of the hash table is valid or not. In other words, will the parameters L, H, λ and N result in a mapping of the M prefixes into a single hash table without/with acceptable overflow?

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, OC and the table_overflow counter to zeros (lines 1–2). The set of hash values $\{r_0, \cdots r_{H-1}\}$ for each prefix is calculated (lines 5–6). Then, the algorithm updates the counter HC by using the H hash values of each prefix. If there is a spot for the current prefix in HC then the algorithm will move on to the next prefix (lines 8–10). If not, it will increment the OC counter (lines 11–14).

When Algorithm 1 exits, table_overflow contains the number of prefixes that could not fit in either HC or OC. If that number is not acceptable, then the algorithm can be repeated with more hash functions. That is with $H = H + 1$. If a separate TCAM is used to store the short prefixes as described earlier, then this same TCAM can be used to store the overflow prefixes. Hence the acceptable overflow in Algorithm 1 will depend on the capacity of the added TCAM.

4.2 The Mapping of IP Prefixes in CHAP

The last step in CHAP is to allocate the elements into the hash table using the probing pointers. Before moving to the actual mapping of the prefixes, however, we need to assign values to the probing pointer's array. This is done by running the best fit algorithm [18] to set the values of the probing pointer. 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 call this a *hole*). 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 the largest counter.

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.

4.3 Search in CHAP

As discussed in Section 2.2, hardware implementation of hash tables reads a full row (bucket) of the table into a buffer and uses a set of comparators to determine, in parallel, the longest prefix match among the elements in that bucket. Hence, a metric that will be used to measure the efficiency of the search in CHAP is the Average Successful Search Time, **ASST**, the average number of rows accessed for successful search.

The CHAP search algorithm itself is straightforward. Given a key k_x as an input IP address, we calculate the row address $h_0(k_x)$ and then match the prefix against all the elements in this row (in parallel). If we find a hit at that row, we stop searching. If not, we try the next hash functions (*i.e.,* $h_1(k_x), \cdots, h_{H-1}(k_x)$). After we are done with all the H hash functions we start looking at the probing pointers. First the probing pointer $\beta_0[h_0(k_x)]$, then $\beta_1[h_1(k_x)], \cdots, \beta_{H-1}[h_{H-1}(k_x)]$. In other words, the order of accessing the pointers used in searching is the same order used in inserting the prefixes. This constraint has to be satisfied to guarantee the LPM feature in searching. Specifically, if we do not use the insertion order when searching, we might search in a row that contains shorter prefixes than the longest prefix that should match the address k_x. This order is maintained through the dedication of one probing pointer per hash function. Without this dedication, we cannot preserve LPM as the probing pointers will be shared between multiple hash functions.

5 The Incremental Updates

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 [1,2]. 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 and decrement the row counter RC, where a counter $RC[i]$ is associated with each row i to record the number of prefixes stored in that row. This counter will be used during the insert/update operation to keep track of full rows.

Algorithm 2. CHAP Insert Update Algorithm

```
1: subroutine CHAP_Insert_Update (prefix $k_n$)
2: for($i = 0$; $i < H$; $i++$)
3:      $T[i] = h_i(k_n)$
4:      $T[i + H] = \beta_i[h_i(k_n)]$
5: By searching the rows $T[0], \cdots T[2 \times H - 1]$, find:
6:      $k_l$ = longest prefix that matches $k_n$ and $r_l$ = row that contains $k_l$
7:      $k_s$ = shortest prefix that matches $k_n$ and $r_s$ = row that contains $k_s$
8: if($k_l$ is not defined AND $k_s$ is not defined ), then
9:      return(Insert_in_Rows($k_n$, $T[0]$, $T[2 \times H - 1]$) /* if no matching: insert $k_n$ in any row */
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($k_n$, $T[0]$, $r_l$))
14: else if($|k_n| < |k_s|$), then return(Insert_in_Rows($k_n$, $r_s$, $T[2 \times H - 1]$))
15: else, return(Insert_in_Rows($k_n$, $r_l$, $r_s$))
16:
17: subroutine Insert_in_Rows (prefix $k_x$, row $r_a$, row $r_b$)
18: for($i = r_a$; $i <= r_b$; $i++$)
19:      if($RC[i] < L$), then
20:           insert $k_x$ in row $i$ and $RC[i]++$
21:           return ($true$)
22: return ($false$)
```

The basic idea of the insert/update operation, which is detailed in Algorithm 2, is to find the appropriate row r 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 2 consists of two "Boolean" subroutines, **CHAP_Insert_Updtae()** and **Insert_in_Rows()**. The second subroutine is where the actual insertion is made, as it take a prefix k_x and tries to insert it in a series of rows starting from row r_a all the way to r_b. The first subroutine, CHAP_Insert_Updtae(), will determine the appropriate rows to insert the new prefix, k_n.

In the first routine a single dimension array $T[\cdot]$ of size $(2 \times H)$ is used to store the computed values of the hash functions of k_n and the corresponding probing pointers (lines 2–4). For each row in $T[i]$ 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 r_l and r_s, that include k_l and k_s, if a matching is found.

Depending on the length of k_n relative to the length of both k_l and k_s, we will try to insert k_n in one of the $2 \times H$ 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 [1], is when k_n is equal either k_l or k_s. In this case we replace either k_l or k_s with the new prefix k_n (lines 10–12). The third case is if the length $|k_n|$ of k_n is larger than $|k_l|$, the length of k_l. We will try to insert k_n into one of the buckets $T[0], \cdots, r_l$ if they have a space (line 13). In the next case we check to see if $|k_n| < |k_s|$ and if it is true, then we try to insert k_n in a row among $r_s, \cdots, T[2 \times H - 1]$ (line 14). The final case is when $|k_s| < |k_n| < |k_l|$ and in this case we will call Insert_in_Rows() to try to put k_n in any row between

r_l and r_s (line 15). In any case, the subroutines terminate successfully if we were able to insert k_n.

6 Evaluation

We used C++ to build our own simulation environment. This environment allows us to choose and arrange different types of hash functions. The BGP (Border Gateway Protocol) routing tables of Internet core routers were obtained from the routing information service project [19]. The statistics for the routing tables used are listed later in Table 9. When measuring the average lookup time, we used synthetic traces (uniform distribution) generated from these tables.

The hash functions used in our experiments are from three different hashing families: bit-selecting, CRC-based, and $\mathbf{H_3}$ [20] hashing families. Those families have the advantage of being simple and fast enough to be easily realized in hardware.

For a given hardware implementation, the number of rows, N, and the number of entries per row, L, are fixed and the performance of the CHAP scheme 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 smaller value 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 addition to H and λ, the performance of CHAP depends on the load factor α. Clearly, α depends on the size of the actual IP routing table and the size of the physical memory used to store the hash table. For a given α, the hashing overflow depends on the aspect ratio of the memory N/L. In what follows, we will define a "configuration" by specifying both N and L.

6.1 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 will use the table "rrc07-AS21202" obtained from [19] and use two different configurations C_1: $L = 200$, $N = 1024$ and C_2: $L = 100$, $N = 2048$. We tried many different configurations and they all led to results similar to those shown in Figure 4. In addition, these two configurations have a high average load factor $\alpha = 91.27\%$ for the "rrc07-AS21202" table, which articulates the strength of CHAP.

Figure 4 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 71.61% more overflow reduction than linear probing on average. Moreover, we can see

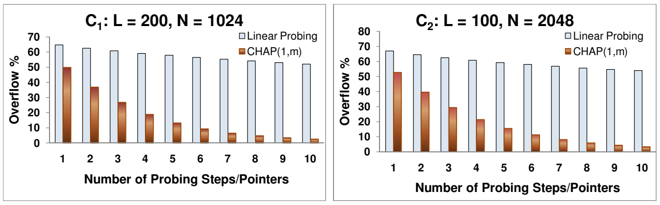

Fig. 4. Overflow of CHAP(1, m) vs. linear probing for table rrc07-AS13645

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 reduction 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.

6.2 Sensitivity Analysis of CHAP (H,H)

In this section we study the effect of varying λ or the maximum value of the overflow counter, OC, in the CHAP setup algorithm (Section 4.1). We report the results for table "rrc07-AS13645" since all other tables have similar results. We will also show the results for slight variations of the configurations C_1 and C_2. In both case we use $H = 3$.

Figure 5 shows the values of overflow versus λ for the reported configurations. For the Figure 5(a), we set $N = 1024$ rows and $L = 180, 200, 220$ and 240 entry per row, which results in $\alpha = 96.03\%, 91.27\%, 82.98\%$ and 76.10% respectively. As for Figure 5(b), we set $N = 2048$ rows and $L = 90, 100, 110$ and 120 entry per row which will result in the same loading factors. Note that $\lambda \in [0, L]$.

From the figures we can see that 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 becomes almost zero in Figure 5(b) while it is actually zero in Figure 5(a). The low overflow determined at $\lambda = \frac{L}{2}$ means 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 the following section we will use $\lambda = \frac{L}{2}$.

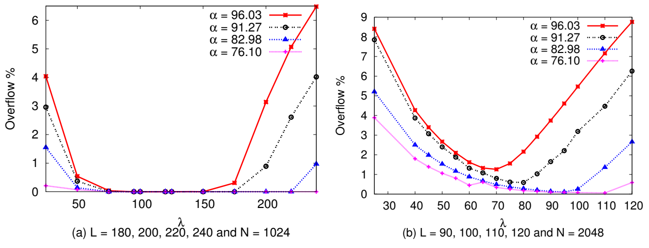

Fig. 5. The overflow vs. λ

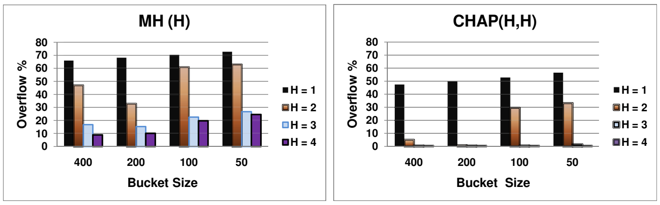

Fig. 6. Average overflow of 4 bucket widths for MH(H) and CHAP(H,H)

6.3 CHAP(H,H) Versus Multiple Hashing(H)

In this section we compare the **CHAP(H,H)** scheme against the regular multiple hash function scheme (which we call **MH(H)**) [15], where H is the number of hash functions used. We compare the two schemes in terms of the ASST (Average Successful Search Time) and the overflow. Again, we use the routing table rrc07-AS13645.

In Figure 6 we show the average values of overflow for different number of hash functions (between 1 and 4) and for four different bucket sizes. To keep α constant at 91.27%, we set $N = 512, 1024, 2048$ and 4096 where $L = 400, 200, 100$ and 50. It's obvious from this figure that CHAP(H,H) has much less overflow than multiple hashing for the same number of hash functions.

The results shown in Figure 7 indicates that the average ASST over the four bucket sizes for MH(H) is 1.7, while it's 2.24 for CHAP(H,H). Although the difference between the two schemes seems large, we have to take into consideration that at $H = 3$ the overflow of CHAP is already zero for the bucket sizes of $L = 400, 200$ and is less than 2% when $L = 100, 50$. Thus adding more hash functions in this case makes the average memory access time worse. If we recalculate the average ASST over the four bucket sizes for CHAP (without taking $H = 4$ into account) we find it to be 1.87 which is only 10% higher than MH. What looks like the classical tradeoff between the overflow and the average memory access time can be seen in Figures 6 and 7. However, a better understanding of the tradeoff that CHAP and MH present can be obtained by comparing CHAP(H,H) with MH($2H$) since both has $2 \times H$ as the maximum number of table accesses. For example at $H = 2$ with a bucket size $L = 200$, the overflow for CHAP(2,2) is 1.01% with an average access time of 2.2 memory cycles, while for the MH(4) is 10.12% with an average access time of 2.1 memory cycles.

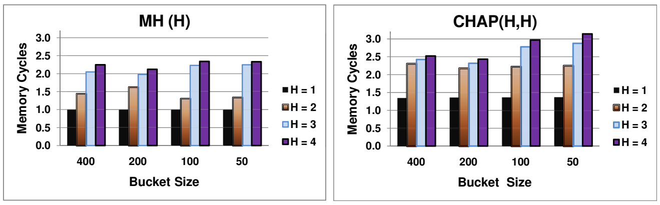

Fig. 7. Average ASST of 4 bucket widths for MH(H) and CHAP(H,H)

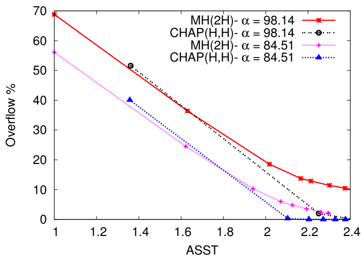

Fig. 8. Overflow vs. the ASST for MH(H) v.s. CHAP(H,H) for different load factors

	No# Tables	Avg. size	Avg. α	A.short prefixs
rrc04	3	185	93	0.77
rrc05	4	179	89.5	0.76
rrc07	3	133	66.5	0.96
rrc11	4	198	99	0.78

Fig. 9. The Statistics of the IP lookup tables used in Figure 10

In order to show that CHAP can achieve both lower overflow and average access time than MH, we plot in Figure 8 the average access time versus the overflow of both schemes for two load factors (98.14% and 84.51%) for $L = 200 \pm 25$ elements and $N = 1024$ rows. We used the same table as before and we monitored the variation of ASST and the overflow for the same load factor α. We varied the number of hash functions from 1 to 10. Each curve on the graph corresponds to 10 values of H ($H = 1, \cdots, 10$), where the left most point corresponds to $H = 1$ and the rightmost point corresponds to $H = 10$. CHAP(3,3) has zero overflow, thus we do not need to use more than three hash functions. Figure 8 shows that for the region of interest (low overflow), for any $H > 1$, the CHAP(H,H) curve has both smaller ASST and overflow than the MH(H) curve. In other words, CHAP(H,H), for $H > 1$, is more efficient (smaller ASST and fewer overflows) than the MH(H) under the same system configurations and under the same load factor.

In a different experiment we compare CHAP(3,3) with MH(6). For this experiment we map each of the 14 IP tables of [19] into a fixed hash table of 200 K entries. A summary of the properties of the 14 tables is given in Table 9, where the third column in this table indicates the average load factor (percentage) when the IP tables are mapped to a 200 K entries. All the 14 tables are from the date January 19th, 2007. The last column in the table indicates the average percentage of short prefixes in each IP routing tables group. We have indicated that these short prefixes are excluded from the main CHAP hash table and inserted in a separate TCAM table. We repeated the experiment for 4 different configurations, namely ($L = 400$, $N = 512$), ($L = 200$, $N = 1024$), ($L = 100$,

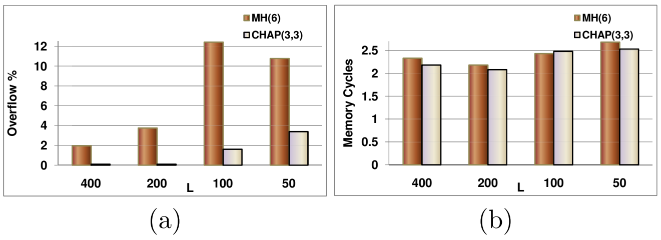

Fig. 10. Average (a) overflow and (b) ASST for CHAP(3,3) vs. MH(6)

$N = 2048$) and ($L = 50$, $N = 4096$), and we measured the average overflow and ASST for the 14 tables (Figure 10).

As we can see, CHAP(H,H) is better than MH(2H) in all cases in terms of both the ASST and the overflow. However there is only one case, $L = 100$ and $N = 2048$, where CHAP(H,H) has a slightly higher ASST (2.52 cycles) than the MH(2H) (2.51 cycles). However, for this configuration, MH(2H) has its worst-case overflow (12.3 %), while CHAP(H,H) incurs only 1.98% overflow. This means that CHAP(H,H) table contains around 20 K entries more than MH(2H) table at this configuration.

7 Conclusions and Future Work

In this paper we have described and studied CHAP, a new hash-based IP lookup scheme that eliminates the overflow problem by utilizing content-based probing and multiple hash functions. We showed that CHAP is very effective in eliminating the overflow, and at the same time, achieves a low average memory access time. We also illustrated that CHAP can be realized in hardware by taking advantage of state-of-the-art search memory architectures.

Unlike other hash-based schemes, the CHAP scheme does not require complex preprocessing of the IP tables. Simply sorting the prefixes from long to short while setting up is the only preprocessing required. CHAP uses simple functions that can be easily realized in hardware. Moreover, CHAP has simple setup and incremental update algorithms. Simulation results show that content-based hash probing 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 achieves lower average memory access time than the multiple hash function scheme while also reducing the overflow.

In this paper we did not study the general CHAP(H,m) scheme when $m \neq H$ and we intend to study this further in the future. Future work also includes the possibility of using CHAP in other IP protocol processing tasks (e.g., packet filtering and inspection).

References

1. Huston, G.: Analyzing the internet's bgp routing table. The Internet Pro. J. (2001)
2. Varghese, G.: Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices. Morgan Kaufmann, San Francisco (2005)
3. Jiang, W., Prasanna, V.K.: A memory-balanced linear pipeline architecture for trie-based ip lookup. In: HOTi 2007, August 2007, pp. 83–90 (2007)
4. Rekhter, Y., Li, T.: An architecure for ip address allocation with cidr. RFC (1993)
5. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Comm. of the ACM 13(7), 422–426 (1970)
6. Kirsch, A., Mitzenmacher, M.: Simple summaries for hashing with multiple choices. IEEE/ACM Trans. on Net (2007)
7. Song, H., et al.: Fast hash table lookup using extended bloom filter: An aid to network processing. In: Sigcomm 2005, August 2005, pp. 181–192 (2005)

8. Cormen, T., Leiserson, C., Rivest, R., Stien, C.: Introdcution to Algorithms. McGraw Hill, New York (2003)
9. Lakshminarayanan, K., Rangarajan, A., Venkatachary, S.: Algorithms for advanced packet classification with ternary cams. In: Sigcomm 2005, May 2005, pp. 193–204 (2005)
10. Panigrahy, R., Sharma, S.: Reducing tcam power consumption and increasing throughput. In: HOTi 2002, August 2002, pp. 107–112 (2002)
11. Shah, D., Gupta, P.: Fast updating algorithms for tcams. IEEE Micro. Mag. 21(1), 36–47 (2001)
12. Zane, F., Narlikar, G., Basu, A.: Coolcams: Power-efficient tcams for forwarding engines. In: Infocom 2003, April 2003, pp. 42–52 (2003)
13. Cho, S., Martin, J., Melhem, R.: Ca-ram: A high-performance memory substrate for search-intensive applications. In: Ispass 2007, April 2007, pp. 230–241 (2007)
14. Kaxiras, S., Keramidas, G.: Ipstash: A power-efficient memory architecture for ip-lookup. In: Micro 2003, November 2003, pp. 361–373 (2003)
15. Azar, Y., Broder, A.Z., Karlin, A.R., Upfal, E.: Balanced allocations. In: ACM SOC, pp. 593–602 (1994)
16. Vocking, B.: How asymmetry helps load balancing. ACM J., 568–589 (July 2003)
17. Basu, A., Narlikar, G.: Fast incremental updates for pipelined forwarding engines. In: Infocom 2003, pp. 64–74 (July 2003)
18. Randell, B.: A note on storage fragmentation and program segmentation. Comm. of the ACM 12, 365–372 (1969)
19. RIS: Routing information service (2006), `http://www.ripe.net/ris/`
20. Ramakrishna, M., et al.: Efficient hardware hashing functions for high performance computers. IEEE Trans. on Comp. 46(12), 1378–1381 (1997)