

**ADVANCED HASHING SCHEMES FOR PACKET  
FORWARDING USING SET ASSOCIATIVE  
MEMORY ARCHITECTURES**

by

**Michel Hanna**

M.S., Cairo University, 2004

Submitted to the Graduate Faculty of  
Computer Engineering Program,  
The School of Arts and Sciences  
in partial fulfillment  
of the requirements for the degree of  
**Master of Sciences**

University of Pittsburgh

2009

UNIVERSITY OF PITTSBURGH  
THE SCHOOL OF ARTS AND SCIENCES

This thesis was presented

by

Michel Hanna

It was defended on

November 16<sup>th</sup>, 2009

and approved by

Prof. Rami Melhem, Computer Science Department

Prof. Steven Levitan, Department of Electrical and Computer Engineering

Prof. KyoungSoo Park, Computer Science Department

Thesis Advisors: Prof. Rami Melhem, Computer Science Department,

Prof. Sangyeun Cho, Computer Science Department

# ADVANCED HASHING SCHEMES FOR PACKET FORWARDING USING SET ASSOCIATIVE MEMORY ARCHITECTURES

Michel Hanna, M.S.

University of Pittsburgh, 2009

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 in wire speed which is why scalability and low power consumption are features that PF engines must have while maintaining high throughput.

It is common for PF engines to use hash tables; however, the classic hashing downsides have to be dealt with (e.g., collisions, worst case memory access time,  $\dots$  etc.). 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 leads to bucket overflows.

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 exactly one memory cycle. Hence, PF engine architectures based on associative memory will outperform those based on the conventional Ternary Content Addressable Memory (TCAM) in terms of power and scalability since they use regular RAM technology. However, such architectures need special algorithms to handle the overflow and the incremental updates.

The two standard solutions to the overflow problem are either to use some sort of pre-defined 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, or CHAP. As the name suggests, CHAP is a probing scheme that is based on the content of the hash table to avoid 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, or 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 hashing schemes.

**Keywords:** Hardware Hashing, Set Associative Memories, IP Lookup, Packet Forwarding, Hash Schemes.

## TABLE OF CONTENTS

<b>PREFACE</b> . . . . .	ix
<b>1.0 INTRODUCTION</b> . . . . .	1
<b>2.0 BACKGROUND</b> . . . . .	4
2.1 General Open Addressing Hash . . . . .	4
2.2 Hashing in the presence of wildcards . . . . .	5
2.3 Set Associative Memory Architecture Overview . . . . .	7
<b>3.0 CONTENT-BASED HASH PROBING</b> . . . . .	10
3.1 The CHAP(H,H) Scheme . . . . .	12
3.2 The CHAP Setup Algorithm . . . . .	13
3.2.1 The Mapping of IP Prefixes in CHAP. . . . .	14
3.3 Search in CHAP . . . . .	14
3.4 The Incremental Updates in CHAP . . . . .	16
<b>4.0 THE PROGRESSIVE HASHING SCHEME</b> . . . . .	20
4.1 The PH Setup Algorithm . . . . .	21
4.2 Searching in PH . . . . .	22
4.3 The Incremental Updates in PH . . . . .	23
<b>5.0 EVALUATION</b> . . . . .	26
5.1 The Evaluation of Content-based Hash Probing . . . . .	27
5.1.1 The Advantages of Content-based Hash Probing . . . . .	27
5.1.2 Sensitivity Analysis of CHAP (H,H) . . . . .	28
5.1.3 CHAP(H,H) versus Restricted Hashing(H) . . . . .	29
5.1.4 CHAP(H,H) versus Restricted Hashing(2H) . . . . .	31

5.2 The Evaluation of Progressive Hashing . . . . .	32
5.3 Applying Content-based Hash Probing to PH . . . . .	34
5.3.1 Memory Overhead of CHAP and PH . . . . .	35
<b>6.0 CONCLUSIONS AND FUTURE WORK . . . . .</b>	<b>37</b>
<b>BIBLIOGRAPHY . . . . .</b>	<b>39</b>

## LIST OF TABLES

1	The Statistics of the IP lookup tables on January 31 <sup>st</sup> 2009. . . . .	26
---	--	----

## LIST OF FIGURES

1	Splitting the Hashing Space into Groups. . . . .	7
2	The CA-RAM As An Example of Set Associative Memory Architectures. . . . .	8
3	The CHAP basic concept. . . . .	11
4	The CHAP(3,3). . . . .	11
5	The Evolution of The PH Scheme. . . . .	24
6	Applying the PH Scheme. . . . .	24
7	Overflow of CHAP(1, m) vs. Linear Probing(1, m) for table rrc07. . . . .	28
8	The overflow vs. $\lambda$ . . . . .	29
9	Average overflow of 4 bucket widths for RH(H) and CHAP(H,H) for table rrc05. . . . .	30
10	Average AMAT of 4 bucket widths for RH(H) and CHAP(H,H) for table rrc05. . . . .	30
11	(a)Average Overflow and (b) AMAT for CHAP(3,3) vs. RH(6) for 15 Lookup Tables for C1: $\{L = 180, N = 2048\}$ . . . . .	31
12	(a)Average Overflow and (b) Average AMAT for CHAP(3,3) vs. RH(6) for 3 configurations. . . . .	32
13	(a) Average Overflow (b) AMAT of RH(5) vs. GH(5) vs. PH(5) for 15 Lookup Tables for C1: $\{180 \times 2048\}$ . . . . .	33
14	(a) Average Overflow and (b)Average AMAT for of RH(5) vs. GH(5) vs. PH(5) for 3 configurations. . . . .	34
15	(a) Average Overflow and (b) Average AMAT of CHAP(5,5) vs. PH(5) vs. PH.CHAP(5,5) for 3 Configurations. . . . .	35

## **PREFACE**

### **Acknowledgments**

To both Dr. Rami Melhem and Dr. Sangyeun Cho, my advisors and also to both Dina and Mira, my little beautiful family

## 1.0 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 [10]. 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 with affordable prices [26, 11].

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 [21].

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, that is 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 unbounded 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 [4, 13, 24].

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 or by using multiple hash functions [6].

The hardware schemes use special hardware such as Ternary Content Addressable Memory (TCAM) [14, 16] 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 [11]. Hence many researchers proposed optimizations to the TCAM architecture [15, 18, 23, 30].

In this thesis, we assume open addressing hash schemes for which a number of efficient hardware prototype implementations have been proposed recently [5, 12, 29]. 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 in general:

- The introduction of the new concept of content-based hash probing 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 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 thesis 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 which 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. We also discuss in this

section the state-of-the-art set associative memory architectures by showing a representing example. In Section 3 we describe CHAP, our first scheme. We discuss our second scheme Progressive Hashing in Section 4. Section 5 shows 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 6.

## 2.0 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 scheme 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:

$$h(k), h(k) + \beta_0, h(k) + \beta_1, \dots, h(k) + \beta_{m-1} \quad (2.1)$$

where each  $\beta_i$  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” [6]. 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:

$$h(k, i) = h'(k) + c_1 \times i + c_2 \times i^2, i = 0, 1, \dots, m - 1 \quad (2.2)$$

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” [6].

Instead of probing, we can apply a second hash function to find an empty bucket, which is known as *double hashing* [6]. 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*) the probing sequence of inserting a key into the hash table is given as follows:

$$h_0(k), h_1(k), \dots, h_{H-1}(k) \tag{2.3}$$

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, 27]. Note that using a different hash table for each hash function in Equation 2.3 is a valid design option; however, using different hash tables leads to memory fragmentation that results in poor space utilization. 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 Equation 2.3.

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  $\alpha$ , or the *load factor*, defined as  $M/(N \times L)$ . With a smaller  $\alpha$ , the average number of hash table accesses can be made smaller, however at the expense of more unused memory space, which leads to increase the power consumption [2].

## 2.2 HASHING IN THE PRESENCE OF WILDCARDS

Applying hash functions in packet forwarding is very challenging due to the fact that wildcards, or don’t care, bits are heavily present in the IP lookup tables. Hashing with wildcards requires one of the two solutions: restricted hashing or grouped hashing [9]. 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 [25] 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 [8, 9]. 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. The number of prefixes that are longer than 16 is less than 2% of the lookup table population.

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:

- 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*,  $\dots$ , and  $h_4()$  that uses 8 bits can be associated to group *S8*. This scheme is similar to the one used in [12]. Figure 1(a) shows the five groups and their associated hash function. We represent the 32-bit address space with bold line and *MSb* and *LSb* stand for most significant bit and least significant bit, respectively. The prefixes that are less than 8 bits long, which are less 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.

Grouped hashing will be used in Section 4 to derive the progressive hashing (PH) scheme, which is our second proposed scheme.

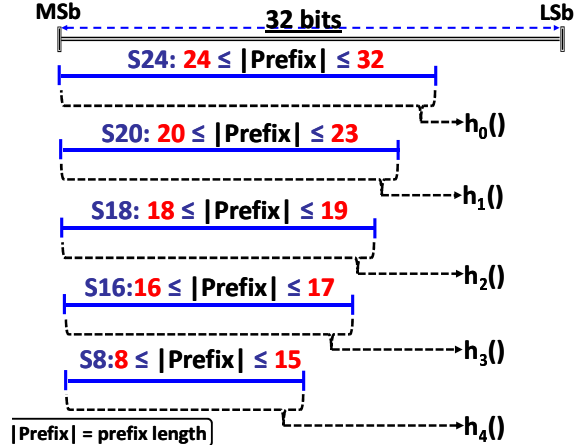


Figure 1: Splitting the Hashing Space into Groups.

### 2.3 SET ASSOCIATIVE MEMORY ARCHITECTURE OVERVIEW

We use the CA-RAM (Content Addressable-Random Access Memory) as a representative of a number of set associative memory architectures proposed for IP lookup [5, 12, 29]. 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 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 [5].

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 Figure 2.

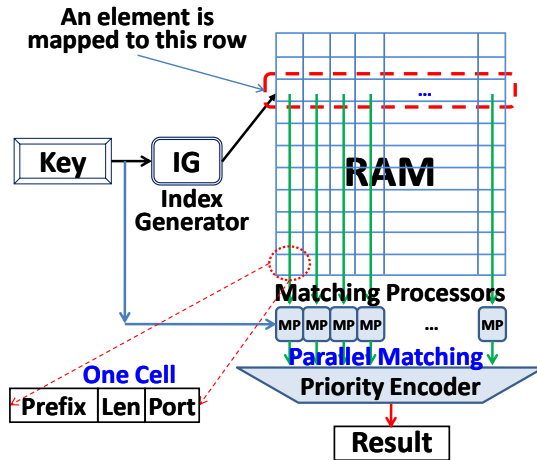


Figure 2: The CA-RAM As An Example of Set Associative Memory Architectures.

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 Figure 2 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*, which is 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 candi-

date keys with the search key in parallel, resulting in constant-time matching. Each match processor performs comparison quickly using a hardware comparator.

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 light-weight, 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 [5] shows that CA-RAM is over 26 times more power-efficient than the 16T SRAM-based TCAM [14], and over 7 times improved over the 6T dynamic TCAM [16]. The CA-RAM cell size is over  $12\times$  smaller than a 16T SRAM-based TCAM cell, and  $4.8\times$  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 [5].

### 3.0 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 (2.1). 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 Equation (2.1), CHAP uses the same probing sequence, but with the constants  $\beta_0, \beta_1, \dots, \beta_m$  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:

$$h(k), \beta_0[h(k)], \beta_1[h(k)], \dots, \beta_{m-1}[h(k)] \quad (3.1)$$

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 3 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  $\beta_1[i]$ .

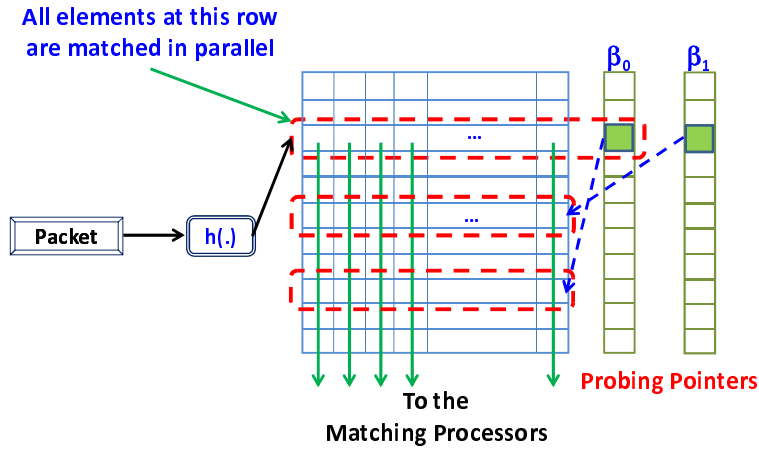


Figure 3: The CHAP basic concept.

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.

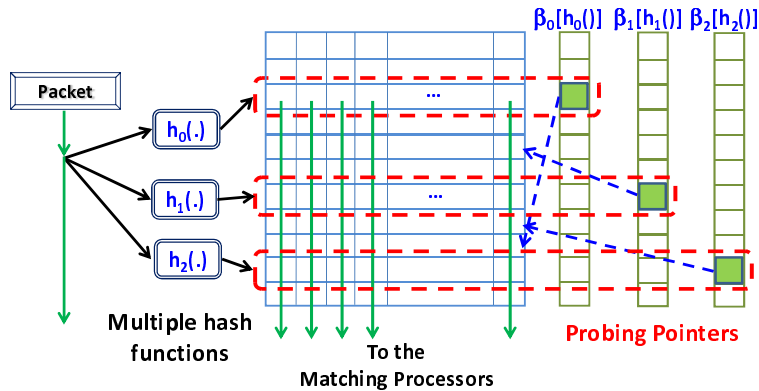


Figure 4: The CHAP(3,3).

The content-based probing can also be applied to the multiple hashing scheme. Specifi-

cally, 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:

$$h_0(k), h_1(k), \dots, h_{H-1}(k), \beta_0[h_0(k)], \beta_1[h_1(k)], \dots, \beta_{m-1}[h_{H-1}(k)] \quad (3.2)$$

In essence, we dedicate to each hash function a pointer per row. An example is shown in Figure 4 for a three hash functions CHAP scheme, or CHAP(3,3), where a key is mapped to three different buckets. 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 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 work, 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 Equations 3.1 and 3.2 for both 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 [10]. 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 that we call “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 which is a common practice[30, 3, 8, 9].

### 3.2 THE CHAP SETUP ALGORITHM

Algorithm 1 lays out the setup phase of CHAP. In that algorithm,  $j = 0, \dots, M - 1$  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,  $HC[\text{row index}]$ , 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  $OC[\text{row index}][\text{hash function index}]$  to count the overflow elements for each hash function per row. The maximum value of a single counter in this array is equal to  $\lambda$ , 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, do the parameters  $L$ ,  $H$ ,  $\lambda$  and  $N$  result in a mapping of the  $M$  prefixes into a single hash table with acceptable overflow, or not?

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  $\{r_0, \dots, r_{H-1}\}$  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 the 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 [20]. 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  $HC[J]$ , which we call a *hole*. Then the  $I^{th}$  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**[N][L]”, where N and L are the number of rows and the row capacity respectively. For each element in H\_Table[N][L] consists of the actual prefix, H\_Table[N][L].key, and the prefix length, H\_Table[N][L].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  $i = 0, \dots, H - 1$  (lines 2–4).

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 5–7). If we do not find a match in these rows, we simply search the overflow\_buffer (line 11).

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 [8]. 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 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 restrictive 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:  $r_0(k_m), \dots, r_{2H-1}(k_m)$ , where  $r_i(k_m) = h_i(k_m)$  for  $i = 0, \dots, H - 1$  and  $r_i(k_m) = \beta_i[h_i(k_m)]$  for  $i = H, \dots, 2H - 1$ . 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:  $r_0(P_X), \dots, r_{2H-1}(P_X)$ , we will match  $k_X$  at row  $r_X$  before matching  $k_Y$  at row  $r_Y$ .  $\square$

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

### 3.4 THE 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 [10, 26]. 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  $HC$  which is used to keep track of the rows' populations.

The basic idea of the insert/update operation, which is detailed in Algorithm 3, 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 3 consists of two boolean functions, **CHAP\_Insert\_Update()** and **Insert.in.Rows()**. The first subroutine, **CHAP\_Insert\_Update()**, determines the appropriate rows to insert the new prefix  $k_n$  (lines 16–21). The second subroutine 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  $r_i$ , which 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 will be 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  $r_l$  and  $r_s$  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 [10], 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  $|k_n|$  is larger than  $|k_l|$ , then we try to insert  $k_n$  into one of the buckets  $\{r_0, \dots, r_l\}$  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  $\{r_s, \dots, r_{2H-1}\}$  (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 subroutines terminate successfully if we are able to insert  $k_n$ . Otherwise, we try either insert  $k_n$  into the `overflow_buffer`, or use a backtracking scheme like “Cuckoo hashing” [17] 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 [7].

---

**Algorithm 1** CHAP(H,H) Setup Algorithm.

---

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

---

---

**Algorithm 2** The CHAP Search Algorithm.

---

```
1: Search_Hash_Table(Packet  $P$ )
2:   for( $i = 0 ; i < H ; i++$ ) {
3:      $r_i = h_i(P)$ 
4:      $r_{i+H} = \beta_i[h_i(P)]$  }
5:   for( $i = 0 ; i < 2H ; i++$ ) {
6:     if( $P$  matches  $H\_Table[r_i][j].key$ ), then
7:       return  $H\_Table[r_i][j].port$ 
8:     else
9:       continue
10:   }
11:   search the overflow_buffer
```

---

---

**Algorithm 3** CHAP Insert Update Algorithm.

---

```
1: subroutine CHAP_Insert_Update (prefix  $k_n$ )
2: for( $i = 0; i < H; i ++$ ) {
3:    $r_i = h_i(k_n)$ 
4:    $r_{i+H} = \beta_i[h_i(k_n)]$  }
5: By searching the rows  $r_0, \dots, r_{2H-1}$ , find:
6:    $k_l =$  longest prefix matching  $k_n$  and  $r_l =$  row containing  $k_l$ 
7:    $k_s =$  shortest prefix matching  $k_n$  and  $r_s =$  row containing  $k_s$ 
8: if( $k_l$  is not defined AND  $k_s$  is not defined ), then
9:   return(Insert_in_Rows( $k_n, 0, (2H - 1)$ ) /*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( $k_n, 0, r_l$ ))
14: else if( $|k_n| < |k_s|$ ), then return(Insert_in_Rows( $k_n, r_s, (2H - 1)$ ))
15: else, return(Insert_in_Rows( $k_n, r_l, r_s$ ))

16: subroutine Insert_in_Rows (prefix  $k_x, a, b$ )
17: for( $i = a; i \leq b; i ++$ )
18:   if( $\text{HC}[r_i] < L$ ), then
19:     insert  $k_x$  in  $r_i$  and  $\text{HC}[r_i]++$ 
20:   return (true)
21: return (false)
```

---

## 4.0 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 Figures 5(a) and (b) where the hashing space is represented as a circle. In the restricted hashing scheme (Figure 5(a)) the hash functions  $h'_0(), \dots, h'_3()$  are applied to all the keys in the hashing space. On the other hand, in the grouped hashing (Figure 5(b)) we split the hashing space into groups and a single hash function is associated with each group. In Figure 5(b), functions  $h_0(), \dots, h_3()$  are associated with Groups 0,  $\dots$  3 respectively.

In this section we group the prefixes based on their lengths (i.e., use grouped hashing or GH). Consequently, groups with longer prefix length can use the hash functions of other groups that have shorter prefix lengths. For example, in Figure 1, group  $S_{24}$  can use the hash functions of groups  $S_{20}$  and  $S_{16}$ . Motivated by this observation, we propose to apply the hash functions in a progressive manner as illustrated in Figure 5(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 Figure 1, the hash function  $h_0(\cdot)$  is more specific than  $h_1(\cdot)$ ,  $h_2(\cdot)$ ,  $h_3(\cdot)$  and than  $h_4(\cdot)$ . Figure 6 demonstrate the PH scheme applied to the same groups of Figure 1. As an example, group  $S_{24}$ , which is assigned to hash function  $h_0(\cdot)$ , can use the less specific hash functions of groups  $S_{20}$ ,  $S_{18}$ ,  $S_{16}$  and  $S_8$  as illustrated in Figure 6.

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,  $j = 0, \dots, M - 1$  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,  $H\_Table[\text{row index}][\text{bucket size}]$ , with  $L = \text{maximum bucket size}$ , and  $N = 2^R$  maximum number of rows, where  $R$  is the maximum number of bits used to index the hash table. Each entry in  $H\_Table[N][L]$  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,  $HC[N]$ , is used to count the number of elements that are mapped to each row of the hash table and *table\_overflow* records the number of overflow elements and 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 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 make the first prefix that matches any packet to be its LPM. Unfortunately, Theorem 1 cannot be used for PH scheme as some prefixes have a different insertion's hash order than their search's hash order.

For example, if a packet  $P_X$  matches prefix  $k_X \in (S18)$  and  $k_Y \in (S16)$  in Figure 6(a), then  $k_X$  is the LPM of  $P_X$ . Assume that during the prefixes mapping that 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  $P_X$  we try all the five hash functions  $r_0 = h_0(P_X), \dots, r_4 = h_4(P_X)$ . Assume that one of the hash functions that were not used to store either  $k_X$  or  $k_Y$  generates the row  $r_Y$  when it is applied to  $P_X$ , 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 as there might be a better matching, which is  $k_X$  in this case. This is why we store the hash function index in PH setup algorithm, Algorithm 4, (line 11).

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  $r_0 = h_0(P), \dots, r_{H-1} = h_{H-1}(P)$  (lines 2–3). 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 5). If we did not find any match, then we search the overflow\_buffer (line 10).

### 4.3 THE INCREMENTAL UPDATES IN PH

In Section 3.4 we talked about the importance of the incremental updates issue for the IP forwarding engine. In this section we describe how to perform the incremental updates for the progressive hashing scheme.

Deleting a certain prefix is straight forward in PH scheme. It involves locating this prefix, deleting it and adjusting the *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  $i = H, \dots, 2H - 1$ . Also, we use only the hash functions that are applicable to the prefix being inserted. Specifically, we replace the lines 2–4 from Algorithm 3 with the following lines:

```
2: for( $i = 0 ; i < H ; i ++$ ) {  
3:   if ( $k_n \in \mathfrak{G}_i$ ), then  
4:      $r_i = h_i(k_n)$  }
```

After that, we decide to which bucket we 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 aforementioned 3 lines.

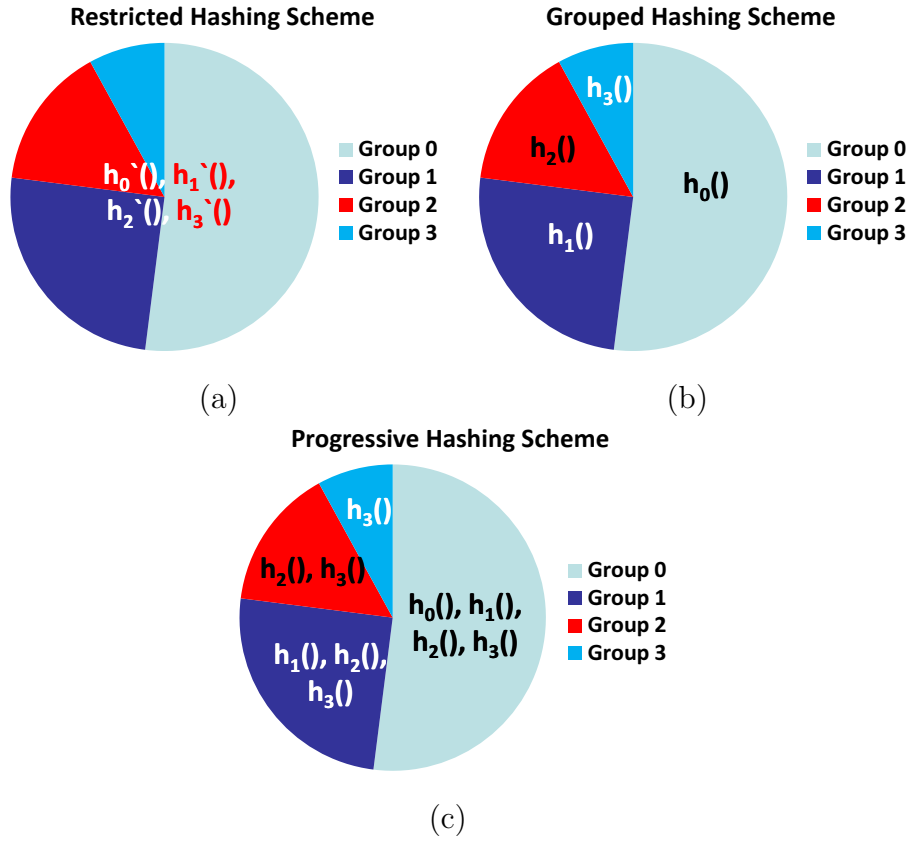


Figure 5: The Evolution of The PH Scheme.

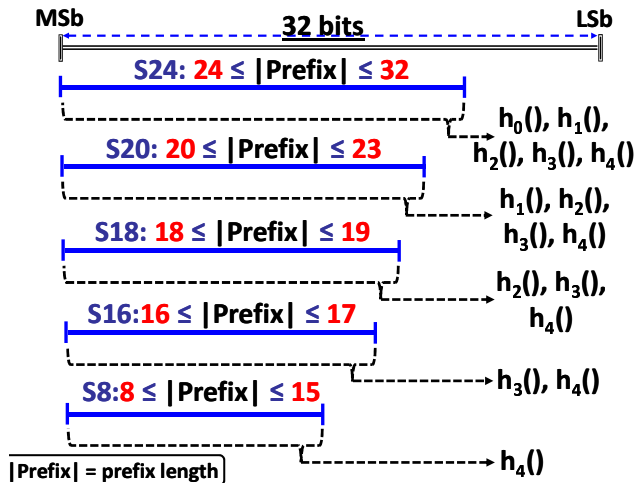


Figure 6: Applying the PH Scheme.

---

**Algorithm 4** The PH Setup Algorithm.

---

```
1: Sort the IP prefixes from longest to shortest and define the groups
2: Initialize  $HC[N]$  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 ( $k_j \in \mathfrak{G}_i$ ), 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 5** The PH Search Algorithm.

---

```
1: Search_Hash_Table(Packet  $P$ )
2:   for( $i = 0 ; i < H ; i ++$ ) {
3:      $r_i = h_i(P)$ 
4:     if( $P$  matches  $H\_Table[r_i][j].key$ ), then
5:       if( $i == H\_Table[r_i][j].h$ ), then
6:         return  $H\_Table[r_i][j].port$ 
7:       else
8:         continue
9:     }
10:   search the  $overflow\_buffer$ 
```

---

## 5.0 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  $\mathbf{H}_3$  [19] hashing families. Those families have the advantage of being simple and fast enough to be easily realized in hardware.

For the evaluation, we collected 15 tables from the Border Gateway Protocol (BGP) Internet core routers of the routing information service project [22] on January 31<sup>st</sup> 2009. Table 1 lists the 15 routing tables, their sizes, and the percentage of prefixes, which we call “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.

IP Table	Size	% Short prefixes	IP Table	% Short prefixes	Size
rrc00	292,717	0.78	rrc10	0.82	276,912
rrc01	276,224	0.82	rrc11	0.82	275,903
rrc02	272,743	0.79	rrc12	0.82	277,132
rrc03	283,147	0.80	rrc13	0.81	280,961
rrc04	283,075	0.81	rrc14	0.82	274,824
rrc05	301,383	0.77	rrc15	0.82	275,828
rrc06	277,555	0.81	rrc16	0.81	280,744
rrc07	274,479	0.83	<b>Average</b>	<b>0.81</b>	<b>280,242</b>

Table 1: The Statistics of the IP lookup tables on January 31<sup>st</sup> 2009.

We define a “configuration” by specifying both  $N$  = the number of rows, and  $L$  = the number of entries per row. The performance of 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  $\alpha$ , the hashing overflow depends on the aspect ratio of the memory  $N/L$ .

In Section 5.1 we evaluate the CHAP scheme and in Section 5.2 we evaluate the PH scheme. Finally, Section 5.3 evaluates the combined scheme of PH and CHAP.

## 5.1 THE EVALUATION OF CONTENT-BASED HASH PROBING

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,  $\lambda$ , and the number of hash functions used,  $H$ , which is also the number of probing pointers per row in CHAP(H,H). Intuitively, if  $\lambda$  is small, then the setup algorithm (Algorithm 1) may not be able to eliminate the overflow. On the other hand, if  $\lambda$  is large, then Algorithm 1 may terminate with every *OC* having a value smaller than  $\lambda$ , 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 analysis of CHAP(H,H) for  $\lambda$ .

### 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 Figure 7. In addition, these two configurations have a high average load factor  $\alpha = 98.5\%$  for the “rrc07” table, which articulates the strength of CHAP.

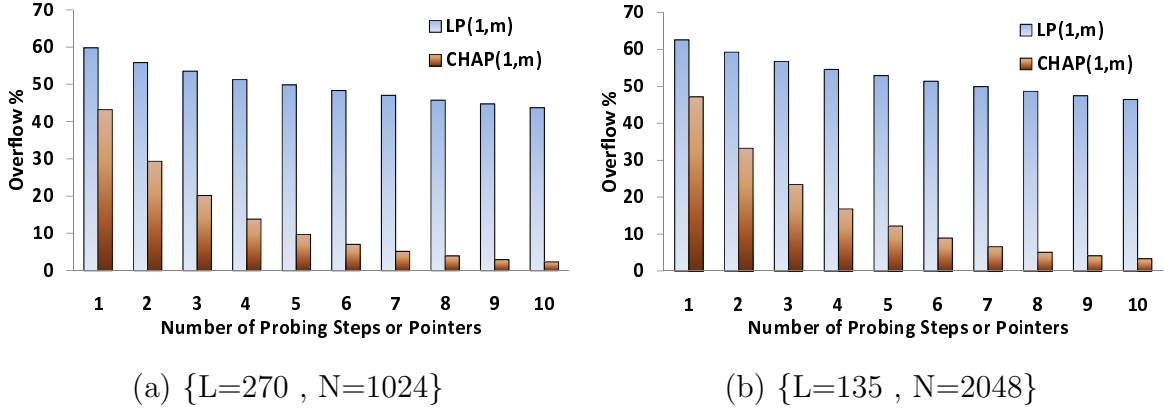


Figure 7: Overflow of CHAP(1, m) vs. Linear Probing(1, m) for table rrc07.

Figure 7 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  $\lambda$  in the CHAP setup algorithm (Section 3.2). We report the results for table “rrc07” since all other tables have 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  $H = 3$ .

Figures 8(a) and (b) show the values of overflow versus  $\lambda$  for the two groups. For Figure 8(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 Figure 8(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  $\lambda \in [0, L]$ .

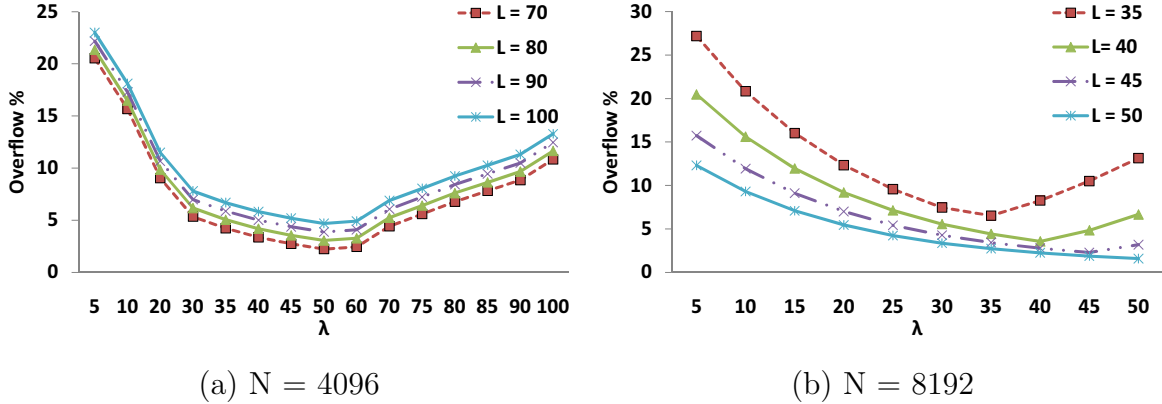


Figure 8: The overflow vs.  $\lambda$ .

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 Figure 8(a), which is an indication that the average hole size in the hash table is equal to  $\frac{L}{2}$ . For larger values of  $\lambda$ , the maximum hole size becomes smaller than  $\lambda$  and thus we are unable to insert all the elements that were counted by *OC* into the hash table. This increases the overflow. For Figure 8(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) between 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.

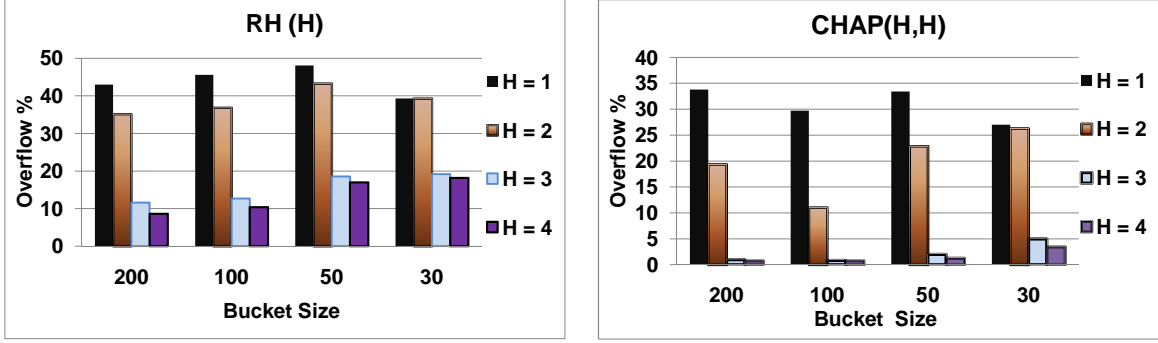


Figure 9: Average overflow of 4 bucket widths for RH(H) and CHAP(H,H) for table rrc05.

In Figure 9 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  $16384$  where  $L = 200, 100, 50$  and  $30$ , respectively. It is obvious from Figure 9 that CHAP(H,H) has much less overflow than multiple hashing for the same number of hash functions. On average CHAP(H,H) is 49.0% lower than RH(H) over all four bucket sizes.

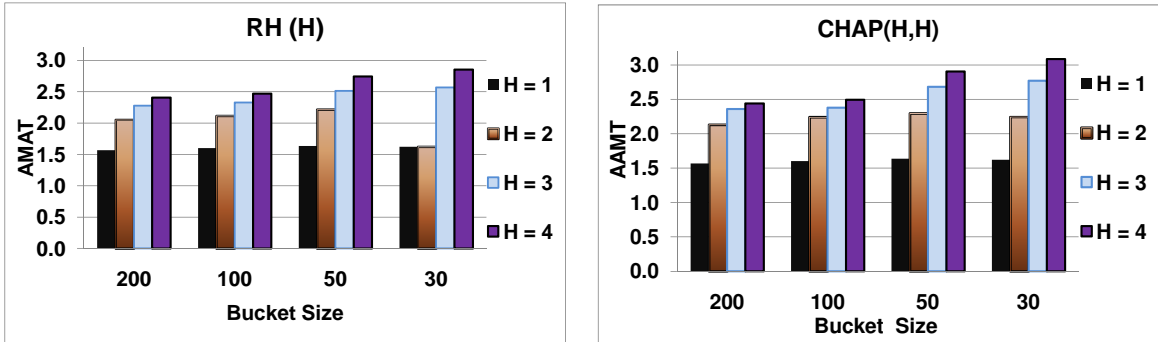


Figure 10: Average AMAT of 4 bucket widths for RH(H) and CHAP(H,H) for table rrc05.

The results shown in Figure 10 indicates that the average AMAT over the four bucket sizes for RH(H) is 2.16, while it's 2.28 for CHAP(H,H) which is only 5% higher than RH. We note here that both CHAP and RH schemes 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 access memory 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 tradeoff between the overflow and the average memory access time can be seen in Figures 9 and 10. However, a better understanding of the tradeoff that CHAP and RH present can be obtained by comparing CHAP( $H,H$ ) with RH( $2H$ ) since both has  $2H$  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 average access time compared to RH( $2H$ ), we plot in Figure 11 (a) the overflow and (b) the average memory access time of both schemes for one configuration C1: $\{L = 180, N = 2048\}$ . For this experiment we map each of the 15 IP tables into a fixed hash table of 368,640 entries.

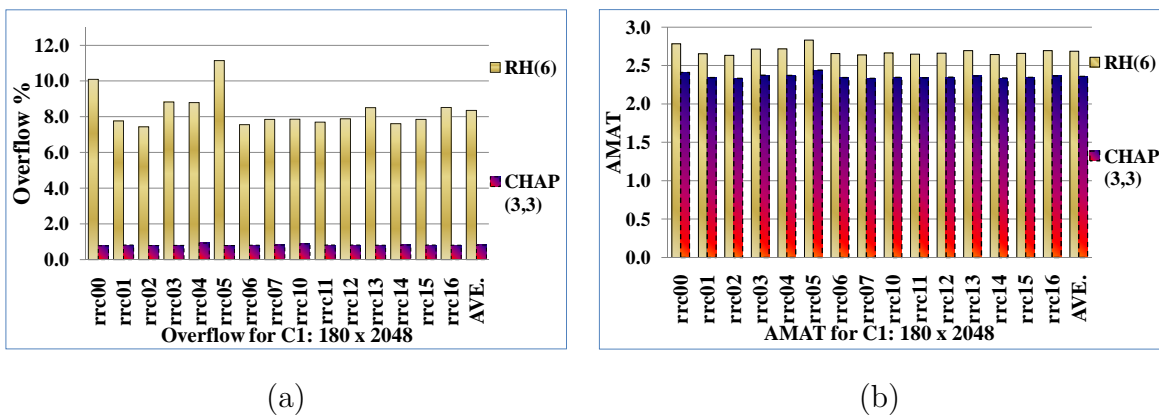


Figure 11: (a)Average Overflow and (b) AMAT for CHAP( $3,3$ ) vs. RH( $6$ ) for 15 Lookup Tables for C1:  $\{L = 180, N = 2048\}$

Note that we represent the average of all files as another independent point that is called AVE in Figure 11. 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 90.2% and at the same time improves the AMAT by 12.2% for this configuration.

To evaluate the CHAP scheme 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 Figure 12(a) and (b). Figure 12(a) shows the average overflow over all the 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. Figure 12(b) shows the AMAT of the same three configurations for RH(6) and CHAP(3,3).

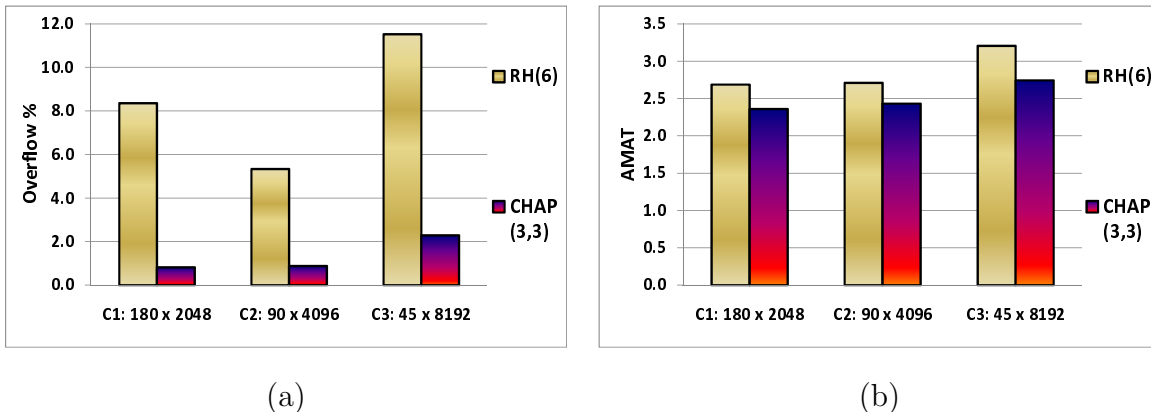


Figure 12: (a) Average Overflow and (b) Average AMAT for CHAP(3,3) vs. RH(6) for 3 configurations.

For the three configurations, CHAP(3,3) reduces the overflow by 90.2%, 83.5% and 80.1% respectively over RH(6). At the same time, CHAP(3,3) improves the AMAT over the RH(6) for these configurations by 12.2%, 10.2% and 14.4% respectively.

## 5.2 THE EVALUATION OF PROGRESSIVE HASHING

In this section we compare the PH scheme 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 Figure 1 for both GH(5) and PH(5) schemes.

Figure 13(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, the PH reduces the overflow by 86.5% compared to the RH scheme and by 66.9% compared to the GH scheme. At the same time, the AMAT (Figure 13(b)) of the PH is improved by 22.0% over the RH scheme and 3.4% over the GH. Note that the overflow prefixes are added to the overflow\_buffer that is searched after exhausting all possible buckets in the CA-RAM.

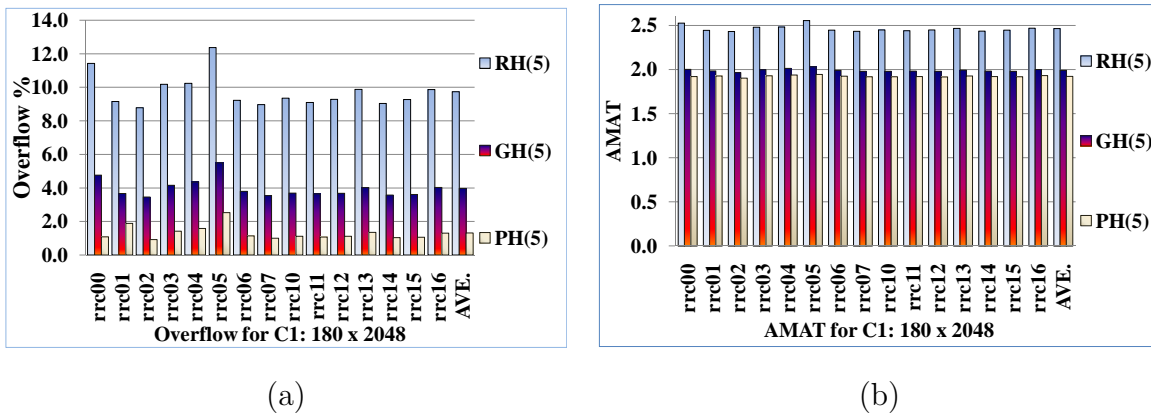


Figure 13: (a) Average Overflow (b) AMAT of RH(5) vs. GH(5) vs. PH(5) for 15 Lookup Tables for  $C1: \{180 \times 2048\}$ .

To show that the PH scheme 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 Figure 14(a) and (b). Figure 14(a) shows the average overflow over all the 15 lookup tables for the three schemes RH, GH and PH. Figure 14(b) shows the AMAT for the same configurations and schemes.

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

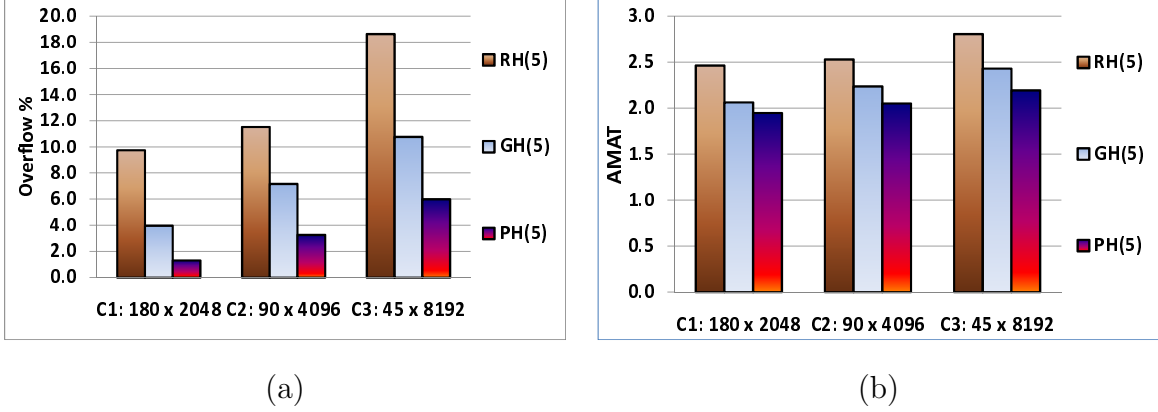


Figure 14: (a) Average Overflow and (b) Average AMAT for of RH(5) vs. GH(5) vs. PH(5) for 3 configurations.

### 5.3 APPLYING CONTENT-BASED HASH PROBING TO PH

In this section we will 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 progressive hashing scheme instead of RH scheme to get better performance than both schemes.

In Figure 15(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 5.2 and 5.1. The largest average overflow belongs to PH(5) scheme with 3.5% 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) scheme with a reduction of 100%. For the third configuration, C3, the PH\_CHAP(5,5) combined scheme reduced the overflow by 87.5% 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 (Figure 15(b)) than CHAP(5,5) and PH(5) schemes with an average of 19.7% improvement over CHAP(5,5) and of 2.3% improvement over PH(5). The improvement in the AMAT comes from the fact that the combined scheme, PH\_CHAP(5,5), uses PH to reduce the overflow in addition to the

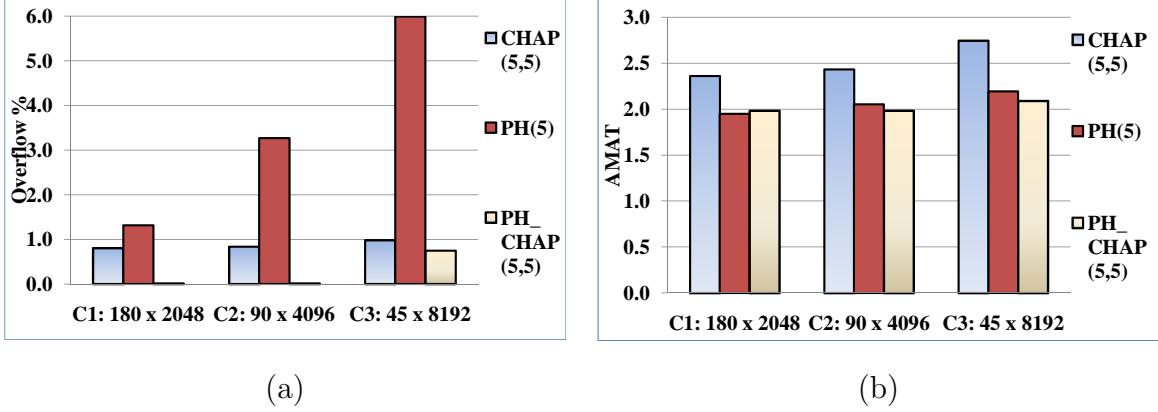


Figure 15: (a) Average Overflow and (b) Average AMAT of CHAP(5,5) vs. PH(5) vs. PH.CHAP(5,5) for 3 Configurations.

probing pointers, in contrast to the fact that CHAP(5,5) 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.

### 5.3.1 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. We estimate the memory requirements for one configuration C1:  $\{L = 180, N = 2048\}$  as an example for these three schemes. In general, C1 requires  $180 \times 2048 = 368,640$  or 370K entry, where an entry is represented by 5 bytes prefix plus 5 bits prefix length and there is 1 byte per row for the row counter.

For the PH(5) scheme we need 3 bits per entry for the hash function index. Thus, the total memory requirement for PH(5) for the configuration C1 is  $\sim 2.12$  MB. For the CHAP(5,5) scheme we add 5 pointers per row where each pointer 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  $\sim 2.00$  MB for the configuration C1. Finally, for the sam C1 configuration, the combined PH\_CHAP(5,5) scheme needs  $\sim 2.13$  MB.

## 6.0 CONCLUSIONS AND FUTURE WORK

In this thesis we have described and studied 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 illustrated that both schemes can be realized in hardware by taking advantage of state-of-the-art search memory architectures. In this work we use simple hash functions that can be easily realized in hardware. We provided simple setup and incremental update algorithms for both schemes.

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 improves the average memory access time over the restricted multiple hash function scheme while reducing the overflow.

While we introduce Progressive Hashing as a new open addressing hash-based packet processing scheme, it can also work for closed addressing hash systems. Progressive hashing is effective in reducing the classical hashing overflow by 93% on average over the restricted hashing.

A state-of-the-art CMOS technology SRAM memory design [28] reports of a single chip of 36.375 MB that runs on 4.0GHz. Since our scheme depends on a set associative RAM, we conservatively assume that the clock rate is 2.0 GHz. If we assume AMAT of  $\sim 2.0$  (according to Figure 14(b)), then we have a forwarding speed of 2.0 Giga packets per seconds or 640 Gbps for the minimum packet size of 40 bytes.

The future work includes applying both schemes to other packet processing applications such as Packet Classification (PC). In addition, we plan to introduce other optimizations to

reduce the worst case memory access time of both CHAP and PH schemes. Finally, we will study a fully synthesized PH.CHAP(H,H) packet forwarding engine.

## BIBLIOGRAPHY

- [1] Yosi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. pages 593–602. ACM SOC, 1994.
- [2] Florin Baboescu, Dean M. Tullsen, Grigore Rosu, and Sumeet Singh. A tree based router search engine architecture with single port memories. volume 1, pages 123–133. IEEE ISCA, 2005.
- [3] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. pages 64–74. IEEE Infocom, July 2003.
- [4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.
- [5] Sangyeun Cho, Joel Martin, , and Rami Melhem. Ca-ram: A high-performance memory substrate for search-intensive applications. pages 230–241. Ispass’07, April 2007.
- [6] T. Cormen, C. Leiserson, R. Rivest, and C. Stien. *Introdcution to Algorithms*. McGraw Hill, 2003.
- [7] Michel Hanna, Socrates Demetriades, Sangyeun Cho, and Rami Melhem. An efficient hardware-based multi-hash scheme for high speed ip lookup. pages 103–110. IEEE HOTi, August 2008.
- [8] Michel Hanna, Socrates Demetriades, Sangyeun Cho, and Rami Melhem. Chap: Enabling efficient hardware-based multiple hash schemes for ip lookup. IFIP Networking, IFIP, May 2009.
- [9] Michel Hanna, Socrates Demetriades, Sangyeun Cho, and Rami Melhem. Progressive hashing for packet processing using set associative memory. IEEE/ACM ANCS, ANCS, October 2009.
- [10] G. Huston. Analyzing the internet’s bgp routing table. *The Internet Pro. J.*, 2001.
- [11] Weirong Jiang and Viktor K. Prasanna. A memory-balanced linear pipeline architecture for trie-based ip lookup. pages 83–90. HOTi’07, August 2007.

- [12] S. Kaxiras and G. Keramidas. Ipstash: A power-efficient memory architecture for ip-lookup. pages 361–373. *IEEE Micro*, November 2003.
- [13] A. Kirsch and M. Mitzenmacher. Simple summaries for hashing with multiple choices. *IEEE/ACM Trans. on Net.*, 2007.
- [14] H. Noda K. Inoue H. J. Mattausch T. Koide and K. Arimoto. A cost-efficient dynamic ternary cam in 130nm cmos technology with planar complementary capacitors and tsr architecture. Proc. Int’l Symp. VLSI Circuits, June 2003.
- [15] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary cams. pages 193–204. ACM Sigcomm, May 2005.
- [16] H. Noda and *et al.* A cost-efficient high-performance dynamic team with pipelined hierarchical searching and shift redundancy architecture. *IEEE J. Solid-State Circuits*, 40(1):245–253, January 2005.
- [17] R. Pagh and F. Rodler. Cuckoo hashing. *Lec. Notes in Comp. Sci.*, pages 121–133, 2001.
- [18] R. Panigrahy and S. Sharma. Reducing team power consumption and increasing throughput. pages 107–112. IEEE HOTi, August 2002.
- [19] M. Ramakrishna and et Al. Efficient hardware hashing functions for high performance computers. *IEEE Trans. on Comp.*, 46(12):1378–1381, December 1997.
- [20] B. Randell. A note on storage fragmentation and program segmentation. *Comm. of the ACM*, 12:365–372, July 1969.
- [21] Y. Rekhter and T. Li. An architecure for ip address allocation with cidr. *RFC*, 1993.
- [22] RIS. Routing information service. <http://www.ripe.net/ris/>, 2009.
- [23] D. Shah and P. Gupta. Fast updating algorithms for tcams. *IEEE Micro Mag.*, 21(1):36–47, Jan./Feb. 2001.
- [24] Haoyu Song and et Al. Fast hash table lookup using extended bloom filter: An aid to network processing. pages 181–192. ACM Sigcomm, August 2005.
- [25] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, 17(1):1–40, 1999.
- [26] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2005.
- [27] B. Vocking. How asymmetry helps load balancing. *ACM J.*, pages 568–589, July 2003.

- [28] Y. Wang, U. Bhattacharya, F. Hamzaoglu, P. Kolar, Y. Ng, L. Wei, Y. Zhang, K. Zhang, and M. Bohr. A 4.0 ghz 291 mb voltage-scalable sram design in 32nm high-k metal-gate cmos with integrated power management. In *IEEE ISSCC*, pages 456–457, 2009.
- [29] SangKyun Yun. Hardware-based ip lookup using n-way set associative memory and lpm comparator. *Lecture Notes in Computer Science (LNCS)*, 4017/2006:406–414, 2006.
- [30] F. Zane, G. Narlikar, and A. Basu. Coolcams: Power-efficient tcams for forwarding engines. pages 42–52. *IEEE Infocom*, April 2003.