Progressive Hashing for Packet Processing Using Set Associative Memory.

Michel Hanna
Dept. of Comp. Sci.,
University of Pittsburgh
mhanna@cs.pitt.edu

Socrates Demetriades
Dept. of Comp. Sci.,
University of Pittsburgh
socrates@cs.pitt.edu

Sangyeun Cho
Dept. of Comp. Sci.,
University of Pittsburgh
cho@cs.pitt.edu

Rami Melhem
Dept. of Comp. Sci.,
University of Pittsburgh
melhem@cs.pitt.edu

ABSTRACT

As the Internet grows, both the number of rules in packet filtering databases and the number of prefixes in IP lookup tables inside the router are growing. The packet processing engine is a critical part of the Internet router as it is used to perform packet forwarding (PF) and packet classification (PC). In both applications, processing has to be at wire speed. It is common to use hash-based schemes in packet processing engines; however, the downside of classic hashing techniques such as overflow and worst case memory access time, has to be dealt with. Implementing hash tables using set associative memory has the property that each bucket of a hash table can be searched in one memory cycle outperforming the conventional Ternary CAMs in terms of power and scalability.

In this paper we present “Progressive Hashing” (**PH**), a general open addressing hash-based packet processing scheme for Internet routers using the set associative memory architecture. Our scheme is an extension of the multiple hashing scheme and is amendable to high-performance hardware implementation with low overflow and low memory access latency. We show by experimenting with real IP lookup tables and synthetic packet filtering databases that PH reduces the overflow over the multiple hashing. The proposed PH processing engine is estimated to achieve an average processing speed of 160 Gbps for the PC application and 320 Gbps for the PF application.

Categories and Subject Descriptors

C.2.6 [**Internetworking**]: Routers

General Terms

Architecture, Design, Performance

Keywords

Packet Forwarding, Packet Classification, Hardware Hash

1. INTRODUCTION

High speed Internet routers and firewalls require wire speed packet forwarding and filtering while the sizes of the filter sets are increasing at a high rate [7, 25, 26]. In addition, the advancement of optical networks keeps pushing the link rates already beyond 40 Gbps [17, 26].

In packet forwarding (**PF**), the destination address of every incoming packet is matched against a routing table to determine the packet's next hop on its way to the final destination. An entry in the forwarding table, or an IP prefix, is a binary string of a certain length (prefix length) followed by don't care bits and an associated port number. The actual matching requires finding the “Longest Prefix Matching” (LPM) as instructed in the CIDR protocol [15].

In packet classification (**PC**), a packet header is matched against a database of rules, or filters, to obtain the best matching rule. There is a priority tag that is appended to each rule and the packet classifier must return the rule with the highest priority as the best matching rule in case of multiple matches. Each filter consists of multiple field values. The number of fields per rule and the number of bits associated with a field are variable and depend on the application. Typically, filtering is applied to the following fields: IP source address, IP destination address, source port, destination port and the protocol. IP addresses are represented by prefixes, which calls for prefix matching, while ports are encoded as ranges, which requires range matching. Finally, the “protocol” field, uses exact matching. At any field, it is possible to specify a wild card to indicate that no matching is needed for that field. Rules are associated with either an “action” (e.g., firewall) or a “flow ID” (e.g., QoS).

Packet classification is a more complex process than packet forwarding. This complexity arises from two factors: (1) more fields need to be matched and (2) different kinds of matching are required for each field. Packet forwarding, on the other hand, is simpler since it tackles a single field and a single matching type. A few researchers introduced schemes that can be applied to both applications [2, 11, 18].

The two main streams of PF and PC research are: algorithmic and architectural. Some researches propose algorithmic solutions for PF and PC and provide space and time complexity bounds [8, 6, 9, 21]. These solutions were subsequently modified and extended to enhance their performance [17, 18, 24]. Some of these solutions are amenable to hardware implementations [24]. This motivated the introduction of architectural solutions that mostly rely on the Ternary Content Addressable Memory (TCAM) technology [10, 20]. A TCAM is a fully-associative memory that can store binary values, 0's and 1's as well as don't care bits. TCAMs have been the *de facto* standard for packet processing in industry [12, 25]. However, TCAM comes with significant deficiencies: high power consumption, low bit density, poor scalability to long input keys and higher cost per bit compared to other memories.

Architectural solutions based on hashing have also been proposed [5, 11, 18]. Hash tables come in two flavors: closed addressing hash (or *chaining*) and open addressing hash. The hash table in closed addressing hash has a fixed height (number of buckets), and each bucket is an infinite size linked list. In open addressing, the hash table has a fixed height and a fixed bucket width. The overflow in open addressing hashing is handled through probing [4]. The authors in [21] introduce a special type of hash-based packet classification scheme that is called Tuple Space Search (TSS). In TSS, the five PC fields are represented by a "tuple" which is simply a group of integers substituting the actual fields. A prefix is presented by an integer that equals the number of its defined bits, each range is converted to two integers using range encoding and each protocol is presented by an integer. Although TSS has been extensively studied theoretically, few papers addressed the actual implementation issues of TSS such as overflow handling, memory efficiency and memory delay. The work by Song et al. [19] is the first to show how an actual "coarse-grained" TSS system can be built by splitting the $2D$ TSS into coarse clusters. The work in [11] enhances that of [19] by using multiple hash tables, one table per hash function. In general, the closed addressing hash scheme suffers from low space utilization and the use of a relatively large number of hash functions [11, 18].

In this paper, we assume open addressing hash schemes for which a number of efficient hardware prototype implementations have been proposed recently [3, 9]. 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. The overflow problem is treated by first splitting the keys into groups based on their lengths and for each group a hash function is assigned. Our new idea is to define the groups and the hash functions in a way that allows us to reuse the hash functions on different categories in a progressive way, hence the name Progressive Hashing or PH. We enhance the average search time of the PF by sorting the lookup tables and by applying during the search the same hash function order used for insertion. We also introduce an optimization that we call "I-Mark" where we distinguish keys into two sets: independent and dependent; then we insert the independent set at any order to reduce the overflow for both the PC and PF applications. In addition, the I-Mark can be used to also enhance the average search time for PC. Our goal is to fit the packet processing database in a single fixed size hash table with minimal overflow, high space utilization and low average memory access time.

The rest of the paper is organized as follows: Section 2 gives a brief background on open addressing hashing and state-of-the-art set associative memory architectures. In Section 3 we describe our main scheme PH. Optimizations to the PH are discussed in Section 4 and Section 5. We show the experimental results and evaluation in Section 6. Finally, we conclude and talk about 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*, inserting an element into a hash table is done by testing each bucket for a free space starting at the hash index generated by this element. Another way is to use a second order equation of the generated hash index to specify which bucket should be tested, which is called *quadratic probing*. Both probing schemes suffer from "primary" and "secondary key clustering" respectively [4].

Instead of probing, one can apply a second hash function to find an empty bucket, which is known as *double hashing* [4]. In general, the use of $H \geq 2$ hash functions is shown to be better in eliminating hash overflow than probing [1]. Given a hash table with M records and N buckets, the average number of hash table accesses to find a record is heavily affected by the choice of hash function(s), the number of keys per bucket, L, and the load factor, α, defined as $M/(N \times L)$. With a smaller α, the average number of hash table accesses can be made smaller at the expense of unused memory space.

2.2 Set Associative Memory Architectures

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 [3, 9]. 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 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. The matching processors are programmable and the format of each memory row is flexible. The left corner of Figure 1 shows how a row may be divided into entries (cells) to store prefixes, their length and their port number for the PF application [3, 8].

3. PROGRESSIVE HASHING

The predominant issue of any hashing system is the collision handling method. In this section, we propose the

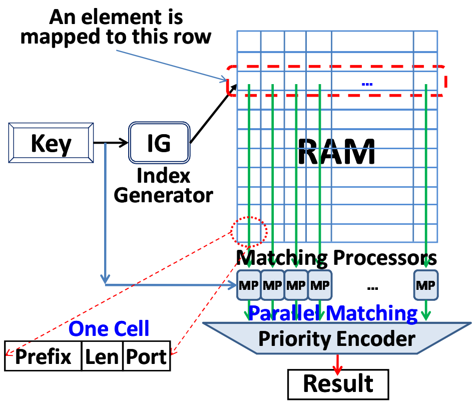


Figure 1: The basic CA-RAM Architecture.

PH scheme as an effective mechanism for reducing collisions (hence overflow) for open-addressing hash systems. The PH scheme can also be applied to closed-addressing hash systems. As we mentioned in Section 2, using multiple hash functions is efficient in reducing collisions. For this reason we use the multiple hash scheme throughout this paper. In general, using different hash tables for different hash functions is a valid design option; however, using different hash tables leads to memory fragmentation and poor space utilization. To achieve high space utilization (the ratio between the required memory to store the database and the capacity of the actually used RAM) we apply multiple hash functions on a single hash table (per application). Specifically, a key is inserted in the hash table using any of the H hash functions.

In Section 3.1 we discuss the issues of hashing with wildcards and in Section 3.2 we give the details of the PH scheme and in Section 3.3 we talk about the search in PH. Finally Section 3.4 talks about the incremental updates.

3.1 Multiple Hashing with Wildcards

In both applications, PF and PC, wildcards (or don't care) bits are heavily present in the routers' databases. Hashing with wildcards requires one of the two solutions: restricted hashing or grouped hashing, as described next.

3.1.1 *Restricted Hashing*

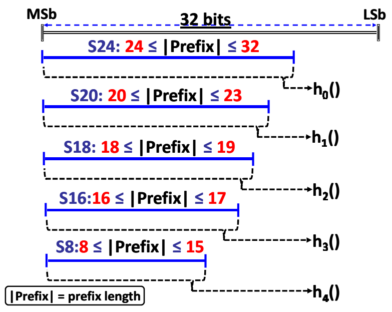
In restricted hashing **RH**, the hash functions are restricted to be using only the non-wildcard bits of the keys. For example, in the PF case, prefixes can be either expanded [22] to increase the number of non-wildcard bits or use only a specific prefix length, say 16 bits, for hashing. In the latter case, the shorter prefixes are kept in a small fast memory that is searched in parallel with the main lookup table [8]. On the other hand, the RH scheme cannot be used in PC since some rules have all wildcards bits in one or two of the source or the destination fields; expanding those rules results in huge database.

3.1.2 *Grouped Hashing*

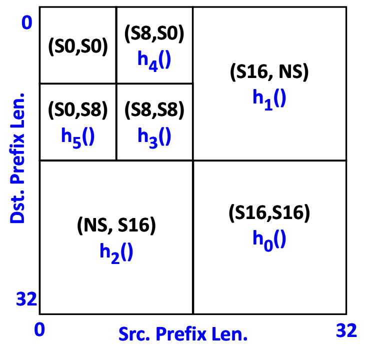
In grouped hashing **GH**, keys are grouped based on their prefix lengths, then a different hash function is applied to each group. For example, for PF, the 32 bit IPv4 wide address space can be split into 5 groups as follows:

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

Then, each group can be associated with a different hash function. For example, $h_0()$ that uses 24 bits can be associated with group $S24$, $h_1()$ that uses 20 bits can be associated with group $S20$, $\cdots$, and $h_4()$ that uses 8 bits can be associated to group $S8$. This scheme is similar to the one used in [9]. Figure 2(a) shows the five groups and their associated hash function. We represent the 32-bit address space with a 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 in parallel with the main hash table.



(a)



(b)

Figure 2: Splitting the Hashing Space into Groups for: (a) PF and (b) PC.

Grouped hashing can be also applied to PC using the tuple space concept. For example, a coarse-grained tuple space [19] can divide the PC hashing space (or keys to be hashed) into 7 groups as shown in Figure 2(b). First, the filters are split into 4 groups based on the source and the destination prefixes lengths: $(S16, S16)$, $(S16, NS)$, $(NS, S16)$ and (NS, NS), where "S16" means that the prefix has 16 specific bits or more, while the "NS" stands for "Non-Specific", i.e., the prefix is less than 16 bits. The (NS, NS) group is then split once more into 4 groups: $(S8, S8)$, $(S8, S0)$, $(S0, S8)$ and $(S0, S0)$ where "S8" means that the prefix is less than 16 and greater than or equal to 8 bits long and "S0" includes all the prefixes that are less than 8 bits long. There are many options to classify the rules in each PC database. However we pick this example because of its simplicity.

It is possible to associate a different hash function to each group as follows:

- Group $(S16, S16)$ uses the hash function $h_0()$ which uses the 16 most significant bits from each dimension and merge them to make one word of 32 bits to compute the hash index.
- Groups $(S16, NS)$ and $(NS, S16)$ use hash functions $h_1()$ and $h_2()$ respectively, where each function uses the 16 most significant bits of either the source or destination prefixes to compute the hash index.
- Group $(S8, S8)$ uses the hash function $h_3()$ which takes the 8 most significant bits from each dimension and merges them to make one word of 16 bits to compute the hash index.
- Groups $(S8, S0)$ and $(S0, S8)$ use hash functions $h_4()$ and $h_5()$ respectively, where each function uses the 8 most significant bit of either the source or the destination prefixes to compute the hash index.
- Group $(S0, S0)$ is added to the overflow_buffer since it usually contains very few rules.

3.2 Using the PH Scheme

In the previous sections we described the two multiple hashing schemes for packet processing, which are abstracted in Figures 3(a) and (b) where the hashing space is represented as a circle. For the restricted hashing scheme (Figure 3(a)) the hash functions $h'_0() \cdots h'_3()$ are applied to all the keys in the hashing space. For the grouped hashing (Figure 3(b)) we split the hashing space into 4 groups, Group 0, to Group 3 and a single hash function is associated with each group as follows: $h_0()$ is associated with Group 0, $\cdots$, and $h_3()$ is associated with Group 3.

Since the restricted hashing scheme cannot be applied to the PC application, we consider further the grouped hashing scheme. Note that the grouped hashing scheme's search algorithm has to try all the hash functions of all the groups to find the key with the highest priority that matches a certain packet. This means that for each incoming packet, the grouped hashing search algorithm will access the memory H times if there are H groups. Also, since the groups are based on the prefix lengths, groups with longer prefix length can use the hash functions of other groups that have shorter prefix lengths. For example, in the PF application of Figure 2(a), group $S24$ can use the hash functions of groups $S20$ and $S16$. Motivated by these two observations, we propose our main scheme, Progressive Hashing (PH) where we apply the hash functions in a progressive manner to other groups as illustrated in Figure 3(c) to give some keys more chances to be mapped to the hash table to reduce 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" between the groups. To clarify, note that a key in a packet processing

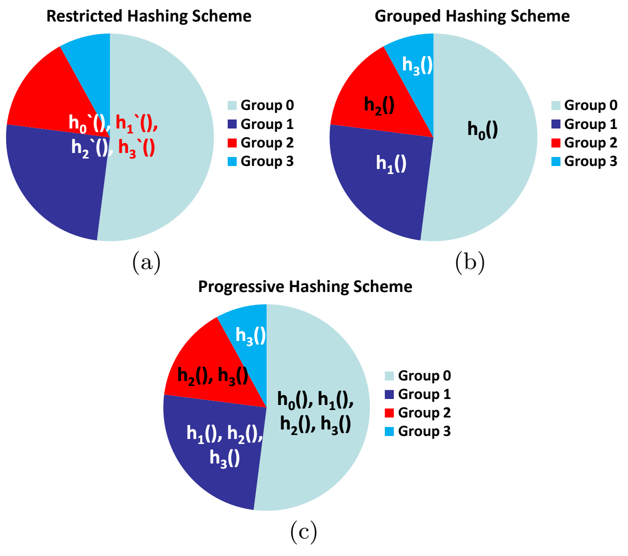


Figure 3: The Evolution of The PH Scheme.

application can be one of two things: a single prefix in case of the PF application (destination prefix) or two prefixes in case of the PC application (source and destination prefixes). We define the "hash function specificity" as follows:

DEFINITION 1. *In packet processing applications, 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 $h_i(\cdot)$.*

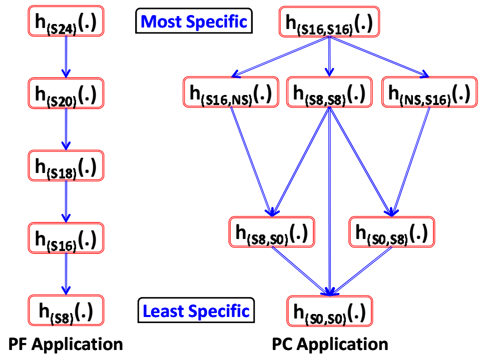


Figure 4: The Hash Function Specificity Hierarchy of PF and PC Applications.

For example, the PF hash function $h_{S24}(\cdot)$ in Figure 4 is more specific than $h_{S20}(\cdot)$ and the PC hash function $h_{(S16,S16)}(\cdot)$ in Figure 4 is more specific than $h_{(S8,S8)}(\cdot)$. Note that for the PC case in Figure 4, we cannot define a specificity relationship between the two hash functions $h_{(S16,NS)}(\cdot)$ and $h_{(NS,S16)}(\cdot)$ nor between $h_{(S8,S0)}(\cdot)$ and $h_{(S0,S8)}(\cdot)$. Figure 4 is a directed graph that represents the hierarchy between different hash functions for the two applications. Specifically, there is a directed edge between group pairs that have a direct specificity relation. Note that only in this graph we replaced the hash functions subscript numbers, such as $h_0(\cdot)$, with the actual group name to eliminate any ambiguity. For example, the PF $h_0(\cdot)$ is $h_{S24}(\cdot)$, while $h_0(\cdot)$ of the PC is $h_{(S16,S16)}(\cdot)$.

Progressive hashing for the PF application is based on the observation that group $S24$ can use the less specific hash functions of the groups $S20$, $S18$, $S16$ and $S8$, and the same for group $S20$ which can use the less specific hash functions of the groups $S18$, $S16$ and $S8$, $\cdots$, etc. This is illustrated in Figure 5(a). Similarly, for the PC application, group $(S16, S16)$ can use other less specific hash functions of the groups $(S16, NS)$, $(NS, S16)$, $(S8, S8)$, $(S8, S0)$, and $(S0, S8)$, and the same applies for group $(S16, NS)$ which can use hash functions of the less specific group $(S8, S0)$, $\cdots$, etc., as illustrated in Figure 5(b).

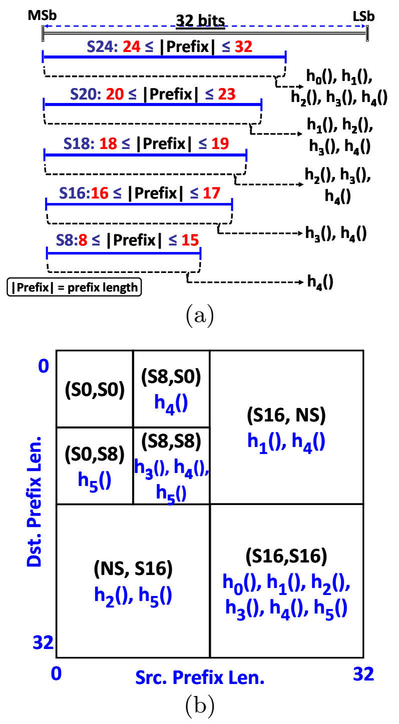


Figure 5: Applying PH on: (a) PF and (b) PC.

After dividing the keys into groups as we saw, we run the setup algorithm, Algorithm 1, to store the keys into the hash table. In the setup algorithm, $j = 0, \cdots, M-1$ is used to index the keys, where M is the total number of prefixes/rules in a database. The goal is to map the keys into a hash table, H_Table[row index][bucket size], with L = 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 multiple fields depending on the application. For example, in the PF case we store the actual prefix, its length (or its mask) and the port number, while in the PC case we need to store the source and destination addresses prefixes, ports, action and so on. H is the maximum number of hash functions and an array of counters, HC[row index], is used to count the number of elements that are mapped to each row of the hash table and we use a counter OC to count the overall overflow. Group number 'i' is represented by $\mathfrak{G}_i$.

Algorithm 1 attempts to allocate K_j, (line 5) in the hash table, if the attempt is not successful, it stores the key in the overflow_buffer that is searched in parallel with the main hash table. The amount of overflow to be handled depends on the overflow_buffer capacity.

Algorithm 1 The PH Setup Algorithm.

```
 1: Initialize $HC[N]$ array to zeros and $OC = 0$
 2: **for**($j = 0; j < M; j++$) {
 3:     define binary flag: $inserted = false$
 4:     **for**($i = 0\ ;\ i < H\ ;\ i++$) {
 5:         **if** ($K_j \in \mathfrak{G}_i$), **then** {
 6:             $r_i = h_i(K_j)$
 7:             **if**($HC[r_i] < L$), **then** {
 8:                 H_Table[$r_i$][$HC[r_i]$].key = $K_j$
 9:                 $HC[r_i]++$, $inserted = true$ }
10:         }
11:     }
12:     **if**($inserted == false$), **then**
13:         Store $K_j$ in overflow_buffer, $OC$++ }
```

3.3 Searching in PH

In this section we show how to search for a key in the progressive hashing. The goal is to find the highest priority key that matches any given packet since we might find multiple matches. For PF, the priority is simply the length of the prefix, while PC assigns a priority tag to each rule.

The general PH search algorithm works as follows: for each packet P_x that arrives at the packet processing unit, in the first cycle we calculate the row index addresses $r_0 = h_0(k_x), \cdots, r_{H-1} = h_{H-1}(k_x)$, where k_x represents the header of the packet P_x. For each row r_i we match P_x against all the elements in that row in parallel in a single clock cycle using the matching processors. In case of PC we have a different matching processor for each field such as prefix matching processors, range matching processors, etc. The matching processors calculate the best matching key in the bucket and store the intermediate key. After H cycles, there might be a maximum of H intermediate matching keys. In the final cycle, the best matching key among the matched intermediate keys is identified. In case there is no match in the hash table, we search the overflow_buffer for a match or we apply the default rule or route.

Each successful search requires $H + 2$ cycles to calculate the best matching key (1 cycle to calculate hash indices + H cycles for the H memory accesses of the H rows + 1 cycle to calculate the best match). In Section 4, we discuss an optimization that improves the average memory access time (AMAT) for the PF application and in Section 5 we give another optimization that improves the AMAT for PC.

3.4 The Incremental Updates

The incremental update operations in PH are straightforward. An incremental update operation can be either key deletion, key addition or key update.

To delete a key is simple; just search for the key in the hash table then delete it and update the row counters accordingly. To insert a new key or to update the information of an existing key K_N we take the following steps: (1) first we try to find if K_N already exists by searching through maximum of a H buckets. (2) If it exists, then we update its information. For example, for PC, an action of an existing rule is needed to be changed from accept to drop or vice versa or we need to assign a new port for an existing prefix in case of PF. (3) If we do not find K_N in the hash table, then we try to find a row with an empty spot to store it using one of the H hash functions. If there is no space, then we append K_N to the overflow_buffer.

4. IMPROVING PF AMAT

We show in Section 3.3 that finding the best matching key for a given packet requires searching through all the H hash functions. In this section we seek to improve the average search time by guaranteeing that the first matching key is the best matching key. In other words, for the PF applications, the first matching prefix will be the longest matching prefix (LPM) under certain constraints. In the following subsection we describe how such optimization can be applied for any RH system and then we extend it to PH.

4.1 Stop at First Matching in RH

To be able to stop at the first matching prefix during search in a PF restricted multiple hashing system, 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 apply the hash function in a consistent order ("hash order") during both insertion and search time. Theorem 1 proves that these two conditions are enough to find the LPM first.

THEOREM 1. *In restricted multiple hashing's search algorithm, the first matching prefix is the LPM if:*

1. *The prefixes are inserted from the longest to shortest.*
2. *The search's hash order 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 the groups of 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: $h'_0(K_m)$, $\cdots$, $h'_{H-1}(K_m)$, $\forall K_m$ and $0 \leq m \leq M-1$. 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 $h_l(K_X) = h_l(K_Y)$ = row index 'l'. We can see that it is not possible for K_Y to find a space in row 'l' if K_X could not find a place.

This means that if $h_i(K_X) = X$ and $h_j(K_Y) = Y$, then $i < j$. Hence while searching for a match for P_X in the order: $h_0(P_X)$, $\cdots$ $h_{H-1}(P_X)$, we will match K_X at row X before we go to row Y. $\square$

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

4.2 Stop at First Matching in GH and PH

Unfortunately Theorem 1 cannot be used for both GH and PH schemes 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 5(a), then K_X is the LPM of P_X. Assume that during the prefixes mapping, the two prefixes are stored in two different rows as follows: $h_2(K_X) = X$ and $h_3(K_Y) = Y$. During the search for P_X we try all the five hash functions $h_0()$ to $h_4()$. Assume that one of the hash functions that were not used to store either K_X or K_Y generates the row Y when it is applied to P_X, i.e., $h_0(P_X) = Y$ or $h_1(P_X) = Y$. This means that we will search row Y before row X, thus, we incorrectly report K_Y as the LPM instead of K_X.

To solve the above 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, K_X in this case. This requires that the PH setup algorithm, Algorithm 1, has to be changed to store also the hash function index that is used to store each prefix. That is to add the following line after line 6 in Algorithm 1: "H_Table$[r_i][HC[r_i]]$.h $= i$". The search algorithm has to be changed as well since it must stop only at the first matching prefix for which the stored hash function index, ".h", is identical to the hash function index that is used to lookup the prefix during the search. Algorithm 2 shows the modified search algorithm for the PF application.

Algorithm 2 The PH Modified Search Algorithm For The PF Application.

```
1: Search_Hash_Table(Packet $P$)
2:     for($i = 0$ ; $i \leq H-1$ ; $i++$) {
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 $H\_Table[r_i][j].port\_number$
7:             else continue
8:     }
```

The algorithm uses the hash index field "$.h$" to check for the hash function that is used to store each prefix (line 5). Note that applying the same procedure for PC is more complicated by the fact that in some cases it is not clear which group of rules is more specific than the other. For example, group $(S16, NS)$ and group $(NS, S16)$ do not have a clear indication about which one is more specific than the other. Section 5 introduces an optimization that improves the AMAT of PC.

5. THE I-MARK OPTIMIZATION

In this section we introduce an optimization that reduces the overflow for the PF application. In addition to that, it can be used to reduce the AMAT for the PC application. Note that if we insert the keys that can be hashed by more hash functions after the keys that can be hashed by fewer hash functions then we may reduce the collisions and hence the overflow. This is simple for the PC application since there are no order restrictions over the rules during the insertion; however, there is a strict order of insertion in the PF application if we want to stop at the first matching prefix as being the LPM. Definition 2 formally defines the dependence relationship between keys:

DEFINITION 2. *If there is a packet P_i that matches two keys, K_i and K_j, K_i and K_j are called dependent keys.*

Therefore, for the packet forwarding, the independent prefixes can be inserted in any order and not according to the priority, while the dependent prefixes have to be inserted in order according to the priority as in Section 4.2. To use this, we define a binary flag, I-Mark, which is set to 1 if the key is independent and is reset to 0 if not for all the keys in the database. We call the set of keys which have I-Mark equal to 1 the "independent" set and the other set of keys the "dependent" set. There are many ways to insert the independent set, one particularly is to insert the independent keys first before the dependent ones but in a reverse order according to each key hash function specificity. For example, given the PF example in Figure 5(a) we insert the independent prefixes that use only the hash function $h_4()$ first, then we insert the independent prefixes that use $h_3()$ and $h_4()$ second and so on. This order can be justified logically as we try to accommodate those keys that can be hashed only by fewer number of hash functions before trying to accommodate other keys that can be hashed by more hash functions. Based on the available data, we find that the independent set represents 42.1% on average of the IP lookup table, while it ranges between 64.0% to 89.7% for the PC database.

In addition to allowing keys insertion in an arbitrary order, note that once one of the independent keys is matched during the search, then we can stop the search since there is no more matches. This helps us to reduce the AMAT of the PC application significantly as we will see in Section 6.

6. EXPERIMENTAL EVALUATION

For the evaluation of the PH scheme, we use C++ to build a simulation environment that allows us to choose between different types of hash functions. The hash functions used in the experiments are from four hashing families: bit-selecting, additive and rotative [13], CRC-based, and $\mathbf{H_3}$ [14] hashing families. Those families can be realized in hardware which is an advantage. For the PF evaluation, we collected 15 tables from the Border Gateway Protocol (BGP) Internet core routers of the routing information service project [16] on January 31st 2009. Table 1 lists the 15 routing tables and their sizes. To measure the average search time, we generate uniformly distributed synthetic traces using the same tables.

Table	Size	Table	Size
rrc00	292,717	rrc10	276,912
rrc01	276,224	rrc11	275,903
rrc02	272,743	rrc12	277,132
rrc03	283,147	rrc13	280,961
rrc04	283,075	rrc14	274,824
rrc05	301,383	rrc15	275,828
rrc06	277,555	rrc16	280,744
rrc07	274,479	**Average**	**280,242**

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

The PC case is studied using the ClassBench tool [23]. The tool provides synthetic packet classification databases from real databases. The ClassBench defines 3 families of PC applications: IPC (an old version of Firewalls), ACL (Access Control List) and FW (modern version of Firewalls). We generated 11 synthetic databases and their traces using ClassBench and their statistics are given in table 2.

For a given hardware implementation, the number of rows, N, and the number of entries per row, L, are fixed. We define a "configuration" by specifying both N and L. The PH performance, in terms of both overflow percentage and AMAT, depends on the number of groups for the application (hence the number of hash functions), H. In addition to H, the performance of PH depends 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. Section 6.1 compares the modified PH scheme (defined in Section 4) against two other schemes: grouped hashing (GH) scheme and restricted

Table	Size	(S16,S16) %	(S16,NS) %	(NS,S16) %	(NS,NS) %
ACL1	8905	92.49	6.00	0.00	1.52
ACL2	8072	48.20	20.69	28.43	2.68
ACL3	7714	60.72	17.14	18.58	3.56
ACL4	8960	59.63	15.33	20.42	4.61
ACL5	6593	92.08	0.00	7.92	0.00
FW1	7573	15.34	29.54	51.46	3.66
FW2	8874	14.59	73.91	11.46	0.03
FW3	6361	6.02	27.07	62.18	4.73
FW4	7144	19.16	29.16	44.85	6.83
IPC1	8310	72.97	11.07	15.17	0.78
IPC2	10000	36.60	11.14	52.26	0.00
Average	8046	36.60	11.14	52.26	2.58

Table 2: The Statistics of the Packet Classification databases from ClassBench Tool.

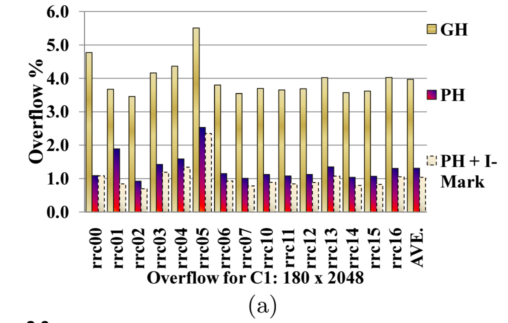
hashing (RH) scheme for the packet forwarding application. For the RH, all 5 hash functions use the most significant 16 bits and are applied to all the prefixes in the lookup tables. Those prefixes that are less than 16 bits are inserted in the overflow_buffer since they represent less than 2% of the routing tables. For the packet classification application, Section 6.2 compares the PH scheme only against the GH scheme. In all cases, we compare the schemes that have the same worst-case memory access time (WMAT). The results for the I-Mark optimization are also reported for each application. Finally, in Section 6.3, we apply the PH scheme for the state-of-the-art content-based hash probing scheme for the PF application.

6.1 Packet Forwarding Evaluation

We show in Figure 6(a) the overflow percentages, which is the ratio of the overflow to the total number of prefixes in the routing table, of the 15 routing tables for one configuration, $C1 : \{L = 180, N = 2048\}$ and for three schemes: GH, modified PH (after sorting the prefixes) and modified PH with the I-Mark optimization. We refrained from plotting restricted hashing (RH) scheme as its average overflow is high (18.0%). On average, the modified PH reduces the overflow by 95.0% compared to the RH scheme and by 66.7% compared to the GH scheme. At the same time, the AMAT (Figure 6(b)) of the modified PH is decreased by 62.6% over the GH scheme which has a constant AMAT of 5. The modified PH with the I-Mark optimization reduces the overflow by 72.8% and decreases the AMAT by 62.4% over the GH scheme.

To show that the PH scheme is robust under other configurations, we use three configurations to evaluate PH for PF applications: $C1 : \{L = 180, N = 2048\}$, $C2 : \{L = 90, N = 4096\}$ and $C3 : \{L = 45, N = 8192\}$ in Figure 7(a) and (b). Figure 7(a) shows the average overflow over all the 15 lookup tables for the GH, modified PH and PH with I-Mark schemes. These three configurations have the same average load factor of 76.0% which is considerably high. Figure 7(b) shows the AMAT of the same three configurations but only for the modified PH and the modified PH with I-Mark.

The modified PH with I-Mark has the lowest overflow percentage among the three schemes, then the modified PH has a little higher overflow percentage. The average (over the three configurations) overflow reduction percentages of the modified PH and the modified PH with I-Mark are 51.6% and 57.8% compared to the GH scheme. The modified PH



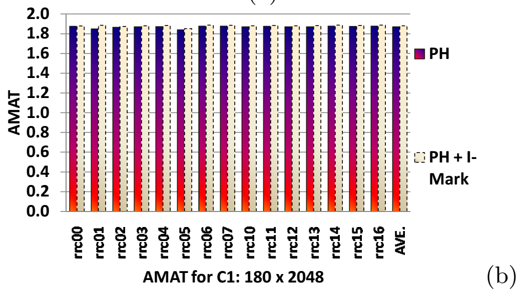
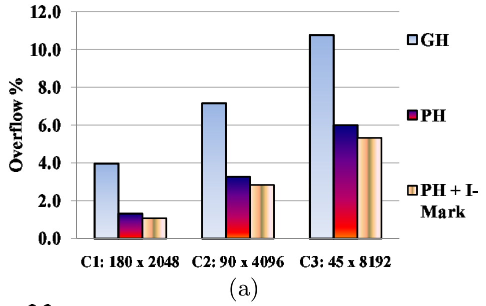


Figure 6: (a) Average Overflow (b) AMAT of GH vs. PH vs. PH +I -Mark for 15 Lookup Tables for Config. C1: $\{180 \times 2048\}$.



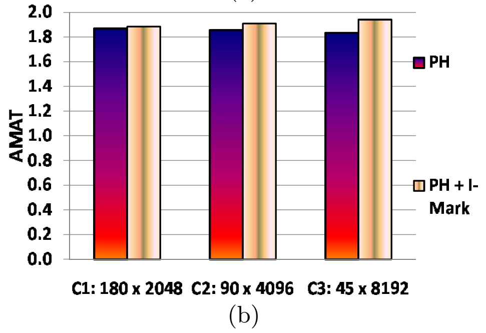


Figure 7: (a) Average Overflow of GH vs. PH vs. PH + I-Mark and (b)Average AMAT for PH vs. PH + I-Mark for PF for 3 Config's.

decreases the AMAT by 62.9% and the modified PH with I-Mark decreases the AMAT by 61.7% compared to the GH scheme which has a constant AMAT of 5.

6.2 Packet Classification Evaluation

We show in Figure 8 a comparison between the GH scheme vs. the PH scheme with and without I-Mark optimization for all the 11 PC databases for one configuration: $C1 : \{L = 100$ and $N = 128\}$. The PH reduces the overflow (Figure 8(a)) by 75.3% on average compared to the GH, while the PH with I-Mark reduces it by 84.2% on average compared to the GH. For the PH with I-Mark, the AMAT is decreased by 43.2% compared to both the GH and the PH schemes. We do not show the AMAT of GH and PH since both of them have a constant AMAT of 6.

We believe that the comparison for the PC application should be for each PC family as the average comparison among the 11 databases is not fair in the sense that each family, and sometimes even each database, has its own characteristics that requires different configurations for better performance. As a result, we show in Figure 9(a) the average overflow and in Figure 9(b) AMAT of the three PC families for three different configurations: $C1 : \{L = 100$ and $N = 128\}$, $C2 : \{L = 80$ and $N = 256\}$ and $C3 : \{L = 50$ and $N = 512\}$. The average loading factors for these configurations are 62.9%, 39.3% and 31.4% respectively.

From Figure 9(a) we can see that both PH with I-Mark optimization and PH outperform GH in terms of overflow reduction. The PH achieves zero percent overflow for the FW family, while reduces the overflow by 66.25% for the ACL family and by 94.35% for the IPC family compared to the GH. Similarly, the PH with I-Mark achieves zero overflow for both FW and IPC families and reduces it by 74.3% for the ACL family compared to the GH. The AMAT of the PH with I-Mark is decreased by 31.0%, 56.0% and 48.4% respectively for the three families compared to both the GH and the PH schemes as illustrated in Figure 9(b).

6.3 Applying PF to Content-based Probing

In this section the progressive hashing is applied to the Content-based HAsh Probing (CHAP) for the PF application [8]. While regular probing uses predetermined offsets to solve that problem, CHAP [8] uses a mix of restricted multiple hash functions (all use the most significant 16 bits) and probing in the following probing sequence:

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

The probing pointers, $\beta_0, \beta_1, \cdots, \beta_m$, are determined dynamically for each value of $h_i(k)$, where k is the key to be hashed, depending on the distribution of the data stored in a particular hash table. The overall scheme is called **CHAP(H,m)** since it has H hash functions and m probing pointers per row. Figure 10 shows CHAP(3,3) when $m = H = 3$. As presented in [8], CHAP uses restricted multiple hash functions where each function takes the most significant 16 bits; we apply our PH scheme to enhance CHAP performance.

In Figure 11(a) we show the average overflow of the RH CHAP, PH CHAP and PH with the I-Mark CHAP for the same three configurations we used in Section 6.1. The first two configurations have zero overflow for both the PH schemes (with and without the I-Mark) with a reduction of 100% in this case. For the third configuration, $C3$: 45×8192, PH reduced the overflow by 23.6% and PH with I-Mark reduced it by 28.7% compared to RH.

We note that both schemes PH + CHAP with and without I-Mark have a lower AMAT (Figure 11(b)) than the RH + CHAP scheme with an average of 21.3% reduction in case of PH only and 23.1% in case of PH with I-Mark.

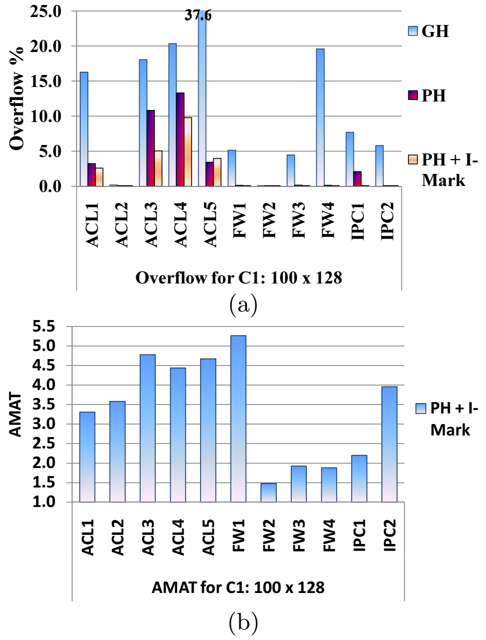


Figure 8: (a) Average Overflow of GH vs. PH vs. PH + I-Mark and (b) AMAT of PH + I-Mark for the 11 PC databases for configuration C1: $\{100 \times 128\}$.

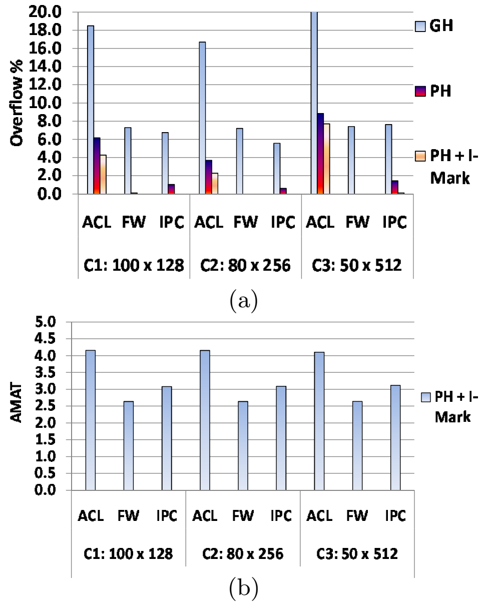
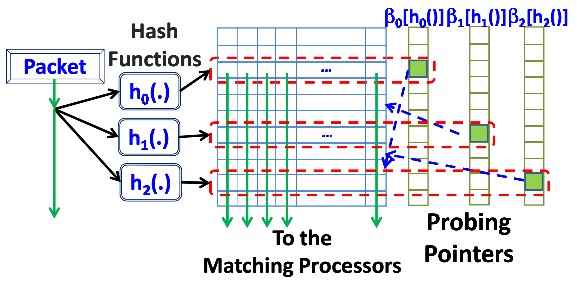


Figure 9: (a) Average Overflow for GH vs. PH vs. PH +I-Mark and (b) Average AMAT of PH + I-Mark for 3 PC Families for 3 Config's.



Packet
Hash Functions
$h_0(.)$
$h_1(.)$
$h_2(.)$
$\beta_0[h_0()]\beta_1[h_1()]\beta_2[h_2()]$
To the Matching Processors
Probing Pointers

Figure 10: The CHAP(3,3).

6.4 Actual Performance Estimation

In this section we estimate the actual processing rate of the PH packet processing engine using sound approximations. We estimate the memory requirements for two configurations C1 for the PF and C1 for the PC. For PF, the configuration C1 requires $180 \times 2048 = 368,640$ or 370K entry, where an entry is represented by 5 bytes prefix plus 5 bits prefix length plus 1 bit for the I-Mark plus 3 bits for the hash function flag, which is 49 bits and there is 1 byte per row for the row counter. The total memory requirement for this PF configuration is ~ 2.16 MB. For PC, the configuration C1 requires $100 \times 128 = 12,800$ or 13K entry, where an entry is represented by 2×5 bytes for prefixes plus 2×5 bits prefix length plus 4×2 bytes for port ranges plus 1 byte protocol and other fields encoding plus 1 bit for the I-Mark plus 3 bits for the hash function flag, which is 166 bits and there is 1 byte per row for the row counter. The total memory requirement for this PC configuration is ~ 0.25 MB.

A state-of-the-art CMOS technology SRAM memory design [27] 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. For the PF application, and if we assume AMAT of ~ 2.0 (according to Figure 7(b)), then we have a forwarding speed of 1.0 Giga packets per seconds or 320 Gbps for the minimum packet size of 40 bytes. For the PC application, and if we assume AMAT of ~ 4.0 (according to Figure 9(b)), then we have a filtering speed of 0.5 Giga packets per seconds or 160 Gbps for the minimum packet size of 40 bytes.

7. CONCLUSIONS AND FUTURE WORK

In this paper we introduce Progressive Hashing (PH), a new open addressing hash-based packet processing scheme that can also work for closed addressing hash systems. PH is effective in reducing the classical hashing overflow on average by 95% compared to restricted hashing (RH) and by 66.7% compared to grouped hashing (GH) for the packet forwarding (PF) application and 73.3% for the packet classification (PC) application compared to GH. We also introduce an optimization to improve the average memory access time for the PF application with an estimated 320 Gbps average processing speed. The PH processing engine is estimated to achieve an average processing speed 160 Gbps for the PC application with the I-Mark optimization which is used to reduce both the overflow and the AMAT at the same time. PH is also applied to the state-of-the-art PF hash-based system, CHAP, and showed that CHAP achieves a better performance than that with RH.

This paper also introduced optimizations that reduce the average memory access time, however we have some preliminary results of optimizations that reduce the worst case memory access time. The optimal insertion order of keys for the I-Mark optimization into the hash table is not discussed here, rather a heuristic is introduced. We believe that finding such an order needs further investigation and calls for studying the effect of the number of keys per group on this

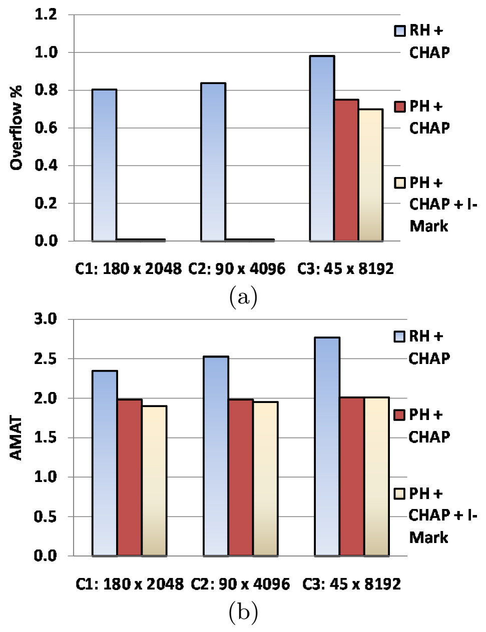


Figure 11: (a) Average Overflow and (b) Average AMAT of PH vs. PH + CHAP vs. PH + CHAP + I-Mark for the Lookup Tables for 3 Config's.

order. We will study a fully synthesized PH packet processing engine in a future work. The future work will also propose a full design of a pipeline-based PH packet processing engine to achieve a single lookup per clock cycle.

8. REFERENCES

[1] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 2000.

[2] F. Baboescu, D. Tullse, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. *SIGARCH Comput. Archit. News*, 33(2):123–133, 2005.

[3] S. Cho, J. Martin, R. Xu, M. Hammoud, and R. Melhem. Ca-ram: A high-performance memory substrate for search-intensive applications. pages 230–241. IEEE ISPASS, 2007.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stien. *Introdcution to Algorithms.* McGraw Hill, 2003.

[5] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. Fast ppacket classification using bloom filters. In *ACM/IEEE ANCS*, pages 61–70, New York, NY, USA, 2006.

[6] P. Gupta and N. Mckeown. Packet classification using hierarchical intelligent cuttings. In *IEEE Hoti*, pages 34–41, 1999.

[7] P. Gupta and N. Mckeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[8] M. Hanna, S. Demetriades, S. Cho, and R. Melhem. Chap: Enabling efficient hardware-based multiple hash schemes for ip lookup. pages 756–769. IFIP Networking, 2009.

[9] S. Kaxiras and G. Keramidas. Ipstash: A power-efficient memory architecture for ip-lookup. pages 361–373. Micro'03, 2003.

[10] R. A. Kempke and A. J. McAuley. Ternary cam memory architecture and methodology. http://www.freepatentsonline.com/5841874.html, 1998. United States Patent 5841874.

[11] A. Kirsch and M. Mitzenmacher. Simple summaries for hashing with choices. *IEEE/ACM Trans. Netw.*, 16(1):218–231, 2008.

[12] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary cams. pages 193–204. ACM Sigcomm, 2005.

[13] A. Partow. General purpose hash function algorithms library. http://www.partow.net/.

[14] M. Ramakrishna and et Al. Efficient hardware hashing functions for high performance computers. *IEEE Trans. on Comp.*, 46(12):1378–1381, 1997.

[15] Y. Rekhter and T. Li. An architecure for ip address allocation with cidr. *RFC*, 1993.

[16] RIS. Routing information service. http://www.ripe.net/ris/, 2006.

[17] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. pages 213–224, New York, NY, USA, 2003. ACM SIGCOMM.

[18] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *ACM SIGCOMM*, pages 181–192, New York, NY, USA, August 2005. ACM.

[19] H. Song, J. Turner, and S. Dharmapurikar. Packet classification using coarse-grained tuple spaces. In *ACM/IEEE ANCS*, pages 41–50, New York, NY, USA, 2006. ACM.

[20] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended tcams. pages 181–192. IEEE ICNP, 2003.

[21] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, pages 135–146, New York, NY, USA, 1999.

[22] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, 17(1):1–40, 1999.

[23] D. Taylor and J. Turner. Classbench: A packet classification benchmark. In *IEEE INFOCOM*, volume 15, pages 499–511, 2007.

[24] D. E. Taylor and J. S. Turner. Scalable packet classification using distributed crossproducing of field labels. volume 1, pages 269–280. IEEE Infocom, 2005.

[25] D. E. Turner. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys*, 37(3):238–275, 2005.

[26] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices.* Morgan Kaufmann, 2005.

[27] 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.