HIGH-PERFORMANCE PACKET PROCESSING ENGINES USING SET-ASSOCIATIVE MEMORY ARCHITECTURES

by

Michel Hanna

B.S., Cairo University at Fayoum, 1999

M.S., Cairo University, 2004

M.S., University of Pittsburgh, 2009

Submitted to the Graduate Faculty of

The Computer Engineering Program,

Dietrich School of Arts and Sciences

in partial fulfillment

of the requirements for the degree of

Ph.D.

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Michel Hanna

It was defended on

May 7^{th}, 2013

and approved by

Prof. Rami Melhem, Computer Science Department

Prof. Steven Levitan, Department of Electrical and Computer Engineering

Prof. Prashant Krishnamurthy, School of Information Sciences

Prof. Taieb Znati, Computer Science Department

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

Prof. Sangyeun Cho, Computer Science Department

HIGH-PERFORMANCE PACKET PROCESSING ENGINES USING SET-ASSOCIATIVE MEMORY ARCHITECTURES

Michel Hanna, PhD

University of Pittsburgh, 2013

The emergence of new optical transmission technologies has led to ultra-high Giga bits per second (Gbps) link speeds. In addition, the switch from 32-bit long IPv4 addresses to the 128-bit long IPv6 addresses is currently progressing. Both factors make it hard for new Internet routers and firewalls to keep up with wire-speed packet-processing. By packet-processing we mean three applications: packet forwarding, packet classification and deep packet inspection.

In packet forwarding (PF), the router has to match the incoming packet's IP address against the forwarding table. It then directs each packet to its next hop toward its final destination. A packet classification (PC) engine examines a packet header by matching it against a database of rules, or filters, to obtain the best matching rule. Rules are associated with either an "action" (e.g., firewall) or a "flow ID" (e.g., quality of service or QoS). The last application is deep packet inspection (DPI) where the firewall has to inspect the actual packet payload for malware or network attacks. In this case, the payload is scanned against a database of rules, where each rule is either a plain text string or a regular expression.

In this thesis, we introduce a family of hardware solutions that combine the above requirements. These solutions rely on a set-associative memory architecture that is called CA-RAM (Content Addressable-Random Access Memory). CA-RAM is a hardware implementation of hash tables with the property that each bucket of a hash table can be searched in one memory cycle. However, the classic hashing downsides have to be dealt with, such as collisions that lead to overflow and worst-case memory access time. The two standard

solutions to the overflow problem are either to use some predefined probing (e.g., linear or quadratic) or to use multiple hash functions. We present new hash schemes that extend both aforementioned solutions to tackle the overflow problem efficiently. We show by experimenting with real IP lookup tables, synthetic packet classification rule sets and real DPI databases that our schemes outperform other previously proposed schemes.

Keywords: Hardware Hashing, Set Associative Memories, IP Lookup, Packet Classification, Deep Packet Inspection.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF EQUATIONS

PREFACE

Acknowledgments

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

1.0 INTRODUCTION

A router, as shown in Figure 1, consists of multiple interface cards (egress and ingress), a switch fabric, a CPU or a network processor (NP) that is attached to a memory module (this could be a hierarchy of caches [11]), a forwarding engine, and either a deep packet inspection (DPI) engine (in case the router has a firewall capabilities) or a QoS engine for traffic shaping. Both the DPI engine and the QoS unit contain a packet filtering (classification) unit.

The router's main function is to receive data packets and forward them to their correct destinations. It inspects the packet's IP (internet protocol) headers and extracts the destination of this packet via looking it up in its forwarding table. High-speed routers have become very desirable as they facilitate the rapid transfer of packets. In addition to their basic packet forwarding (PF) functionality, most modern routers also inspect the remaining packet (e.g., TCP) headers to determine what access rights or what service (bandwidth) this packet might have. This is called packet classification (PC) functionality.

High-speed Internet routers and firewalls require wire speed packet-processing, while the sizes of their DPI and PC rule databases and PF tables are increasing at a high rate [31, 53, 54]. In addition, the advancement of optical networks keeps pushing the link rates, which are already beyond 40 Gbps [43, 54]. In this thesis, we focus on the three main components of the high-speed Internet router: the packet forwarding engine, the packet filtering engine and the deep packet inspection engine. We note that these three engines are similar in the sense that they all rely on search-intensive operations.

In packet forwarding (**PF**), the destination address of every incoming packet is matched against a forwarding table to determine the packet's next hop on the way to its final destination. An entry in the forwarding table, or an IP prefix, is a binary string of a certain length followed by wildcard (don't care) bits and an associated port number. The actual

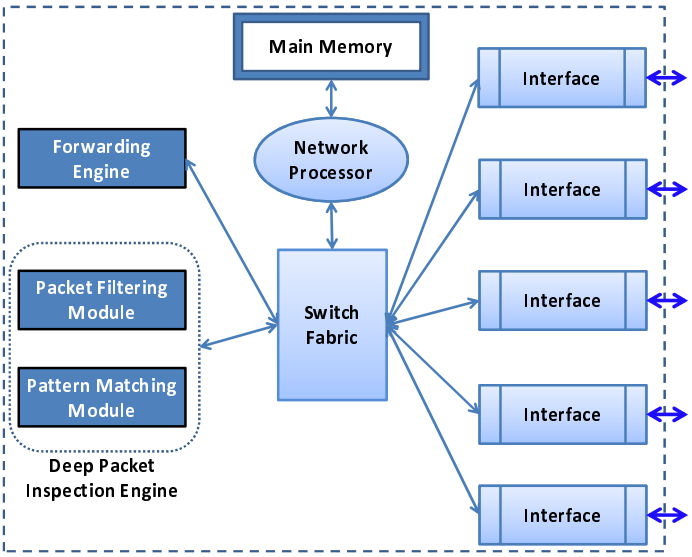


Figure 1: A generic router architecture with deep packet inspection capability.

matching requires finding the "Longest Prefix Matching" (LPM) as instructed in the CIDR protocol [39].

In packet classification (**PC**), a packet header is matched against a database of rules–or filters–to obtain the best matching rule. A priority tag 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 (tuples [47]): IP source address, IP destination address, source port, destination port and protocol identifier. Rules are associated with either an "action" (e.g., in case of firewall) or a "flow ID" (e.g., in case of QoS).

In deep packet inspection (**DPI**), the firewall examines the packet's payload for traces of either network attacks, such as network intrusion detection system (NIDS) or malware signatures, such as virus scanning systems [33]. The main components of a DPI engine are the pattern-matching (PM) unit [59], and the packet filtering unit [33]. The packet pattern-matching problem is defined as follows: given a set of k patterns $\{P_1, P_2, \cdots, P_k\}$, $k \geq 1$,

and a packet of length n, the goal is to find all the matching patterns in the packet. Note that each pattern (string) has its own length. If we match one or more of these substrings, we have a "partial" match and the pattern-matching unit should return the longest matched substring to the software layer of the firewall for further investigation. In addition to DPI, content filtering, instant-messenger management, and peer-to-peer identification applications all use pattern (string) matching for inspection [33]. Throughout this thesis we will refer to any of a prefix, a PC rule or a DPI pattern as a "key."

The two main streams of packet-processing research are: algorithmic and architectural. Many researchers have devised algorithmic solutions that provide space and time complexity bounds for difficulties arising in packet-processing [19, 18, 26, 47, 15]. Some of these solutions are feasible for hardware implementations [51, 33, 15]. In general, the algorithm-based solutions have lower throughput than their hardware counterparts. This motivated the introduction of architectural solutions that mostly rely on the Ternary Content Addressable Memory (TCAM) technology [28, 46]. A TCAM is a fully associative memory that can store binary values, 0s and 1s as well as wildcard (don't care) bits. TCAMs have been the *de facto* standard for packet-processing in industry [31, 53, 33]. However, TCAM comes with significant inefficiencies: high power consumption, low bit density, poor scalability to long input keys, and higher cost per bit compared to other memories.

In addition to these two research streams, the industry recently started to adopt embedded DRAM [58] (eDRAM) technology. eDRAM is a capacitor-based dynamic RAM that is integrated on the same die as the main ASIC (application specific integration circuit) or processor, which allows the network processor chip to have more memory. This direction is pioneered by "Huawei" Technologies [6] who call their technology *Smart Memory*, "NetLogic" and "Cavium."

Architectural solutions based on hashing have also been proposed [16, 29, 44, 47, 15]. Hash tables come in two flavors: closed addressing hash (or *chaining*) and open addressing hash. A 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, a hash table has both a fixed height and a fixed bucket width. Overflow in open addressing hash is handled through probing [13], as described in Section 3.1. We define the overflow as being the percentage of keys that did

not fit into the main hash table to the total number of keys available for storage. In this thesis, we assume open addressing hash and 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 power consumption issue will be addressed almost automatically by storing this hash table in an efficient SRAM or DRAM memory array.

The remainder of this thesis is organized as follows:

- In Section 3, we give a brief background on: general open addressing hashing, hashing with the presence of wildcards, and finally we illustrate our set-associative memory architecture.
- In addition, in Section4, we describe our hashing schemes, which are used later as tools in our packet-processing engines.
- in Sections 5, 6, and 7 we describe our solutions and experimental results to each of the three packet-processing applications.
- Finally, we give a summary of the thesis in Section 8.

2.0 THESIS OUTLINE, MOTIVATION AND CONTRIBUTIONS

In this section, we describe the 'big picture' of our thesis. We mentioned that the subjects of this thesis are the router's search-intense packet-processing units; namely, the forwarding engine, the packet filtering engine and the deep packet inspection engine. Since the target of today's and future Internet routers and firewalls is to keep up with both IPv6 and new ultra high link rates, we need to think about packet-processing units designs that have high throughput and have low power consumption all at the same time. We choose a design that is based on a set-associative memory hardware realization of hash tables, as it provides a general homogeneous platform for all the three applications. The actual architecture we are using is called CA-RAM (Content Addressable-Random Access Memory) and is described in some detail in section 3.4.

In general, The CA-RAM provides a reliable RAM architecture for search intensive applications [12]. It allows for concurrent searching through multiple keys. In addition, it is based on regular RAM technology rather than the expensive TCAM technology. Finally, the CA-RAM is better than the classical hardware TCAM-based solution in its ability to support different types of matching. For example, the packet classification application requires three different types of matching: prefix matching, exact matching and range matching. While the TCAM naturally supports both exact and prefix matchings, it requires the addition of special complex hardware to deal with the range matching [46]. On the other hand, CA-RAM supports all aforementioned types of matching, due to its unique feature of separating the storage capability from the matching logic.

We believe that such architecture is capable of answering the wire speed and the power consumption issues much better than other hardware and software solutions. The high throughput requirement is boosted by using pipelining. For CA-RAM to work efficiently,

it needs good hashing schemes to fully utilize its capabilities. For example, in the PC and PF applications, the keys that are being mapped into the hash table have low entropy and tend to collide. Also, in the DPI application, some strings are substrings of other strings in the DPI rule database. Thus CA-RAM needs new, efficient, collision-resolving schemes that are more efficient than classical linear and quadratic probings. These new hashing schemes should allow for high space utilization of the CA-RAM. I talk more about probing in the next section, Section 3.1.

At the same time, CA-RAM also needs to maintain a high throughput, which is achieved by lowering the memory access time and using pipelining. Thus our proposed hashing schemes should lower the memory access time for the CA-RAM while resolving the collision problem, which are two contradictory goals. In addition, in packet-processing applications, the search engine, in this case the CA-RAM, has to report first the best matching key in the database. In case of PF, the CA-RAM has to return the longest matching prefix, while in the DPI, the CA-RAM has to return the longest matching pattern. Thus, our hashing schemes have to support this property. Finally, since packet-processing applications require dealing with wildcards, my hashing schemes have to deal with this issue as well.

In this thesis, I am proposing three hash-based schemes (techniques) that enable the use of the CA-RAM for the three packet-processing applications. These are (in order of presentation): content-probing pointers, progressive hashing and independent marking (I-mark). These techniques are fully developed for two applications, namely PF and P,C and are extended to the DPI application with some additional hardware customization. These schemes are defined as our building blocks in Section 4. In addition, each scheme will be optimized for each application, as described in Sections 5, 6 and 7.

My first scheme, the content-hash probing or CHAP, is–as its name indicates–a probing technique that relies on the actual stored data in the hash table to probe for overflow keys. This is in contrast to the classical systematic probing techniques (e.g., linear and quadratic) that are known in the literature, which do not take into account the stored data and its distribution. The third way to handle probing in open address hashing is double hashing. My second hashing scheme, progressive hashing, extends double hashing, which is a known technique for open address hashing (see Section 3.1), by first splitting the keys into different

groups. We then assign to each group a unique hash function to reduce the collision probability. This is different than double, or multiple, hashing where all keys can use all hash functions that are used by the system. Finally, I propose my third technique, the I-mark, to take advantage of the fact that some keys can be parts of other keys, like in DPI where some short strings are substrings of other longer strings. In this technique, I use the fact that some keys are truly independent or unique (i.e., have no other keys shorter or longer that are parts of these keys, nor are these keys part of other keys) to store the keys efficiently inside the CA-RAM. This is so we can insert those unique keys before or after other keys to lower the chance of collisions.

As I am going to show in the rest of this thesis, the CA-RAM, along with my proposed hashing schemes, outperforms the TCAM. I prove this by conducting a battery of experiments and simulations to compare both CA-RAM and TCAM in terms of throughput (speed), power consumption and space area. For example, my modified CA-RAM is estimated to be capable of processing up to 320 Gbps for PF application, and is 160 Gbps for the PC application, while our CA-RAM speed runs at a clock frequency of 540MHz. While achieving such high rates of processing in these three application, the CA-RAM power consumption is less than its equivalent size TCAM by 77% for the PF application and 71% for the DPI application. Moreover, we consider that the CA-RAM is much more scalable than the TCAM due to the fact that it separates the storage from the matching part.

3.0 BACKGROUND

In this chapter, in Section 3.1, I describe open addressing hash from the theoretical point of view, while in Section 3.2, I address the issue of using single versus multiple hash tables. This issue stems from the fact that some of the proposed solutions for packet-processing rely on multiple hash tables. In Section 3.3, I talk about hashing keys with wildcards and how they can cause collisions. In Section 3.4, I describe in some detail the CA-RAM general architecture and it advantages over the TCAM. At the end of this chapter, in Section 3.5, I summarize some of the related work.

3.1 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 schemes achieve 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 (or more) distinct keys $k_i \neq k_j$ hash to the same value: $h(k_i) = h(k_j)$. Such an occurrence is called "collision." A worst-case (pathological) situation that restricts the effectiveness of hashing is when all the keys are mapped to the same row. There are two solutions to the collision problem in this case: 1) Make the row large enough to hold all the possible colliding prefixes at the cost of a large amount of wasted memory. 2) Control the row size and handle the overflow prefixes in a different way, such as "probing."

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

$$h(k), h(k) + \beta_0, h(k) + \beta_1, \cdots, h(k) + \beta_{m-1} \tag{3.1}$$

where each β_i is a constant, and m is the maximum number of probes. Linear probing is simple, but often suffers from what is called "primary key clustering" [13].

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:

$$h(k, i) = (h'(k) + c_1 \times i + c_2 \times i^2) mod(N), i = 0, 1, \cdots m - 1 \tag{3.2}$$

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, called "secondary key clustering" [13].

Instead of probing, we can apply a second hash function to find an empty bucket, a procedure know as *double hashing* [13]. In general, the use of $H \geq 2$ hash functions is shown to be better in reducing the overflow than probing [2]. In this case (which we refer to as *multiple hashing*) the probing sequence of inserting a key into the hash table is given as follows:

$$h_0(k), h_1(k), \cdots, h_{H-1}(k) \tag{3.3}$$

where H is the number of hash functions. To achieve high space utilization (or load factor, which is the ratio between the size of the database and the capacity of the actual RAM used to store it) we apply multiple hash functions on a single hash table, rather than using a separate table for each hash function [8].

3.2 USING SINGLE HASH TABLE VS. MULTIPLE HASH TABLES

Note that using a different hash table for each hash function in Equation 3.3 is a valid design option; however, using different hash tables leads to more overflow, Hence results in poor space utilization [14, 26, 54]. In what follows, we argue that using a single hash table is better than using multiple hash tables.

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

$$overflow_A = max\{0, (a - L)\}$$

$$overflow_B = max\{0, (b - L)\}$$

$$overflow_C = max\{0, (c - 2 \times L)\}$$

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

To achieve high space utilization (load factor) 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 3.3. 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 α, or the *load factor*, defined as $M/(N \times L)$. With a smaller α, the average number of hash table accesses can be made smaller, however at the expense of more unused memory space [3].

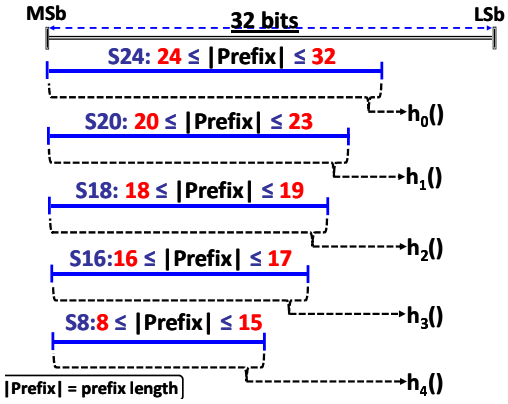
3.3 HASHING WITH WILDCARDS

Applying hash functions in packet-processing applications is very challenging due to the fact that wildcards, or don't care bits, are heavily present in the packet-processing engine's database. Hashing with wildcards requires one of two solutions: restricted hashing or grouped hashing [20]. In restricted hashing **RH**, the hash functions are restricted to using only the non-wildcard bits of the keys. We use RH only for PF application. In grouped hashing, **GH**, keys are grouped based on their lengths, then different hash functions are applied to each group. By length we mean the part of the key that does not contain any don't care bits, which we call "specific bits." For example, the 32-bit IPv4 address can be split into 5 groups as follows:

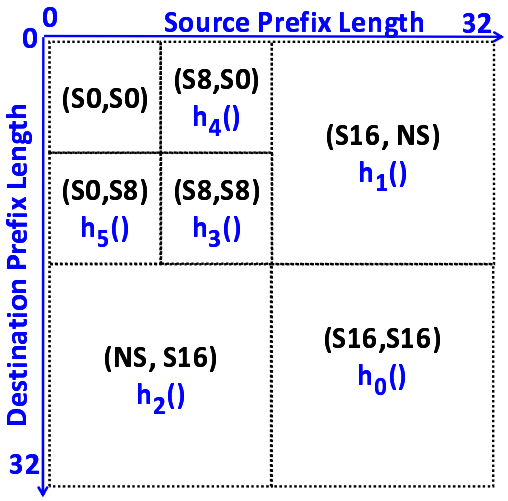
- Group $S24$, containing prefixes with at least 24 specific bits.
- Group $S20$, containing prefixes of length between 20 and 23 bits.
  - Group $S18$, containing prefixes of length 18 and 19 bits.
  - Group $S16$, containing prefixes of length 16 and 17 bits.
- Group $S8$, containing prefixes of length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

Then, each group is associated with a different single hash function as shown in Figure 2(a). We represent the 32-bit address space with a bold line, where MSb and LSb stand for most significant bit and least significant bit, respectively.

Grouped hashing can also be applied to PC using the tuple space concept. For example, a coarse-grained tuple space [45] (we describe this scheme in Section 3.5), where a key is of the form (source address prefix, destination address prefix), can divide the PC hashing space (or keys to be hashed) into 7 groups, as shown in Figure 2(b). The filters are split into 4 groups based on the source and the destination prefix lengths: $(S16, S16)$, $(S16, NS)$, $(NS, S16)$ and (NS, NS), where "S16" means that the prefix has 16 or more specific bits, while the "NS" stands for Non-Specific, i.e., the prefix is less than 16 bits. The (NS, NS) group is split once more into 4 groups: $(S8, S8)$, $(S8, S0)$, $(S0, S8)$ and $(S0, S0)$, where "S0" includes all prefixes that are less than 8 bits long.



(a)



(b)

Figure 2: Splitting the hashing space into groups for (a) PF application, and (b) PC Application.

3.4 THE CONTENT ADDRESSABLE-RANDOM ACCESS MEMORY (CA-RAM) ARCHITECTURE

We use the CA-RAM as a representative of a number of set-associative memory architectures proposed for packet-processing [12, 26, 60]. 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. The key features of

the CA-RAM are that it separates the matching logic from the storage, allowing for greater space saving over regular CAMs. At the same time keeping the matching logic near the memory bulk, allowing for lower I/O bandwidth and lower processing latency [12].

CA-RAM uses high-density memory (*i.e.*, SRAM, DRAM or eDRAM [58]) 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 [12]. A CA-RAM takes a search key as an input, 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 3. The task of the index generator is to create an index from an input key. The actual function of the index generator depends highly on the target application. 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 3, where a CA-RAM entry (cell) stores a key, and its corresponding data. Alternatively, two bits can be used to store a ternary digit to represent 0, 1 and don't care, rather than binary (as in TCAM arrays, except that the comparison hardware in this case is shared among all the rows in the memory array). Optionally, each row can be augmented with an auxiliary field to provide information on the status of the associated bucket (e.g., how many keys are stored in this row). We use the auxiliary field in our hashing schemes as we will see later.

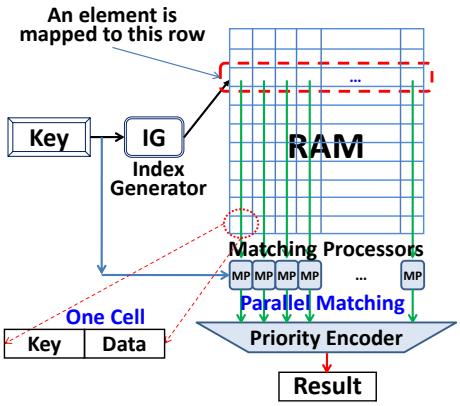


Figure 3: The CA-RAM as an example of set-associative memory architectures.

Once the index is generated from the input key, the memory array is accessed and L candidate keys are fetched simultaneously. The match processors then compare the candidate keys with the search key in parallel, resulting in constant-time matching. Each match processor performs comparison quickly using an appropriate hardware comparator. For example, if we are comparing PC range field, then the matching processor is simply a range comparator, while when we are comparing either PF prefix field or a DPI string, the matching processor consists of an exact matching comparator.

Figure 4 shows a simple key matching circuit for a generic CA-RAM. We assume that CA-RAM is being queried for a certain key (stored key) using another external key. Throughout my thesis, the external key will be extracted from some IP packet (header or payload). After the key is fetched from the storage section of the CA-RAM (i.e., RAM), it is stored in a buffer that we call a "key buffer." This key is going to be compared against the key part of the incoming element, which is stored in another buffer, "I/O buffer." For simplicity, we assume that this is exact matching where we use just simple XOR gates to make sure that both the stored key and the external key are identical.

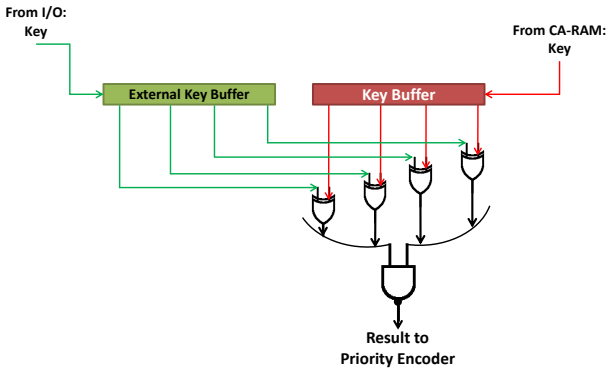


Figure 4: A simple key matching circuit for a generic CA-RAM.

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 completely separates the dense memory array from the common match logic (i.e., match processors). Since the match processors are simple and lightweight, the overall area cost of CA-RAM will be close to that of the memory array used. At the same time, by performing a number of candidate key matching operations in parallel, low-latency, constant-time search performance is achieved.

CA-RAM was compared against TCAM in terms of performance, power and area (cost). The result obtained in [12] shows that CA-RAM is over 26 times more power-efficient than the 16T SRAM-based TCAM [30], and over 7 times more than the 6T dynamic TCAM [34]. 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 [12].

3.5 RELATED WORK

In this section, we review hash-based solutions that were proposed for each of the packet-processing applications. Most of the work that is done in hash-based packet-processing uses closed addressing hash [7, 29, 44, 8]. One of the few solutions that are based on open address hashing in the PF application is the IPStash [26, 27]. The IPStash architecture is similar to the CA-RAM architecture [12, 26, 27]. However, IPStash uses a special form of the grouped hashing scheme as it classifies the prefixes into only three groups according to their lengths, and uses only 12 bits for hash table indexing. In addition, IPStash uses controlled prefix expansion (CPE) [48] to expand prefixes of lengths between 8 and 15 bits to 16 bits, and then choose any 12 bits to index the hash table [27]. In our work [21], we compared our CA-RAM-based schemes against IPStash and showed that our schemes are superior.

	Prefix	Port		Prefix	Port
a	000*****	0	h	1*******	0
b	000101**	1	i	1001****	0
c	0001111*	2	j	11011***	2
d	0010****	0	k	110101**	1
e	00111***	2	l	111101**	1
f	0110****	0	m	1111111*	2
g	01111***	2			

Table 1: An example of an 8-bit address space forwarding table.

The most well-known PF algorithm-based solutions use the binary trie data structures. According to [42], a trie is: "a tree-like data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching." Table 1 shows an example of an IP forwarding table assuming 8-bit addressing. Figure 5 is the equivalent binary trie of the IP forwarding table shown in Table 1, where a, $b, \cdots, m$ are symbols given to the prefixes for easy identification. The main advantages of trie-based solutions are that they provide simple time and space bounds. However, with the 128-bit IPv6 prefixes, both trie height and enumeration become an issue when the prefixes are stored inside the nodes.

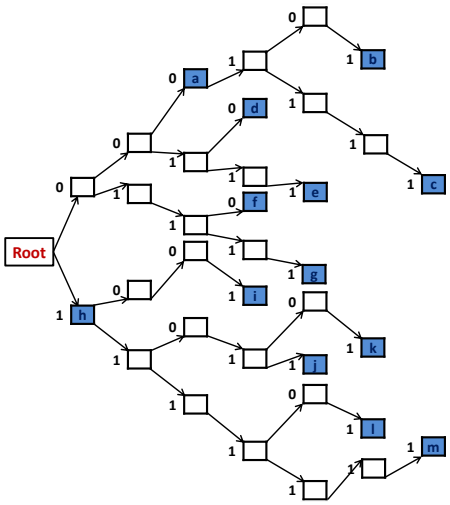


Figure 5: The binary trie representation of the forwarding table given in Table 1.

For the PC application, the authors in [47] introduce a special hash-based scheme 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 represented 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 represented by an integer.

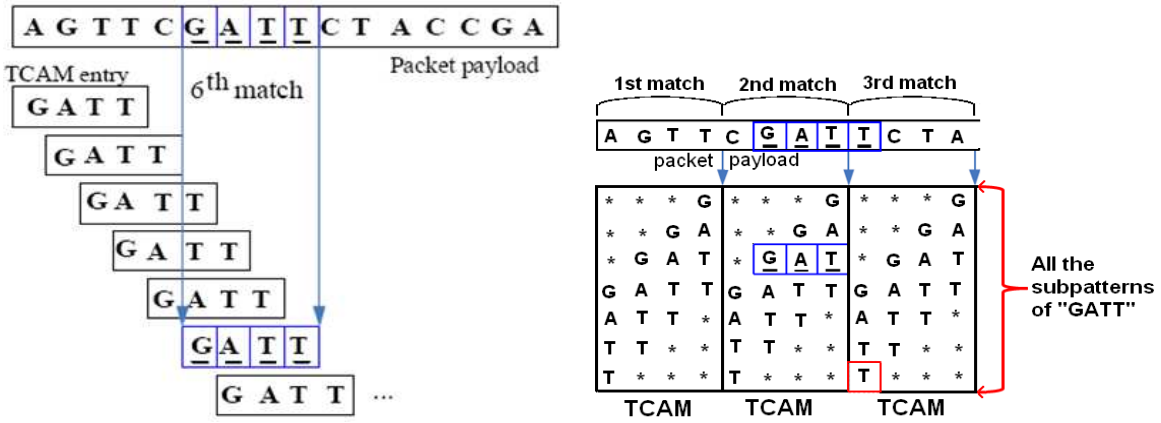
The original TSS scheme suffers from having high worst-case search time of $O(W^2)$, where W is the prefix length in bits. The authors then introduced an optimization called "markers and computations." Each tuple has two pointers to two filters; a marker (filer) and a precomputation filter. During a search for the best matching filter for any incoming packet, we probe to a tuple T_i, where we have two choices:

- The packet we have matches one of the filters that belong to T_i, in which case we need to find a more specific filter.
- The packet does not match any filter, in which case we need to find a less specific filter.

A marker filter is a pointer to the less specific filter than the current matching filter in T_i, while a precomputation filter is a pointer to the more specific filter. Note that the specificity of a certain filter in the 2D TSS is determined by the length of both prefixes of this filter.

In [32], the authors introduced an optimization to the TSS scheme by introducing a binary search scheme on 2D tuples space. However, this optimization [32] did not address

the practical issues of the TSS such as: how to handle the multiple hash tables, and how to select the hash functions. The work by Song et al. [45] is the first to show how a practical TSS system functions by splitting the $2D$ TSS into coarse clusters, and then assigning one hash table to each cluster. The work in [29] enhances [45] by using other kinds of summaries instead of the Bloom filter-based [7]. A Bloom filters (BF) are simple array of bits used for representing a set to support fast approximate membership queries [7]. The main disadvantage of BFs is that they make false positive membership queries.



(a) (b)

Figure 6: The (a) Sliding window example, and (b) Its jumping window equivalent where $w = m = 4$.

A DPI engine consists of two units: a PC engine and a pattern-matching engine [33]. Some of the DPI patterns are "simple" patterns, meaning they are concatenations of single-byte ASCII code characters (in which some wildcard bytes could appear in place) while there other patterns are called "composite" patterns. An example of the composite patterns would be what is known as "correlated" patterns, where subpatterns appear in specific locations inside the packet. Throughout our work, we assume only simple patterns and subpatterns. To process a composite pattern, the PM engine detects all the simple subpatterns that make that composite pattern and sends them to the network processor for further investigation.

TCAM-based pattern-matching engines perform well for the small-sized pattern databases,

but suffer great performance deficiency for large ones [33]. The TCAM width 'w' is an important factor in the design of the pattern-matching unit. In [59], the authors describe a byte-by-byte scanning TCAM solution, which is referred to as a "sliding window" scheme [49]. In a sliding window scheme, a window of 'w' characters of each packet is compared to the content of the TCAM for a match. The problem with this approach is that the scanning speed is low, since each time a new character is being added to the window, another is being dropped, as depicted in Figure 6(a).

The authors in [49] propose an extension to the sliding window that is called "jumping window," where at every clock cycle, a window of 'm' characters is jumped and a new window is inspected, as we can see in Figure 6(b). The throughput in this case is claimed to be multiples of that of the sliding window, however at the cost of replicating the same patterns $m-1$ times, as can be seen in Figure 6(b). In addition to TCAM solutions, a few closed addressing hash-based solutions are proposed [15].

In addition to these two research streams, there are also the automaton-based approaches [33, 5, 4]. However, these approaches have an inherent complexity that limits the total number of Regular Expressions (or complex patterns) that can be detected using a single chip [4]. There has been a lot of progress in this area through either using the DFA (Deterministic Finite Automaton) or Non-Deterministic Finite Automaton (NFA) [10].

4.0 OUR DEVELOPED HASHING SCHEMES AND TOOLS

As we mentioned earlier in Section 2, classical hash collision resolution schemes are inefficient for our packet-processing application while using the CA-RAM architecture. On one hand, the keys tend to have more collisions than the typical hashing of keys that are assumed to be uniformly distributed, while on the other hand, these schemes do not perform well with the CA-RAM architecture. For example, if we resolve hash collisions with linear probing, we might end up searching the entire CA-RAM for a certain key, which is not acceptable.

In this section we describe our proposed hash-based schemes that best suit both the three packet-processing applications and our main hardware architecture, CA-RAM. All our schemes are based on open addressing hash methodology; however, they could also be applied to the closed addressing hash methodology. These are all general schemes that are meant to work for the three applications. We introduce more specific optimizations in Chapters 5, 6 and 7 that exploit particular features of each application.

In Section 4.1, we describe our first hashing scheme, "Content-based HAsh Probing," or CHAP. CHAP is an extension to the hash probing techniques introduced in Section 3.1. In Section 4.2, we discuss our "Progressive Hashing" scheme, which is an extension to the grouped hashing introduced in Section 3.1.

Finally, in Section 4.3, we describe our "I-Mark" scheme. The I-Mark is a general scheme to mark those keys that are unique so that if we find one of them during a matching process, then we can stop looking for more matches. This helps in reducing the average memory access time (AMAT) and the overflow at the same time.

4.1 CONTENT-BASED HASH PROBING (CHAP)

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 (3.1). 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 (3.1), CHAP uses the same probing sequence, but with the constants $\beta_0, \beta_1, \cdots, \beta_m$ determined dynamically for each value of $h(k)$, depending on the distribution of the data stored in a particular hash table. Specifically, the probing sequence to insert a key "k" is:

$$h(k), \beta_0[h(k)], \beta_1[h(k)], \cdots, \beta_{m-1}[h(k)] \tag{4.1}$$

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 these 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 7 shows the basic idea of CHAP when $m = 2$. To match the overflow excess keys to specific rows, we need to collect all the overflow elements across all the rows. We achieve this by counting the excess elements per row, and finding for each row i two rows in which these overflow elements can fit. These two rows' indices are recorded in $\beta_0[i]$ and $\beta_1[i]$.

Assume that we are searching for a key k. If the hash function points to row $i = h(k)$ and it turns out that the input key k is not in this row, we check to see if the probing pointers at row i are defined or not. If defined, this means that there are other elements that belong to row i, but that 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.

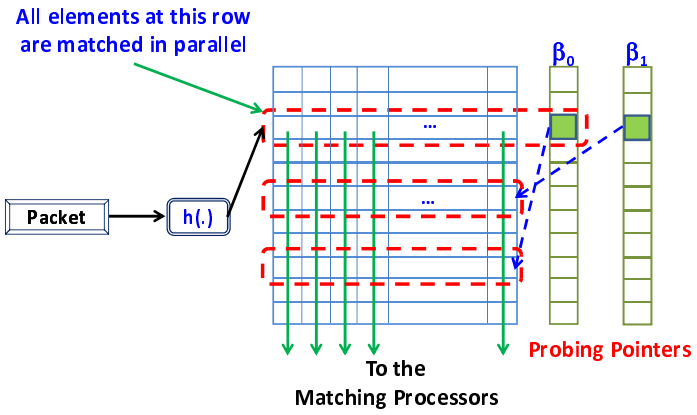


Figure 7: The CHAP basic concept.

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

$$h_0(k), h_1(k), \cdots, h_{H-1}(k), \beta_0[h_0(k)], \beta_1[h_1(k)], \cdots, \beta_{m-1}[h_{H-1}(k)] \tag{4.2}$$

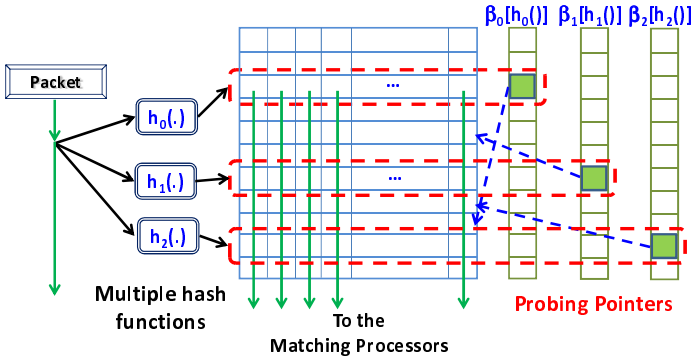


Figure 8: The CHAP(3,3).

In essence, we dedicate to each hash function a pointer per row. An example is shown in Figure 8 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 could share probing pointers among hash functions. Yet a third organization could assign multiple pointers for each hash function, which is the only possible organization for CHAP(1,m), when $m > 1$. In Chapters 5 and 6, 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 4.1 and 4.2 for the two organizations, respectively.

4.1.1 The CHAP Setup Algorithm

The main idea in this section is to prepare the keys and insert them into the CA-RAM hash table, which we call "setup phase." Algorithm 1 lays out the setup phase of CHAP. Before we describe the actual algorithm, we discuss an important issue, which is to reduce the search time for the CHAP algorithm. To achieve this, we have to stop at the first matching key during search in the CHAPs search algorithm, Algorithm 2. This is done by storing the keys according to their **specificity** (priority) from the most to the least [19]. The priority (specificity) of a key depends on the key type. In PF application, the key priority is just the prefix length; the same goes for the strings (signatures [1]) of the DPI application. However, there is a distinct priority value associated with each PC rule.

In Algorithm 1, $j = 0, \cdots, M-1$ is used to index the keys, where M is the total number of keys in a packet-processing database. Our hash table has $2^R = N$ rows, where R is the number of bits used to index the hash table. We use i as an index for hash functions and H as the maximum number of hash functions. An array of counters, $HC[\,]$ of size N, is used to count the number of elements that will be mapped to each row of the hash table. We define a two-dimensional array of counters $OC[\,][\,]$ of size $N \times H$ to count the overflow elements for

[1] Throughout this thesis we will use both terms *signature* and *string* interchangeably to present the same thing.

each hash function per row. The maximum value of a single counter in this array is equal to λ, where $\lambda \leq L$, and L is the number of keys stored per row. This bound comes from the fact that a hole, or an empty space in any row of the hash table, can never exceed L. The CHAP setup phase determines if the configuration parameters of the hash table are valid or not. In other words, do the parameters L, H, λ and N result in a mapping of the M keys into a single hash table with acceptable overflow or not?

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

```
1: CHAP_Setup(packet-processing Database)
2: Sort the keys according to specificity from highest to lowest
3: initialize the arrays HC[ ] and OC[ ][ ] to zeros
4: table_overflow = number of keys that will not hashed to the CA-RAM
5: for(j = 0; j < M; j++) {
6:     inserted = false
7:     for(i = 0; i < H; i++) {
8:       r_i = h_i(k_j) }
9:    for (i = 0; i < H AND inserted == false; i++) {
10:      if(HC[r_i] < L), then {
11:          HC[r_i]++
12:          inserted = true }
13:    }
14:    for (i = 0; i < H AND inserted == false; i++) {
15:      if(OC[r_i][i] < λ), then {
16:          OC[r_i][i]++
17:          inserted = true }
18:    }
19:    table_overflow++
20: }
```

Algorithm 1 calculates the number of keys to be assigned to each row. By "assigned," we mean not only the keys that are hashed to this row, but also the overflow keys that are supposed to be in this row but that will reside in other rows to which this row's probing

pointers point. It starts by sorting keys from highest priority to lowest priority, then initializing the two arrays HC and OC to zeros, while the table_overflow counter is initialized to the number of keys that are not going to be inserted in the hash table, because they are shorter than the minimum length the CA-RAM can handle (lines 1–2). Sorting the keys helps to stop at the first matching key, as will be proved in Section 4.1.2. The set of hash values $\{r_0, \cdots r_{H-1}\}$ for each key is calculated (lines 6–7). Then, the algorithm updates either HC or OC as follows: if there is a spot for the current key in HC, then the algorithm increments HC (lines 8–11); if not, it increments the corresponding OC counter (lines 12–15). In any case, the algorithm moves on to the next key.

When Algorithm 1 exits, table_overflow will include the number of keys that could not fit in either HC or OC (lines 16–17). If this 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 [2]. The progressive hashing scheme discussed in Section 4.2 may be applied in conjunction with CHAP to further reduce the overflow.

After completing the setup algorithm, we run a simple best-fit algorithm over the two arrays OC and HC, where we assign each overflow counter an address to a row that has a suitable hole that can accommodate the overflow keys. These addresses are what we call "probing pointers," and they are one per row per hash function. Finally, we populate the hash table by simply mapping the keys into their rows using the same hash functions that we used in Algorithm 1. Once we have overflow for a certain row and a certain hash function, we use the probing pointer associated with this pair to put the key into a different row. Note that we are using a small TCAM chip to store the overflow, i.e., "overflow_buffer," where we us it to store prefixes for PF application, rules for PC application and filters for the DPI application. We search the overflow_buffer after searching the main CA-RAM, which is covered in the next section, Section 4.1.2.

[2]Throughout this thesis we use the term "overflow_buffer" to represent a small embedded TCAM module that holds all the keys that will not stored in the CA-RAM.

4.1.2 Search in CHAP

In this section I discuss how search for a certain key is done in a CHAP-based hash table. We call our main hash table "**H_Table[ ][ ],**" of size $N \times L$ where N and L are respectively the number of rows and the row capacity. Each element in H_Table[ ][ ] consists of the actual key, H_Table[ ][ ].key, and a couple of corresponding fields: an ID number, H_Table[ ][ ].ID and the priority, H_Table[ ][ ].pri. The key priority is used to determine the most specific key (MSK), while the key ID is a unique ID number that is reported back to the control plane software, which is used for incremental updates of the keys database.

As discussed in Section 3.4, 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 most specific key in that bucket. A complete search might need to search more than one bucket. To measure the efficiency of the search in CHAP, we use the "Average Memory Access Time", **AMAT**, which is the average number of rows accessed for successful search.

Algorithm 2 The CHAP Search Algorithm.

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

The CHAP search algorithm, Algorithm 2, is straightforward. Given a key K, we calculate the row address $r_i(K) = h_i(K)$ and $r_{i+H} = \beta_i[h_i(K)]$, where $i = 0, \cdots, H-1$ (lines 1–3). For each row of the $2 \times H$ rows, we match the key against all the keys in this row in parallel, and if we hit at this row, we return the key ID number associated with the key (lines 4–6). If we do not find a match in these rows, we search the overflow_buffer (line 8).

In Section 4.1.1, we discussed the importance of storing the keys according to their priorities to be able to stop at the first matching key during the CHAP search algorithm. In addition to sorting and inserting the keys according to their priority, 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 MSK first.

Theorem 1. *In CHAP, the first matching key is the MSK if:*

1. *The keys are inserted in order of most specific to least specific.*
2. *The search's hash order, which includes both the order of accessing the probing pointers and the order of applying the hash functions, is the same as the insertion's hash order.*

Proof. In a restricted multiple hashing scheme, all the H hash functions are applied to all keys. For example, in the PF application, 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: $r_0(k_m), \cdots, r_{2H-1}(k_m)$, where $r_i(k_m) = h_i(k_m)$ for $i = 0, \cdots H-1$ and $r_i(k_m) = \beta_i[h_i(k_m)]$ for $i = H, \cdots, 2H-1$. In addition, assume that there exists a packet K 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 K as follows: $r_0(K), \cdots r_{2H-1}(K)$, we will match k_X at row r_X before matching k_Y at row r_Y. □

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

4.1.3 The Incremental Updates Under CHAP

An important issue in the packet-processing domain is the incremental updates of the databases. The number of keys included in a packet forwarding database grows with time [22, 54]. The updates consist of two basic operations, *Insert/Update* and *Delete* a key. In CHAP, the delete operation is straightforward. For any key deletion operation, we find the key first, then we delete it by storing all zeros and then decrement the row counter HC, which is used to keep track of the rows' populations. Deleting a key from any row does not require shifting, since the matching processors will ignore the deleted key spot as it will contain all zeros.

The basic idea of the insert/update operation, which is detailed in Algorithm 3, is to find the appropriate row that the new key should fit in. In this algorithm, Algorithm 3, we take into account that we report the most specific key. If we find out that the key already exists in our CA-RAM, the existing entry will be updated.

Algorithm 3 consists of two functions, **CHAP_Insert_Update()** and **Insert_in_Rows()**. The first function, CHAP_Insert_Update(), determines the appropriate rows to insert the new key K_n (lines 16–21). The second function is where the actual insertion is made, as it take a new key K_n, then it tries to insert it in a row from a range of rows starting from row index a all the way to row index b.

In the first function, the row array r_i, which has a size of $2 \times H$, is used to store the computed values of the hash functions of the new key, K_n, and the corresponding probing pointers (lines 2–4). Note that r_i is a global variable ,because it has to be accessed by the second function, Insert_in_Rows(). For each row r_i we match K_n against all the keys in this row and extract both the most specific key, K_l, and the least specific key, K_s, that match K_n (lines 5–7). We record the rows indices l and s that include K_l and K_s if such matchings are found. Depending on how specific K_n is related to 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 [22], is when K_n is equal to 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

Algorithm 3 The CHAP Insert Update Algorithm.

0: define r_i as an integer array of $2 \times H$ elements

1:**CHAP_Insert_Update** (New Key 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 $r_0, \cdots r_{2H-1}$, find: {

6: K_l = most specific key matching K_n and l = index of row containing K_l

7: K_s = least specific key matching K_n and s = index of row containing K_s }

8:**if**(K_l is not defined **AND** K_s is not defined), **then** {

9: **return**(Insert_in_Rows(K_n, 0 ,$(2H-1)$)) }

/* insert k_n in any of rows $r_0, \cdots, r_{2H-1}$ */

10:**else if** (($|K_n| == |K_l|$) **OR** ($|K_n| == |K_s|$)), **then** {

11: Replace K_l or K_s with K_n /*an update operation*/

12: **return** $(true)$}

13:**else if**($|K_n| > |K_l|$), **then** { **return**(Insert_in_Rows(K_n, 0, l))}

14:**else if**($|K_n| < |K_s|$), **then** { **return**(Insert_in_Rows(K_n, s, $(2H-1)$)) }

15:**else**, **return**(Insert_in_Rows(K_n, l, s))

}

16:**Insert_in_Rows** (key K_n, a, b) {

/* insert K_n in any of the rows $r_a, r_{a+1} \cdots, r_b$ */

17:**for**($i = a$; $i <= b$; $i++$) {

18: **if**(HC[r_i] $< L$), **then** {

19: insert K_n in r_i and HC[r_i]++

20: **return** $(true)$}

}

21:**return** $(false)$ }

if $|K_n|$[3] is larger than $|K_l|$, then we try to insert K_n into one of the buckets $\{r_0, \cdots, r_l\}$ 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 $\{r_s, \cdots, r_{2H-1}\}$ (line 14). Finally, if $|K_s| < |K_n| < |K_l|$, then we try to put K_n in any row between r_l and r_s (line 15).

In any case, the functions terminate successfully if we are able to insert K_n. Otherwise, we try to either insert K_n into the overflow_buffer, or use a backtracking scheme like "Cuckoo hashing" [35] to replace an existing key, say K_y, from the hash table by K_n, then try to recursively reinsert K_y back into the hash table [14].

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

[3]Throughout this thesis, we use the notation $|K|$ to represent the how specific a key 'K' is.

4.2 THE PROGRESSIVE HASHING SCHEME

We introduce our Progressive Hashing scheme (PH) as another effective mechanism for reducing collisions (hence overflow) for keys with wildcards (don't care bits) using open addressing hash systems. As we mentioned in Section 3.3, using multiple hash functions is efficient in reducing collisions in general. In the same section, the two multiple hashing schemes for dealing with don't care bits are described. In the restricted hashing scheme (see Figure 9(a)) the hash functions $h'_0(), \cdots, h'_3()$ are applied to all the keys in the hashing space, hence all hash functions are restricted to the specific (not the don't care) bits. On the other hand, in the grouped hashing (see Figure 9(b)) we split the hashing space into groups and a single hash function is associated with each group. In Figure 9(b), functions $h_0()$, $\cdots$, $h_3()$ are associated with groups 0, $\cdots$, 3 respectively.

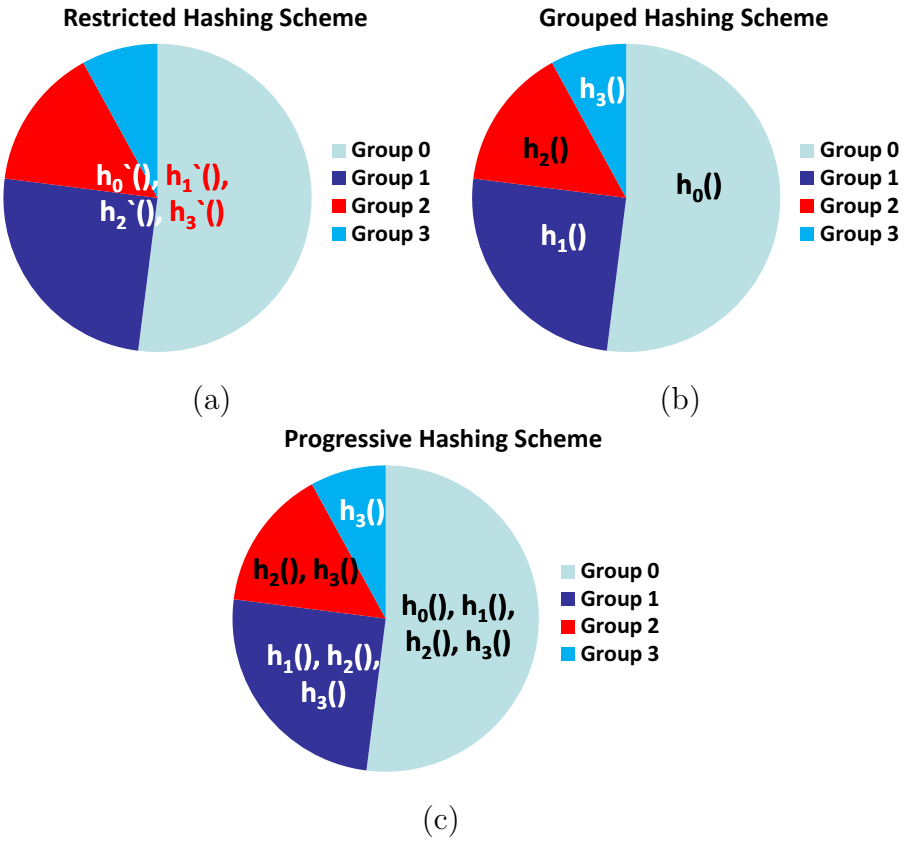


Figure 9: The evolution of the PH scheme.

In progressive hashing, we group the keys based on their lengths, as in the grouped hashing (GH). Consequently, groups with longer key lengths can use the hash functions of other groups that have shorter key lengths. For example, in Figure 2, for the PF application, group $S24$ can use the hash functions of groups $S20$ and $S16$. Motivated by this observation, we propose to apply the hash functions progressively as illustrated in Figure 10 to give some keys more chances to be mapped into 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 $h_i(\cdot)$.*

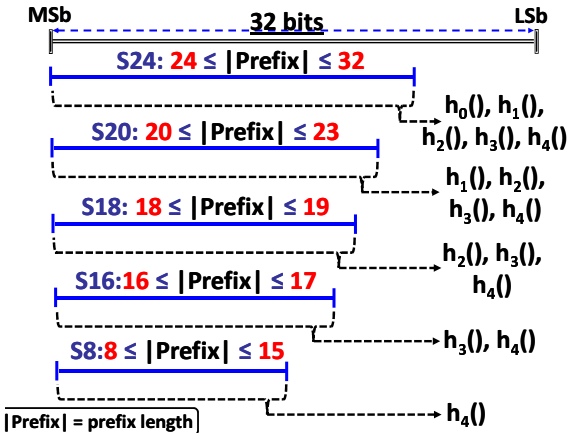


Figure 10: Applying the PH scheme on PF application.

For example, in Figure 2, the hash function $h_0(\cdot)$ is more specific than $h_1(\cdot)$, $h_2(\cdot)$, $h_3(\cdot)$ and $h_4(\cdot)$. Figure 10 demonstrates the PH scheme applied to the same groups in Figure 2. As an example, group $S24$, which is assigned to hash function $h_0(\cdot)$, can use the less specific hash functions of groups $S20$, $S18$, $S16$ and $S8$, as illustrated in Figure 10. In the next two sections, we show the PH setup and search algorithms.

4.2.1 The PH Setup Algorithm

In this section, we introduce the PH setup algorithm, Algorithm 4, where we prepare the keys to be mapped to the CA-RAM. Before dividing the keys into groups, we sort the keys according to their priorities from highest to lowest and insert them in that order. In this algorithm, Algorithm 4, $j = 0, \cdots, M-1$ is used to index the keys, where M is the total number of prefixes in an IP routing table. The goal is to map the prefixes into a hash table, H_Table[ ][ ] of size $N \times L$, where L = maximum bucket size, $N = 2^R$ maximum number of rows and R is the maximum number of bits used to index the hash table.

As indicated in the last section, each entry in H_Table[ ][ ] contains the field "*key*," which consists of the actual key (PC rule, PF prefix or DPI string), its length (or mask), the key ID number and the hash function field (lines 9–12) [4]. In the next section, Section 4.2.2, we show the importance of the "hash function" field. H is the maximum number of hash functions and an array of counters, $HC[]$ of size N, is used to count the number of elements that are mapped to each row of the hash table. A counter, *table_overflow*, that records the number of overflow elements, is initialized by the number of keys that are shorter than the minimum length to be hashed to the hash table. 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.

4.2.2 Searching in PH

In this section, we show how to find the Most Specific Key (MSK) in the PH scheme. But since we might find multiple matches, we want to guarantee that the first key that matches a packet is the best. Unfortunately, Theorem 1 cannot be used for progressive hashing as some keys have a different insertion's hash order than their search's hash order.

For a PF example, if a key K matches two prefixes $K_X \in (S18)$ and $K_Y \in (S16)$ in Figure 10(a), then K_X is the LPM of K. Assume that during the prefixes mapping, both

[4] The hash function index is used to store the index of the hash function that is used to store the prefix.

Algorithm 4 The PH Setup Algorithm.

PH_Setup(Keys Table){

1: Sort the keys from highest to lowest priorities and define the groups

2: Initialize $HC[]$ array to zeros and table_overflow = number of keys that will not be hashed

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++ }

18: }

Algorithm 5 The PH Search Algorithm.

Search_Hash_Table(Key K) {

1: **for**($i = 0$; $i < H$; $i++$) {

2: $r_i = h_i(K)$

3: **if**((K matches H_Table[r_i][j].key) **AND** ($i ==$ H_Table[r_i][j].h))

,**then** /* done in parallel for all values of j */

4: **return** H_Table[r_i][j].$port$

5: }

6: search the overflow_buffer }

prefixes are stored in two different rows as follows: $h_2(K_X) = r_X$ and $h_3(K_Y) = r_Y$. During the search for P we try all the five hash functions $r_0 = h_0(P), \cdots, r_4 = h_4(P)$. 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 K, i.e., $r_0 = r_Y$ or $r_1 = r_Y$. This means that we search r_Y before r_X, thus, we report K_Y as the LPM instead of K_X, which is wrong.

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

The PH search algorithm is given in Algorithm 5. It works as follows: for each key K that arrives at the packet-processing unit, we calculate the row index addresses $r_0 = h_0(K), \cdots, r_{H-1} = h_{H-1}(K)$ (lines 1 and 2). For each row r_i we match K 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 MSK in the bucket if and only if the stored hash function index ".h" is identical to the hash function index that is used to look up the key during the search (line 4). If we do not find any match, then we search the overflow_buffer (line 6).

4.2.3 Incremental Updates in PH

In Section 4.1.3, we discussed the importance of the incremental updates issue for the packet-processing engine. In this section, we describe how to perform the incremental updates for the progressive hashing scheme.

Deleting a certain key is straightforward in PH. It involves locating this key, deleting it by storing all zeros in its place and adjusting the corresponding HC row counter, as we do in CHAP 4.1.3. 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 $i = H, \cdots 2H - 1$. Also, we use only the hash functions that are applicable to the key being inserted. Specifically, we replace lines 1–3 in Algorithm 3 with the following lines:

2: **for**$(i = 0 \ ; \ i < H \ ; \ i++) \ \{$

3: **if** $(K_n \in \mathfrak{G}_i)$, **then**

4: $r_i = h_i(K_n) \ \}$

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

4.3 THE INDEPENDENT (I)-MARK SCHEME

The I-Mark scheme is an optimization that reduces the overflow for the PF application, and at the same time reduces the AMAT (average memory access time), which is average number of memory accesses per lookup for both the PC and DPI applications. The unique feature of this scheme is that it is orthogonal to both CHAP and PH schemes; and hence it can be applied in conjunction with both.

The I-Mark scheme is motivated by the observation that under the progressive hashing scheme, some keys can use more hash functions than others. So, if we insert those keys last (after the others), then we may reduce the collisions and hence the overflow because of the flexibility of using multiple hash functions. However, this is not simple, since some keys have to be inserted before other keys because of some "dependence" relationship. Definition 2 formally defines the dependence relationship between keys:

Definition 2. *If the key in a packet P_i matches two keys, K_i and K_j, then K_i and K_j are called dependent keys.*

As we discussed earlier, we insert the keys into the hash table according to their priorities, starting from the highest to the lowest. This helps us find the most specific key first during searching. Independent keys, however, can be inserted in any order, and not according to their priority.

Following this observation, we define a binary flag, I-Mark, which is set to 1 if the key is independent of any other key in the database, and is reset to 0 if not. 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. To determine if a certain key is independent or not, this key has to be neither a subkey (e.g., substring in case of DPI database or subprefix in case of PF) of any other keys, nor a super key (i.e, there are other key(s) that is (are) subkey of this key). In other words, we run an $O(n^2)$ algorithm that compares each key against all other keys in the database. If there is at least one key that is either a subkey or a superkey of the current key; we mark this current key's flag to 0, otherwise we mark it 1.

There are many ways to insert the independent set; one in particular is to insert the independent keys first before the dependent ones, where we insert the independent keys in a

reverse order according to the number of hash functions each key is assigned to. For example, given the PF example in Figure 10, 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 by fewer hash functions before trying to accommodate other keys that can be hashed by more hash functions.

In addition to allowing key insertions in arbitrary order, we note that once one of the independent keys is matched during the search; then we can stop the search, since we are guaranteed that there are no more matches. This helps us reduce the AMAT of the PC and DPI applications significantly.

4.4 CONCLUSION

In conclusion, I introduced three novel hashing techniques for keys with wildcards. My first technique, CHAP, is a probing scheme that is more efficient than other non-content-based classical probing schemes. Instead of searching the subsequent buckets blindly (like in linear and quadratic probings) for a key that does not exist in a certain bucket, we intelligently search those buckets that contain the overflow of this bucket. This technique limits the number of buckets to be searched; hence it lowers both the AMAT and WMAT.

In progressive hashing I introduced a new way of handling the keys with wildcards in the packet-processing units that use hashing in storing those keys. Instead of using all hash functions to map all the keys, we split these keys into groups based on the length of the specific (non-wildcard) part of the keys. Then we assign each group a hash function. Finally, for these groups that have longer keys than other groups, we allow the former to use the hash functions of the latter. Our progressive hashing provides a better collision resolution mechanism than classical restricted hashing, where all hash functions are applied to all keys, but only on the shortest specific parts of those keys.

As for our last technique, the I-mark, we use the observation that some keys are neither subkeys nor superkeys of or from other keys. Thus, while searching for a certain key and a

key that is I-marked is found, we can stop the search as there is no other key that might be more specific than this key. This lowers the average memory access time (AMAT).

In addition, I-marked keys can be inserted in any order to reduce the chance of collisions. This is unlike those keys that are not I-marked where they have to be inserted in a specific order (longer or most specific first, then shorter or less specific after that) to guarantee finding the MSK (most specific key) in the first hit during the search. Note that the I-mark scheme can be applied independently from the hashing scheme (i.e, progressive hashing, restricted hashing and CHAP).

In the next three chapters, we describe how to make use of these three techniques to design a CA-RAM-based packet forwarding engine (Chapter 5), packet classification engine (Chapter 6), and pattern-matching unit for the DPI application (Chapter 7).

5.0 THE PACKET FORWARDING APPLICATION

In this chapter, we discuss application of the presented tools discussed in Chapter 4 to the PF application. We start with Section 5.1 that demonstrates the required CA-RAM hardware changes to convert it into a packet forwarding engine. Then in Section 5.3 we demonstrate the effectiveness of our CHAP scheme, when applied to PF. CHAP uses restricted hashing (RH), where we use only the first 16 bits of the IPv4 prefixes for hashing. Since these 16 bits are not good enough to prevent collisions, and hence overflow, we use multiple hash functions.

In Section 5.4, we introduce our Progressive Hashing-based PF engine augmented with the I-Mark scheme. Unlike CHAP, we use more bits in hashing the prefixes in PH, which effectively reduces the overflow. Section 5.5 combines both PH and CHAP schemes in a single solution, while in Section 5.6 we combine the PH and I-Mark solutions. Finally, in both Sections 5.7 and 5.8 we show the performance estimation using state-of-the-art SRAM technologies and also by using the CACTI [52] memory simulator.

5.1 THE CA-RAM ARCHITECTURE FOR PACKET FORWARDING

In this section, we describe the hardware customization that is needed to enable the CA-RAM as a packet forwarding engine. In Figure 11 we show a packet forwarding CA-RAM with two hash functions and two probing pointers. The main difference between the generic CA-RAM, Figure 3, and the CA-RAM that is enabled for PF, Figure 11, is that the key part is replaced by the prefix, while the data is replaced by the prefix length (or mask). In addition, we include one bit for each CA-RAM cell used for the I-Mark field. Another

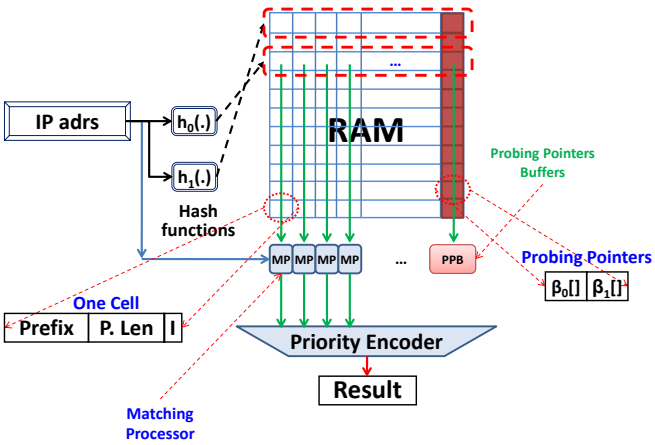


Figure 11: The CA-RAM as a packet forwarding engine.

difference is that we dedicate at the end of each row (bucket) extra bits that are used to store the probing pointers.

During the search process, the probing pointers for each indexed bucket are stored in a special buffer as shown at the bottom right corner of Figure 11. These buffers are used to index the RAM part of the CA-RAM during the search and after exhausting the buckets that are indexed by the hash functions.

The CA-RAM matching processors in this case are actually prefix matching circuits. Figure 12 shows an example of the prefix matching circuit for the CA-RAM, where we assume IPv4 IP's (i.e., 32 bit long).

The main component in the prefix matching circuit is the small RAM-based "prefix mask table" that is used to store the masks for the IP prefixes. The other alternative to this table is to directly store the mask alongside the prefix, which leads to almost double the space required to store any IP lookup table. The prefix mask table consists of 33 entries for IPv4, where each entry is 32 bits wide for a total size of 132 bytes. For IPv6, the prefix mask table size is going to be 129 entries, where each entry is 128 bits wide for a total size of 2064 bytes. The remaining architecture is straightforward, where for each incoming IP address; a mask is applied to this address given the length of the stored IP prefix that we are trying

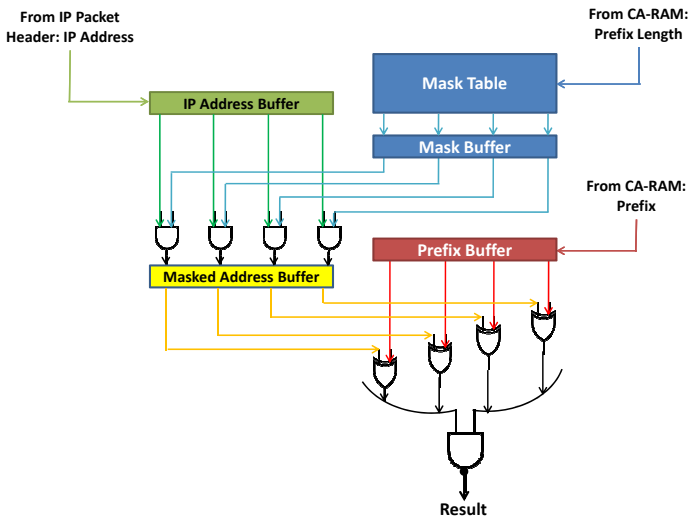


Figure 12: The CA-RAM prefix matching circuit for packet forwarding application.

to match against this IP address. Note that the IP prefix addresses are of different lengths, because they represent different networks.

Then, the masked IP address is XORed with the stored prefix to see if both matched or not. If there are multiple matchings on the same row, we use the priority encoder in Figure 11 to determine which prefix is the LPM. In this case, we assume that the entire process is pipelined into the following stages:

- Once the packet arrives, the router extracts its destination IP address.
- Next, this IP address is fed to the CA-RAM to calculate the hash index that is used to access the specific row.
- At the same time, the same IP is fed to each of the matching processors of the CA-RAM.
- Once a row is indexed, it is fetched from the RAM part of the CA-RAM and fed to the matching processors.
- Each matching processor will access its mask table for the mask that is indexed by the stored prefix length field.
- In the next step, the MP will AND the incoming IP address with the fetched mask, creating the masked address.
- The PM then XORs the masked incoming IP address with the stored prefix.

- The final step is to see if we have a match or not, which is done by having a NAND gate to sum all XOR gates output.

- If the result is zero, then we have a match, otherwise, we do not.

In the next section we describe our evaluation methodology for our CA-RAM-based PF engine. The actual results are depicted in the sections that are after this section.

5.2 EVALUATION METHODOLOGY

In this section, I start by describing the evaluation methodology of my PF solution. The reason is that we want to show that we are using real-life data and to define the configuration of our experiments, , as well as to define the metrics that we will use later to describe the effectiveness of our solution against other solutions.

For evaluation purposes, we used C++ to build our own simulation environment that 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 $\mathbf{H_3}$ [38] hashing families which are simple and 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 [41] on January 31, 2009. Table 2 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. 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.

We generally use two metrics: overflow percentage and the AMAT. The overflow is defined as the ratio of the prefixes that overflowed to the total number of prefixes of a routing table. The AMAT (average memory access time) is the average number of memory lookups to find the longest prefix match (LPM) for all the packets in a given trace.

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	**Average**	**0.81**	**280,242**

Table 2: The statistics of the IP forwarding tables on January 31^{st} 2009.

5.3 THE RESTRICTED HASHING-CHAP-BASED SOLUTION

We discuss our first PF engine design, where we used the restricted hashing (RH) in addition to our CHAP probing technique to accommodate all the prefixes in a single CA-RAM chip. Although we later show that RH is in general less effective than the GH hashing scheme, here we want to demonstrate our content-based probing is very effective in reducing the overflow. Here we show some of the obtained results [1].

Before we evaluate the CHAP, we shed some light on the restricted hashing. Figure 13 shows a histogram of how many prefixes share the most significant 16 bits for 15 real IP tables given in Table 2. The number of prefixes that are shorter than 16 is less than 2% of the lookup table population, and these are ignored in the figure.

[1] More results can be found in our work [19].

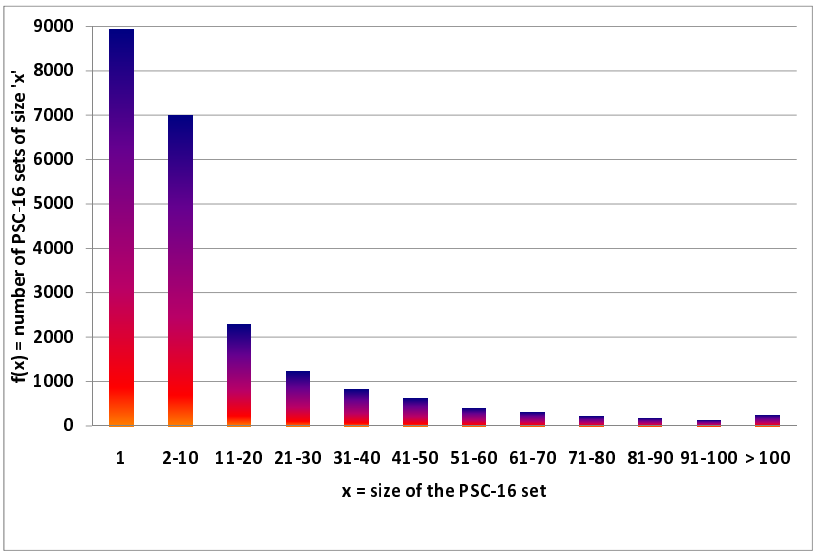
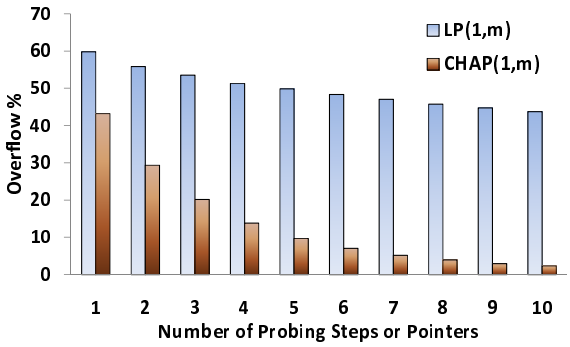


Figure 13: The histogram of the prefixes sharing the first 16 bits.

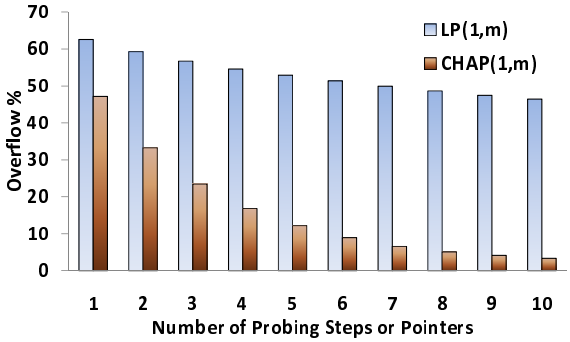
To explain the figure, we define PSC-16 (Prefix Set with Common 16 bits) as a set that contains prefixes from the same IP forwarding table having an identical 16 first bits (sharing a common 16-bit prefix). The size of a PSC-16 set is the number of prefixes in that set. We then define $f(x)$ as the number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

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

In Figure 14, we compare CHAP against a simple linear probing (LP) technique.



(a) {L=200 , N=1024}

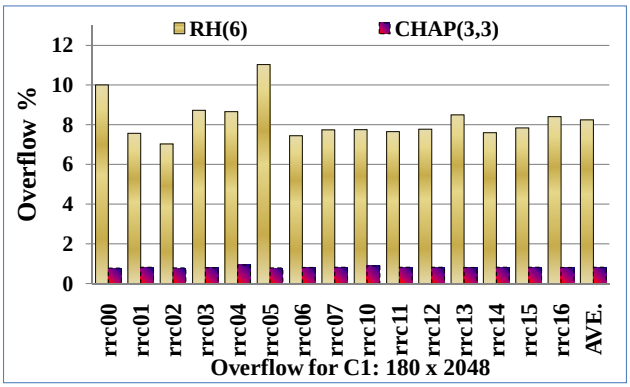


(b) {L=100 , N=2048}

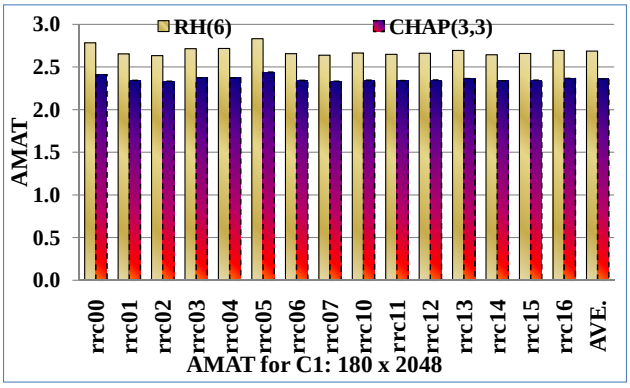
Figure 14: The overflow of CHAP(1, m) vs. linear probing(1, m) for table rrc07.

Figure 14 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. Next we compare our CHAP(H,H) against RH(2H). We plot in Figure 15 (a) the overflow and (b) the average memory access time of both schemes for one configuration C1:{L = 180, N = 2048}. We increased the bucket size by one prefix for RH(2H) to compensate for the CHAP overhead, as we discussed previously.

As we can see, CHAP$(3, 3)$ is better than RH(6) for all files in terms of both the AMAT and the overflow. In fact CHAP$(3, 3)$ reduces the overflow by 90% and at the same time improves the AMAT by 12.2% for this configuration. The details of evaluating CHAP performance under other configurations can be found in [19, 21].



(a)



(b)

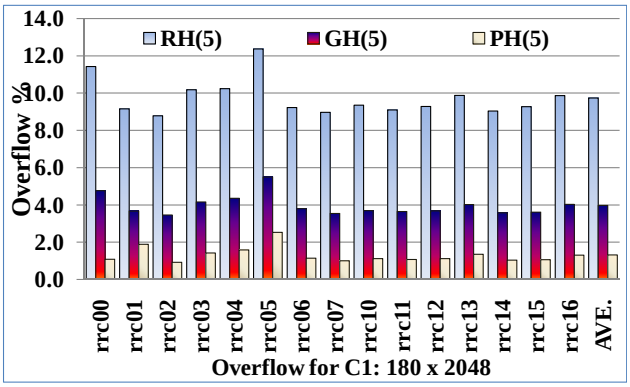
Figure 15: The (a) Average overflow, and (b) AMAT for CHAP(3,3) vs. RH(6) for fifteen forwarding tables for C1: {L = 180 , N = 2048}

5.4 THE PROGRESSIVE HASHING-BASED SOLUTION

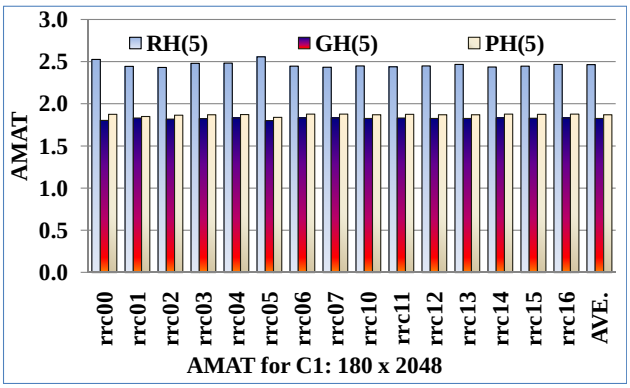
In this section, we show some results [2] where we used the progressive hashing for the PF application. In Figure 16(a) we show the overflow percentage for all 15 routing tables for configuration C1:{L = 180, N = 2048}, for the three schemes: RH, GH and PH. On average, PH reduces the overflow by 86.5% compared to RH and by 66.9% compared to GH. At the same time, the AMAT (Figure 16(b)) of PH is improved by 22.0% over RH and 3.4% over GH. Note that the overflow prefixes are usually added to the overflow_buffer that is always

[2]More results can be fond in our work [20].

searched after exhausting all possible buckets in the CA-RAM [20, 21].



(a)



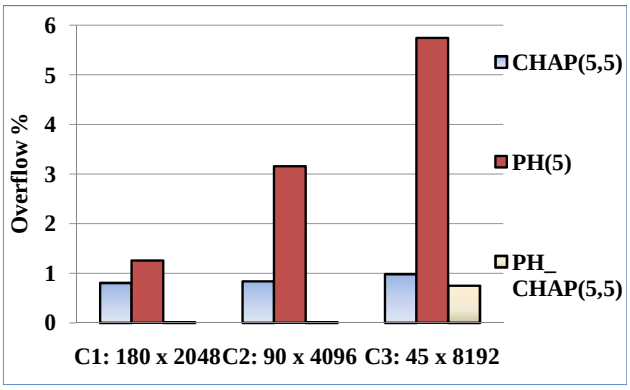
(b)

Figure 16: The (a) Average overflow, and (b) AMAT of RH(5) vs. GH(5) vs. PH(5) for fifteen forwarding tables for C1: $\{180 \times 2048\}$.

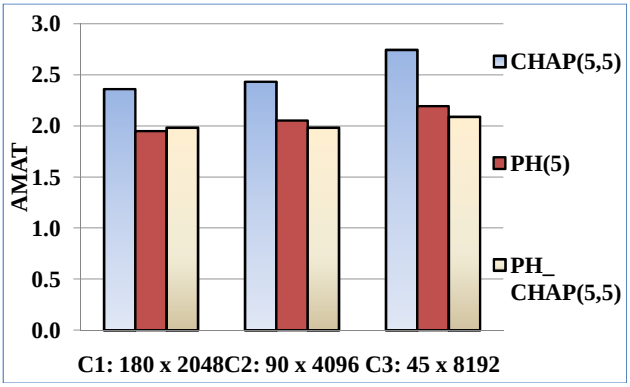
5.5 ADDING CHAP TO PROGRESSIVE HASHING

In this section, we combine both PH and CHAP schemes into a third hybrid scheme that we call **PH_CHAP(H,H)** [3]. In essence, after applying PH to the prefixes, we add probing pointers to radically reduce the overflow. As described before, CHAP uses restricted hashing where the hash functions use only the most significant 16 bits of all the prefixes. However, we use PH instead of RH to get better performance than either scheme.

[3]For more about this please refer to our work in [20, 21].



(a)



(b)

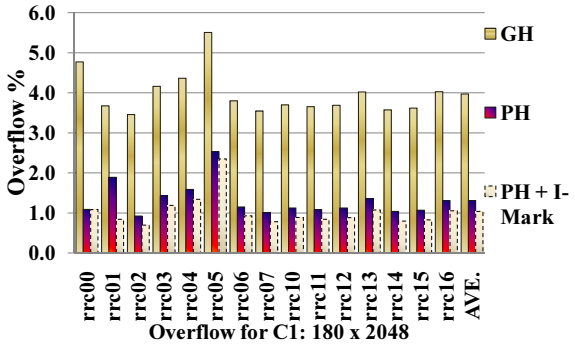
Figure 17: The (a) Average overflow, and (b) Average AMAT of CHAP(5,5) vs. PH(5) vs. PH_CHAP(5,5) for three configurations.

In Figure 17(a) we show the average overflow of CHAP(5,5), PH(5) and PH_CHAP(5,5) for the same three configurations $C1 : \{L = 180, N = 2048\}$, $C2 : \{L = 90, N = 4096\}$ and $C3 : \{L = 45, N = 8192\}$. The largest average overflow belongs to PH(5) with 3.38% and the lowest average overflow is 0.3% for PH_CHAP(5,5) over the 3 configurations. The first two configurations, C1 and C2, have zero overflow for PH_CHAP(5,5) with a reduction of 100%. For the third configuration, C3, PH_CHAP(5,5) reduced the overflow by 86.9% over PH(5) and by 23.6% over CHAP(5,5). At the same time, we note that PH_CHAP(5,5) has a lower AMAT (Figure 17(b)) than CHAP(5,5) and PH(5) with average of 19.7% and 2.3% improvements.

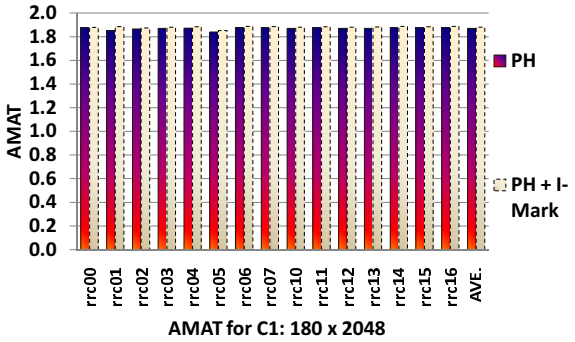
5.6 ADDING THE I-MARK SCHEME TO THE PROGRESSIVE HASHING

Now, we apply the I-mark scheme in addition to PH on the stored prefixes. This enables us to insert some of those prefixes (the I-marked ones) in any order, which will also reduce the overflow. In addition, it also reduces the AMAT during the search; if we find any of those prefixes that are I-marked, we terminate the search, as we will not find any longer prefix that matches the same IP address. In Figure 18(a) we show the overflow of the 15 routing tables for the same configuration, $C1 : \{L = 180, N = 2048\}$ and for three schemes: GH, PH and PH + I-Mark.

On average, the PH reduces the overflow by 66.7% compared to the GH scheme. At the same time, the AMAT (Figure 18(b)) of the modified PH is decreased by 62.6% over the GH scheme, which has a constant AMAT of 5. The PH + I-Mark reduces the overflow by 72.8%, and decreases the AMAT by 62.4% over the GH scheme.



(a)



(b)

Figure 18: The (a) Average overflow, and (b) AMAT of GH vs. PH vs. PH+I-Mark for fifteen forwarding tables for C1: $\{180 \times 2048\}$.

5.7 PERFORMANCE ESTIMATION OF CHAP AND PH

We estimate both the average and the minimum throughput of our hybrid "PH_CHAP(H,H)" scheme using two different SRAM memory architectures. The first memory architecture that we use is the 500MHz QDR III SRAM [37]. This is the latest architecture among the famous Quad Data Rate (QDR) SRAM family, soon to be commercially available soon. The QDR is an SRAM architecture that can transfer up to four words of data in each clock cycle [37]. The other memory architecture is a state-of-the-art CMOS technology SRAM memory design [55], which reports an experimental single chip of 36.375 MB that runs at 4.0GHz. Since our scheme depends on a set-associative RAM, we conservatively assume that the original RAM clock rate is halved. This will take care of the delay for the hash functions computation and the matching logic.

Assuming an AMAT of ∼2.0, if the clock rates are 250 MHz and 2.0GHz for the aforementioned architectures, then we have a forwarding throughput of 125 mega packets per second and 1.0 giga packets per second. In other words, 40 Gbps and 320 Gbps for the minimum packet size of 40 bytes.

In addition to the AMAT, we use another throughput metric, which is the worst-case memory access time (WMAT). The WMAT of the hybrid PH_CHAP(5,5) scheme is $10+1 = 11$ ($2 \times H$ + searching the overflow_buffer). Thus, the worst-case throughput is slightly less than 20% of the AMAT throughput. Note that all these estimated rates are for a single CA-RAM chip.

To increase both throughputs (AMAT-based and WMAT-based), we can use multiple CA-RAM chips per line card as described in [25, 24]. In addition, a typical router has multiple line cards [11]. The aggregate throughput of a router is calculated as the sum of throughputs of the line cards.

5.8 PERFORMANCE ESTIMATION USING CACTI

In addition to the above estimations, we use "CACTI" (version 5.3) the cache simulator [52] to estimate the throughput in addition to the area and the power requirements of our PH-CHAP(5,5) scheme. CACTI is a standard cache simulator that takes in the following input parameters: cache (memory) capacity, cache block size (also known as cache line size), cache associativity, technology generation, number of ports, and number of independent banks (not sharing address and data lines). CACTI produces the cache (memory) configuration that minimizes delay, along with its power and area characteristics. In addition to cache, CACTI also simulates different SRAM and DRAM memories [52].

We estimate a total memory of 2.25MB (actually CACTI takes only the total amount of memory as a power of 2) and that the row width is 512 bytes. Note that this memory is enough to store the largest IP table (rrc05), which has over 301K IPv4 prefixes. Since we propose to build our schemes as high-performance hardware ASIC chip; we use the High Performance International Technology Roadmap for Semiconductors SRAM 45nm technology model (ITRS-HP).

There is a special version of CACTI that is modified to simulate TCAMs [1], which we call *TCAM-CACTI*. We assume that the TCAM chip should be able to store the maximum IP table from Table 2, and that each IPv4 prefix needs 4 bytes plus 4 bytes for the next hop address. Thus the total TCAM capacity is $301,383 \times 8 = 2.29$MB.

	Total Power (nJ)	Total Delay (nS)	Max Freq. (MHz)	Total Area (mm^2)
CARAM	2252.30	86.34	423.98	17.86
TCAM	9948.01	382.86	24.90	58.85

Figure 19: The CACTI results of CA-RAM vs. TCAM, where both has sizes of 2.5MB.

In Figure 19, we show the CACTI results for both the TCAM and CA-RAM. Note that we simulated 2.5MB as both CACTI simulators use power-of-two numbers for the rows and their widths. In terms of total dynamic power, the CA-RAM consumes about 22.6% of the TCAM. The total delay (access time) represents the amount of time in which a packet traverses a chip from end to end. Our CA-RAM has a lower access time than the equivalent TCAM configurations by 77.4%. The maximum frequency is the frequency of the smallest

pipelining stage, where CA-RAM has a maximum frequency 94% greater than the TCAM. Finally, the CA-RAM has an area of 30.3% that of the TCAM.

5.9 CONCLUSION

In this chapter, I introduced multiple architectures for a CA-RAM-based packet forwarding engine. These architectures are based on the three schemes that I introduced earlier in Chapter 4, namely the CHAP, the progressive hashing and the I-Mark scheme.

We showed that by the addition of CHAP to restricted hashing, though restricted hashing is less effective than progressive hashing in reducing hash collisions, we achieve considerable overflow reduction. The CHAP achieves more than 72% overflow reduction over the equivalent linear probing, where both schemes have exactly the same WMAT. Also, CHAP reduces the overflow by 90% over the RH, while CHAP achieves more than 12% reduction in the AMAT over the PH.

The overall results of PH against both the equivalent GH and RH CA-RAMs are as follows: PH reduces the overflow by more than 85% and 66% over RH and GH respectively. At the same time, PH has improved the AMAT by 22% and 3.4% over RH and GH respectively.

I also added these schemes in conjunction with each other (e.g., CHAP with PH, PH with I-Mark) to improve the overall performance of our CA-RAM PF engine. For example, the CHAP plus PH hybrid scheme (PH_CHAP) improved the overflow reduction by 100% over both individual PH and CHAP schemes (for two out of three configurations). At the same time, PH_CHAP reduces the AMAT by more than 19% and 2% over CHAP and PH schemes respectively.

I used the I-Mark with the PH, which reduced the overflow by 95% and 66% over both RH and GH schemes, while reducing the AMAT. Finally, I showed that my schemes can have an average forwarding speed of 320 Gbps if using future SRAM technology and 40 Gbps with the standard QDR III SRAM, where we used less than 2.5MB of RAM to store, 301K IPv4 prefixes. The CACTI simulator shows that CA-RAM can run on a frequency of 423MHz while maintaining a moderate area and power versus the TCAM.

6.0 THE PACKET CLASSIFICATION APPLICATION

In this chapter, we discuss how to use our hash-based algorithmic tools with the CA-RAM memory architecture to solve the packet classification problem. Because of the fact that some packet classification rules have all wildcard bits in one or two of the source or the destination prefixes fields, we will not be able to apply the restricted hashing scheme to the PC application. Instead, we consider both the grouped hashing and its extension the progressive hashing scheme. As we mentioned in Section 1, for the PC application, we use a special type of hashing that is called Tuple Space Search (TSS).

The original TSS is a $2D$ space where one dimension represents the source prefix length and the second dimension is the destination prefix length as shown in Figure 2-b. Each dimension starts from 0 and runs till 32, which is the maximum length of the IPv4 address. Thus, the TSS has $33 \times 33 = 1089$ cells, where each cell originally is an independent hash table that is augmented with special data structures that are called "markers and precomputations" [47]. Note that our TSS is different from the original TSS in that we store all the rules inside one big hash table, CA-RAM, but we use the same notion of splitting rules into a $2D$ space, then use hashing.

The markers and precomputations of the original TSS scheme are tools to find the best matching rule out of multiple rules. Each cell has a "marker" that points to the next cell where the best matching rule may exist in case no match is found at the present cell. In addition, a cell has a "pre-computed" cell address marking the next best cell to search for a match, in case a match is found at the current cell. The main problem with the original TSS scheme is that many of the hash tables are empty [54, 45]. In addition, finding the optimal (best or most cost efficient) matching rule presents a problem, since we may have to search the entire TSS to find it. In other words, the original TSS has a high average memory

access time that is not practical (i.e., searching 1089 hash tables). This problem has been tackled by the introduction of markers and precomputations [32], which represent extra-high preprocessing overhead [45].

One solution to the original TSS is to group multiple cells into a single cell, or in other words, to merge multiple hash tables into one hash table, called "coarse-grained TSS" [45]. This solution eliminates the TSS problem by augmenting the $2D$ TSS into multiple segments and merging all the hash tables in each segment (partition) into a single hash table. However, coarse-grained TSS still suffers from the overhead of computing the markers and precomputations, in addition to space fragmentation due to the use of multiple hash tables.

We introduce three different hash-based TSS solutions in this chapter. The three solutions are based on our version of the coarse-grained TSS optimization and the progressive hashing. We either use a single CA-RAM (hash table) or two CA-RAMs to store our version of the TSS. Moreover, we avoid using markers and precomputations via applying our I-Mark scheme [20, 21]. We note that the I-mark overhead computation algorithm is $O(N^2)$, where N is the number of filters, while each of the markers and the precomputations require $O(W^d)$, where W is the number of bits in one dimension of the TSS and d is the number of dimensions where $2 \leq d \leq 5$.

We start in Section 6.1, where we discuss in general the special hardware requirements for utilizing our CA-RAM as a packet classification engine. In Section 6.2, we show our first scheme, which uses the progressive hashing in addition to the I-Mark. It also serves as a baseline for our subsequent schemes as it simply adds all the PC rules into a single hash table (CA-RAM). Section 6.3 is our second PC solution, where instead of using the same coarse-grained $2D$ TSS we use a dynamic partitioning scheme to split the TSS. Our main goals in this solution are to: enhance the space utilization, reduce the average memory lookup time, and hence, reduce use of power. Then, in Section 6.4, we show our two CA-RAMs PC solution, where we use the fact that some PC databases have non-unique pairs of source and destination prefixes of their rules. We use one CA-RAM to store the source and destination prefixes of the PC rules; while the second CA-RAM is used to store the rest of the rules. Finally, in Section 6.5, we discuss the simulation results and the performance estimation of our three PC CA-RAM-based solutions.

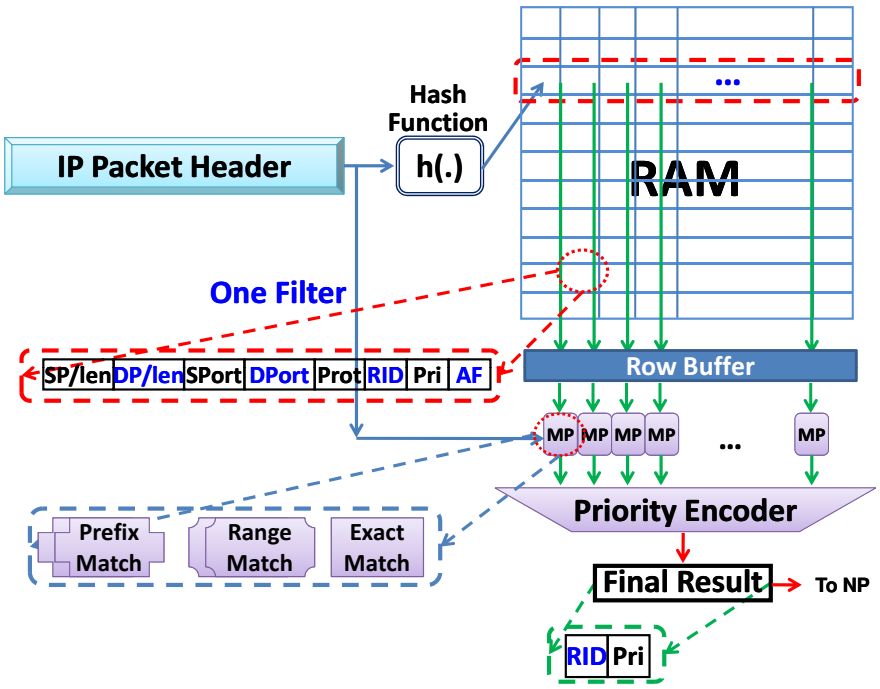


Figure 20: The CA-RAM detailed architecture for the packet classification application.

6.1 THE CA-RAM ARCHITECTURE FOR PACKET CLASSIFICATION USING PROGRESSIVE HASHING

In addition to the algorithmic part of our solutions, there is an important architectural part that has to be added to the original CA-RAM architecture in order for it to be capable of handling the packet classification process. In addition to the prefix matching function, the matching processors (at the bottom of Figure 3) also have to support exact matching for the protocol field and range matching for the source and destination ports. Figure 20 shows the details of the CA-RAM architecture that has been enabled (modified) for PC application.

Each cell in the RAM part of the CA-RAM in Figure 20 consists of the following fields:

- Source Prefix (SP)
- Destination Prefix (DP)
- Source Port Range (SPort)
- Destination Port Range (DPort)

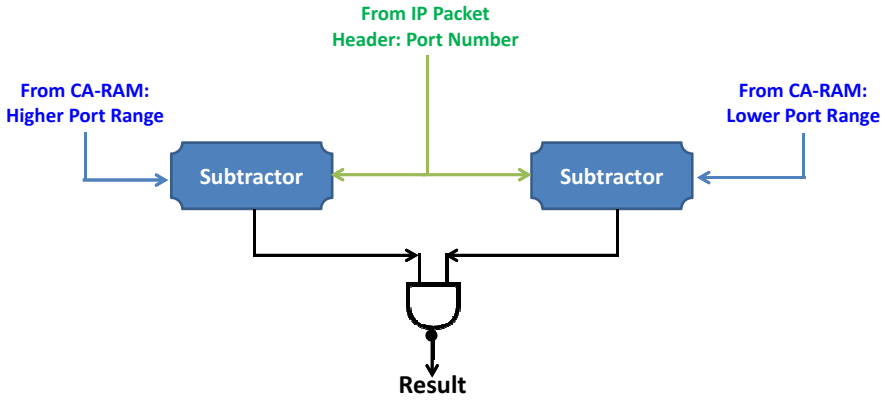


Figure 21: The range matching circuit for the CA-RAM PC application.

- Rule ID (RID)
- Priority (Pri)
- Auxiliary Field (AF)

Thus each cell represents one PC filter (or rule). The rule ID, RID, is a unique 16-bit number to distinguish between different rules, while the priority filed, Pri, is to determine the best matching rule in case of multiple matches and it ranges from 4 to 16 bits long. The auxiliary field, AF, consists of 4-bit hash function index (we talk about this field in Section 6.2) and a 1-bit I-Mark. The matching logic is shown in the bottom of Figure 20, where there are three types of matching circuits plus the priority encoder to find the best matching rule in a certain row.

The hardware implementation of the prefix matching circuit is covered in detail in Section 5.1. In Figure 21, we show an example of the range matching circuit for the CA-RAM matching processors. The main component is what in Figure 21 we call "Range Comparator", which is basically a subtractor. The range compactor idea is simple: to store the port range in the CA-RAM, high range and low range, and subtracts the input port from both the high range and the low range. If the port lies in the stored range, both subtractors will have a zero carry (borrow) and hence the NAND gate will generate '1' indicating a match, and 0 otherwise.

The exact matching circuit is made out of comparators which are implemented using "XOR" gates plus "NAND" gates, as shown in Figure 22. We use the exact matching circuit for both the port matching and the hash function ID matching. The XOR gates will make sure that the stored protocol and the input protocol in this example are identical. Finally, the NAND gate will generate '1' if we have a match and '0' otherwise. This same circuit can be used to check the hash function ID and the I-Mark bit to stop the search, as we describe in the next section, Section refsec:PC:phimark.

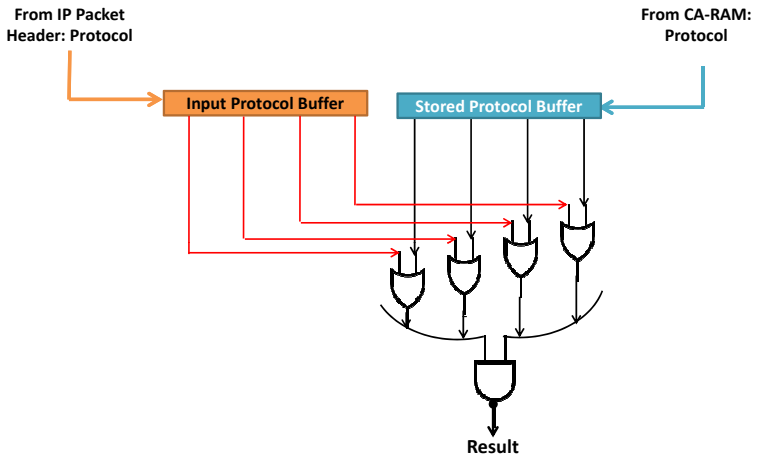


Figure 22: The exact matching circuit for the CA-RAM PC application.

6.2 THE PH AND THE I-MARK HYBRID SOLUTIONS

We start by showing Figure 23, which is the progressive hashing equivalent of the grouped hashing for TSS shown in Figure 2-(b). We call Figure 23 a "partitions plan" or simply a *p*artition. By a "**partition**" we mean the actual groups we make out of the tuple space, which includes both their numbers (number of groups) and their associated physical boundaries (which bits are used from the source and destination addresses to define the group). In the next section, Section 6.3, we are going to add one more metric to the partitions and that is the number of hash functions that are associated with each partition (group). However in this section we use one-to-one mapping between groups and hash functions as we did with

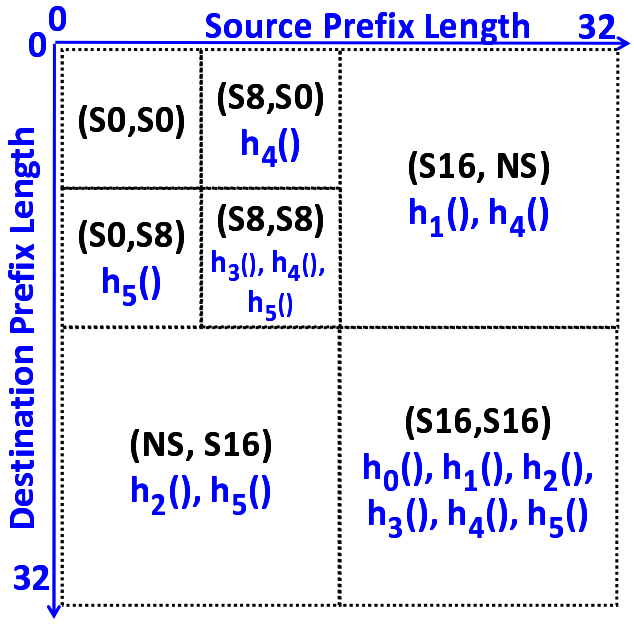


Figure 23: Applying PH on packet classification application.

the packet forwarding application.

Since we want to reduce the average memory-access time (AMAT) we would like to find the best matching filter first. However, this is not possible, because there is no strict order priority among the filters. Unlike the PF application, where the longer the prefix, the higher its priority, there are cases in the PC application where this is not applicable due to the nature of the PC filters. For example, consider a packet that matches two different filters: f_1 from group (S16, NS) and f_2 from group (NS, S16) of Table 3. Which filter is the best match is actually up to the explicit priority tag field of the filters.

To tackle this problem, we use the I-Mark scheme. The idea is simple; we mark those filters that are independent of other filters, hence called independent filters, by setting their I-Mark bit to 1. The definition of independent filters follows the same concept that is introduced in Definition 2 in Section 4.3. We insert the independent filters first in the hash table, followed by the dependent filters. Thus, we are guaranteed to find the independent filters first.

However, and due to the nature of the progressive hashing where some keys are using

some hash functions and not all the hash functions, we need an extra caution before we decide to stop at the first independent matching filter. Here is an example: assume there is a packet P_X that matches filter $K_X \in (NS, S16)$ and filter $K_Y \in (S16, NS)$ from Figure 23, and that K_X has a higher priority than K_Y. Assume that during the filter mapping, the two filters are stored in two different rows as follows: $h_2(K_X) = X$ and $h_4(K_Y) = Y$. During the search for P_X we try all the six hash functions $h_0()$ to $h_5()$. Assume that one of the hash functions was 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 best match filter 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 $h_4()$ and not $h_0()$ or $h_1()$, hence K_Y has to be skipped as a match as there might be a better match, K_X in this case. In Section 6.5.1 we compare between grouped hashing scheme and progressive hashing with the I-Mark scheme.

6.3 THE DYNAMIC TSS PH SOLUTION

In the last section, Section 6.2, we used only a single grouping strategy to split all the databases according to the partition plan shown in Figure 23. As we are going to learn from the experiment section, Section 6.5.1, our first solution, will not work efficiently for large data sets, or in other words, is not scalable. Hence, instead of using exactly the same partitions for all the files, we use different partitions for different files. In other words, we vary the number of groups per partition along with the physical boundaries that we use for each PC database. Moreover, we assign multiple hash functions for each group and use the probing pointers to eventually eliminate the overflow. For example, if one of the database groups (NS, S16) has more rules than other groups, we will assign two or more hash functions to this group.

Table 3 lists the 11 PC databases (each is $30K+$ rules) that we generate using the ClassBench [50] tool. In this table, we show the detailed stats of the near-optimal partitions

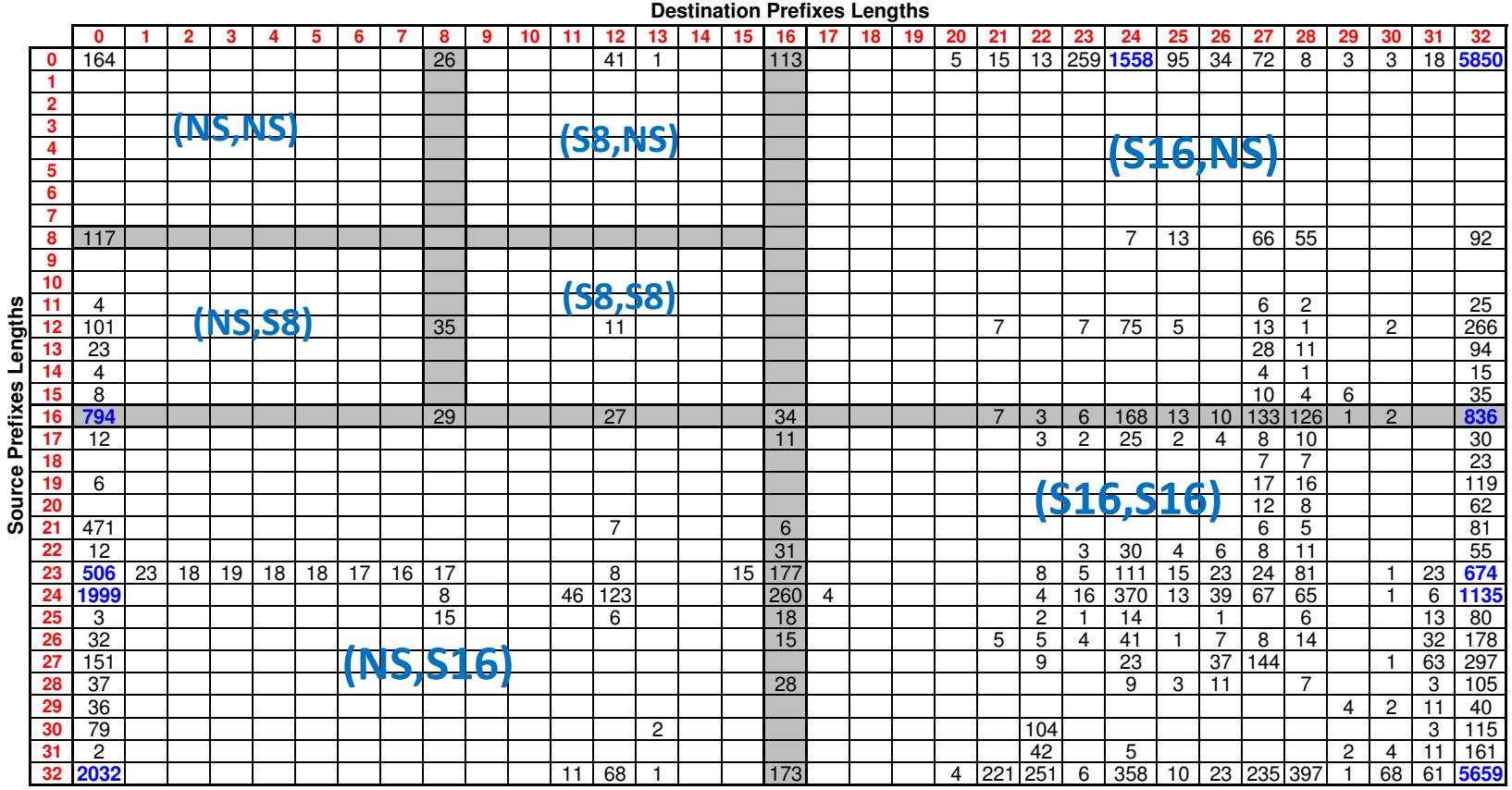
plan for each of the 11 files. In the next subsection, we talk about how we generated such partitions.

The idea stemmed from the observation that not all PC databases that we experiment with responded well, in terms of overflow, using the initial partitions shown in Figure 23. Figure 24 shows this observation, where the X-axis represents the destination prefix length and the Y-axis is the source prefix length. This figure represents the average number of PC rules that belong to each tuple based on the PC databases shown in Table 3. In this figure, each cell, $c_{i,j}$, represents the average number of rules that have at least i bits source prefix and j bits destination prefix. For example, $c_{16,16}$ has 34 rules and $c_{0,32}$ has 2032 rules on average. We use light gray shades for the boundary cells in Figure 24 to show our original partitions shown in Figure 23.

We notice that more than 75% of the tuples are empty (exactly 819 out of 1089 tuples). Also that most of the rules are concentrated in the lower right-hand side and that, in general, the rules are not evenly distributed among any group of the groups marked in gray. The main observation is that groups $(S8, S8)$, $(NS, S8)$ and $(S8, NS)$ are almost empty, hence there is no need to assign those cells distinct groups.

Table Name	Size	G_0	%	G_1	%	G_2	%	G_3	%	Non-unique S^8&D^* Pairs %
$ACL1_{30K}$	30,072	S8,NS	99	NS,S8	93	S30,S22	74	S23,S24	72	0.34
$ACL2_{30K}$	30,609	NS,S8	89	NS,S12	86	S12,NS	78	S8,NS	69	0.58
$ACL3_{30K}$	30,990	S8,NS	83	NS,S20	80	S21,NS	67	S12,S24	63	20.67
$ACL4_{30K}$	30,533	S8,NS	85	NS,S8	83	NS, S20	79	S12,S8	68	26.01
$ACL5_{30K}$	30,482	S8,S27	100	S16,S27	92	S24,S27	68	S8,S32	62	48.58
$FW1_{30K}$	30,021	NS,S27	64	NS,S32	60	S8,NS	46	S16,NS	44	0.76
$FW2_{30K}$	30,174	S16,NS	88	S24,Ns	65	NS,S24	26	S16,S24	14	0.00
$FW3_{30K}$	30,513	NS,S32	65	S8,NS	35	S16,NS	33	S21,NS	31	1.10
$FW4_{30K}$	30,251	NS,S8	67	NS,S23	66	NS,S24	57	S8,Ns	49	2.81
$IPC1_{30K}$	30,211	NS,S16	87	S16,NS	83	NS,S24	73	S16,S16	71	4.97
$IPC2_{30K}$	30,000	NS,S32	89	S22,S32	47	S32,NS	36	S32,S32	25	0.00
Ave_all$_{30K}$	30,351	NS,S8	78	NS,S16	76	S8,NS	72	NS,S24	70	9.62

Table 3: The statistics of the ClassBench's $30K$ PC databases, where $G_0 \cdots G_3$ represent the top four groups containing the most rules of each database.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0	164								26				41	1			113				5	15	13	259	1558	95	34	72	8	3	3	18	5850
1																																	
2																																	
3																																	
4																																	
5																																	
6																																	
7																																	
8	117																								7	13		66	55				92
9																																	
10																																	
11	4																											6	2				25
12	101								35				11									7		7	75	5		13	1		2		266
13	23																											28	11				94
14	4																											4	1				15
15	8																											10	4	6			35
16	794								29				27				34					7	3	6	168	13	10	133	126	1	2		836
17	12																11						3	2	25	2	4	8	10				30
18																												7	7				23
19	6																											17	16				119
20																												12	8				62
21	471												7				6											6	5				81
22	12																31							3	30	4	6	8	11				55
23	506	23	18	19	18	18	17	16	17				8			15	177						8	5	111	15	23	24	81		1	23	674
24	1999								8			46	123				260	4					4	16	370	13	39	67	65		1	6	1135
25	3								15				6				18						2	1	14		1		6			13	80
26	32																15					5	5	4	41	1	7	8	14			32	178
27	151																						9		23		37	144			1	63	297
28	37																28								9	3	11		7			3	105
29	36																													4	2	11	40
30	79													2									104									3	115
31	2																						42		5					2	4	11	161
32	2032											11	68	1			173				4	221	251	6	358	10	23	235	397	1	68	61	5659

Figure 24: The average tuple space representation of the 11 PC databases given in Table 3.

6.3.1 The Setup Algorithm for Dynamic TSS Solution

In this section, we describe how to process the rules in a PC database for our dynamic TSS solution. The main idea is to show how to select the best TSS partitions for each file we have. By *TSS partitions* we mean the following variables: C = number of groups, H = total number of hash functions. Our classical progressive hashing has both of these variables constant at 6, where each group has only one hash function. In general, $1 \leq C \leq 1089$, where $1089 = 33 \times 33$ and 33 represents the length of the IPv4 prefixes from 0 up to 32.

Our algorithm is a heuristic to find the partitions that lead to the minimal overflow for a given PC database. We start our algorithm by assuming a constant number of groups, and for each group we assign initially one hash function. Note that if we increase the number of groups and the number of hash functions per group, the worst-case memory access time will also be increased. Usually, we use the classical progressive hashing partitions as initialization for each database.

Next, we calculate the overflow by running multiple iterations of the progressive hashing setup algorithm, Algorithm 4. After each iteration, we store the overflow in a variable, OF_i, where the subscript i represents the iteration number. We try to adjust (fine tune) the partition for each iteration so that the overflow is reduced by doing the following steps:

- We first adjust the partitions boundaries according to the TSS representation of the current PC database. For example, if the first group was $S22, S16$ for a ceratin PC database, we can make it $S24, S16$ if the overflow will be lower in this case.
- Second, if needed, we may merge two or more groups into one group or split a large group with two or more smaller groups. For example, group $S8, S8$ has more than 90% of the rules; we may consider replacing it by $S8, NS$ and $NS, S8$.
- Finally, if needed, we add more hash functions to any of the groups that have more rules than can be handled by a single hash function.

Not all the files have to go through the last two steps; i.e., some files do not have to have more hash functions per group or more groups, but all files have to go through at least one boundary reform. For example, file $ACL5_{30K}$ has 6 groups: $(S32, S32)$, $(S32, S27)$, $(S24, S27)$, $(S16, S27)$, $(S8, 32)$ and $(S8, S27)$ as shown in Figure 25. We notice that the

rules are again concentrated at the bottom-left corner and that most of the groups we choose in Figure 23.

So, instead of using the old groups, we ended up the six new groups we mentioned earlier. In this example, group $(S32, S32)$ has two hash functions while the other groups have only one hash function. This is because this group has 43% of the entire database, which means that it needs more hash functions than other groups that have fewer rules.

6.3.2 Incremental Updates For The Dynamic TSS Solution

Since the only significant difference between this scheme and our progressive scheme is that the partitions and number of hashing functions are chosen differently for each file, then the basic incremental updates scheme for progressive hashing will work for dynamic TSS. This is true for the basic dynamic TSS solution.

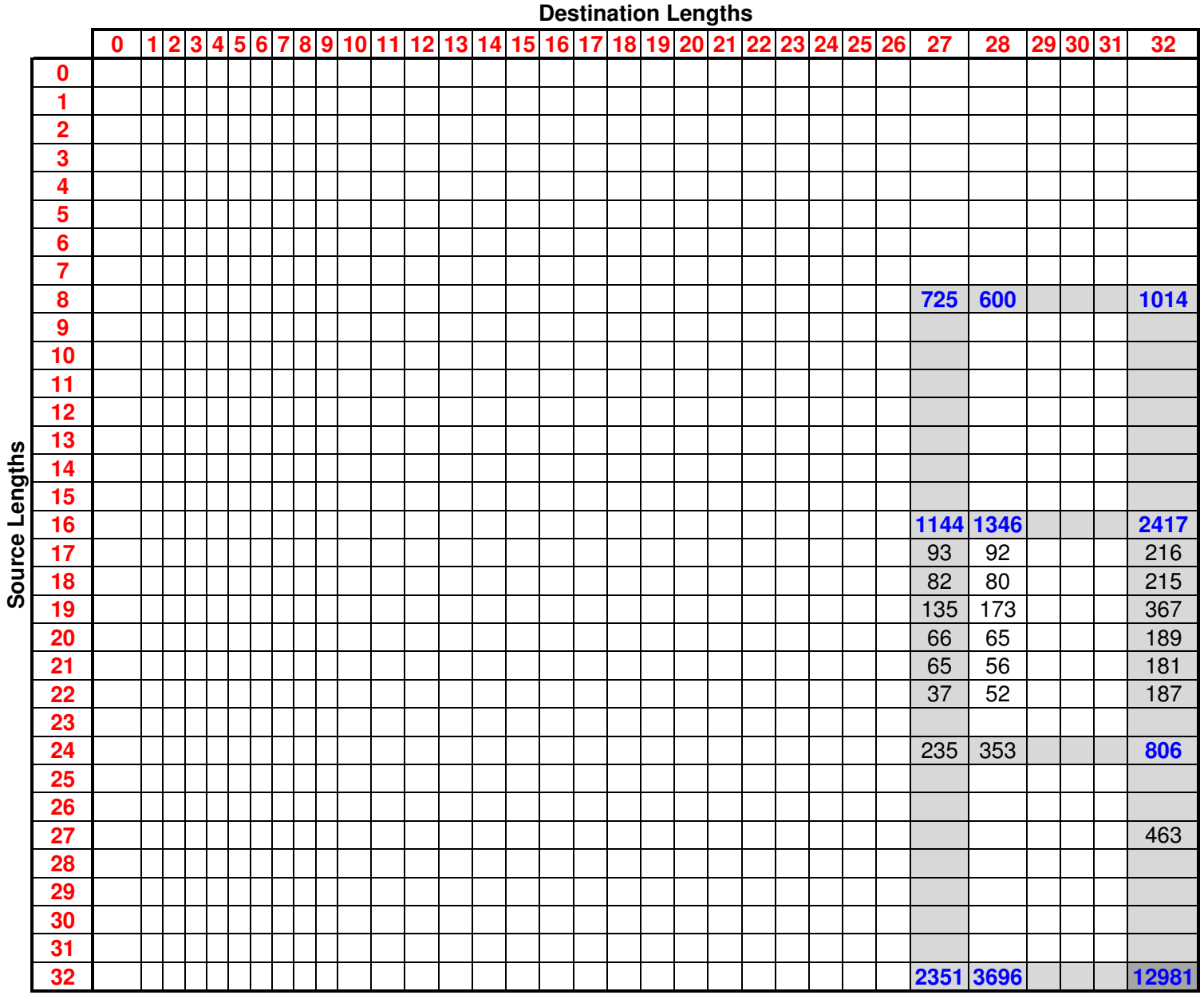


Figure 25: An example of the TSS representation of $ACL5_{30K}$ db of Table 3.

6.4 THE TWO CA-RAM ARCHITECTURE FOR DYNAMIC TSS SOLUTION

In this section, we describe an alternative CA-RAM architecture in case we want to take advantage of the fact that some PC databases have non-unique pairs of source and destination prefixes. Also, we want to make sure that the width of our CA-RAM hash table is physically feasible, which is very important as the PC filters are very wide (roughly 160 bits per filter). Figure 26 shows the overview of this architecture.

Throughout this thesis, we call the first CA-RAM the "**main**" CA-RAM and the second CA-RAM the "**auxiliary**" CA-RAM. The main idea is to split the PC rule storage between two CA-RAMs, where for a certain PC rule we store the IP prefix pair in the main CA-RAM and the remaining of the tuples (e.g., source and destination ports, protocol, flags, etc) are stored in the auxiliary CA-RAM. After finding the right pair of prefixes in the first CA-RAM, we use pipelining to search the second CA-RAM for the rest of the rule. This split allows us to store the unique prefix pair only once to save space and to reduce the collisions. In this solutions, we use dynamic TSS with progressive hashing and I-Mark as we did in the last PC solution, Section 6.3, to store the PC rules. Thus, this solution does not need special algorithms adjustment as it uses the same algorithms as the previous one-CA-RAM solution. In the next section, Section 6.4.1, we show the architectural details of this solution.

6.4.1 The Architectural Aspects of The Two CA-RAMs PC Solution

Figure 27 shows more details of the two CA-RAMs PC solution architecture. We start by describing the main CA-RAM architecture, which is similar to the PC CA-RAM shown in Figure 20 that is used for the PH+I-Mark and dynamic TSS PC solutions. The left part of Figure 27 shows a sample element from the main CA-RAM row, which consists of the following fields:

- Unique source and destination prefix (SP and DP) pair.
- Rule ID (RID), which is used to uniquely identify each PC rule.
- Rule priority (Pri) which is used in case we want to report the best matching rule.

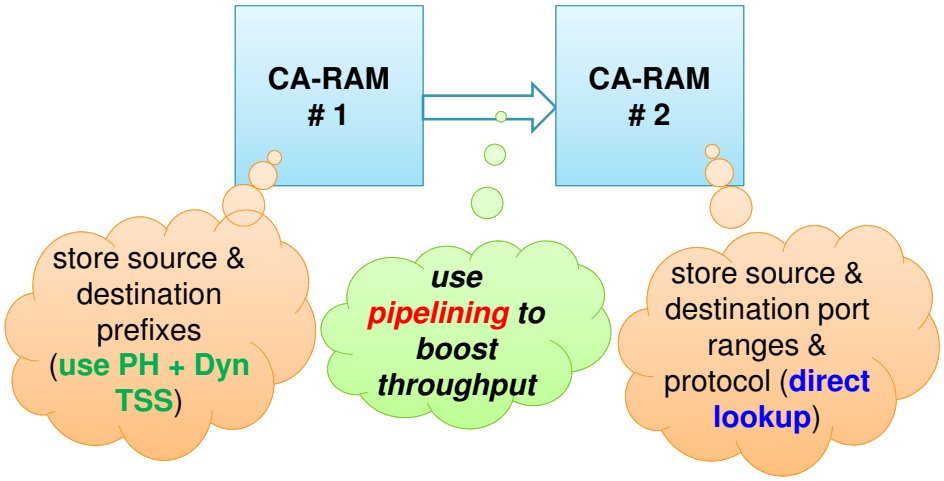


Figure 26: The two CA-RAM architecture: an overview.

- Multiple bit (m), to indicate if the source and destination pair is unique or not.

The differences between the main CA-RAM of Figure 27 and that of Figure 20 are that the matching processors of the main CA-RAM contain only prefix matching, while the CA-RAM of Figure 20 has three types of matching circuits. In addition, the main CA-RAM stores source and destination prefix pair, not the entire PC rule. In order to identify if a prefix pair is unique or not, we use a single bit, m, to indicate this fact. This bit is set to 1 if this pair of prefixes is shared among more than one rule and is reset to 0 otherwise.

During the setup phase, when we try to insert a new rule to a specific row, r, we first check to see if this rule's source and destination prefix pair are already stored at r or not. If the pair is already stored, we set its m bit to one, and store the remainder of the rule at the associated row in the auxiliary CA-RAM. Note that we associate one row of the auxiliary CA-RAM to each row of the main CA-RAM (i.e., one-to-one mapping). If the prefix pair is new, we store it at r if it has space, while initializing its m bit to 0, and then store the remainder of its rule in the auxiliary CA-RAM. Each unique pair of prefixes is stored with a single rule ID. This rule ID is the first rule that has that prefix pair and we use it as a unique identifier to all the rules that share this pair. This identifier is also used to generate an index to the auxiliary CA-RAM row that contains a list of all the rules that share this

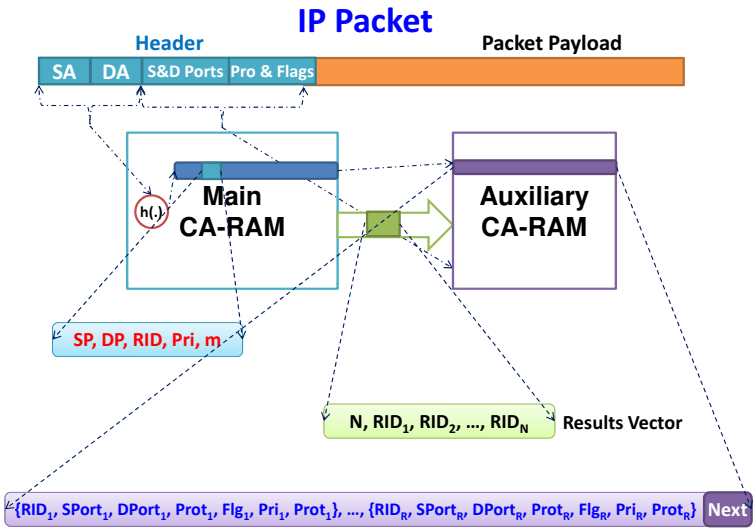


Figure 27: The two CA-RAM architecture: the main CA-RAM row element format, the results vector format and the auxiliary CA-RAM row format.

prefix pair, as shown in Figure 28.

Once we receive an IP packet header, we use the source and destination addresses to generate the main CA-RAM row(s) index/indices. In this phase, we match only the source and destination addresses to the source and destination prefixes stored at each row. If we find a match, or multiple matches, we record each rule ID that has been matched in a data structure that we call the "results vector." For each main CA-RAM row that is matched, we generate a single results vector.

In the next phase, we use pipelining to submit the current results vector to the auxiliary CA-RAM to finalize the match. Note that the main CA-RAM matching is a partial matching, as we match only the source and destination prefixes, while the final "full" matching happens only at the auxiliary CA-RAM. At the same time, we continue searching the main CA-RAM using the progressive hashing search algorithm, Algorithm 5.

Figure 28 shows the details of the auxiliary CA-RAM, which is a bit more complicated than the main CA-RAM. In addition, the bottom part of Figure 28 shows the details of each row of the auxiliary CA-RAM. Each row consists of multiple elements, where each element contains the following fields:

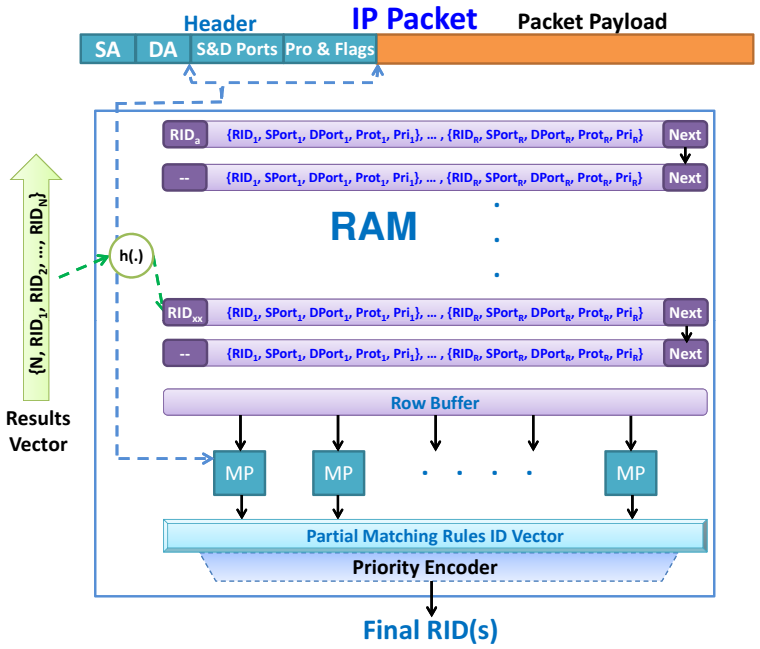


Figure 28: The auxiliary CA-RAM detailed architecture and its row format.

- Rule ID (RID), used to uniquely identify each PC rule.
- Source and Destination Ports (SPort & DPort).
- Protocol (Prot) field.
- Flags (Flg) field.
- Rule priority (Pri), used when we want to report the best matching rule.

Each row of the auxiliary CA-RAM is indexed by a unique rule ID and at the end of each row there is a pointer to the next row. The idea is to store only the ports protocol and flags of all the rules that share the same prefix pair but differ in the rest of the fields in a single row. In case a single row does not fit all these rules, we use its "$NEXT$" pointer which points to another row that contains the remainder of the rules. These $NEXT$ pointer values are set almost exactly the same way we set the probing pointers for our CHAP algorithm (Algorithm 1 in Section 4.1), where we use the best fit algorithm to find a row that can hold the excess (overflow) for each overflowing row. Thus, these pointers depend on the content of each PC database that is being stored in the CA-RAM.

For each matching vector received, the auxiliary CA-RAM calculates a series of row indices from the RIDs of the matching vector. Since each rule ID is 16 bits, we use a simple hash index generator to extract the required row index. The auxiliary CA-RAM starts processing each row trying to find a match.

The full PC matching is done once we match the rest of the tuples of a certain rule that has an I-Mark flag equal to 1. The matching is also terminated if we search all the possible rows and find multiple rules that match the current IP packet header, but with no I-Mark flag set. In this case we choose the rule with the highest priority through the priority encoder shown in the bottom of Figure 28. Next, we discuss the incremental updates of this PC solution.

6.4.2 Incremental Updates For The Two CA-RAM Solution

In this section, we discuss the incremental updates for our third PC solution, the two CA-RAM architecture. In the case of using a two CA-RAM architecture, as in Figure 26, the basic progressive hashing incremental updates scheme does not work directly. First, we have to find where the updated rule is stored, in the main and auxiliary CA-RAMs, or only in the auxiliary CA-RAM. The rule might be stored only in the auxiliary CA-RAM as it might share the same source and destination prefix pair with another rule. In any case, the control plane software should have a full data structure keeping track of where each rule is stored.

In the case where the rule is stored in the auxiliary CA-RAM only, the incremental update will be straightforward: delete the rule or update its priority or action. The other case is when the updated rule is stored in both CA-RAMs. If the update operation is to delete or modify this rule, then we have to make sure that such a rule does not share its prefixes with other rules. If this is the case, we go ahead and remove or modify the rule from both CA-RAMs while updating the counters of each CA-RAM. In the other case, we keep this rule content in the main CA-RAM, but delete or modify the appropriate part of this rule in the auxiliary CA-RAM.

There is one final case that should be considered when the rule does not already exist; i.e., a new rule is inserted. In this case, we check to see if this new rule shares its prefix

pair with an already stored rule. If so, we store the new rule in the auxiliary CA-RAM by finding an empty space in the row that shares the same prefix pair. If not, we apply the basic incremental update scheme for progressive hashing, Section 4.2.3, since the new rule has a unique prefix pair.

6.5 THE SIMULATION RESULTS AND THE EVALUATION METHODOLOGY FOR OUR PC SOLUTIONS

We used C++ to build our own simulation environment that is similar to the one given in Section 5. The ClassBench tool [50] was used to provide synthetic PC databases, as well as traces, from real databases. There are three families of PC applications defined in the ClassBench: 1- IPC, which is a legacy version of firewalls format, 2- ACL, which is an Access Control List format and 3- FW, which is a modern version of firewalls format. We generated 11 synthetic databases and their traces using ClassBench and their statistics are given in Table 3, where all the tables have more than $30K$. In Table 3 we show the first four groups that have most of the PC rules.

In addition to the fact that most of the PC databases need different partitions, Table 3 shows the number of non-unique source and destination pairs. On average, the number of repeated source and destination prefix pairs is a little less than 10%. This is a very misleading ratio as some files (two to be exact) have 0% repeated prefixes, while others have between 20% and 48% ($ACL3_{30K}$, $ACL4_{30K}$ and $ACL5_{30K}$).

As we did for the PF application (Section 5.2), we define a hardware configuration, C_i, by the number of rows, N and the number of entries per row, L. In this section, we define our hardware configurations in Table 4:

$C_1 : \{60 \times 1K\}$	$C_2 : \{40 \times 1K\}$	$C_3 : \{32 \times 1K\}$
$C_4 : \{30 \times 2K\}$	$C_5 : \{20 \times 2K\}$	$C_6 : \{16 \times 2K\}$
$C_7 : \{15 \times 4K\}$	$C_8 : \{10 \times 4K\}$	$C_9 : \{8 \times 4K\}$

Table 4: The nine CA-RAM hardware configurations that we use in validating our PC solutions.

These configurations are split into three sub-configurations: three $1K$ configurations ,

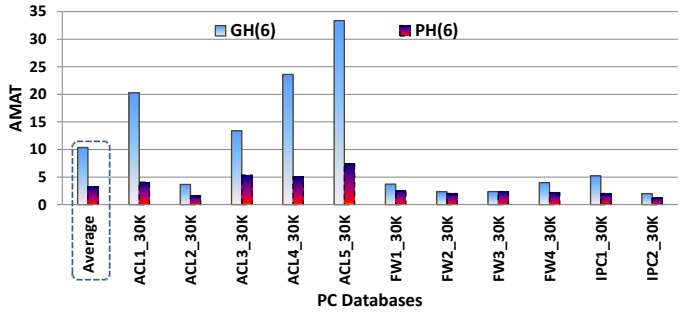
three $2K$ configurations, and three $4K$ configurations. Note that the configurations $1K \times 60$, $2K \times 30$ and $4K \times 15$ all share the same loading factor of 49%, while $1K \times 40$, $2K \times 20$ and $4K \times 10$ have a loading factor of 73% and finally, $1K \times 32$, $2K \times 16$ and $4K \times 8$ have 89% loading factor. The higher the loading factor, the better the RAM space utilization is. The configurations $1K \times 60$, $2K \times 30$ and $4K \times 15$ which have less than 50% loading factor are very important for the industry, as they represent the trend of having two data structures: one online for the lookup process, and one standby for the control plane to incorporate the incremental updates. After a certain time period, usually a few milliseconds, the network processor switches the two copies to reflect the most recent packet-processing database.

6.5.1 Experimental Results for Progressive Hashing and I-Mark Hybrid Solution

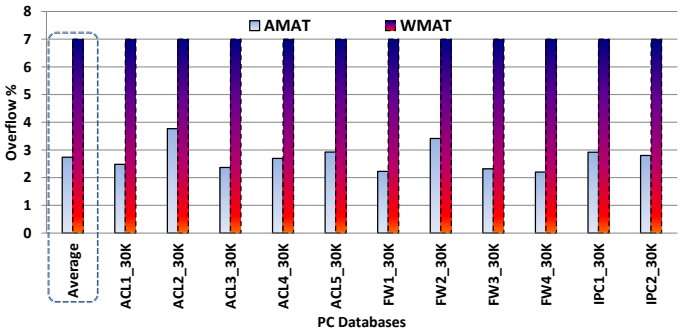
Our first PC solution uses PH in addition to I-Mark optimization. This solution is going to serve mainly as a proof of concept and we use the results obtained here to describe the other two solutions. In a previous paper [20] we used different benchmarks in terms of sizes to test this solution, where the average number of rules per benchmark was around $8K$. However, the benchmarks used in [20] are generated from the same seeds that we used to generate our benchmarks in Table 3. In this chapter, we use similar benchmarks, but with an average of $30K$ rules per database, which is three times as large as the benchmarks used in [20].

Figure 29 shows a comparison between the GH scheme versus the PH scheme with the I-Mark optimization (to stop at the first match) for all the 11 PC databases for one configuration: $C_1 : \{60 \times 1K\}$. We chose this configuration as an example of the overflow and AMAT performance of our progressive hashing solution. In addition, we use the notation $GH(X)$ or $PH(X)$ to indicate how many groups each scheme is using. PH with I-Mark reduces the overflow (Figure 29(a)) by 80% on average compared to GH. For PH with I-Mark, the AMAT is lower than the Worst-case Memory Access Time (WMAT), which is 7, by 61%. We do not show the AMAT of GH since it has a constant AMAT of 7 (6 groups plus 1 to search the overflow_buffer [1]).

[1] Refer to Chapter 4 for more information about overflow_buffer.



(a)



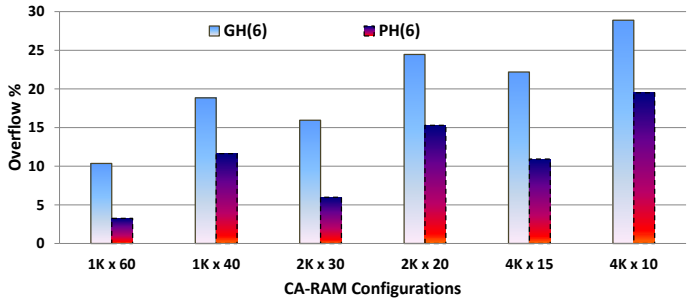
(b)

Figure 29: The (a) Average overflow of GH(6) vs. PH(6) + I-Mark, and (b) AMAT & WMAT of PH(6) + I-Mark for the PC databases given in Table 3 for $C_1 : \{60 \times 1K\}$.

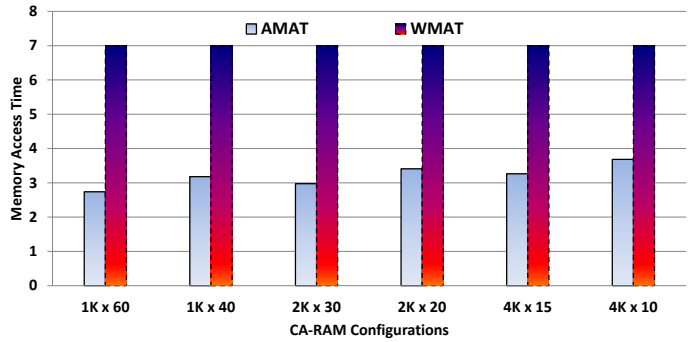
Instead of showing all the 11 files for different configurations, we show only in the next figure, Figure 30, the average of both overflow and AMAT for six of the nine configurations. These configurations are: $C_1 : \{60 \times 1K\}$, $C_2 : \{40 \times 1K\}$, $C_4 : \{30 \times 2K\}$, $C_5 : \{20 \times 2K\}$, $C_7 : \{15 \times 4K\}$, and $C_8 : \{10 \times 4K\}$. From Figure 30(a) we can see that PH with I-Mark optimization outperforms GH in terms of overflow reduction and AMAT. The PH(6) with I-Mark achieves the following overflow reduction percentages over the GH(6) for each configuration: 69%, 38%, 62%, 37%, 51% and 32%. Similarly, the AMAT reduction percentages for the same 6 configurations over the GH(6) are: 61%, 55%, 58%, 51%, 53% and 47%.

Note though, that PH achieved a good reduction overflow; it did not reduce the overflow to zero in any case. These results are unlike the results we got in [20] where 5 out of the 11 benchmarks we used had a zero overflow under PH(6). This is why we proposed our other two solutions, the dynamic TSS and the two CA-RAM architecture. The main reason why the other two solutions will have better results is that they have better space utilization as

the two CA-RAM solution eliminates the redundancy of the rules, while the dynamic TSS solution uses a better partitioning plan for each database file.



(a)



(b)

Figure 30: The (a) Average overflow for GH(6) vs. PH(6) + I-Mark, and (b) Average AMAT of PH(6) + I-Mark for the average PC databases in Table 3 for six hardware configurations.

6.5.2 Experimental Results for Dynamic TSS Solution

Recall that the partitions variables are: C = number of groups and H = total number of hash functions. In our experiments, we varied C between five and ten while assigning one to three hash functions per group. Any group boundaries have to be always more than eight and less than or equal to 32, which are the minimum and maximum IPv4 prefix lengths. Overall, our algorithm runs in a polynomial time with $O(C \times n^2 \times I)$, where $1 \leq n \leq 33$, $5 \leq C \leq 10$ and $1 \leq I \leq C$ and I is the number of iterations we enforce for choosing the boundary for each group.

The average partition $PH(8)$ is chosen to be the one that produces the least average total overflow over all files. It has eight groups, namely: $(S24, S24)$, $(S24, NS)$, $(NS, S24)$,

$(S16, S16)$, $(S16, NS)$, $(NS, S16)$, $(S8, NS)$, and $(NS, S8)$. We use this partition plan to show that a single partition plan cannot outperform the custom partitions for each file.

Each group has its own hash function, $h_0()$ for group $(S24, S24)$, $\cdots$ $h_7()$ for group $(NS, S8)$. As we mentioned in the beginning of this dynamic TSS solution, we use the progressive hashing idea. So, hash function $h_7()$ of group $(NS, S8)$ can be used by other groups such as $(S24, S24)$, $(NS, S24)$, $(S16, S16)$, and $(NS, S16)$. The same goes for all other hash functions that can be used by other groups.

In Figure 31, we show the overflow of three different partitions: 1- the original progressive partitions, or $PH(6)$, 2- an average partition plan over all the files, or $PH(8)$ and 3- custom partitions for each of the 11 files given in Table 3, or $PH(C)$. One can see that any static partitions over all files (like PH(6) and PH(8)) produce higher overflow than the custom partitions (i.e., $PH(C)$).

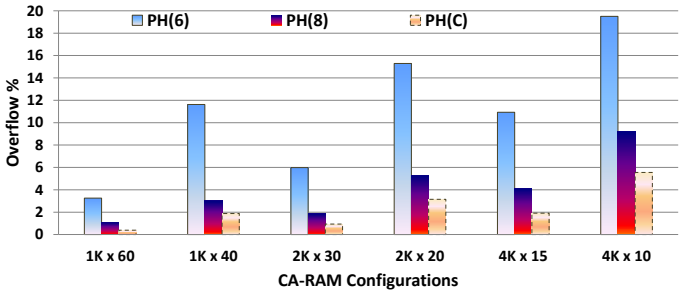


Figure 31: The Average Overflow of PH(6), PH(8) and PH(C) for the PC databases given in Table 3 for six hardware configurations.

After investigating the overflow, in Figure 32 we show the average memory access time of the same three schemes: PH(6), PH(8) and PH(C) for the same six hardware configurations. The PH(6) scheme has the lowest AMAT among the three schemes but at the price of having the most overflow among them too. Our PH(C) has on average 16% more AMAT than PH(6), and at the same time PH(C) reduced the overflow by 82% on average as well. On the other hand, PH(C) reduced the overflow by 84% on average over PH(8) while increasing the AMAT by only 6%.

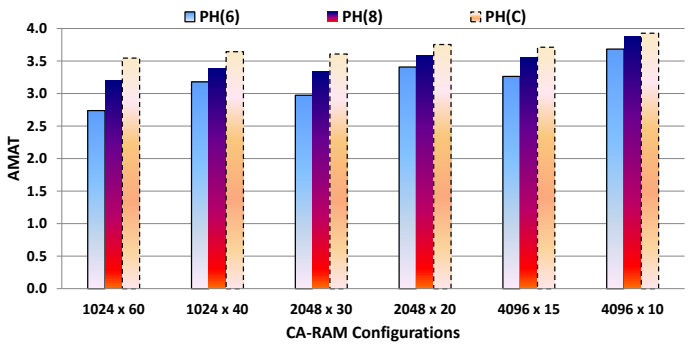


Figure 32: The Average AMAT of PH(6), PH(8) and PH(C) for the PC databases given in Table 3 for six hardware configurations.

In addition to the AMAT, we investigate the worst-case access memory time (WMAT) in Figure 33, where we show the worst-case memory access time of the same three schemes. PH(6) has a fixed WMAT of seven, the average partition ($PH(8)$) also has a fixed WMAT of nine, and finally, each one of the 11 files of Table 3 has a different WMAT range from seven to ten with an average of 8.09. Note that we increase the WMAT by one for accessing the overflow_buffer. Though the WMAT is larger in PH(C) by 16%, the overflow is reduced (on average) by 82% over PH(6).

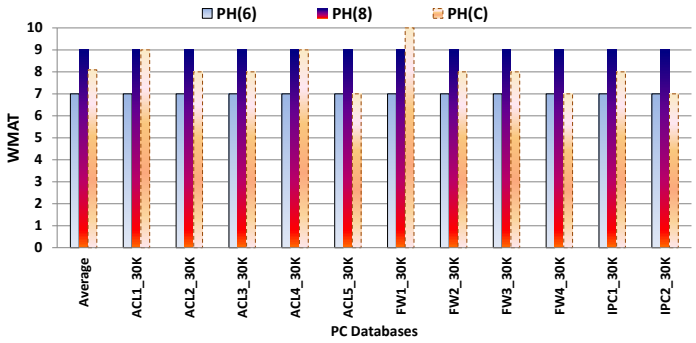


Figure 33: The average WMAT of regular PH(6), average PH(8) and custom cuts PH(C) for the PC databases given in Table 3.

In case we adopt a fully pipelined architecture, like many state-of-the-art NP [6, 40], we then should not pay much attention to the AMAT, but rather the WMAT. I witnessed two ASIC-based algorithmic packet-processing solutions during my industrial internship. These two solutions use deep pipelining (between 20 and 30 stages) to split the lookup process into

a smaller number of pipelined actions (including reading the input packet header and output rule ID). So, our solutions should be acceptable, since we have at most 10 stages.

6.5.3 Results for The Two CA-RAM Memory Architecture Solution

In this section, we present the two CA-RAM architecture results discussed in Section 6.4. We compare our 2 CA-RAM architecture with the single CA-RAM architecture, which is similar to the one used in Section 6.3. In Figure 34, we show the basic single CA-RAM architecture v.s. the equivalent two CA-RAM architecture.

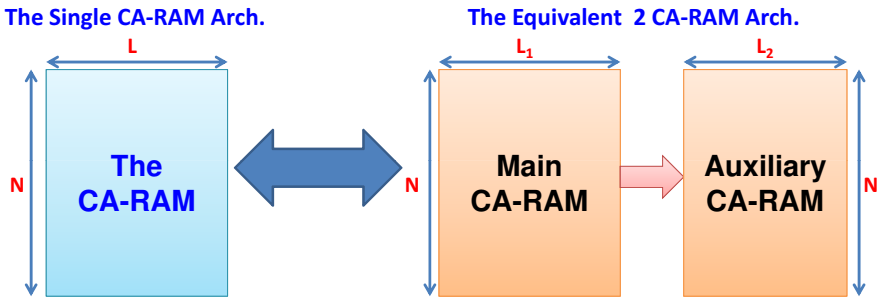


Figure 34: The regular single CA-RAM architecture vs. the equivalent two CA-RAM architecture, where $L = L_1 + L_2$.

The main reason for the comparison is to find a good practical width for the CA-RAM in addition to making use of the fact that some rules share the same prefix pair. The single CA-RAM has an aspect ratio of L rows times N bytes. We split the single CA-RAM vertically into two CA-RAMs each having the same number of rows, N, but with different row width, L_1 and L_2, where $L = L_1 + L_2$. Thus, both architectures are the same in terms of memory capacity.

Figure 35 shows overflow for the 11 PC files given in Table 3 for two architectures: 1- single CA-RAM architecture, where all the five fields (tuples) of the PC rule are kept in the same CA-RAM, and 2- two CA-RAM architecture, where the source and destination prefixes of each PC rule are stored in the main CA-RAM in Figure 26, and the rest of the tuples are kept in the auxiliary CA-RAM. We calculate the overflow for the two CA-RAM architecture using the same methodology for the single CA-RAM architecture, i.e., the number of keys

that were not inserted into the CA-RAM (the main CA-RAM) divided by the total number of keys.

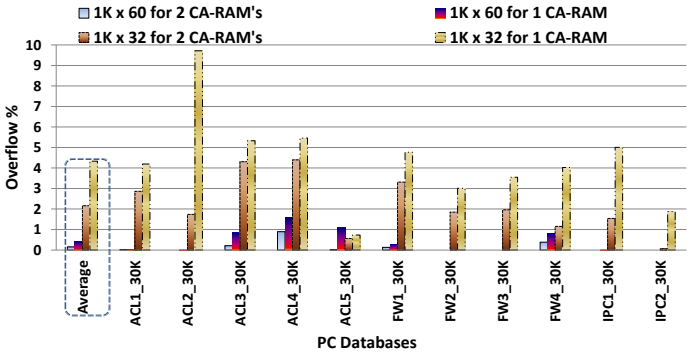


Figure 35: The overflow of the two CA-RAMs vs. the single CA-RAM architectures for the PC databases given in Table 3 for two hardware configurations.

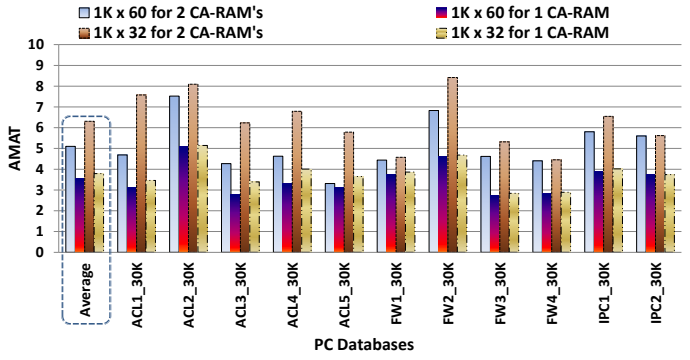


Figure 36: The AMAT of the two CA-RAMs vs. the single CA-RAM architectures for The PC databases given In Table 3 for two hardware configurations.

In this experiment, we used only two hardware configurations: $C_1 : 1K \times 60$ and $C_3 : 1K \times 32$. On average, our two CA-RAM architecture eliminates 61.4% of the overflow over the single CA-RAM architecture. In some cases we were able to totally eliminate the overflow (e.g., like $ACL1_{30K}$, $ACL2_{30K}$, $Fw2_{30K}$, and $Fw3_{30K}$). The AMAT results for the same experiments are shown in Figure 36, where the AMAT of the two CA-RAM architecture is higher than that of the single CA-RAM architecture by 31%, 37% and 40% (with an average of 36%). In the next couple of paragraphs we analyze the reason for this increase.

We repeat the same experiments but with all the nine different memory configurations

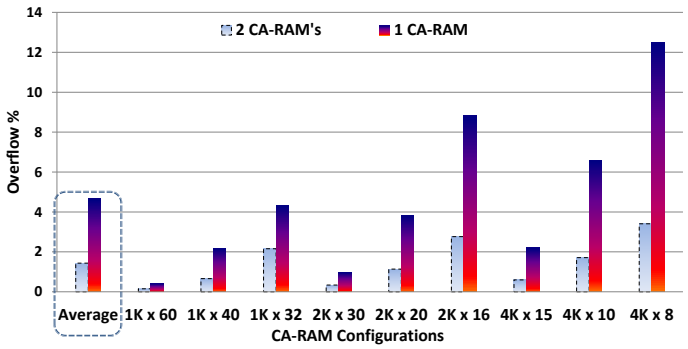


Figure 37: The average overflow of single CA-RAM vs. two CA-RAM architectures for nine hardware configurations.

that are given in Section 6.5 and the results are shown in Figure 37. The results in Figure 37 show an improvement in the overflow for all configurations. The $4K$ configurations have 73%, 74% and 73% average overflow reduction of the two CA-RAM architecture versus the single CA-RAM architecture. The same goes for the $2K$ configurations which have 66%, 70% and 69% overflow reduction. Finally, the three $1K$ configurations have the lowest overflow reduction of 65%, 70%, and 50%. We notice that as the number of rows is increasing, the overflow reduction percentage is also increasing. This stems from the fact that the single CA-RAM architecture has more overflow for a large number of rows.

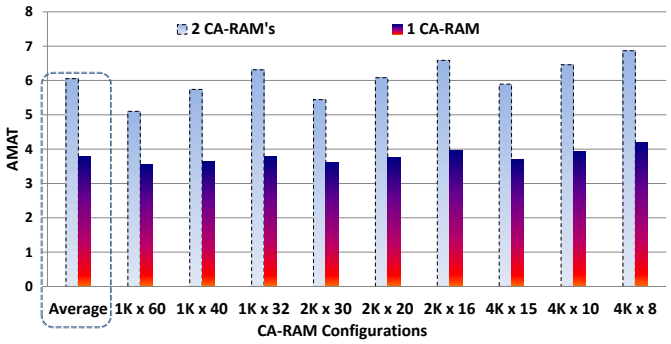


Figure 38: The average AMAT of single CA-RAM vs. two CA-RAM architectures for nine hardware configurations.

In addition to the overflow, in Figure 38 we show the AMAT of both architectures.

Though we may expect that the AMAT of the two CA-RAM is much higher (around two times higher), the data in Figure 38 show that on average the two CA-RAM architecture AMAT is only 37% higher than that of the single CA-RAM architecture. This comes from the fact that we use pipelining; hence, on average the AMAT should rise by only '1', which represents the auxiliary CA-RAM access. However, in some cases, the AMAT for the two CA-RAM architecture is slightly higher because of the need to access more than one row in the auxiliary CA-RAM, as one row might not be enough to store all rule ports and protocol tuples. In other words, the AMAT is higher because of the use of the "$NEXT$" pointers. Note that the maximum number of these $NEXT$ pointers is two per row according to our simulation results.

6.5.4 The Performance Estimation

In this section, we approximate 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. The configuration $C_2 : \{40 \times 1K\}$ requires $40K$ entries, 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 $\sim$ 850.92MB or roughly 1MB.

A state-of-the-art CMOS technology SRAM memory design [55] 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 2.0GHz. If we assume AMAT of $\sim$ 4.0, according to Figure 32, then we have a filtering speed of 0.5 Giga packets per second or 160 Gbps for the minimum packet size of 40 bytes. Since we assume that we use pipelining for the two CA-RAM solution, then it is safe to assume that the same performance can be obtained for this solution as well.

6.6 CONCLUSION

In this chapter, we presented three main CA-RAM based solutions for the packet classification problem. In the first solution, progressive hashing, we showed how the progressive hashing with the I-Mark schemes can be used to solve the PC problem for a relatively low number of rules per database, around $10K$. We then concluded that we need better and dynamic partitions (groups) for each individual file. We also concluded that we may need to increase the number of groups given larger rule set sizes, $30K$.

This leads to our second solution, which we call "dynamic tuple space search," where we vary the number of groups, the number of hash functions assigned to each group, and the group borders. Though we vary the groups and their numbers, we have to meet the system worst-case memory access time. On average we were able to reduce the overflow by 82% for the dynamic TSS solution over the regular TSS solution for nine CA-RAM hardware configurations. Our results indicated that such a scheme can reduce the overflow by almost 100% in some cases. However, we have increased the average memory access time by 6%, mitigated by the fact that we use pipelining.

Our final scheme is based on the idea that some packet classification databases have the same source and destination prefix pair appearing in multiple rules. Since we use this pair in hashing, this leads to more collisions, which is the main reason for overflow. This is why we devised a two CA-RAM architecture where the first (main) CA-RAM is to store the source and destination prefixes and the auxiliary (second) CA-RAM is to store the rest of the PC rule. Though the AMAT is larger in this case, for the same amount of memory, we were able to reduce the overflow by 62% on average over the single CA-RAM architecture. In this case, the AMAT is raised only by 36% on average. Finally, we predicted that our solutions can achieve throughput up to 0.5 Giga packets per second or 160 Gbps for the minimum packet size of 40 bytes.

7.0 THE DEEP PACKET INSPECTION APPLICATION

As we discussed in Chapter 1, the deep packet inspection (**DPI**) function allows the firewall to examine the packet content for either network attacks or malware signatures or strings [33] [1]. In the deep packet inspection problem, packets content are matched against a database of signatures or "rules." These rules consist of fixed-length strings and regular expressions, where these two are called "the payload part," in addition to the 5 tuples of the IP headers 6, or "the headers part." Overall, one can consider the DPI rules mainly as regular expressions.

In this chapter, I describe my PM (pattern-matching) solution using the CA-RAM memory architecture. The packet pattern-matching problem is defined as follows: given a set of k patterns $\{P_1, P_2, \cdots, P_k\}$, $k \geq 1$, and a packet of length n, the goal is to find all the matching patterns in the packet. Note that we extract these strings from the DPI rules and each string has its own length (i.e., they vary in length). If we match one or more of these substrings, we say that we have a "partial" matching. Usually, the PM unit returns the longest matched substring to the DPI software of the firewall which takes an action (either drop or forward the packet).

Figure 39 shows that the DPI engine consists of three smaller engines: 1- pattern-matching (PM) engine, which stores and matches the static strings, 2- packet classification (PC) engine, which processes the incoming IP packet headers, and 3- regular expression (RE) engine, which is responsible for matching variable size regular expressions parts of the rule. To accelerate the pattern-matching in a typical DPI system, a TCAM chip is used to store all the patterns (mostly ASCII characters), which works as a pattern-matching (PM) engine. I propose using the CA-RAM for the main component of a DPI engine, which is the

[1] Throughout this chapter we will use the terms *signature* and *string* interchangeably to present the same thing.

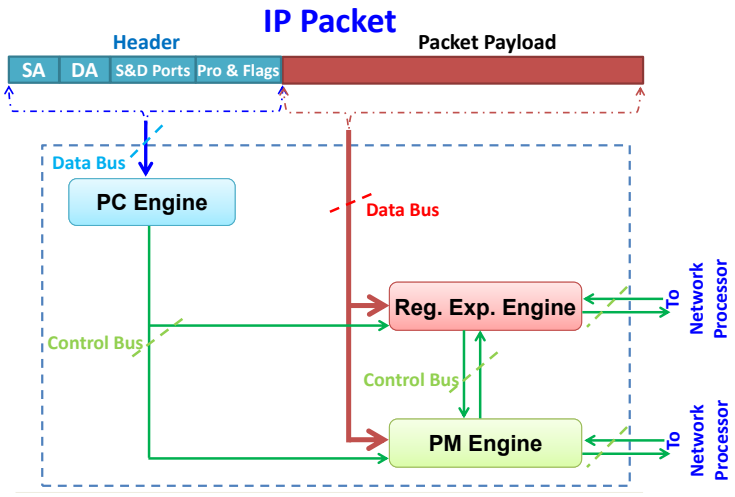


Figure 39: The overview of a DPI engine architecture.

pattern-matching (PM) unit [59]. In other words, my solution is to replace the TCAM-based PM engine with a more efficient CA-RAM.

In this chapter, we use the SNORT DPI rule database, described in Section 7.3, to evaluate our pattern-matching CA-RAM-based solution. SNORT is an open source network intrusion prevention and detection system (IDS/IPS) developed by Sourcefire. Combining the benefits of signature, protocol, and anomaly-based inspection, Snort is the most widely deployed IDS/IPS technology worldwide [23]. The SNORT rules are regular expressions, each having the following form:

"**alert — rule header information — static string — regular expression (encoded in PCRE scripting language) — static string**"

, where '—' is the string concatenation operator.

Figure 40 shows an example of a real SNORT rule. Any SNORT rule starts with the keyword "**alert**". Among the header fields that the PC engine checks: destination & sources IP addresses, protocol, destination & sources port numbers, protocol flags, $\cdots$, etc. These fields are identified by reserved keywords such as "**tcp**" (i.e., TCP protocol), "**any**" for any port number, while the destination and source addresses are defined by the variables, $\$EXTERNAL\_NET$ and $\$HOME\_NET$ in this example.

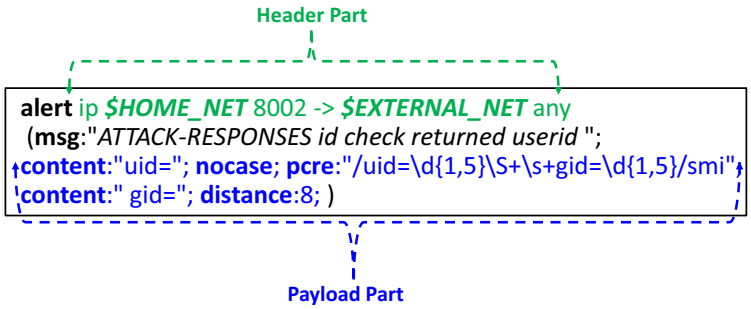


Figure 40: An example of a SNORT [23] rule.

The reserved keyword "**msg**" indicates the warning message that is displayed once the entire rule is matched. The payload part of the rule is defined by the two keywords: "**content**" for a static string, and "**pcre**" for regular expression. By "static string" we mean a consecutive group of ASCII characters that have certain order and maybe ceratin case (upper, lower or mixed). Note that a regular expression may contain more static strings that we extract and store in our simulations. The PCRE stands for *Perl Compatible Regular Expression* and it is a scripting language that describes regular expressions. There are some keyword modifiers that describe the static strings such as "**nocase**," which means that the string characters can be in either upper case or lower case, and "**distance**," which tells the NIDS system the index of the first character of the string.

To summarize, the rule in Figure 40 indicates a PC filter on the source IP address 8002 and for all destination IP addresses. In the payload part of the rule, there are two static strings (patterns) in this rule: "uid" and "gid," in addition to the regular expression that is given after the *pcre* keyword.

The main feature of our CA-RAM so far is that it is storing fixed-length keys (e.g., IP prefixes and PC rules). However, the PM problem of the DPI application has variable-sized keys (strings). Figure 41 shows the histogram of the compiled SNORT rules lengths. Note that we only included the static strings of the SNORT rules in addition to these static strings that exist inside some regular expressions. From this figure we can see that almost 90% of the rules have a length of 36 characters or less. In addition, we note that percentage-wise, there is a variety of rule lengths from one character up to 64 characters.

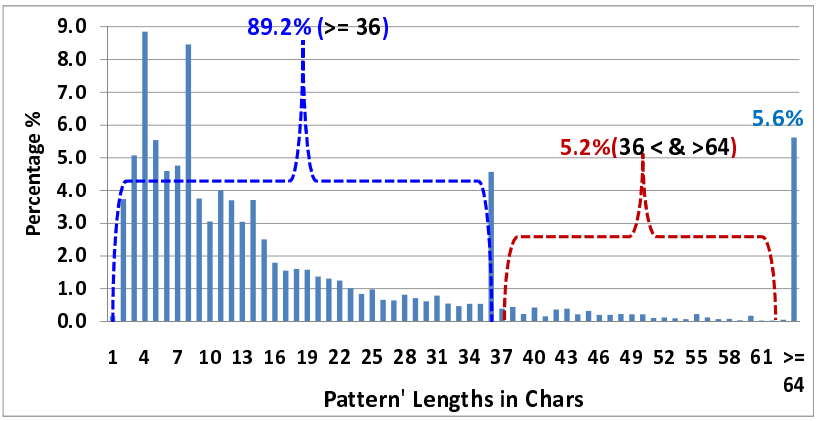


Figure 41: The SNORT statistics: percentage of patterns vs. the pattern lengths.

Note that the DPI engine will perform pattern-matching and RE matching if and only if the header part has been positively matched first by the PC engine. The regular expression function is either realized in FPGA hardware or is performed by the network processor [33]. In this thesis we only focus on the PM engine.

Our main motivation in this chapter is to replace the usually used TCAM chip with our CA-RAM solution, which both consumes less power and has higher storage density than the TCAM. In addition, our solution is capable of storing variable-sized strings unlike the TCAM which stores only strings of the same size.

Our DPI CA-RAM solution has two main components: 1- an architectural part, and 2- an algorithmic part. The architecture part is described in Section 7.1, where I talk about the implementation of the variable length string matching capability inside the CA-RAM architecture. In Section 7.2, I describe the algorithmic part of the solution, which relies on two of my three hash-based schemes, namely the progressive hashing and the CHAP. In Section 7.3, I describe the simulations performed to show the superiority of my proposed solution over the TCAM. Finally, in Section 7.4 I conclude with observations and results.

7.1 VARIABLE-LENGTH STRING MATCHING SUPPORT FOR CA-RAM ARCHITECTURE

Since the PM problem requires matching different pattern lengths, we need to adjust the CA-RAM architecture to support this feature. One possible solution is to fix the rule length to a certain length. However, this requires the padding of all short patterns by don't care bits, hence the need to use ternary logic. Another drawback is that the CA-RAM space waste will increase.

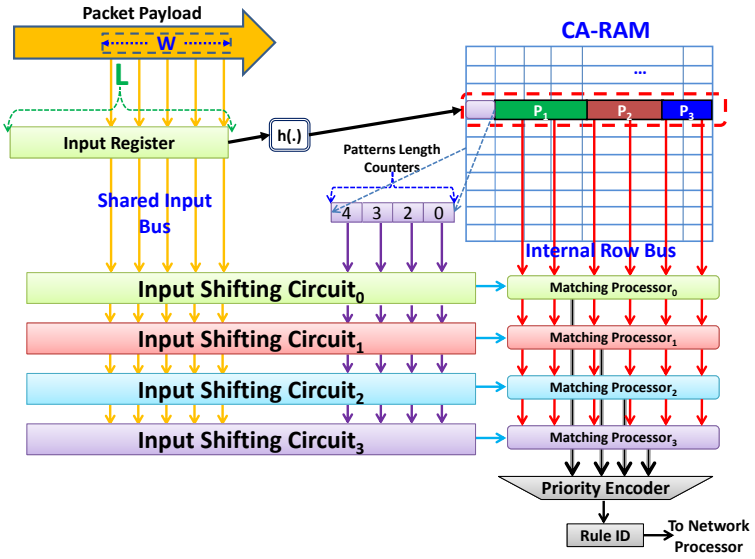


Figure 42: The modified CA-RAM variable-sized length patterns matching architecture.

The importance of supporting variable-length string matching stems from the fact that patterns that are fully detected need less processing from the Network Processor (NP), which is translated to higher throughput. To support storing variable length strings, we modify the CA-RAM basic architecture, Figure 3, to the one depicted in Figure 42. We also use pipelining to achieve an ultra-high throughput.

As depicted in Figure 42, we allow each rule to have its own length that ranges between a minimum length, L_{min}, and a maximum length, L_{max}, where we assume a CA-RAM configuration of N rows (buckets) each having L bytes. Thus, the maximum number of patterns per bucket equals $n_{max} = \lceil \frac{L}{L_{min}} \rceil$. For each row, there are n_{max} counters, each containing the length of each rule, as shown in the upper part of Figure 42. The CA-RAM

in Figure 42 stores the strings of the pattern-matching engine, but the supporting matching circuit is different from that of the basic circuit shown in Figure 3. Note that there could be more than one hash function, but for simplicity we show only one hash function in this figure, where we hash the first L_{min} characters of each of the patterns to generate the index.

My main idea here is that the input string goes through a shifting circuit that creates multiple copies, n_{max}, of the string, and at the same time aligns these copies to be compared with each of the stored strings (patterns). Each shifting circuit, which is shown in Figure 43, has $n_{max} = 4$ input string buffers (input registers) that store the input payload string. These input registers have a unified size of L characters and their outputs are fed to the n_{max} shifting circuits. In the first clock cycle, the PM engine fetches an input window (string) of W characters, where $L > W$, from the currently inspected packet payload and feeds it to the array of input string registers. Then, in the next clock cycle, a new window is created by simply by adding a new character from the payload to the right-hand side of the old window. At the same time, we shift the rest of the characters one position to the left, thus dropping the leftmost character of the previous window [49]. Note that we assume in Figure 43 that the input string is placed at the rightmost position of the input registers and that the remaining, L - W bytes, of each input string register contains zeros. At the same time, the same input string is fed to the hashing function, $h(\cdot)$, to calculate the CA-RAMs row index that stores the patterns that will be matched against this input window.

In the next clock cycle, each input shifting circuit calculates the offset used for the left shift according to the pattern string counters, as shown in Figure 43. Note that the pattern string counters are stored at the end of each row along with the probing pointers, as we will see next. Figure 43 shows the second input shifting circuit of Figure 42. We propose to use Barrel shifter [17] circuits to shift the input multiple position in a single clock cycle [17]. The output of the shifting circuit is stored in the L bytes shifted input register, which is used as an input to its associated string matching processor circuit.

After shifting the input string, the shifted string copies are matched against the patterns stored in the CA-RAM row using string matching processors (MPs). Matching results are then reported to the NP (network processor), as a pattern ID [2]. In case we have multiple

[2]We assume in this thesis that each pattern is associated with a unique ID that can be identified by the

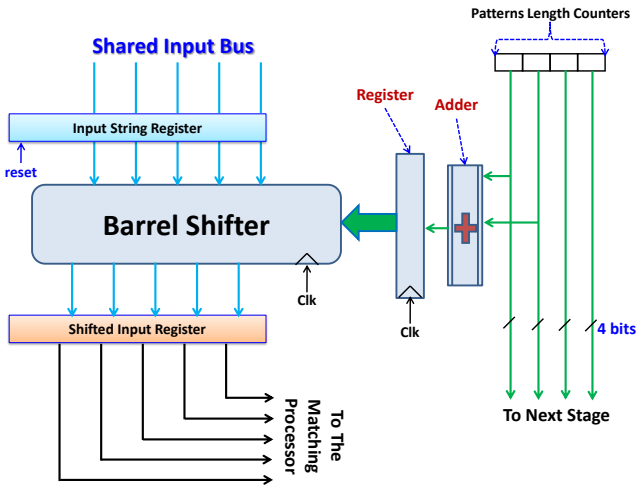


Figure 43: The input shifting circuit$_2$ of Figure 42.

matches, we use a priority encoder to select the best matching (which is the longest matching as well). Sometimes, the NP software requires that we report all the matches for further processing. In this case, instead of reporting the best matching pattern ID, our CA-RAM reports all the matched patterns IDs by encoding them in a single vector (to be sent to the NP). In fact, reporting multiple matches comes for free in CA-RAM, as they co-exist on the same bucket(s).

In case we store the longest pattern first, the priority encoder circuit is going to be very simple. The idea of this simple circuit is that once a matching processor finds a match, it blocks the results of the next matching processors. This is done by compiling the result of each matching processor into an enabling signal to the next matching processors, starting from left to right. In Section 7.1.1 we describe our FPGA synthesis model and we show in detail how these sub-circuits are implemented.

7.1.1 The FPGA Synthesis Results

The research team also did an experiment with Xilinx FPGA kit [56] to demonstrate a proof of concept. We want to emphasis that such a variable-sized string matching circuit can be

NPs control and data planes software.

implemented in hardware. Our design is implemented using the Xilinx ISE Design Suite. A combination of VHDL and schematic design is used to instantiate the pattern-matching CA-RAM circuit that is described in Section 7.1.

The characteristics of the design are defined, as previously discussed, by CA-RAM bucket width, L, minimum pattern length, L_{min} and maximum pattern length, L_{max}. These determine the maximum number of patterns per bucket , $n_{max} = \lceil \frac{L}{L_{min}} \rceil$. Pattern length counters are used to dynamically generate and use a set of up to n_{max} masks on L bytes of CA-RAM row data, which is then compared with the input string. A priority encoder can then determine the highest quality match for further processing.

The gate counts required for each component of the matching circuit scale linearly with n_{max}. Gate width is related linearly to L, but either intelligent CA-RAM packing or an addition index stored (akin to a pattern length counter) could allow gate widths to be greatly reduced and separate portions of a large CA-RAM bucket width to be processed separately in parallel (4 sets of $n_{max}/4$ gates of $L/4$ width instead of one set of n_{max} gates of L width). Much more detailed gate analysis has been performed, relating abstract gate counts to these parameters. The matching logic required is independent of the overall CA-RAM size, and only depends on the width of an individual bucket. CA-RAM size can be adjusted by increasing row count without the need for additional matching logic.

The parameters for the simulations are as follows: $L = 16$ bytes, $L_{min} = 4$ bytes, $L_{max} = 8$ bytes and $n_{max} = 4$. This design was placed and routed for a Spartan6 $XC6SLX75T$ FPGA. The maximum clock frequency at the speed grade used in our simulation is 270 MHz [57]. Pipeline implementations allows for increased throughput as well, as next generation FPGA. At this frequency, our design has an estimated power draw of 94.59 mW, or 0.0352 nJ. Given the worst-case linear scaling of match logic size with n_{max}, it can be seen that the overall power draw will remain reasonable under more realistic packet filtering conditions ($L = 256$ bytes, $L_{min} = 8$ bytes, $L_{max} = 36$ bytes, $n_{max} = 32$). Assuming linear scaling, power draw under these conditions would be 756.7 mW.

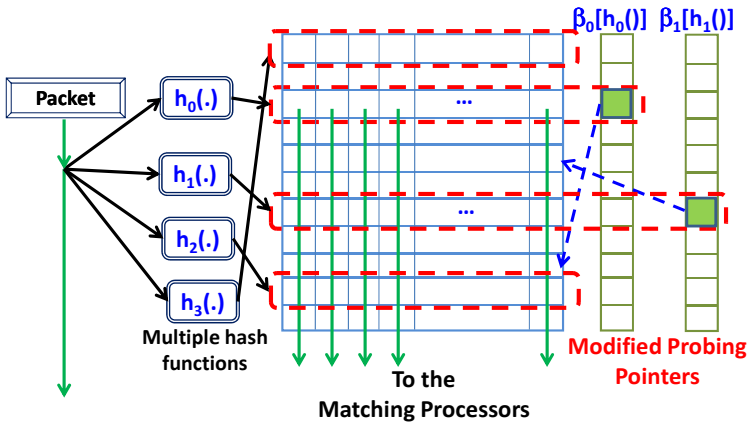


Figure 44: The modified CHAP scheme example, where $H = 4$ and $P = 2$.

7.2 THE HYBRID DYNAMIC PROGRESSIVE HASHING AND MODIFIED CHAP SOLUTION FOR THE PM PROBLEM

In this section, we describe the algorithmic part of our CA-RAM DPI solution. We use the hash tools that we introduced in Chapter 4. Specifically, we use the progressive hashing and a modified version of our CHAP scheme to reduce the overflow and the I-Mark scheme to reduce the average memory access time (AMAT) of the CA-RAM.

First we modify our CHAP(H, H) algorithm, where H represents both the number of hash functions and the number of probing pointers per row (refer to Chapter 4.1). Instead of using one probing pointer for each hash function per row for all hash functions, we choose P hash functions to have probing pointers, where $H \geq P$, while the other hash functions will not have any. We call this algorithm *CHAP(H, P)*, and Figure 44 is an example in which $H = 4$ and $P = 2$. This allows us to reduce both the AMAT and the WMAT. The WMAT in this case is equal to $WAMT_{DPI} = H + P + 1$, where the extra one is to count in the overflow_buffer, as we discussed earlier for the PF and PC applications.

Moreover, these 'P' hash functions will be the last P hash functions. The reason is that the pattern groups that use these hash functions are smallest in length among the other groups. Hence, these groups tend to have more overflow than the other groups; also, these functions are used by all the groups as we are using progressive hashing (refer to Chapter 4.2).

Next, we propose an iterative dynamic algorithm that selects a near-optimal "partition plan" in terms of overflow, and memory access time. By a **partition**, we mean the number of hashing groups and their associated physical boundaries. The groups' boundaries are simply the minimum length of string for each group. The best, near-optimal partition plan is the one that has the lowest overflow for a given hashing space. If there are multiple of these plans, we select the plan with the lowest AMAT and WMAT.

To find the near-optimal partition, we impose a goal, or an upper limit, OL_{max}, on the overflow. This limit depends on a system parameter, which is the capacity of the embedded overflow TCAM buffer. In the DPI application, the overflow_buffer stores the patterns that are shorter than L_{min}, where L_{min} is the minimum pattern length that we store in the CA-RAM, in addition to the actual overflow (collided) strings.

We propose an algorithm that is complementary to our original PH algorithm by choosing the groups dynamically based on the database distribution, similar to what we did for the PC solution in Section 6.3. The idea is first to study the PM database distribution to select the hashing parameters, which are the number of groups and their boundaries. Figure 41 shows the string length distribution for the SNORT database. The next step is to decide what constitutes the maximum number of groups, H, based on the memory access time budget that is entitled to the PM unit. Since most of the modern NPs are deeply pipelined and have between 30 to 40 stages [11], we can assume in this chapter that $8 \leq H \leq 15$ cycles.

We start by defining $OL[i]$ as a linear array that contains the overflow percentage of iteration 'i', where $1 \leq i \leq I_{max}$ and I_{max} is the maximum number of iterations of our algorithm. Next, we use a simple heuristic algorithm that dictates how to find a good partition, group, by minimizing the overflow, $OL[i]$, given H, $H \leq WMAT$, which is the number of groups. We reserve $OL[0]$ for the percentage of the strings that have length less than minimum string length, L_{min}. The algorithm, starts by sorting the patterns from longest to shortest length, then partitioning them into H_1 groups based on their length, where each group has one hash function. As we already know, this sorting is useful in finding the longest pattern during the insertion of the actual strings into the hash table.

The algorithm then simulates the mapping of the patterns into the hash table using a

progressive hashing scheme. For each iteration, 'i', we calculate the overflow $OL[i]$ of the iteration, by adding $OL[0]$ to the percentage of the collided patterns. If $OL[i] \leq OL_{max}$, we stop and report both the number of groups, H_i, and their lengthes. If not, then we iterate one more time by changing the hashing parameters. This is done by splitting the groups that generated the most collisions each to two groups. We keep iterating until I_{max}. If $OL[I_{max}] > OL_{max}$, then we use the modified CHAP(H_i, P) algorithm for the last hash P functions where H_i is the number of groups for the best partition plan. This modified CHAP(H, P) setup algorithm is the same CHAP(H, H) set algorithm, Algorithm 1, but with P probing pointers instead of H.

The exact mapping of the signatures into the CA-RAM hash table is as follows: We first use our I-mark technique, Chapter 4.3, to mark these strings that are not substrings of other strings. After that, we have two clusters of strings, those that are I-marked and those that are not. We insert the I-marked cluster from shortest to longest strings since the shortest have a lower chance of being mapped to the hash table. The non-independent strings are then inserted from longest to shortest (to enable finding the longest matching string first).

The search algorithm for this dynamic PH + modified CHAP(H, P) scheme is identical to that of the progressive hashing algorithm, Algorithm 5 while, its incremental update algorithm is identical to the CHAP(H, H) Algorithm 3, but with P probing pointers rather than H.

7.3 THE SIMULATION RESULTS AND THE EVALUATION METHODOLOGY FOR OUR DPI SOLUTION

In this section, we describe our software simulation results, in Section 7.3.1 and Section 7.3.2 to evaluate the effectiveness of our solution. Finally, Section 7.3.3 compares our CA-RAM solution against the equivalent TCAM solutions using two CACTI [1, 52] simulators.

For evaluation purposes, we used C++ to build our own simulation environment that 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

$\mathbf{H_3}$ [38] hashing families that are simple and easily realized in hardware. We used the SNORT database for evaluation [23].

In all our experiments we set $L_{min} = 2$, where all the patterns that are one character long are stored in the overflow_buffer. We assume that we use a small TCAM chip as the overflow_buffer. Also, this overflow_buffer can work as a temporary buffer for the incremental updates (if any).

Configuration	L_i	N_i	Size (KB)
C_1	512	256	128
C_2	1024	256	256
C_3	2048	256	512
C_4	512	512	256
C_5	1024	512	512
C_6	2048	512	1024

Table 5: The six variable string CA-RAM hardware configurations for DPI application.

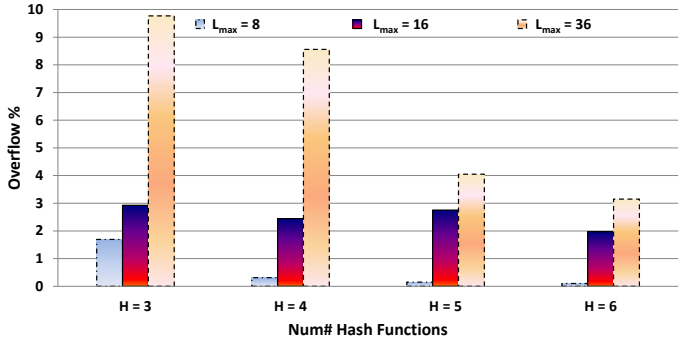
In our simulations we use six hardware configurations that are defined in Table 5. Namely, $C_i = L_i \times N_i$, where L_i = row width in bytes and N_i = number of rows. The sizes of these six configurations range from 128 KB all the way up to 1 MB, as shown in the last column of Table 5. Since our goal is to replace the TCAM chip that is used as the pattern-matching engine, we need our CA-RAM configurations to have sizes compatible with the TCAM chip sizes. In Table 6, we show the TCAM chip sizes, in binary KB RAM.

L_{max}	8	16	36
TAM_{size} (KB)	128	256	1024

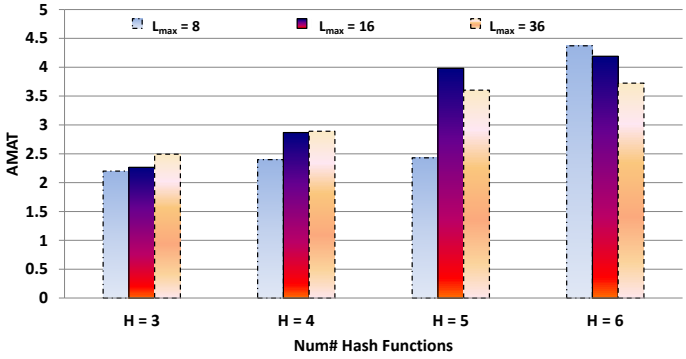
Table 6: The TCAM equivalent sizes for different widths using Equation 7.1.

7.3.1 Sensitivity Analysis Results

We start our simulations by running some sensitivity analysis to find the best partitions in terms of minimizing both the overflow and the AMAT. Searching for these partitions includes finding the number of hash functions, H, as well as their boundaries. We use a very simple heuristic, as we described in Section 7.2, that calculates the partitions by running the progressive hashing setup algorithm, Algorithm 4, with different partitions, till we find the partition with the smallest overflow percentage for a given number of hash functions. Figure 45-(a) shows the overflow of C_1 for four numbers of hash functions $H = 3, 4, 5, 6$, and for three different $L_{max} = 8, 16, 36$. Note that we not only needed to change the number of hash functions, but also to see the effect of changing L_{max}.



(a)



(b)

Figure 45: The overflow and the AMAT (using I-Mark) of SNORT for C_1 : $\{512 \times 256\}$, $H = 3, 4, 5, 6$ and $L_{max} = 8, 16, 36$.

We can see that the overflow is reduced by 81%, 91% and 100% as we linearly increase the number of hash functions from 3 to 6 for $L_{max} = 8$. At the same time, the AMAT is

increased by 12% and 25% for increasing H from 3 to 4 and 5 respectively. For $L_{max} = 36$, the decrease in overflow is 12%, 59% and 68% for increasing H from 3 to 4, 5 and 6 respectively, while the increase in AMAT is 16%, 45 and 49%. It is obvious that regardless of increasing the number of hash functions for $L_{max} = 36$, the overflow reduction is minimal at best, as it cannot be eliminated at all. This is due to the fact that longer strings require larger (wider) bucket sizes. Given all these facts, we use $H = 5$ and 6 in the following experiments.

Groups	G_0	G_1	G_2	G_3
Number of Chars	16	8	4	2

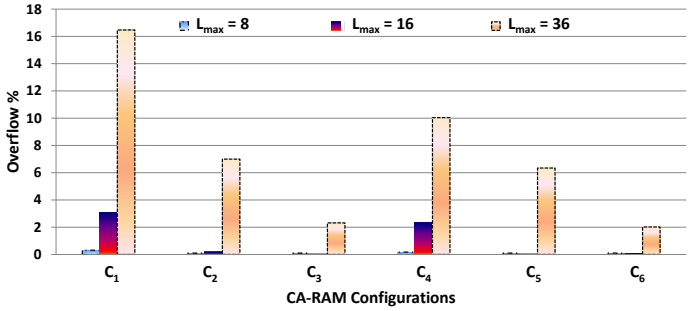
Table 7: The six optimal groups that we use in our DPI simulations.

Before we go ahead and show results for these two numbers of hash functions, we shed some light on how we choose the partitions. We varied the number of groups and their boundaries (the smallest string in the group) till we found the partition that gives the minimal (optimal) overflow for the given number of hash functions. In Table 7 we show the resultant optimal four groups that we obtained, where each cell equals the number of characters that each group has. At the same time, these characters are the characters that each hash function is using in hashing. For example, group G_0 has 16-character-long strings and its hash function, $H_0()$, uses these 16 characters to calculate the hash index. Note that for any string that is longer than L_{max}, we store only its first "L_{max}" characters. Once this string is reported as a (partial) match to the DPI software, it will try to find the full match in this case. This is due to the fact that the DPI software stores the entire full length pattern.

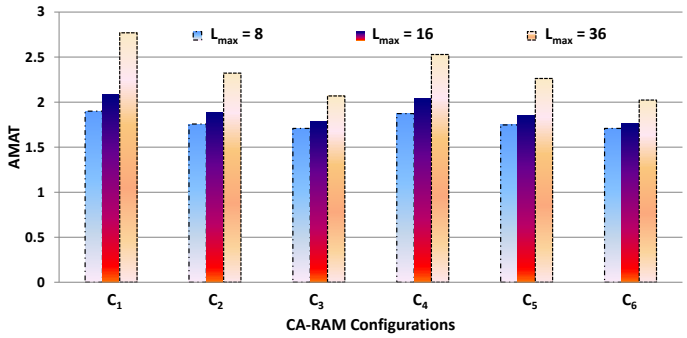
Note that for the configurations where $L_{max} = 8$, we can only use up to eight characters. So in this case we simply do not use G_0. We use the dynamic progressive hashing that we introduced in Section 7.2, where we can assign more than one hash function to each group. For both groups, G_2 and G_3, we assign two hash functions each. Thus, we can have up to six hash functions in total.

7.3.2 The Dynamic PH and Modified CHAP Results

In Figures 46-(a) and 47-(a), we show the overflow when using $H = 4\&5$ both for $L_{max} = 8, 16, 36$ characters. For $H = 5$ and $L_{max} = 36$, the maximum overflow is almost 6.3% for C_4, while the minimum overflow is around 0% for 4 configurations, with an overall average of 1.45%. On the other hand, for $H = 4$ and $L_{max} = 36$, the maximum overflow is almost 12% for C_1, while the minimum overflow is also 0% for four configurations. The overall average overflow is 2.6% in this case.



(a)

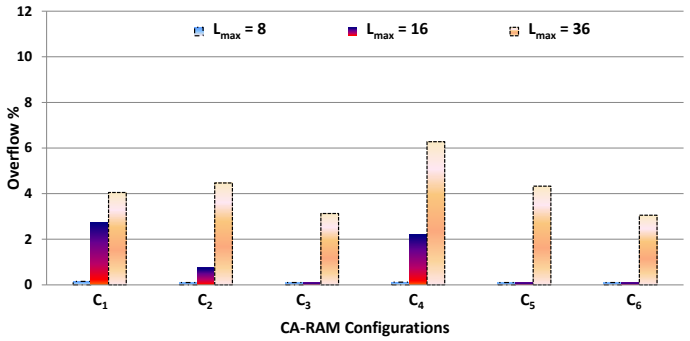


(b)

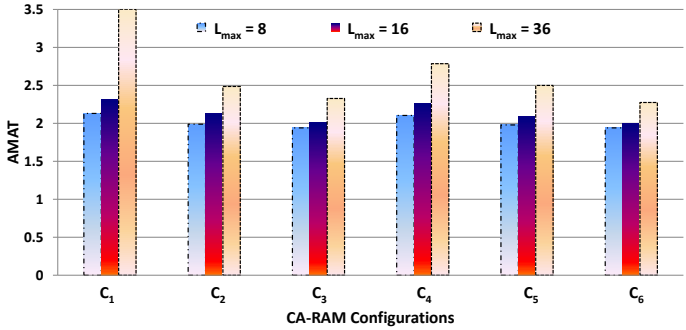
Figure 46: The dynamic PH with I-Mark: (a) Overflow, and (b) AMAT of SNORT for $H = 4$ and $L_{max} = 8, 16, 36$.

To calculate the AMAT we used the I-mark scheme where we found that 76.6% of the SNORT signatures are independent (i.e., I-marked). The average memory access time for $H = 4$, which is shown in Figure 47-(b), ranges between 1.7 (C_3 and $L_{max} = 8$) and 2.7 (C_1 and $L_{max} = 36$). The AMAT overall average of the six CA-RAM configurations is 2.0 for $H = 4$, while the WMAT is 5. As for $H = 5$, the AMAT, which is shown in Figure 47-(b), ranges between 1.9 (C_3 and $L_{max} = 8$) and 3.0 (C_1 and $L_{max} = 36$). We note that the overall

AMAT average of the six CA-RAM configurations is 2.2 for $H = 5$, while the WMAT is 6.



(a)

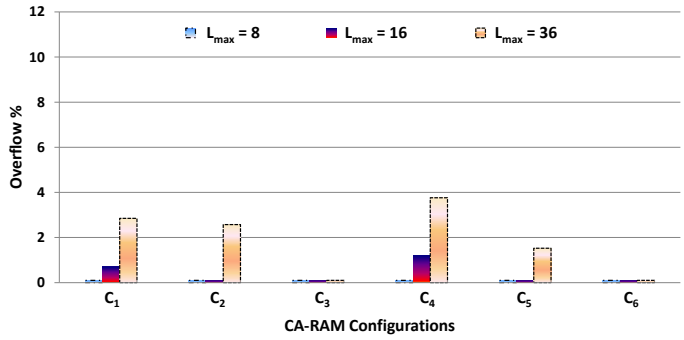


(b)

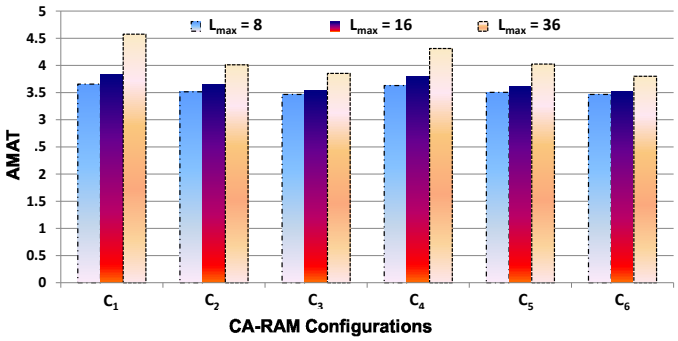
Figure 47: The Dynamic PH + I-Mark: (a) Overflow, and (b) AMAT (I-Mark) of SNORT for $H = 5$ and $L_{max} = 8, 16, 36$.

Finally, we show in Figure 48 the overflow and the AMAT for $H = 5$ of the hybrid dynamic progressive with I-mark and the modified CHAP with four probing pointers (i.e., $P = 4$). The four probing pointers we use are accessed at the last hash function, $H_4()$. Thus the total WMAT $= 5 + 4 + 1 = 10$ in this case.

For the six configurations, the overall average overflow is just 0.8% while the average AMAT is around 3.8%. We notice that in each configuration there is at least one case where overflow is zero. This is achieved at the price of having 33.8% larger AMAT than when $H = 5$.



(a)



(b)

Figure 48: The dynamic PH + modified CHAP(H, P) and I-Mark: (a) Overflow, and (b) AMAT (I-Mark) of SNORT for $H = 5$, $P = 4$ and $L_{max} = 8, 16, 36$.

7.3.3 Comparison with TCAM

Next, we compare several CA-RAM configurations against their equivalent TCAM configuration in terms of power, delay, and frequency (throughput). The TCAM chip size is calculated via a simple equation, Equation 7.1, that first calculates the size of the TCAM in number of ternary cells, then multiplies it by a scaling factor to convert it to binary RAM cell.

$$TCAM_{size} = \Uparrow ((\Uparrow Patterns\_Number \Uparrow \times \Uparrow L_{max} \Uparrow) \times \frac{TCAM\_Cell_{size}}{RAM\_Cell_{size}}) \Uparrow \qquad (7.1)$$

Note that in Equation 7.1 we use the $Uparrow$ to represent the next power of two of an integer number. We approximate to the next power of two to find: the suitable number of

TCAM rows, the equivalent binary width and the overall TCAM chip size. Note that most of the TCAM commercial chips have a width between four and up to 16 characters.

In case of SRAM the scaling factor is $\frac{16}{6}$ where 16 is the number of transistors per TCAM cell and six is the number of transistors per SRAM cell. If we use a DRAM technology, then the scaling factor is going to be $\frac{16}{1}$, where one is the number of transistors per DRAM cell. The size of the TCAM chip depends on two variables: 1- the number of patterns to be stored, and 2- the TCAM width, 'W'. It is clear that we should pick $W = L_{max}$, as shown in Equation 7.1, since L_{max} represents the maximum length of the stored patterns.

The CA-RAM storage part can be simulated using the standard *CACTI* [52] memory simulation tool. However, there is a special version of the CACTI tool that simulates TCAM memories [1]. We assume 90nm technology for both TCAM and SRAM simulations in CACTI. The SRAM template we used is the "ITRS-HP," which stands for International Technology Roadmap for Semiconductors, a high performance cell. We compare the three TCAM configurations in Table 6 with their equivalent CA-RAM configurations that are given in Table 5.

Note that the RAM CACTI just gives the power for the SRAM part of the CA-RAM. In addition to CACTI, we had to approximate the matching part of our CA-RAM simply by extrapolating from the CACTI results of both the SRAM and TCAM. First, we define the total CA-RAM power as follows:

$$P_{CA-RAM} = P_{RAM} + P_{match} + P_{pri.encoder} \tag{7.2}$$

The TCAM CACTI tool splits the total power into the following components:

$$P_{TCAM} = P_{read} + P_{write} + P_{search} + P_{pri.encoder} \tag{7.3}$$

It is obvious that we can use the same priority encoder power component from the TCAM in the CA-RAM. Based on [12], the authors calculated that the matching component in the CA-RAM is around 7% of the total RAM component. In this work, our team simulated and synthesized a CA-RAM-based IP-lookup engine prototype with a single hash function while using linear probing to resolve hash collisions for 130nm technology. However, for our more complicated DPI solution we had to compare the number of transistors, the TCAM memory cell size, the SRAM memory cell size, and the overall power in order to extrapolate the

matching part of the variable-length string matching CA-RAM solution. We estimate that for the three CA-RAM configurations, the matching circuit overhead (component) represents about 40% of the RAM component, on average.

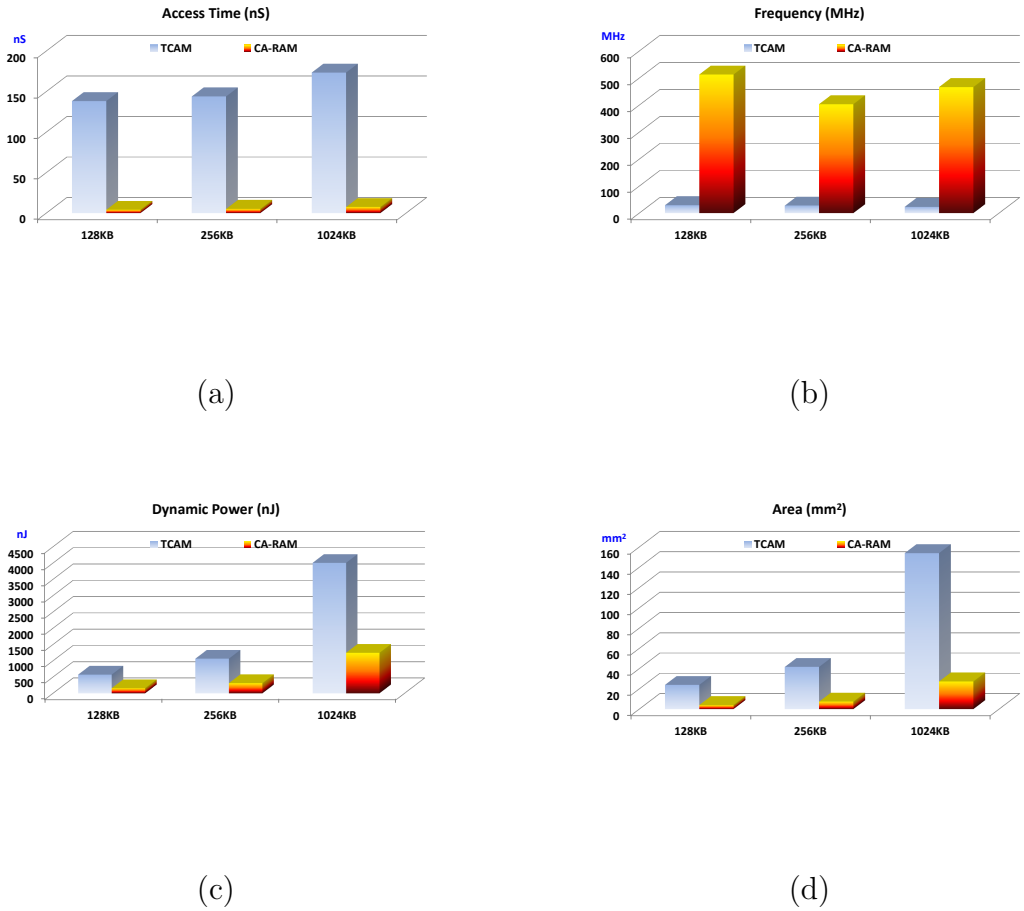


Figure 49: The TCAM vs. CA-RAM in terms of: (a) Total time delay, (b) Maximum operating frequency, (c) Total dynamic power and (d) Total area.

In Figure 49-(a) we compare both the CA-RAM and the TCAM in terms of access time, which can be an indicator for the total delay. On average, over the three configurations, the CA-RAM has a lower access time than the equivalent TCAM configurations by 78%. The average TCAM max frequency, Figure 49-(b), for the three configurations is 94% lower than that of the equivalent CA-RAM configurations. While TCAM provides the result in one clock cycle, our CA-RAM requires multiple (up to 9 or 10 cycles) to provide the same result. However, if we use pipelining, CA-RAM can achieve the same throughput as TCAM.

In terms of the dynamic power, Figure 49-(c), the CA-RAM uses 71% less dynamic power than the TCAM on average for the three configurations. The CA-RAM has, on average, 82.4% smaller area than the TCAM according to Figure 49-(d). Finally, on average, The CA-RAM has an access time of 1.83ns per stage, while the TCAM has 15.18ns access time.

7.4 CONCLUSION

In this chapter we described our CA-RAM-based solution for the pattern-matching problem. The most interesting feature of our solution is that, unlike the TCAM, which can only store single-sized patterns, it can store variable-length patterns. Our solution stored all the SNORT patterns from 2 to 36 characters.

We also introduced a new, dynamic progressive hashing algorithm that took into account the data distribution while calculating the groups and their number. At the same time, we allowed some groups to have more than one hash function.

Moreover, we introduced a modified flexible CHAP algorithm where we varied the number of probing pointers, P which is not equal to H, the number of hash functions used. This modified CHAP algorithm allowed us to reduce both AMAT and WMAT. Finally, we allowed these probing pointers to be accessed only from certain hash functions (i.e., modified CHAP). So, the overall CA-RAM with the dynamic PH + modified CHAP has an average overflow that is less than one percent and an average AMAT that is four percent less than PH + I-mark.

We showed by both simulations and syntheses that a variable-length pattern-matching CA-RAM is feasible. Moreover, we showed that our CA-RAM is superior to the TCAM solution in terms of power consumption, area, and maximum operating frequency. For example, the CA-RAM has 71% lower dynamic power and 82% smaller area on average than does the TCAM. The only disadvantage is that the TCAM evaluates the answer in a single clock cycle, while with our CA-RAM we have to use pipelining to achieve the same performance, but the cycle time is much smaller (i.e., higher frequency).

8.0 CONCLUSION AND SUMMARY FOR THE THESIS

Achieving scalability and low power consumption are our strongest motivations for this thesis. In the packet-processing area, very few solutions combine these two motivations without overshooting the maximum time delay a packet has to incur while traveling across a router or a firewall. This is why we chose set-associative memory architectures (CA-RAM) to pursue this thesis. In addition, the CA-RAM keeps the packet-processing engine computation (i.e., matching logic) close to the stored database (forwarding tables, classification rules, $\cdots$, etc.), which allows for higher processing rates and lower I/O latency.

In this thesis, we discussed our fully developed hash-based schemes, which are the content-based hash probing (CHAP), the progressive hashing (PH) and the independent mark technique (I-Mark). Note that most of our developed schemes are for the multiple hashing domain with emphasis on open address hashing; but at the same time they could be applied to closed address hashing. We verified these tools for the following packet-processing engines:

- CHAP-based IP packet forwarding engine,
- Progressive Hashing-based (with and without I-Mark) IP packet forwarding engine,
- Progressive Hashing-based (with and without I-Mark) IP packet classification engine,
- Hybrid CHAP plus PH-based (with and without I-Mark) IP packet forwarding Engine.

We used both synthetic simulation tools and software simulation to prove that CA-RAM is superior to TCAM in these three areas. The CA-RAM for PF application is estimated to have an average forwarding speed of 320 Gbps with future SRAM technology and 40 Gbps with the standard QDR III SRAM, while storing 301K prefixes in approximately 2.5MB of RAM. For the same application, the standard CACTI memory simulator showed that such

architecture can run on a frequency of 423MHz, which is 94% more than the equivalent TCAM, while maintaining a moderate area and power requirements.

For the PC application, we introduced two CA-RAM architectures. In the first architecture, we use a single CA-RAM with dynamic PH to store on average 30K packet classification filters, while on the second architecture we used two CA-RAMs and split the rule between them.

The main reason for introducing the two CA-RAM solution is to take advantage of the fact that many rules share their two IP (source and destination) prefixes. Though in this case, the AMAT is larger by 36% on average over the single CA-RAM solution, but for the same amount of memory, we were able to reduce the overflow by 62% on average. We estimated that our solutions can achieve a throughput of up to 0.5 Giga packets per second or 160 Gbps for the IPv4 minimum packet size of 40 bytes.

Finally, for the DPI application, we introduced a variable-length CA-RAM for pattern-matching. We used SNORT database, which is an open source network intrusion prevention and detection system from Sourcefire, to evaluate our pattern-matching engine. In comparison to the TCAM, we estimated through the use of CACTI that the CA-RAM has 71% lower dynamic power and approximately 82% smaller area than the TCAM on average. Our CA-RAM can have a clock cycle of roughly 540MHz while the same sized TCAM can run roughly on a 70MHz clock.

9.0 APPENDIX I

In this appendix, I discuss the hash functions that I used in my experiments. The important property of these hash functions is that they must be easy to be implemented in hardware.

The hash functions that I used in my experiments in Sections 5.2, and 6.5 are from three different hashing families: bit-selecting, CRC-based and H3 [38] [9] hashing families. By bitt-selecting, we mean selection of the specific bits, randomly, out of the given keys to construct the hash index. These families have the advantage of being simple, and efficient to realize in hardware. We found that most of those functions perform quite well in both overflow elimination and distribution of the keys uniformly among the table rows. Moreover, we found that direct bit selection and random selection are also good hash functions especially in both PF and PC applications. The authors in [61] came to the same conclusion.

The H3 families of hash functions were first reported by Carter et al. in [9]. Let Q denote the set of all possible $i \times j$ Boolean matrices, where the hashing function is selecting j bits out of the i bits of the key. For a given $q \in Q$, let $q(k)$ be the bit string which is the kth row of the matrix q, and let $x(k)$ denote the kth bit of x. The hashing function $h_q(x) : A \rightarrow B$ is defined as:

$$h_q(x) = x(1) \times q(1) \oplus x(2) \times q(2) \oplus \cdots \oplus x(i) \times q(i) \tag{9.1}$$

, where $\oplus$ denotes XOR operation and $\times$ denotes AND operation on the bit-level.

I used general-purpose string-based hash functions from an open-source library [36] for the DPI application. In addition to the string hash functions from [36], I used the CRC-based and the H3 for the experiments in Section 7.3.

BIBLIOGRAPHY

[1] B. Agrawal and T. Sherwood. Modeling TCAM Power for Next Generation Network Devices. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2006.

[2] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced Allocations. pages 593–602. ACM SOC, 1994.

[3] F. Baboescu, D. M. Tullse, G. Rosu, and S. Singh. A Tree Based Router Search Engine Architecture with Single Port Memories. *ACM SIGARCH Comput. Archit. News*, 33(2):123–133, 2005.

[4] M. Bando, S. Artan, and J. Chao. Lafa: lookahead finite automata for scalable regular expression detection. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS*, pages 40–49, 2009.

[5] M. Becchi, C. Wiseman, and P. Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS*, pages 30–39, 2009.

[6] Bill Lynch and Sailesh Kumar. Smart Memory for High Performance Packet Processing. In *The Proceedings of the IEEE Symposium on High performance Chips, HotChips*, August 2010.

[7] B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] A. Broder and M. Mitzenmacher. Using Multiple Hash Functions to Improve IP Lookups. pages 1454–1463. IEEE Infocom, 2001.

[9] J. Carter and M. Wegman. Universal Classes of Hash Functions. *J. of Computer Science*, 10(2):143–154, 1979.

[10] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40:20–26.

[11] H. J. Chao and B. Liu. *High Performance Switches and Routers*. Wiley-IEEE Press, 1^{st} edition, 2007.

[12] S. Cho, J. Martin, , and R. Melhem. CA-RAM: A High-Performance Memory Substrate for Search-Intensive Applications. pages 230–241. ISPASS, 2007.

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

[14] S. Demetriades, M. Hanna, S. Cho, and R. Melhem. An Efficient Hardware-based Multi-hash Scheme for High Speed IP Lookup. pages 103–110. IEEE HOTi, 2008.

[15] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro*, 24:52–61, 2004.

[16] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. Fast Ppacket Classification Using Bloom Filters. In *ACM/IEEE ANCS*, pages 61–70, 2006.

[17] P. Gigliotti. Implementing Barrel Shifters Using Multipliers. http://www.xilinx.com/support/documentation/application_notes/xapp195.pdf, 2004. Xilinx Application Note.

[18] P. Gupta and N. Mckeown. Packet Classification using Hierarchical Intelligent Cuttings. In *IEEE Hoti*, pages 34–41, 1999.

[19] M. Hanna, S. Demetriades, S. Cho, and R. Melhem. CHAP: Enabling Efficient Hardware-based Multiple Hash Schemes for IP Lookup. IFIP Networking, IFIP, 2009.

[20] M. Hanna, S. Demetriades, S. Cho, and R. Melhem. Progressive Hashing for Packet Processing Using Set Associative Memory. IEEE/ACM ANCS, ANCS, 2009.

[21] M. Hanna, S. Demetriades, S. Cho, and R. Melhem. Advanced Hashing Schemes for Packet Forwarding Using Set-Associative Memory Architectures. *Journal of Distributed and Parallel Computing (JPDC), Elsiver*, 71:1–15, 2011.

[22] G. Huston. Analyzing the Internet's BGP Routing Table. *The Internet Pro. J.*, 2001.

[23] B. S. Inc. SNORT - Users Manual. https://www.snort.org/assets/156/snort_manual.pdf, 2010.

[24] W. Jiang and V. Prasanna. Multi-terabit ip lookup using parallel bidirectional pipelines. pages 241–250, May 2008.

[25] W. Jiang and V. Prasanna. Reducing dynamic power dissipation in pipelined forwarding engines. pages 144–149. IEEE ICCD, October 2009.

[26] S. Kaxiras and G. Keramidas. IPStash: A Power-Efficient Memory Architecture for IP-Lookup. pages 361–373. IEEE Micro, 2003.

[27] S. Kaxiras and G. Keramidas. IPStash: A Set-Associative Memory Approach for Efficient IP-Lookup. pages 992–1001. IEEE Infocom, 2005.

[28] R. A. Kempke and A. J. McAuley. Ternary CAM Memory Architecture and Methodology. http://www.freepatentsonline.com/5841874.html, 1998. United States Patent 5841874.

[29] A. Kirsch and M. Mitzenmacher. Simple Summaries for Hashing with Multiple Choices. *IEEE/ACM Transactions on Networking*, 16:218–213, 2008.

[30] H. N. K. I. H. J. M. 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, 2003.

[31] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for Advanced Packet Classification with Ternary CAMs. pages 193–204. ACM Sigcomm, 2005.

[32] F.-Y. Lee and S. Shieh. Packet classification using diagonal-based tuple space search. *Elsevier Computer Networks*, 50:1406–1423, June 2006.

[33] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, and T.-H. Lee. Using String Matching for Deep Packet Inspection. *IEEE Computer Society*, 41:23–28, 2008.

[34] H. Noda and *et al.* A Cost-Efficient High-Performance Dynamic TCAM With Pipelined Hierarchical Searching and Shift Redundancy Architecture. *IEEE J. Solid-State Circuits*, 40(1):245–253, 2005.

[35] R. Pagh and F. Rodler. Cuckoo Hashing. *Lec. Notes in Comp. Sci.*, pages 121–133, 2001.

[36] A. Partow. General Purpose Hash Function Algorithms Library. http://www.partow.net/programming/hashfunctions/index.html.

[37] M. Pearson. Qdrtmiii: Next generation sram for networking. http://www.qdrconsortium.org/.

[38] M. Ramakrishna and et Al. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Trans. on Comp.*, 46(12):1378–1381, 1997.

[39] Y. Rekhter and T. Li. An Architecure for IP Address Allocation with CIDR. *RFC*, 1993.

[40] Rick Merritt. Huawei Describes Smart Memory Chip. *EE Times*, August 2010.

[41] RIS. Routing Information Service. http://www.ripe.net/ris/, 2009.

[42] M. Ruiz-snchez, E. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*, 15(2):8–23, March 2001.

[43] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. pages 213–224. ACM Sigcomm, 2003.

[44] H. Song and et Al. Fast Hash Table Lookup using Extended Bloom Filter: An Aid to Network Processing. pages 181–192. ACM Sigcomm, 2005.

[45] H. Song, J. Turner, and S. Dharmapurikar. Packet Classification Using Coarse-Grained Tuple Spaces. In *IEEE/ACM ANCS*, pages 41–50, 2006.

[46] E. Spitznagel, D. Taylor, and J. Turner. Packet Classification Using Extended TCAMs. IEEE ICNP, 2003.

[47] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *ACM Sigcomm*, pages 135–146, 1999.

[48] V. Srinivasan and G. Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Trans. Comput. Syst.*, 17(1):1–40, 1999.

[49] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim. A Multi-Gigabit Rate Deep Packet Inspection Algorithm using TCAM. pages 453–457. IEEE Globecom, 2005.

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

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

[52] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1: An Integrated Cache Timing, Power, and Area Model. Technical report, HP Labs.

[53] D. E. Turner. Survey and Taxonomy of Packet Classification Techniques. *ACM Computing Surveys*, 37(3):238–275, 2005.

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

[55] 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.

[56] Xilinx. ISim Hardware Co-Simulation Tutorial: Interacting with Spartan-6 Memory Controller and On-Board DDR2 Memory. http://www.xilinx.com/support/documentation/.../ug818_ddr2_mem_tutorial.pdf, March 2011. Xilinx Free Tutorials.

[57] Xilinx. Spartan-6 FPGA Data Sheet:DC and Switching Characteristics. http://www.xilinx.com/support/documentation/data_sheets/ds162.pdf, October 2011. Xilinx Free Tutorials.

[58] A. Yamazaki, N. Okumura, K. Dosaka, and M. Kumanoya. A fully synchronous circuit design for embedded dram. In *The Proceedings of the 22nd European Solid-State Circuits Conference, ESSCIRC*, September 1996.

[59] F. Yu, R. Katz, and T. Lakshman. Gigabit rate packet pattern-matching using tcam. pages 174–183. IEEE ICNP, 2004.

[60] S. Yun. Hardware-Based IP Lookup Using n-Way Set Associative Memory and LPM Comparator. *Lecture Notes in Computer Science (LNCS), Springer*, 4017:406–414, 2006.

[61] F. Zane, G. Narlikar, and A. Basu. CoolCAMs: Power-Efficient TCAMs for Forwarding Engines. pages 42–52. IEEE Infocom, 2003.