MicroCuckoo Hash Engine for High-Speed IP Lookup

Nikhitha Tata

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter Athanas, Chair
Patrick Schaumont
Joseph Tront

May 5, 2017
Blacksburg, Virginia

Keywords: Cuckoo hash, Packet forwarding, LPM, TCAM

MicroCuckoo Hash Engine for High-Speed IP Lookup

Nikhitha Tata

(ABSTRACT)

Packet classification is a ubiquitous and essential building block for numerous network functions like security, data traffic monitoring, firewalling, multimedia communication and load balancing. The Moore's law of data traffic indicates the network traffic is doubling every two years [1]; thus, tripling the number of hosts deployed to carry out packet routing mechanism. In order to maintain high-speed lookup rates in this extensive data traffic environment, routers must address several challenges like power consumption, memory utilization and lookup delays. Generally, routers employ TCAMs (ternary content-addressable memory) to perform high-speed packet routing. However, due to the intricate memory structure of TCAMs, it consumes excessive power and also makes table reformation task extensively tedious. In recent years, various hash structures like Bloom filters and Cuckoo filters are considered to route the packets [2]. The hash-based IP lookup approach offers exalted query time complexity, while maintaining constant worst-case lookup times. Most of the proposed Cuckoo hash-based techniques use CPE (controlled prefix expansion) and set associative memory to carry out IP lookup operations on prefixes, which adversely affect the storage space complexity of the routers. Thus, to alleviate the memory requirements, a novel hash architecture is proposed, which relies on a micro-Cuckoo hash structures to perform packet routing process. In this design, two powerful engines namely `micro_CHE` and `micro_PL` are developed to carry out complex LPM (longest prefix match) operations on the prefix rules pertaining to each prefix function. Analysis of multiple prefix rules is concurrently executed in all the micro-engines without creating any inter-dependencies; thus, finishing the LPM computations within a single clock cycle. The proposed parallel architecture guarantees constant worst-case query time and also enhance the memory utilization factor, while capable of processing over 280 million packets per second.

MicroCuckoo Hash Engine for High-Speed IP Lookup

Nikhitha Tata

(GENERAL AUDIENCE ABSTRACT)

The internet data traffic is tripling every two years; due to the exponential growth in the number of routers. Routers implement the packet classification methodology by determining the flow of the packet, based on various rule checking mechanisms that are performed on the packet headers. However, the memory components like TCAMs used by these various rules are very expensive and power hungry. Henceforth, the current IP Lookup algorithms implemented in hardware are even though able to achieve multi-gigabit speeds, yet suffer with great memory overhead. To overcome this limitation, we propose a packet classification methodology that comprises of MicroCuckoo-hash technique, to route packets. This approach alleviates the memory requirements significantly, by completely eliminating the need for TCAM cells. Cuckoo hash is used to achieve very high speed, hardware accelerated table lookups and also are economical compared to TCAMs. The proposed IP Lookup algorithm is implemented as a simulation-based hardware/software model. This model is developed, tested and synthesized using Vivado HLS tool.

Acknowledgments

I am indebted to many people for the completion of this work. I am immensely grateful to Dr. Peter Athanas for providing me the opportunity to work on this project. It was because of your constant support and guidance that I was able to explore and develop my research work. Thank you so much for mentoring me throughout my master's journey. I would also like to thank Dr. Patrick Schaumont and Dr. Joseph Tront for agreeing to be a part of my thesis defense committee. I would like to thank my lab mate Jehandad Khan who has been a friendly mentor to me. You have guided me during most crucial phases of the thesis. Thanks a lot for showing me light on various complex topics and patiently answering my queries. I would also like to thank my friends Nishanth, Rishikesh, Swathi, Prerana, Srikanth, Rohit, Fabina, Satya and Sanjuktha for being with me and supporting me at bad times. Finally, I thank my parents and brother, without whom none of this would have been possible. Your support has always strengthened me and helped me to achieve my goals.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Internet is a global electronic communication channel, consisting of millions of interconnected computer networks all around the world. The Internet incorporates Internet protocol suite (TCP/IP) to link all the devices and facilitates the exchange of information using data segments known as packets [3]. TCP/IP suite comprise of two main components - (1) packet transmission medium, and (2) packet processing elements. Packet transmission medium is optical fibers, which provide a means to transfer the packets in between networks. The data packets are formed and analyzed by the packet processing elements known as Internet routers.

High bandwidth is achievable using optical transmission systems. The rate at which the optical fibers transmit data is referred to as the *line rate*, *wire rate* or *transmission rate*. Protocols like SONET, SDH, WDM are employed to transmit gigabits of data per second per fiber [4] [5] . Internet routers implement the packet classification methodology based on various rule checking mechanisms that are performed on the packet headers. Packet classification is a ubiquitous and essential building block for many critical networks functions like

security, data traffic monitoring, firewalling, multimedia communication and load balancing [6]. Thus, it is necessary for the routers to support high-throughput packet classification.

To achieve a remarkable performance gain in packet forwarding mechanisms, routers should match the line rate of optical fibers. In the current technology, optical fibers provide a line rate of 40Gbps [7]. To satisfy a transmission rate of 40Gbps, high-speed routers must be deployed, capable of processing 125 million packets per second. Developing such a powerful large-scale router is strenuous due to two critical concerns - (1) lookup speed - for analyzing one packet, the router must examine millions of entries present in the FIB (forwarding information base) table, and (2) scalability - structure of the networks is dynamically changing due to the addition of new systems, resulting in exponential extension of FIB tables. Henceforth, routers must accommodate all the new prefixes in the FIB table, without augmenting extensive processing delays.

Numerous techniques like TCAMs, trie-based schemes, and hash-based schemes are proposed to enhance the processing time of the router [8]. TCAMs offer excellent lookup time, by completing the longest prefix match mechanism within one clock cycle. Due to the intricate memory structure of TCAMs, it consumes high power and also makes table reformation task tedious [9]. On the contrary, trie-based schemes employ simple data structure to carry out packet classification. However, the trie-based IP lookup engines cannot maintain constant worst-case lookup time due to it's unbalanced structure [10]. Hash-based schemes were developed to overcome these limitations. Hash structures like Bloom filters and Cuckoo filters gained a lot of momentum due to their exalted data query mechanism. But, existing hash-based approaches encompass CPE (controlled prefix expansion) [11] technique to store prefixes, which adversely affect the storage space complexity. Henceforth, there is a lot of scope for the development of new designs and algorithms to achieve high-performance gain, by exploring the packet classification methodologies.

1.2 Contribution

In this work, a new technique called microCuckoo hash-based IP lookup engine is developed; to achieve high lookup throughput, high memory utilization, and also the proposed mechanism maintains constant worst-case time complexity. High lookup throughput is achievable by minimizing the number of memory access required to route a packet. High memory utilization factor is gained by efficiently handling hash collisions and thereby reducing the occurrences of insertion failures. The underlying philosophy is to accommodate all prefixes in small on-chip memory instead of large off-chip memory since on-chip memory access time is trivial compared to off-chip access time [12].

The proposed IP lookup engine is prototyped using Xilinx Vivado HLS based HW/SW codesign tool. The software module drives the control plane functionalities like populating the FIB table, resolving prefix collisions and extracting prefix length information. The hardware module manages the data plane responsibilities like executing LPM technique on prefixes, determining rule match and forwarding data packet based on next-hop value. Unlike previous Cuckoo hash-based IP lookup designs, micro-Cuckoo hash routing do not rely on CPE (controlled prefix expansion) technique to facilitate IP lookup. Instead, a modular pipelined architecture approach is incorporated to accommodate high-speed packet routing. This is accomplished by dividing the tasks based on the prefix lengths of the input rules.

Two powerful engines namely ***micro_CHE*** and ***micro_PL*** are presented in this thesis to carry out the operations on rules pertaining to each prefix function. The `micro_CHE` (Cuckoo hash) engine is embedded into the control plane to rearrange the data present in the FIB tables and henceforth creating space for new incoming rules. On the contrary, `micro_PL` engine effectuates the query mechanism, by querying all the prefixes present in the FIB table. Inspection of the routing table data indicates 90% of the rules have bit length between 16 to 32. Thus, in this design, prefixes having bit length between 16 to 32 are stored in the respective FIB tables, and other rules are not considered to implement packet routing. Each prefix function is affiliated with one `micro_CHE` and one `micro_PL` engine respec-

tively. Analysis of the prefix rules is concurrently executed in all the micro-engines without creating any inter-dependencies, thus finishing the LPM computations within a single clock cycle.

The proposed parallel architecture propels the system to achieve high lookup rate, by processing over 280 million packets per second. This approach also guarantees constant worst-case lookup time $O(1)$. The compact structure of the micro-engines elevates the area efficiency of the FIB tables. In the current design, each prefix function accommodates around 4k prefix rules, which can be implemented by using only 38 18kb BRAMS. The micro-engines also facilitate the parallel IP lookup and store operations of prefixes, which is essential in the routers, enabling high throughput for packet forwarding. The proposed design is developed for IPV4 packet lookup; however, it is easily scalable to IPV6 packets without appending substantial work.

1.3 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 presents a brief review of the existing technologies in the field of IP address lookup. Chapter 3 provides a detailed design and implementation of the proposed Cuckoo hash-based IP lookup engine. Chapter 4 will summarize the results of the feasibility study. Finally, Chapter 5 concludes the work and offers future directions this work can take.

Chapter 2

Background

2.1 Performance Metrics for lookup algorithms

IP lookup is an important category of packet classification handled by the network layer based on the destination address field of an incoming packet. Routers accomplish packet forwarding using the prefix rules stored in the FIB table. They incorporate various mechanisms like LPM (longest prefix match), exact and ternary match techniques, to select a particular rule, based on which packet gets forwarded to an adjacent network [13]. These techniques involve analyses of all the stored prefixes before selection of specific rule. Due to the exponential growth in the FIB table sizes, IP lookup becomes the bottleneck for the routers in maintaining line rates of 40Gbps.

For IP lookup methodology, prefixes stored in FIB table are either 32-bits wide or 64-bits wide, depending on the version of the Internet protocol, i.e., IPV4 and IPV6 respectively [14]. This chapter provides a brief overview regarding the various existing techniques, designed to perform IP lookup methodologies on IPV4 packets. A prefix is a string of data, formulated with '1's, '0's and don't care bits. For any IP lookup algorithm, the main critical performance metrics are search time, memory storage requirement, update speed, and scal-

ability [15]. The search time parameter indicates the number of memory accesses required by the algorithm to route a single packet. The worst-case lookup time has vital importance in contrast to average lookup time, due to the complexity of the FIB tables and rate of packet drop [16]. The second metric indicates, how efficiently the algorithm stores the prefix data in FIB tables, without wasting any storage space. Information associated with the prefixes is dynamically changing due to the updates made to the networks. Thus algorithm should be able to update the FIB table information successfully, without augmenting substantial delays. This process is assessed using an update speed metric. Lastly, the number of sub-networks associated with any application is not persistent. There might be addition of new networks or elimination of existing ones. Hence, the algorithm must be scalable to accommodate all the new prefixes in the FIB table.

Various algorithms are developed to enhance these performance metrics. There are mainly three different categories of the algorithms - (1) TCAM (Ternary Content-Addressable Memory) approach, (2) trie-based schemes, and (3) hash-based schemes.

2.2 TCAM

A ternary CAM is an enhanced version of content addressable memory, which consists numerous small memory elements known as TCAM cells [9]. Each TCAM cell is capable of storing three kinds of bit values, namely '0', '1', and 'x'. This allows them to store and query prefix rules, as prefixes contain don't care bits. Figure 2.1 depicts the architecture of TCAM based LPM routing. TCAM comprising of N cells can accommodate N prefix rules. One TCAM cell gets associated with one prefix rule; thus, each TCAM cell stores a (value,mask) pair corresponding to that particular rule. As shown in Figure 2.1, destination address information broadcasts to all the TCAM cells [17]. In each cell, the received destination address field comparison happens against the value stored in that cell with the help of the mask associated with it. If the stored value of the TCAM cell matches the incoming destination

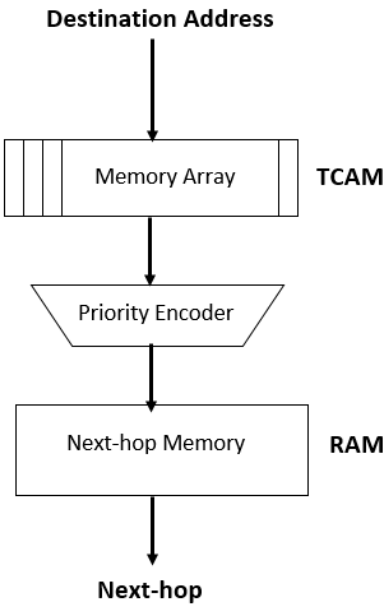


Figure 2.1: Block diagram of TCAM based longest prefix matching

address, it updates the respective bit value in the matched_bitvector array to '1'. If a given destination address matches multiple prefix rules, the priority encoder is employed to choose one particular rule. The matched_bitvector array is the input to the priority encoder; it selects a rule associated with the lowest bit that is set in the bitvector, because, lower bits of the matched_bitvector array gets affiliated to the prefixes of higher lengths. The output of the priority encoder is used as an index to access the RAM memory, to retrieve next-hop information associated with the selected rule. Thus, a TCAM is chosen in high-speed routers, as it can perform LPM matches in one memory clock cycle irrespective of FIB table size [18, 19, 20]. However, it poses two critical challenges. Firstly, each TCAM memory cell requires double the number of transistors compared to SRAM, making them power hungry and costly. Secondly, a TCAM stores prefixes in the descending order of their length, i.e., contiguous TCAM cells stores prefixes of the same length. Hence, to accommodate one new prefix rule, entire TCAM structure should be modified. Various IP lookup mechanisms like tree-based and hash-based schemes are developed to overcome these limitations.

2.3 Trie-based Schemes

In computer algorithms, a trie structure is a unique non-linear data structure used to store a dynamic set of strings. It is also known as a *digital tree*, *prefix tree*, and *radix tree* [21]. A trie data structure can be visualized as a tree consisting of a root and several branches (sub-trees). Further, each sub-tree has its own root and children, and so on. A tree structure can be graphically represented using nodes and edges (links). A node is labeled as an external node if it does not have any children. On the contrary, a node having a single child is called an internal node. An M-ary tree is a tree with its nodes having at most M children. The packet forwarding mechanism employs the 2-ary tree structure, because it deals with only two binary digit numbers '0' and '1'. As a result, each node has at most two branches corresponding '1' and '0' respectively.

The word trie has its origin from the word 'retrieval' [21], which means trie memory can be used to store and retrieve information. The trie data structure stores (key, value) pairs using nodes and edges. The root node of the 2-ary tree consists of two sub-nodes. The left branch of the root node is labelled as '0', and the right branch as '1', which connects the root nodes with the two sub-nodes respectively. To store a given prefix rule (p, v_p) in the tree; router first needs to retrieve the root node information. The first bit of the prefix is extracted to make a branching decision. If the first bit is '0', algorithm access the left node; otherwise, the right node is accessed. In a similar manner, all the remaining bits are selected one bit at a time, which leads to the particular node n in the tree. Now, this node n stores the next-hop information value v_p associated with the given prefix rule. Trie structure represents the key value p of the prefix rule by the concatenation of the labels of all the branches in the path from the root node to node n. Thus, portraying the key value by the path connecting root node to a particular node. Many algorithms are proposed to use trie memory for IP lookup mechanism. Major classifications of Trie-based packet forwarding include - (1) binary trie, and (2) multi-bit trie. The FIB table shown in Table 2.1 is considered, to explain binary trie and multi-bit trie approaches.

Table 2.1: Forwarding table for trie-based approach

	Prefix	Next_hop
R1	0*	a
R2	01000	b
R3	0111*	c
R4	1*	d
R5	100*	e
R6	1100*	f
R7	1101*	g
R8	1110*	h
R9	1111*	i

A *binary tree* is the basic trie-based data structure [22]. In a binary tree, each node can accommodate at most two children. The binary tree structure for the FIB table presented in Table 2.1 is given by Figure 2.2. A prefix rule of length W bits needs to traverse a path of W nodes to store the information associated with that prefix rule. In other words, Rule R3 has a prefix length of four bits. Thus, the binary tree covers a path consisting of four nodes to store next-hop information C associated with Rule R3. Hence, to verify if a given destination address matches any of the prefixes present in the tree, the binary trie technique has to check at most W nodes associated with a given prefix. For IPV4 packet forwarding, the prefix length can be up to 32-bits. Therefore, the binary tree algorithm requires 32 steps to perform longest prefix matching.

To summarize the performance metrics of a binary tree based IP lookup engine; consider a prefix rule with W defined bits (prefix length is W). The time complexity needed to look up this prefix is $O(W)$because it requires W memory accesses. To add this prefix as a new rule, at most W new nodes may be added, making the update speed complexity to be $O(W)$. Lastly, the memory complexity to store N prefixes of W-bits is $O(NW)$. Many enhanced algorithms like a Patricia tree [23], path-compressed tree [24] and leaf pushed tree [25] are proposed to alleviate the storage requirements needed by a binary tree (by sharing nodes between prefixes to store data). However, all these algorithms suffer from

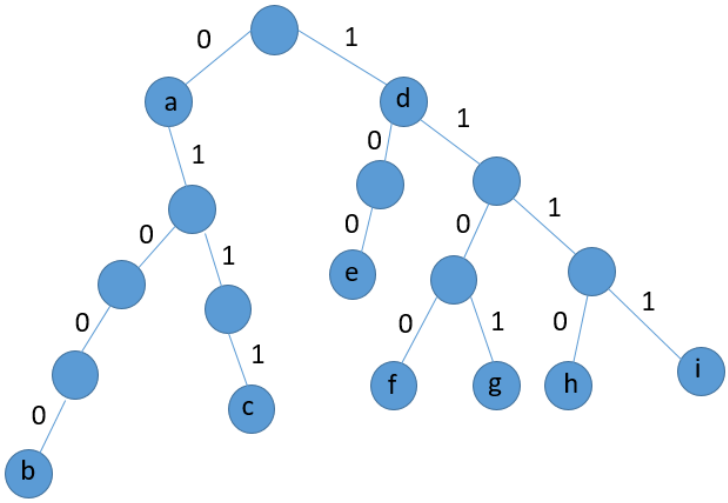


Figure 2.2: A binary trie storing the prefixes of Table 2.1

inflated lookup latencies (lookup steps in the worst-case), due to the sequential bitwise matching technique incorporated. Thus, making these algorithms slow and unsuitable for high throughput packet forwarding.

The query speed is limited in the binary trie structures as each node processes only one bit at a time. Henceforth, the lookup rate can be elevated by analyzing multiple bits at a time [22]. This concept forms the basis for developing multi-bit trie data structures. In the binary trie, to store a prefix of length W bits, it needed W levels. However, in multi-trie approach, if each node is inspecting k bits at a time, it requires a maximum of only $\frac{W}{k}$ levels, to store a given prefix. The resulting trie structure is known as 2^k-ary trie or 2^k-way tree. Thus it requires only k stages to perform IP lookup, compared to binary trie which required W steps.

Stride value of a trie represents the number of bits that can be assessed by a trie node at any given time. Each level gets associated with a unique stride value, which then decides how many bits get processed by the node present in that particular level. If the stride value of Level 1 is k, all the nodes associated with Stage 1 can process k bits at a time. Thus the prefixes stored at these nodes must have k defined bits, which is achieved

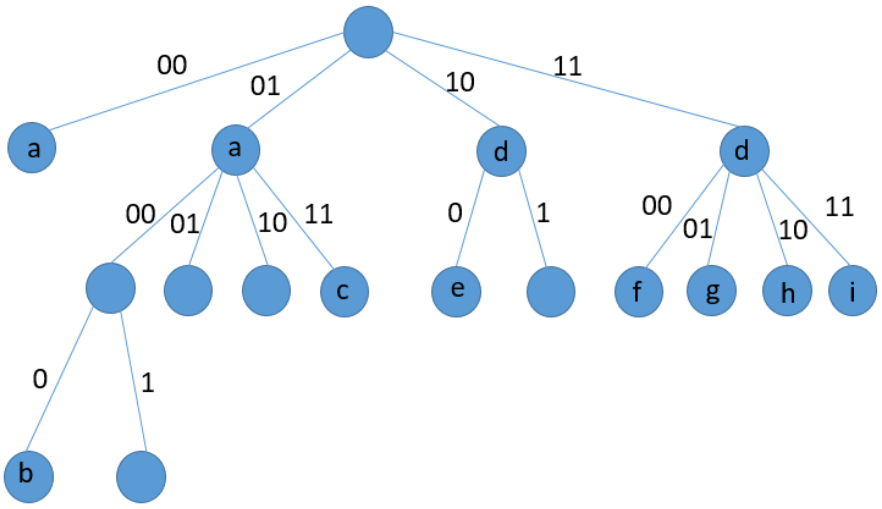


Figure 2.3: A 2-ary multi-bit trie storing the prefixes of Table 2.1

using prefix expansion. Consider Figure 2.3, which represents the multi-bit trie structure of the FIB table. The prefix length is five and stride value is two. Thus three Levels are formed. The stride value implies, all the nodes present in the trie can analyze two bits at a time. Thus, prefix rules stored in this 2-way multibit tree must contain multiples of two defined bits. For example, prefix 0* gets expanded to 00*, 01* (two defined bits) and prefix 100* to 1000*, 1001* (four known bits). Therefore prefix 100* can be stored using only two nodes, hence requires only two memory lookups to locate this prefix, thus, decreasing the lookup time required by the LPM mechanism. However, this prefix expansion process elevates the storage requirement of the multi-bit trie algorithm due to two main reasons - (1) expanded prefixes creates new prefixes, thus increasing the number of nodes, and (2) the next-hop corresponding to a single prefix rule gets stored in all the new prefixes set up by the expansion. In comparison with binary trie, the multi-bit trie is doing the tradeoff between memory size and memory query time.

2.4 Hash-based schemes

Trees are realized using abstract data structures, which make them difficult to implement on hardware. In contrast, the hash-based lookup can be achieved using the flat data structure, which makes them suitable for hardware implementation. Unlike trie structures, where latency of an operation depend on the key length (number of nodes), hash structure latencies can be made constant ($O(1)$) independent of key length. Additionally, hash tables can also potentially scale well with the exponential growth of FIB tables. Hence, hash-based IP lookup engines are suitable for routing IPV4 packets.

In hash-based schemes, the RAM tables (also known as hash tables) store the prefix rules. Two different methods can be used to accommodate data in these hash tables, namely - (1) open hashing/separate chaining, and (2) closed hashing/open addressing. The input to hash tables is a set of key elements. Open hashing hash tables have a fixed height, i.e., fixed number bucket locations. However, each bucket is a linked list of infinite size. Thus, each address of the hash table holds a pointer respective to the header of a linked list. In this approach, a linked list is used to store the keys. Open hashing is used to minimize the lookup time, however, due to the complex structure of linked lists, it is not easy to port it to hardware. On the contrary, a closed hashing table structure has a fixed height, and a fixed width, i.e., the number of elements stored in a bucket is fixed. The hash tables directly store the key value by using it to index to a particular location. Closed hashing based hash tables are more amenable to implement on hardware, due to its simple table structure. Thus, most of the hash-based packet forwarding techniques use closed hashing methodology like Bloom filters, Cuckoo hash, etc. to store prefix values.

However, closed hashing based approach has some glitches. First, the closed hashing technique is prone to data collisions, as multiple key elements might map to the same location. Thus it requires collision resolution techniques to avoid insertion failures. Secondly, RAM memory used to store hash tables doesn't have the capability to manipulate don't care bits; thus, to conduct packet forwarding either prefixes should be expanded, or a

lookup mechanism should be made intelligent enough to handle the don't care bits. Lastly, developing a powerful hash-based scheme for performing LPM based IP lookup is difficult.

Many hash-based algorithms are proposed to overcome these limitations. The first and foremost hash-based IP lookup technique is conducting the binary search on prefixes. Pankaj et al. [17] and Waldvogel et al. [26] propose packet forwarding algorithms based on binary search. The hash table stores all the possible prefix values. Received key value acts as an index to the hash table and retrieves the next-hop information associated with it. Thus, the lookup gets completed in single clock cycle with only one memory access. Also, since it maps one prefix to one location, LPM technique is not required. These algorithms incorporate exact match method. Since IPV4 packet forwarding mechanism deals only with destination field which is 32-bit wide, there are 2^{32} different combinations of prefixes possible. Thus, a hash table should store all these alliances, which increases the storage complexity exponentially resulting in area inefficiency and making the algorithms non-scalable to other applications.

To alleviate the memory utilization problem, Bloom filter based IP lookup algorithms came into the picture [27]. Bloom filters provide a way to store and query prefixes without having to save all possible combinations of IP prefixes. Dharmapurikar et al. proposed the first bloom filter based packet forwarding engine [28]. The technique is to store prefixes of different lengths in different hash tables, and checking if a prefix is a part of any particular group to determine which prefix length rule does the destination address match. Nevertheless, the number of memory accesses made to determine a rule match is high, as the target address gets compared against all the prefixes stored in the database. Song et al. proposed the Extended Bloom filter based IP lookup, to minimize the number of memory lookup accesses [29]. However, these algorithms are not amenable to rational mechanisms of IP lookup, as updating of the FIB table gets curtailed, as the Bloom filter doesn't support deletion of data items. Minlan Yu et al. incorporated counting Bloom filters to resolve this issue, at the cost of space overhead [30]. Despite its high-speed query mechanism, a Bloom filter based approach suffers from false positive occurrences, resulting in the incorrect packet

routing and, thus, making it difficult to maintain constant worst-case lookup time.

Alternative hashing structures like Cuckoo hash were introduced in packet forwarding mechanisms to overcome the failing of bloom filters [31]. Most of the proposed Cuckoo hash-based IP lookup techniques use CPE (controlled prefix expansion) and set associative memory [32] to carry out IP lookup operations on prefixes. As mentioned, the lookup time of IP lookup methodology is arduous to maintain constant, as it has to deal with a set of prefixes with distinct prefix lengths. CPE is a technique used to reduce the number of different prefix lengths present in the FIB table, by expanding each prefix to the desired length. Kaxiras et al., Socrates et al., Minseok et al. and Michel et al. have proposed IPStash [33], MHT [10], LACF [34] and CHAP [35] schemes respectively using CPE based hashing methodology. In these algorithms, the prefix rules of L distinct lengths are categorized into N smaller groups $N << L$. All the prefixes present in one class has same prefix length. Consider a three class CPE expansion, i.e., $(\leq 15, 16-24, \geq 24)$. The prefixes present in class two (16-24) gets expanded to the prefix length of 24-bits irrespective of its original length. Thus, the longest prefix matching technique is executed by comparing incoming destination address against only three classes in, rather than comparing with all prefix functions. The matching process can either be operated sequentially or parallelly. Due to this, the lookup time reduces significantly. However, prefix expansions have adverse effects on the memory storage space.

All the above mentioned algorithms can be broadly categorized into hardware-based and software-based approaches, depending on their implementation platform. The performance of the hardware-based schemes is delimited by the memory utilization and power dissipation factors concerning the lookup engine. On the contrary, software-based schemes offer higher flexibility regarding space management of the FIB tables. However, the credibility of the algorithm is constrained by the throughput of the lookup engine. Henceforth, in this work, an intermediate solution is proposed, by designing an HW/SW codesign based IP lookup engine. This model has been developed, tested and synthesized using the Xilinx Vivado HLS tool [36][37].

Chapter 3

Design & Implementation

In this Section, a novel technique is presented for IPV4 Layer three packet lookup using concurrent micro-cuckoo hash engines. The hash-based IP lookup approach guarantees constant worst-case query time, while capable of processing over 280 million packets per second. There are two Sections in this chapter, (1) robust Cuckoo-hash engine, and (2) high throughput IP lookup. Section 3.1 deals with the development of an effective collision resolving Cuckoo hash engine, which is also proficient of achieving a load factor of 99%. Section 3.2, provides a detailed description of the various algorithms used by the micro-Cuckoo hash IP lookup, namely - a setup algorithm, a lookup algorithm, and an update algorithm.

3.1 Robust Cuckoo Hash Engine

3.1.1 System Overview

The Cuckoo hashing technique is an open addressing hash mechanism that provides an efficient solution for data query operations while maintaining constant worst-case lookup time [38] [39]. As FPGA-based data processing is becoming increasingly relevant in data centers, demand for FPGA-based Cuckoo hash solutions is growing exponentially [40]. In

this work, a high throughput robust Cuckoo hash engine is proposed, which is capable of leveraging the inherent parallelism present within FPGAs. The proposed Cuckoo hash algorithm is implemented as a simulation-based hardware/software model. This model has been developed, tested and synthesized using the Xilinx Vivado HLS tool.

The proposed HW/SW-based Cuckoo hash engine can perform four major operations, namely data_update, data_insert, data_delete and data_query. The HW/SW model is chosen because it helps in enhancing the performance metrics (fmax, throughput, latency) of the design, while at the same time, it alleviates the memory requirements needed for the hardware implementation [41]. For the rest of the discussion, the software part of the model is referred to as **CH_MAIN** and hardware part of the model is called as **CH_FORWARD**. In this design, the `CH_MAIN` and `CH_FORWARD` communicate with each other through three different streams of data, such as **UPDATE_IN** channel, **QUERY_IN** channel and **QUERY_OUT** channel. `UPDATE_IN` channel handles the store operations like data_insert, data_update, and data_delete. The `QUERY_IN` & `QUERY_OUT` channels get dedicated for the lookup operation known as data_query. The computations carried out in store and lookup algorithm is distributed across both `CH_MAIN` and `CH_FORWARD` modules. `CH_MAIN` plays a significant role in store algorithm by handling all the computational work, whereas lookup algorithm is carried out by `CH_FORWARD`. The implication is that the software module has a significant contribution in populating the hash tables and hardware module does the actual memory lookup operation. Thus, the RAM space needed for the implementation of the `CH_FORWARD` is alleviated.

The input to the Cuckoo hash engine is a key-value pair (x, v_x). The key data gets stored in the d hash tables, and the values associated with each key gets stored in another set of d metadata tables at the same location. Hash tables and metadata tables have similar table structure as shown in Figure 3.1. They contain the same number of memory locations. However, the width of tables depends on the size of key data and value data respectively. Hash table locations are associated with a `VALID_BIT` field, to indicate the credibility of the stored data items. Hence, the width of hash table is $(sizeof(key) + 1)$ bits. Only hash

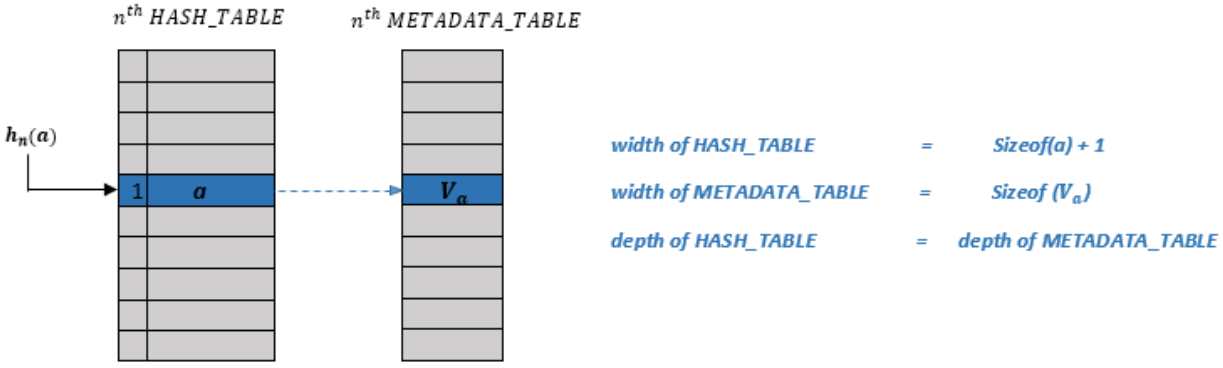


Figure 3.1: Structure of `HASH TABLE` & `METADATA TABLE`

tables participate in all the computations; the metadata table gets updated once it finds the final location.

Each of the d hash tables present in the Cuckoo hash engine gets associated with a maximum of b buckets. Hence, the total number of sub-tables formed will be $n = d \times b$. The number of buckets associated with a hash table indicates the number elements that gets stored per hash table position. Hence, a given key-value pair gets placed in any of the sub-tables $1, 2, ...n$ in positions $b_1, b_2, ..., b_n$ determined by a set of d hash values. The d hash values are generated using a single CRC32 hash function. Sub-tables associated with a particular main table share single hash value. Therefore, n sub-tables require only d i.e. n/b hash values. The sub-table and a specific location in the sub-table are identified using `table_id` and `bucket_id` values. For the rest of the discussion, we assume $d = 4$, $b = 2$ and $n = 8$.

3.1.2 CH_STORE Algorithm

As mentioned in Section 3.1.1, algorithm uses `UPDATE_IN` channel for store operations. The store algorithm (CH_STORE) has two primary objectives - (1) to determine a `table_id` and `bucket_id` for a given key-value pair using `CH_MAIN` (software module), and (2) then transferring this computed information to `CH_FORWARD` (hardware module) to carry

out the actual memory write. This Section first presents the overall structure of the store algorithm, later the functionality of each of the store operations is discussed. `CH_MAIN` and `CH_FORWARD` maintains a duplicate copy of both hash table and metadata table. The duplicity is accomplished using the `UPDATE_IN` channel. To avoid any chaos, the tables of `CH_MAIN` are named as `SW_HASH_TABLE` & `SW_METADATA_TABLE`, and tables of `CH_FORWARD` module are named as `HW_HASH_TABLE and HW_METADATA_TABLE`. All memory locations of both `SW_HASH_TABLE` and `HW_HASH_TABLE` gets concatenated with a `VALID_BIT`. This `VALID_BIT` indicates whether the key stored at a particular location is valid or not.

The key-value pair and operation (operation value can be data_update, data_insert or data_query) are fed as the input to `CH_MAIN` module. Given key-value pair can be stored in any one of the n candidate locations. The `table_id` and `bucket_id` value identifies each of the locations. For $1, 2, ...n$ locations, the table_id value will be $1, 2...n$ respectively, because a location gets mapped to one sub-table. The next step is to determine the `bucket_ids` of these n locations. Hash functions are used to generate `bucket_id data`. The objective of the chosen hash function is to generate unique `bucket_ids` for each key data, which would decrease the collision rate. A straightforward approach is to consider one hash function per sub-table. Even though this method guarantees significant randomness, it entails a lot of computations and also consumes more space when implemented on FPGAs. Hence, in this design, the `CH_STORE` algorithm uses only one hash function, namely CRC32, to create multiple bucket_ids by manipulating the output of the CRC32 hash function.

As discussed in the previous Section, only d hash values are needed to represent n number of candidate locations. Hence, in this design, four different hash values (HV) are generated, which will get shared among the sub-tables. The first two locations use `hash_id_0`, the next two locations use `hash_id_1` and so on. The CRC32 hash function operates on 128-bit input key and generates 32-bit hash data as output. This 32-bit hash data gets grouped into two 16 bit numbers named as `CRC32_high` (upper 16 bits)

and `CRC32_low` (lower 16 bits). After a protracted trial and error method, the following equations are considered to generate four unique `hash_ids` -

$$\text{hash\_id\_1} = \text{crc32\_low}((\text{LOG2\_TABLE\_SIZE} - 1) + 1, 1) \tag{3.1}$$

$$\text{hash\_id\_2} = \text{crc32\_high}(14, (15 - \text{LOG2\_TABLE\_SIZE} + 1) - 1) \tag{3.2}$$

$$\text{hash\_id\_3} = \text{hash\_id\_1} + \text{hash\_id\_2} \tag{3.3}$$

$$\text{hash\_id\_4} = (3 * \text{hash\_id\_1}) + (2 * \text{hash\_id\_3}) \tag{3.4}$$

The next step is to select one location among these eight locations to store a given key-value pair (x, v_x). For this purpose, data from all of the eight locations of `SW_HASH_TABLE` is analyzed, indicated by the `table_id` and `bucket_id` values (one data from one sub-table), and a particular sub-table and bucket is selected. The complexity of computations that are carried out on the garnered `SW_HASH_TABLE` data depends on the operation input (data_update, data_insert or data_delete). Key x is stored in the location `SW_HASH_TABLE [table_id][bucket_id]`, and value v_x is stored in `SW_METADATA_TABLE [table_id][bucket_id]` respectively. The `VALID_BIT` associated with location `SW_HASH_TABLE [table_id][bucket_id]` gets updated to 1, which indicates successful completion of the store operation. Upon completion of store computations, a data frame is created and written into the `UPDATE_IN` channel as shown in Figure 3.2(a). The store algorithm keeps track of the number of writes made to the update stream. The role of `CH_MAIN ends` here, and it calls `CH_FORWARD` to execute further steps.

The `CH_FORWARD` module monitors the `UPDATE_IN` channel continuously using the command *`(!update_in.empty())`*. It doesn't take any action if the update stream is empty. As soon as a data frame arrives, the hardware module flattens the data frame and extracts all the relevant information as shown in Figure 3.2(b). The actual store into the hardware table i.e. memory write takes place in this step. Input key data is stored in

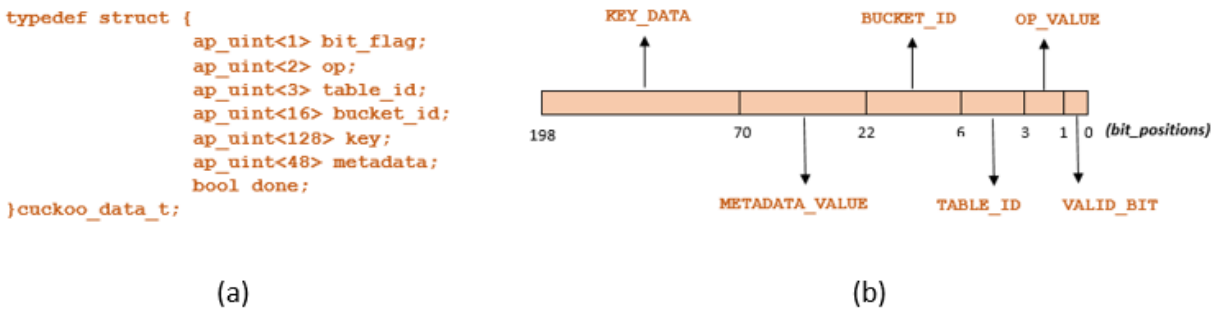


Figure 3.2: (a) Data structure written into `UPDATE_IN` channel by `CH_MAIN`, (b) Data Flattening done in `CH_FORWARD`

`HW_HASH_TABLE [table_id][bucket_id]`, and the value associated with the key, is stored in `HW_METADATA_TABLE [table_id][bucket_id]`. Also, `VALID_BIT` of this location gets changed to 1. Thus, the tables maintained by both `CH_MAIN` and `CH_FORWARD` contains same data items. Therefore, the two primary objectives of the `CH_STORE` algorithm is achieved successfully.

Below Section describes in detail about each of the store operation.

- Data_Update (x, v_x): First, for a given input key x, data from all n candidate locations of `SW_HASH_TABLE` are captured. For any particular location, if `VALID_BIT` is 1 and the stored key data matches item , the value associated with is updated in `SW_METADATA_TABLE`. Otherwise, the operation value gets changed to data_insert, which implies, if a given key entry is not available in the `SW_HASH_TABLE`, store algorithm treats this as a new entry and performs data_insert operation instead of data_update. In the case of data_update operation, the `UPDATE_IN` channel sends only one data frame to `CH_FORWARD`.

- Data_Insert (x, v_x): Initially, the sub-table given by `table_id` $= 1$ is selected and position `bucket_id_1` is accessed. If the `VALID_BIT` of this accessed location is 0, the new element gets inserted here. If not, the second table is selected, and the process

keeps on repeating until it finds an empty position or searches for all n sub-tables. Lack of an empty location indicates, all the eight candidate locations of `SW_HASH_TABLE` associated with the given key x, contain preoccupied data. In this scenario a random table k is selected and the new key x is stored in position `SW_HASH_TABLE [K][bucket_id_k]`, simultaneously value v_x is stored in `SW_METADATA_TABLE [k][bucket_id_k]`. Then, the insertion process gets executed for the entry (y, v_y) that was displaced when inserting x. For adding an entry (y, v_y), sub-table k gets excluded, to avoid occurrences of endless loop. This procedure is recursive and tries to move elements internally, to accommodate new element if needed. The number of writes made to the `UPDATE_IN` channel depends on the number of kick-outs that occur while inserting element x. Thus, the complexity of computation involved in the data_insert operation is more compared to other store operations.

- Data_Delete (x): This operation is similar to data_update operation. If a given element x is found in any one of n candidate locations, the `CH_STORE` algorithm updates `VALID_BIT` associated with that particular memory location to 1. This symbolizes that the data stored in this specific location is no longer valid and can be replaced with any other key in future. In the case of data_update operation, the `UPDATE_IN` channel sends only one data frame to `CH_FORWARD`.

3.1.3 CH_LOOKUP Algorithm

The goal of the lookup algorithm (CH_Lookup) is to verify the existence of a particular key-value pair within `HW_HASH_TABLE`. As discussed in the previous Section, store algorithm populates the `HW_HASH_TABLE` and `HW_METADATA_TABLE` with the key-value pairs respectively. Query operation is performed using this populated `HW_HASH_TABLE`. The input to the lookup algorithm is a set of key data. `CH_MAIN` accepts the key input and writes the input value into the `QUERY_IN` channel. The software module `CH_MAIN`, just act as a buffer to transfer the keys to `CH_FORWARD`. The hardware module performs all the computations

for this algorithm.

CH_FORWARD monitors the QUERY_IN channel continuously using the command *(!query_in.empty())*. If the QUERY_IN stream is empty, no further action is taken by the module. Once the stream gets populated with data, CH_FORWARD reads the input key value. To verify the existence of this particular key data in the HW_HASH_TABLE, the lookup algorithm needs to assume the possible locations for storage of this data. Since the module operates on eight sub-tables, the data could have been stored only in one of these eight locations. The information regarding these locations is obtained by applying a CRC32 hash function on input key data. From the CRC32 output, eight different hash values are calculated using equations 3.1 to 3.4. Hence, with the help of CRC32 function, the search for the query operation is narrowed down to eight locations irrespective of the depth of HW_HASH_TABLE. One location per sub-table is considered, where there is a probability of this input key getting stored. CH_FORWARD compares the given key data against the keys stored in all of these eight locations. If it finds a match, the value from the HW_METADATA_TABLE associated with this particular key is extracted and written into the QUERY_OUT channel else NOT_FOUND command is passed to CH_MAIN module.

Both CH_store and CH_lookup algorithms indicate that the work done by CH_FORWARD is minimal in comparison with CH_MAIN. Thus, high throughput values can be achievable for a Cuckoo hash engine using this HW/SW codesign approach. However, the insertion latency of the model is limited, due to the complexity of computations handled by the data_insert operation. To ensure the reduction in the computational complexity, additional features are added to the existing HW/SW model. Section 3.2.4 and Section 3.2.5 explains the methodology incorporated to boost the insertion timing latency successfully.

3.1.4 Pseudo-Random Kickout Mechanism

Cuckoo hashing is a multi-hash table scheme, which facilitates the internal movement of inserted elements among the tables to make space for a new element. As discussed in Section

3.1.2, the data_insert operation has the capability to resolve collisions to accommodate a new entry, when all n sub-tables are populated. The mechanism used to free preoccupied locations is known as random walk approach [38]. For a given new entry value, if all n candidate locations are occupied, the data_insert operation will kick-out an element from a random location (random location is one among n candidate locations) and continue the process until it can fit back all the kicked out entries into the hash table. Frequent kicking-out operations cause intensive data migration among multiple locations of hash tables, resulting in endless loops and further escalating the time required by the data insertion process. Therefore, the number of kick-outs must be minimum, which in turn means the locations should not be selected randomly to insert elements.

In the random walk scheme, the insertion latency is high because kick-outs from some memory locations may happen more frequently compared to other seldom used ones. Few memory locations participate in frequent collisions due to the randomness involved in selecting them. Henceforth, a new approach known as Pseudo-Random kick-out mechanism is proposed, which alleviates the occurrences of the endless loops and thus enhances the data insertion latency. This Pseudo-Random approach gets incorporated into the HW/SW Cuckoo model explained in Section 3.1.1.

The PR scheme is similar to the random walk approach, where each input entry gets associated with n candidate locations (one location per sub-table). It means that a given key-value pair gets stored in any one of the available n candidate memory locations. A counter is allocated for each sub-table to record the number of collisions that happen in that particular sub-table. Each sub-table encompass 0 to $(tablesize - 1)$ memory locations. Hence, kick-outs happening in any of these memory locations (l_1 to l_{table_size}) increments the counter associated with that particular sub-table. The **`sb_counters`** (sub-table counters) monitor the number of collisions occurring in each sub-table. It helps us to gauge information regarding the number of '**cold**' memory [42]locations present in each sub-table. If hash collisions at a particular location occur infrequently, it is regarded as a cold memory location. For a given sub-table, if the `sb_counter` value is high, it implies most of the locations store

data and participate in kick-outs repeatedly. On the contrary, a low `sb_counter` value insinuates that most of the locations may not participate in the kick-out operations. PR-based kick-out approach curtails the occurrences of the endless loop, by uniformly dividing the store operations among the sub-tables. When kick-outs happen in all the sub-tables uniformly, the probability of collision occurring at a single location gets reduced, and hence chances of infinite loops are also diminished compared to random walk scheme.

For a given input element, if no empty location is available, the scheme selects a location associated with a minimum `sb_counter` value to kick-out the occupied element. Figure 3.3 illustrates the structure of pseudo-random mechanism. The colored locations indicate n candidate locations, where a given element can be stored (n locations are computed using a CRC32 hash function). If other keys have occupied all the $h(x)$ locations, the input item has to replace an existing key through the PR scheme.

Figure 3.4, depicts the actual mechanism of the PR (pseudo-random) scheme. For this example, consider $d = 2$, $b = 2$ and hence $n = 4$; therefore given key x has four candidate locations. These four locations are computed using CRC32(x) function. Since, $d = 2$, only two hash_ids, `hash_id_1` and `hash_id_2` is calculated using Equation 3.1 and Equation 3.2 respectively. `SW_HASH_TABLE` has four sub-tables, first two sub-tables use `hash_id_1`, next two sub-tables use `hash_id_2`. Firstly, the locations l_1, l_2, l_3, l_4 of `SW_HASH_TABLE` should be investigated, to discover an empty location. In the current scenario, all of the four candidate locations of x is occupied by key-1, key-2, key-3 and key-4 respectively. In the next step, the Cuckoo hash engine compares the `sb_counter` values of four sub-tables, and further selects a candidate location associated with a minimum `sb_counter` value. In this example, 15 is minimum. Hence location l_2 is chosen. Further, the existing item key-2 get replaced with input entry x. The value associated with x gets updated in location l_2 of `SW_METADATA_TABLE`. In the meantime, the sb_counter value of the sub-table2 is incremented by one. The kicked-out item key-2 becomes the one that is needed to be inserted, and the insertion procedure goes on until it finds an empty slot in the hash tables.

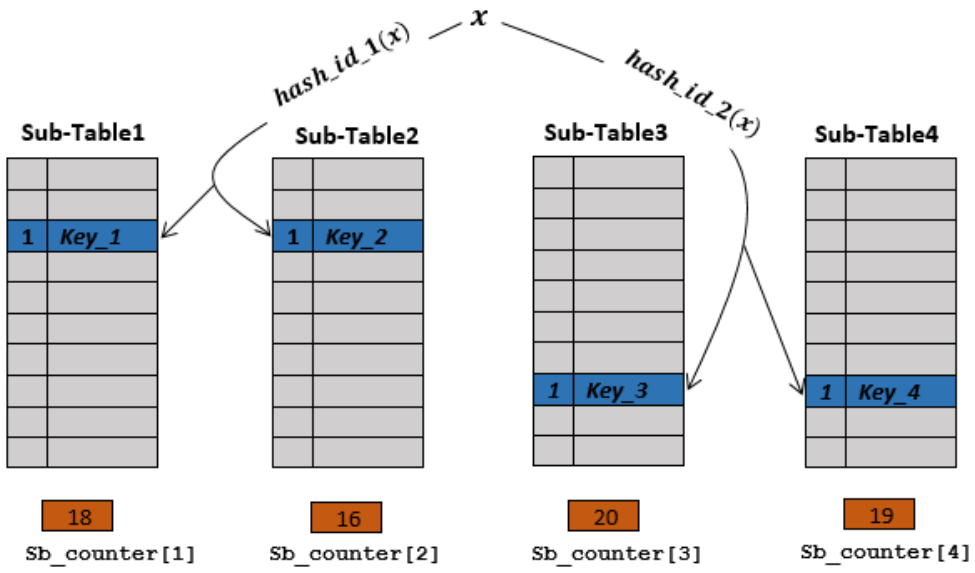


Figure 3.3: Initial status of `SW_HASH_TABLE`

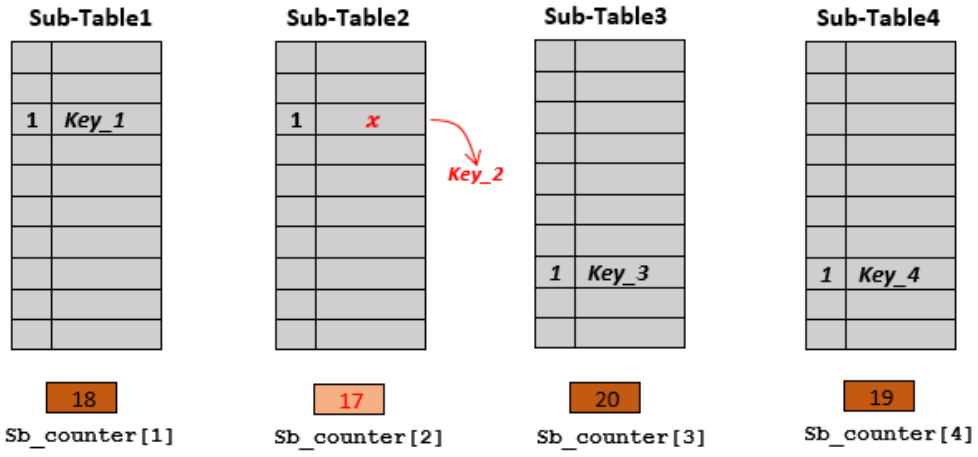


Figure 3.4: Structure of `SW_HASH_TABLE` after 1^{st} iteration

3.1.5 RUN_COUNT Scheme

As discussed in the previous Section, `sb_counters` keep track of the cold buckets and further PR-based Cuckoo hash engine uses these `sb_counters` to minimize the occurrences of kick-outs when moving the stored data across the hash tables. This mechanism helps in improving the data insertion latency; it also reduces the frequency of occurrences of infinite loops. Nevertheless, the semi-random approach still suffers from the probability of insertion failure. Insertion failure occurs when a particular entry doesn't get accommodated into the hash table even after rearranging the stored entries. Thus, the load factor of Cuckoo hash engine reduces. The probability rate of insertion failure is highly dependent on the nature of and the size of the input key data set. For large-scale data set, the rate of collisions will be high, and the insertion failure rate is also amplified. Hence, the next advancement to the Cuckoo hash engine is to further heighten the memory utilization or load factor of the model and make it compatible with large key-value data sets.

In the pseudo-random Cuckoo-hash mechanism, the data_insert operation will allow a maximum of n kick-outs, to store a given entry among one of its n candidate locations. The number of possible chances to fit back all the rejected elements into the hash tables is n i.e. number of sub_tables. Most of the current conventional Cuckoo hash algorithms adapt this rule [43] [44]. For a small set of data in the range of 1-5k elements, this rule works fine, as only a few elements get mapped to the same location. However, for a large set of elements, the CRC32 hash function generates same n locations for multiple entries, resulting in more collisions and thus more insertion failures. A new term called `RUN_COUNT` is introduced, which decides the number of iterations that should take place before declaring an insertion failure. Since the `RUN_COUNT` value doesn't depend on the number of sub-tables, the designer is free to fix this threshold value. Due to which it breaks the internal dependency present between the number of kick-outs and sub-tables. `RUN_COUNT` gets injected into the `CH_MAIN` module. This parameter determines various aspects of the design like the total number of memory access, the total number of kick-outs, load factor and time required to

store a particular key-value pair. Hence, all of these factors must be taken into consideration to figure out a suitable value for `RUN_COUNT`. If the `RUN_COUNT` value is set far higher than n, then load factor of around 99% can be achieved. However, the total number of writes made to the `UPDATE_IN` channel rises, drastically increasing the number of memory access needed to store one entry. On the contrary, if it is set very close to n, the improvements achieved will be negligible, which in turn reduces the load factor of the Cuckoo hash engine. Hence it should be fixed to an optimal value, to gain notable improvement without suffering from a high data insertion latency. The optimal value of `RUN_COUNT` is specific to input key data set; thus it cannot be generalized. This chapter explained the critical issues that limit the efficiency of the Cuckoo hash engine. To overcome these challenges, additional features are added to the basic Cuckoo hash engine, making it resilient to handle data collisions. Using the above described methodology, a load factor of 99% is accomplished, which indicates a very high memory utilization factor. Thus, a robust Cuckoo hash engine is developed, which can handle collisions efficiently by minimizing the number of kick-outs happening to store a given key-value pair. The next Section elaborates an application designed using this robust Cuckoo hash engine.

3.2 High throughput IP Lookup

3.2.1 System Overview

The vital role of a network layer in an Internet protocol suite is to forward packets based on routing table lookup. The exponential growth in the size complexity of IP tables and the high throughput requirements challenges the design of a highly efficient IP lookup machine. IP lookup in high-speed routers must scale with the expanding routing table sizes along with high lookup rates. This chapter presents an IP lookup engine, capable of handling fast lookups that maintain high throughput and also low memory utilization.

In routers, the flow of packets is decided based on pre-existing patterns known as *prefixes*. For IPV4 network layer lookup, a prefix rule is a 32-bit binary array comprised of one's and don't cares. Prefix rules are 32-bit wide, because IP lookup operates only on destination address field of the incoming packet (destination field in IPV4 packet is a 32-bit address). Prefix length of a particular rule can vary from 32-bit to single-bit, depending on the number of defined bits present in the binary array i.e., string of 1's. Routers store prefix rule information in the forwarding table known as *Forwarding Information Base* (FIB). Each prefix rule is associated with a value known as next-hop value, that indicates where to route a packet, if the destination field of an incoming packet matches this specific rule. To make a routing decision, the router compares the destination field address of an incoming packet against all the prefixes stored in FIB database. For a given Layer 3 packet, destination address can match multiple prefix rules of different prefix lengths. The destination field of a packet cannot match two prefix rules of same prefix length. Using longest prefix match and exact match techniques, a single rule is selected amongst the matched rules. This method of routing is called classless inter-domain routing.

The forwarding table shown in Table 3.1 is used to explain the LPM based routing mechanism. This forwarding table comprises of five prefix rules with a maximum width of five bits that is assumed to have next-hop values, i.e., H1, H2, H3, H4 and H5 respectively. The prefix length of R1 is two as only two bits out of five bits is defined, i.e., the prefix length is equal to the difference between the maximum rule width and the number of don't care bits. Accordingly, prefix length of R2, R3, R4, R5 becomes 3,4,5,4 respectively. Consider a packet, with a five-bit destination address field 10101. In the LPM approach, this value is compared against all the prefixes present in the FIB table. In the current example, the routing algorithm compares the destination address and the five prefix rules. In the fisrt step, rule R1 is compared with the destination address. The prefix length of a rule decides the number of the address bits that can be compared with that particular rule, to declare it as a rule match. Since, prefix length of R1 is two, the most significant two-bits of the destination address, i.e., the comparison of bit values 10 with R1 is successful. This success

Table 3.1: FIB table consisting of five prefix rules

	Prefix	Next_hop
R1	**10*****	**H1**
R2	**111****	**H2**
R3	**1010***	**H3**
R4	**10101**	**H4**
R5	**1101***	**H5**

implies that destination address like 10111, 10110, 10001, etc. matches rule R1 as the first two-bits match R1. In a similar manner, the destination address comparison is made for the remaining four rules, resulting in matching R3 & R4. Thus, given destination address 10101 matches three prefix rules R1, R3, and R4. The LPM approach is used to pick one rule amongst these three rules, to route the packet. LPM uses the concept of matching the highest number of address bits on prefix rule that is useful for routing. According to this idea, amongst the rules matched, a rule with highest prefix length value should be selected, since many address bits participate in the comparison. Hence, for destination address 10101, rule R4 (prefix length five) is selected, and next-hop value H4 associated with it is used to route the packet.

Most of the routers in classless inter-domain routing uses TCAMs to perform LPM and exact matches. Ternary Content Addressable Memory is a more flexible type of CAM, that is capable of storing three kinds of states in each TCAM cell, namely '0','1' and 'x'[9]. This makes them compatible to store and query prefix rules, as prefixes contain don't care bits. TCAM is chosen in high-speed routers because they can perform LPM matches in one memory clock cycle irrespective of FIB table size. However, each TCAM memory cell requires double the number of transistors compared to SRAM, and are power hungry and costly. Hence, hash-based IP lookup engines gained more significance, which eliminates the need for TCAM memory.

However, the hash-based approach has some glitches. First, hash-based IP lookup uses RAM memory to store FIB tables, which consumes much less power than TCAMs. Nevertheless, RAM memory doesn't have the capability to manipulate don't care bits, making the lookup mechanism difficult. Second, in TCAMs, prefixes are stored in a descending order of prefix lengths in a single memory. Hence this avoids the need of any collision resolving technique. On the contrary, in hash-based IP lookup, prefixes can be cached into FIB tables in any discipline. Thus multiple rules will be mapped to same location resulting in collisions. Hence, the hashing technique chosen to perform IP lookup must be efficient in resolving these collisions without creating insertion failures. Hence, an IP lookup engine based on cuckoo hash mechanism is proposed in this Chapter. In the previous chapter, an algorithm for a robust Cuckoo-hash engine was developed, which can achieve a load factor of 99%. This chapter presents how this Cuckoo-hash engine is used to build high throughput IP lookup algorithm. This approach guarantees, constant worst-case lookup time and minimum number of memory access to route a packet.

The proposed Layer 3 IPV4 lookup engine is implemented as a simulation-based hardware/software model. For the rest of the discussion in this chapter, the software part of the model is called as IP_MAIN and hardware part of the model is called as `IP_FORWARD`. `IP_MAIN` and `IP_FORWARD` communicate with each other through four different streams of data, such as `st_phv_in`, `masks_in`, `st_update_in`, and `st_phv_out`. Channels `st_update_in` and `masks_in` are used to share prefix rule information between `IP_MAIN` and `IP_FORWARD`. IPV4 packet routing is done using `st_phv_in` and `st_phv_out` channel.

The IP lookup engine consists of two main functionalities - (1) set up FIB tables with predefined prefix rules (FIB_SETUP algorithm), (2) perform packet routing by comparing destination field of the packet with all prefixes stored in the database (lookup algorithm). To accommodate the high-speed packet routing, IP lookup algorithm employs two powerful engines namely **`micro_CHE`** and **`micro_PL`**. The `micro_CHE` engine is an extension of `CH_MAIN` (Section 3.1), which is encompassed in `IP_MAIN` to perform the first func-

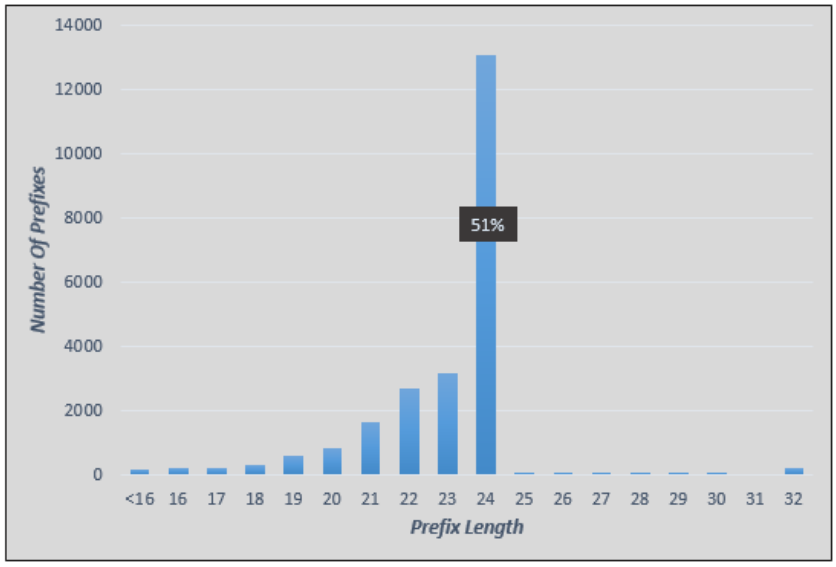


Figure 3.5: Prefix length distribution in routing table

tionality. `micro_PL` engine is an enhancement of `CH_FORWARD` (Section 3.1), contained in `IP_FORWARD` to achieve the second functionality. Henceforth, it can be concluded that `micro_CHE` is a software module and `micro_PL` is implemented in hardware. The bottleneck for any IP lookup algorithm is the number of memory accesses and look up time required to route a single packet. These performance metrics can be enhanced by storing prefix rules of different lengths in different FIB tables. Hence each prefix function (prefix length) is associated with one `micro_CHE` and `micro_PL` engines, to set up and search in the respective FIB tables. Figure 3.5, shows for IPV4 Layer 3 lookup, the length of prefix rules is dominant in the range 16 to 32. Therefore, this design considers 17 micro engines associated with each prefix length from 16 to 32.

The `micro_CHE` engine participates only in the FIB_SETUP algorithm, whereas `micro_PL` engine participates both in FIB_SETUP and lookup algorithm, that is explained in the further Sections. In the FIB_SETUP algorithm, the `micro_PL` engine is invoked to do a memory write into the FIB RAM tables. Whereas, in lookup algorithm, the `micro_PL` engine is called to search if a given destination address matches any of the rules stored in hardware FIB tables. Hence, `micro_PL` engine is encompassed with 5 input parameters -

`store_bit, forward_data, mask, lookup_bit, dest_addr`. These input parameters, notify `micro_PL` engine which action to take. If `store_bit` is set, `micro_PL` engines carry out the FIB_SETUP algorithm, if `lookup_bit` is one, it participates in lookup mechanism. The comprehensive functionality of these parameters is explained in the further Sections.

3.2.2 FIB_SETUP Algorithm

This Section deals with the mechanism of how the FIB tables that contain different prefix functions are populated with their respective prefix rules. As mentioned, each prefix length rules are associated with a different set of FIB tables. The design consists of 17 different sets of FIB sub-tables, since prefix functions from 16 to 32 are considered for packet forwarding. Hence, the rules of bit-length 16 are stored in `FIB_16`, the rules of bit-length 17 are stored in `FIB_17` respectively. Many IP lookup engines, use single FIB table with exalted depth to accrue all prefixes, which is not suitable for collision resolving. Thus, the setup algorithm divides FIB table into granular optimal tables. For example, prefix_16 have four `FIB_16` sub-tables of each 256 locations, rather than having a single 1K `FIB_16` table. However, the number and the depth of sub-tables associated with each prefix function is parameterizable and are not interdependent. For example, prefix_24 can have four `FIB_24` tables, whereas prefix_32 can have only one `FIB_32` table. The designer can select how many tables and depth for each table depending on the prefix rule density. For simplicity, algorithm allocates four FIB sub-tables per prefix function.

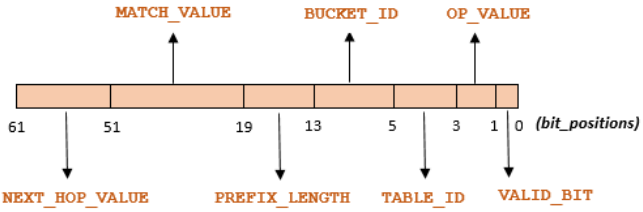
The FIB_SETUP algorithm consists of `IP_MAIN` software module and `IP_FORWARD` hardware module. This algorithm has three primary objectives, (1) read the input prefix rule and extract prefix length of the rule using `IP_MAIN`, (2) invoke particular `micro_CHE` engine (Cuckoo-hash engine) to compute `table_id` and `bucket_id` for a given prefix rule, and (3) consign this computed information to `IP_FORWARD` to acquit actual memory write using respective `micro_PL` engine. The `IP_MAIN` comprises of 17 `micro_CHE`

engines, and the IP_FORWARD consists of 17 micro_PL engines. Analogous to Section 3.2.2, micro_CHE and micro_PL maintains a semi-identical copy of FIB tables using the st_update_in channel. The FIB tables of micro_CHE engine and of micro_PL engine are named as SW_FIB and HW_FIB tables respectively.

The software module IP_MAIN reads the input prefix rule. Prefix rule consists of three fields namely prefix_value (match_value), prefix length and next-hop address. An assumption is made that all micro_CHE engines (Cuckoo-hash engine) would maintain four SW_FIB sub-tables. Hence, a given rule has four candidate locations, where it can be stored. Four candidate locations are calculated using the CRC32 function. Among 128-bit input to the CRC32 function, the least significant 32-bits are given by the sum of match_value and prefix length while the remaining bits are set to zero. SW_FIB table consists of - 32-bit match_value, 6-bit prefix length, 32-bit mask, 128-bit key and a 10-bit next-hop value. For a given rule, 32-bit prefix_mask is generated using prefix length data. If prefix length is 'y', the most significant y bits of the 32-bit mask is set to 1, and remaining ($32 - $'$y$') bits are set to 0. This mask is used to extract the y significant bits of match_value. Each location of SW_FIB sub-table store all this information along with RULE_VALID bit, to indicate the reliability of the prefix rule. Prefix length decides which micro_CHE engine will be accessed. If the prefix length of a given rule is y bits, micro_CHE_y is invoked to store the rule in FIB_Y sub-table. As mentioned previously, each micro_Cuckoo hash engine comprises four SW_FIB sub-tables, hence rule p_y can be stored in any of these four locations given by the CRC32 function. The table_ids for these four locations is 1,2,3,4 and bucket_ids are given by equations (3.1) to (3.4). The micro_CHE engine checks for an empty slot to store a given rule p_y, if there is no empty slot available it rearranges the stored prefix rules across the tables as explained in Section 3.2, by resolving collisions and making space for new data. The micro_CHE engine also performs rule_delete and rule_update operations. In both these operations, it searches for the availability of given rule among the four SW_FIB sub-tables, if found micro_CHE engine updates the next-hop value or deletes the rule respectively. Once the location information is gathered using

```
typedef struct {
                ap uint<1> bit flag;
                ap uint<2> op;
                table addr table id;
                bucket addr bucket id;
                mask length match value;
                pre length prefix length;
                next hop addr length next hop;
                bool done;
}cuckoo data t;
```

(a)



(b)

Figure 3.6: (a) Data structure written into `st_update_in` channel by `IP_MAIN`, (b) Data flattening done in `IP_FORWARD`

`micro_CHE` engines, `IP_MAIN` creates data frame to send to the hardware module (shown is Figure 3.6(a)), i.e., `IP_FORWARD` using the `st_update_in` channel, which writes into FIB RAM table in the memory.

The `IP_FORWARD` module monitors the `st_update_in` channel continuously using the command *`((!st_update_in.empty())`*. If the stream is empty, no action is taken. As soon as a prefix rule arrives, the hardware module flattens the data frame and extracts all the relevant information as shown in Figure 3.6(b). The next step is to send this information to the `micro_PL` engine, and store the prefix rule in respective `HW_FIB` sub-table. As mentioned earlier, the `micro_PL` engine has five input parameters. Since `micro_PL` engines are invoked to make a memory write, the `store_bit` input is set, and the `lookup_bit` input is reset. The rule_data input is also set to forward_data. Hence, the flattened data frame, i.e., forward_data is sent to all the 17 `micro_PL` engines. Each `micro_PL` engine extracts the prefix length information of the rule from the received rule_data. Then, it com-

pares this prefix length value with associated prefix length of the engine. If both the prefix length value does not match, `micro_PL` engine discards the rule_data input. If the prefix length field matches, the `micro_PL` engine reads the rule_data input and stores the match value and the next-hop value in the location `HW_FIB[table_id][bucket_id]`. Also, the `RULE_VALID` bit of this location will be updated to 1. Therefore, all three functionalities of the FIB_SETUP algorithm is completed successfully.

Figure 3.7 represents the complete flow FIB_SETUP algorithm. Given input rule `PREFIX_24` has a prefix length of 24. In the first stage, `IP_MAIN` extracts this prefix length information. In the second stage, since prefix length is 24, `micro_CHE24` is accessed to determine the location to store this specific rule. The highlighted locations within the `SW_FIB24` table, indicates the candidate locations, where this rule can be accommodated. In this example, `micro_CHE24` engine assumes location three is available and hence store rule `PREFIX_24` in `SW_FIB[table_id_3][bucket_id_3]`. This information is wrapped into a data frame and sent to `IP_FORWARD`. `IP_FORWARD` broadcasts this data frame to all the 17 `micro_PL` engines. However, the prefix length of the input rule, matches only the prefix length associated with `micro_PL24` engine. Thus, `micro_PL24` engine carry out the memory write operation by storing input rule `PREFIX_24` in location `HW_FIB24[table_id_3][bucket_id_3]`.

3.2.3 LookUp Algorithm

For an incoming packet, the router has to determine which route to select, so that packet reaches the correct end network. For this purpose, the router has to check whether the destination address of an incoming packet matches any of the prefix rules that are present in FIB sub-tables. If it matches a single rule, exact match technique is performed, and the router selects a route/output-port based on the next-hop value associated with that specific prefix rule. If the destination address of an incoming packet matches multiple rules, then LPM technique is used to select a prefix rule, and the packet is forwarded. The bottleneck

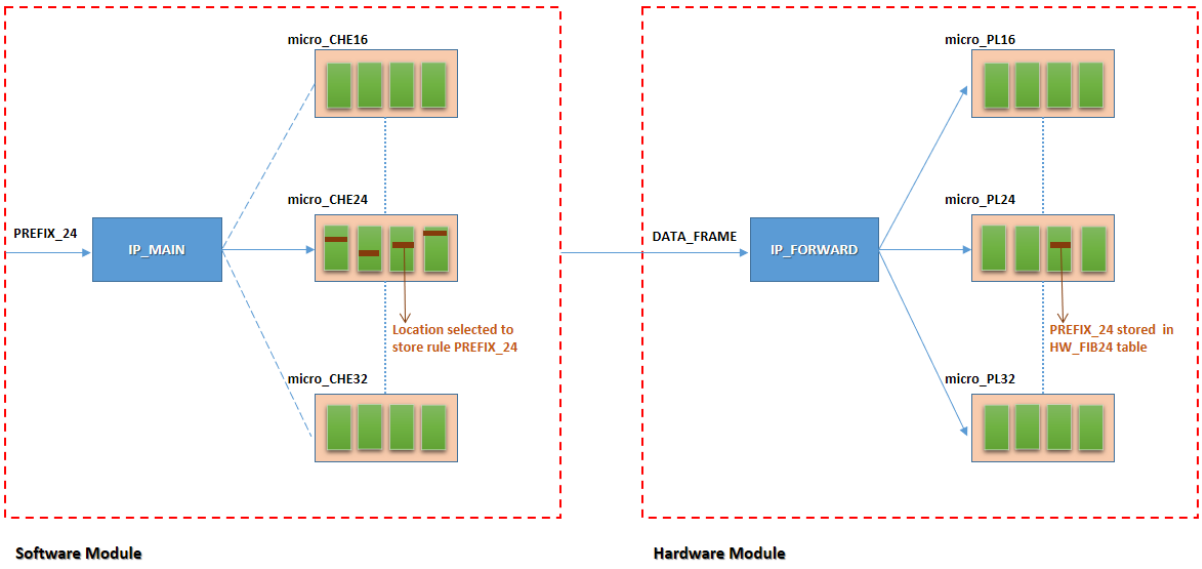


Figure 3.7: Different stages of FIB_SETUP algorithm

in IP lookup engine is the number of memory accesses and the worst-case time required to perform the LPM techniques. This section describes a hardware based lookup algorithm, which can perform LPM match with a minimum number of memory accesses along with maintaining constant worst-case lookup time, irrespective of the size of FIB tables.

Lookup algorithm is carried out by the `IP_MAIN` and the `IP_FORWARD` modules. As discussed in the previous Section, the FIB_SETUP algorithm populates all the 17 sets of `HW_FIB` tables associated with each of the prefix function. IP lookup operation is performed using these populated hardware FIB tables. The input to this lookup algorithm is a set of IPV4 packets. `IP_MAIN` reads the incoming packet and writes the packet into the `st_phv_in` channel. The software module `IP_MAIN` just acts as a buffer to transfer the packets to `IP_FORWARD`. All the computations for this algorithm are done in the hardware module.

The goals of `IP_FORWARD` module are - (1) to find the prefix rules which matches the destination address of an incoming packet using `micro_PL` engines, (2) perform longest prefix match (LPM) on the matched rules to select one prefix rule, and (3) update the

port field of the packet with next-hop value and send back the packet to `IP_MAIN` using `st_phv_out` channel. `IP_FORWARD` monitors the `st_phv_in` channel continuously using command *`((!st_phv_in.empty())`*. If the input packet stream is empty, no further action is taken. Once the stream is populated with packets, the `IP_FORWARD` reads the incoming packets. For IP lookup routing, the only requirement is the destination address of the packet, hence the `IP_FORWARD` extracts the destination address of the incoming packet. This destination address can match any prefix rule present in 17 sets of `HW_FIB` tables. So, the hardware module has to check all the `HW_FIB` sub-tables, to find an accurate prefix rule. Some algorithms incorporate pipelined checking mechanism, where they verify if the destination address matches rules present in `prefix_32` if a match is found it fetches the next-hop information and routes the packet. If a match is not found, it searches for rules present in `prefix_31`, and the process repeats itself. Although this method performs less number of memory reads, it fails to maintain constant lookup time, which affects the line rate of data processing very badly. Hence, this design incorporates a parallel lookup mechanism, where candidate rules (rules that may match the destination address, given by crc32 functions) from all the prefix functions are read in one clock cycle.

The `IP_FORWARD` reads the destination address of the incoming packet through channel `st_phv_in`, and it forwards this data to all the 17 `micro_PL` engines (prefix function16 to `prefix_function32`), by setting `lookup_bit` input of all the engines to one. This tells the `micro_PL` engine that the destination address received is valid and to perform a lookup operation on this destination address. All these `micro_PL` engines return next-hop values to `IP_FORWARD` if there is a match found between the rule in the particular prefix function and the destination address.

Each prefix function (`micro_PL` engine) consists of three main stages to locate a rule matching incoming destination address, namely - Key_generator unit, Memory_data Fetch unit and Rule_checker unit as shown in Figure 3.8. For example, consider `micro_PL24` engine. As stated earlier, `micro_PL24` has four `HW_FIB24` sub-tables associated with it. Hence, destination address will have four candidate prefix rules (one rule per sub-table) to

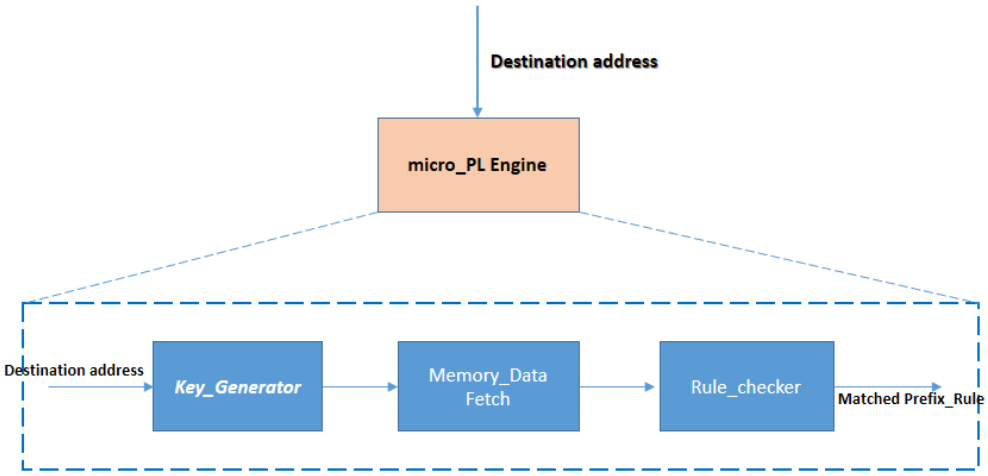


Figure 3.8: Internal structure of `micro_PL` engine used to perform Lookup algorithm

verify. The computation is done using three components shown in Figure 3.8.

Key_generator block generates unique `table_ids` and `bucket_ids` for four candidate locations. `Table_ids` for four `HW_FIB24` sub-tables are 1,2,3,4 respectively. `Bucket_ids` for four candidate locations are generated using CRC32 function. Input to CRC32 hash function is a 128-bit number. Input to crc32 is given using the following equations -

$$\text{key} = (\text{dest\_addr} \wedge \text{mask}_{24}) + 24 \tag{3.5}$$

$$\text{crc32\_input}(31, 0) = \text{key} \tag{3.6}$$

$$\text{crc32\_input}(127, 32) = 0 \tag{3.7}$$

Equation 3.6 indicates, only least significant 32 bits of 128-bit key_input are used to generate `bucket_ids`. This reduces the RAM space required by LUTs used by CRC32 function. The mask value associated with prefix_24 is 0xffffff00. Equation 3.5 indicates, the destination address is bitwise-anded with mask before giving to crc_input, this is carried out to extract most significant 24 bits. This is required because, all the rules present in the `HW_FIB24` is 24 active bit prefixes. Implying the rule matches a particular destination, if

most significant 24 bit matches, remaining eight bits may match or may not match. The output of crc32 is used to generate four `bucket_ids` using equations 3.1 to 3.4.

Next unit of `prefix_func24` fetch all the rules present in these four candidate locations. Rules fetched from four locations of `HW_FIB24`, contain following data - match_value, next-hop, rule_valid. First the `RULE_VALID` bits of all four locations is checked, if it's not set the prefix rule present in it is discarded. If the `RULE_VALID` bit is set to one, the rule present in the location is fetched for further computations. The destination address should be matched against all four rules, if the valid bit of all four locations is set. This is given by equation 3.8.

$$\text{table\_array\_i} = (\text{dest\_addr} \wedge \text{mask\_24} == \text{match\_value\_i}) \tag{3.8}$$

`table_array` is a four-bit array; case statement is used on `table_array` to extract the next-hop address associated with the rule matched. `micro_PL24` returns `rule_match` bit along with this next-hop address. `Rule_match` bit is set to one, if the destination address matches any one of the four rules present in `HW_FIB24`. In a similar manner, all other 16 micro_PL engines compute the keys, compare the destination address with rules stored and send back a `rule_match` bit and next-hop value to `IP_FORWARD`.

Thus, `IP_FORWARD` receives 17 next-hop values associated with each prefix function. Now LPM mechanism is used to select a prefix rule, if more than one prefix rule have matched the destination address. Priority resolver is used to implement LPM technique. `IP_FORWARD` first analyzes the data from `prefix_32` (`micro_PL32`) function, if the `rule_match` bit is set, the next-hop value associated with it is selected and sent to `IP_MAIN`. If the `rule_match` bit of 32 is not set, it implies the destination address did not match any of 32-bit prefix rules. Hence, next the data from `prefix_function31` is analyzed and the process continues until a `rule_match` bit is set. If the destination address did not match any of the prefix function rules, `IP_FORWARD` enables packet_drop field and sends it back to `IP_MAIN`. Figure 3.9 represents the complete flow of lookup algorithm.

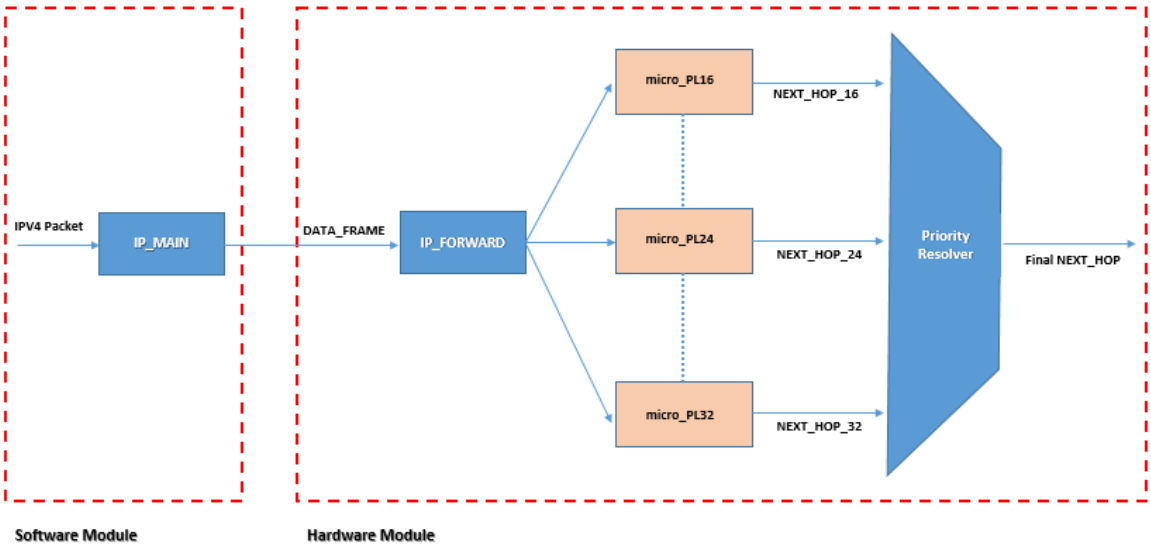


Figure 3.9: Complete Flow of Lookup algorithm

3.2.4 The Incremental Updates

An important issue in the IP forwarding engine is the incremental updates of the prefix rule database. The number of prefixes included in a routing table grows with time. There are three common types of FIB table updates: rule insertion, rule deletion, and rule modification. Rule modification and rule deletion is straightforward. `IP_MAIN` accepts the incremental rule updates. First, the prefix length of the incoming rule is extracted, and the respective micro_CHE engine is invoked. Let's assume that the incoming rule is a 24-bit prefix. For rule deletion, `micro_CHE_24` searches for the existence of a given prefix rule in `SW_FIB_24`, if found the `RULE_VALID` bit associated with the rule is set to 0 and data_frame is formed to send to `IP_FORWARD`. Similarly, in rule_update of prefix_24, the `micro_CHE_24` engine is called, if the rule is found, the next-hop value of the prefix rule is updated with the fetched value, and data frame is sent to `IP_FORWARD`. If the prefix rule is not found, it is considered as rule_insert operation. In both of these operations, the FIB table structure remains intact.

On the contrary, rule insert operation gets a bit tricky as it has to handle the collisions and change the framework of FIB sub-tables. If the IP lookup engines store all the prefix rules

is a single FIB table with prioritizing them, then rule_insert operation gets very complicated as it has to modify the entire structure of FIB table to insert just one rule. Henceforth, this design considered granular FIB sub-tables to store prefixes. The number of modifications done to the FIB_TABLE is dependent on the number of kick outs that take place to store an input prefix rule. As explained in Section 3.1, the desgined Cuckoo_hash algorithm minimizes the number of kick outs by increasing load factor to 99%. Hence, when the prefix_24 rule is to be inserted, `micro_CHE24` is probed, and it searches for an empty position in `SW_FIB_24` based on hash locations generated by CRC32 if found it stores the rule and updates the `RULE_VALID` bit to one. If an empty location is not found then, the collision resolving takes place to make space for a new rule. All the changes made to `SW_FIB_24` tables are sent to `IP_FORWARD` to make the actual writes to RAM tables `HW_FIB_24`. `micro_PL24` does the memory write to update the `HW_FIB_24` table.

One of the important concerns with the incremental updates on the prefix rules is, how to route the packets when there is a continuous change in the FIB table data. The number of updates made to the FIB tables is few hundred updates per second. If the router did not consider the updates correctly, then it results in incorrect routing that can lead to adverse effects.

As discussed in the previous Section, `micro_PL` engine updates the `HW_FIB` tables (FIB_SETUP algorithm) and also later use these populated `HW_FIB` tables to perform packet routing (lookup algorithm). Problem arises, when setup algorithm is sending prefix rule updates to `micro_PL` engine, which modifies the `HW_FIB` table data and parallely lookup algorithm is also sending packets to micro_PL engine. This implies, both FIB_SETUP and FIB_LOOKUP algorithms need to access `HW_FIB` sub-tables, because both channels `st_update_in` and `st_phv_in` have valid data. This scenario is further explained with the help of an example. Assume prefix rule (255.240.60.7/24, 100) is stored in `HW_FIB24[t1][b1]`. Now, `IP_MAIN` sends a packet with destination address 255.240.60.25 (which matches rule present in `HW_FIB24[t1][b1])` to `IP_FORWARD`, at the same time it also sends a rule_update (255.240.60.7/24, 200). `IP_FORWARD` re-

ceives these data_frames, and transfer this information to `micro_PL engines` by setting `store_bit` and `lookup_bit` to one. Since, prefix length is 24 and the `store_bit is set`, `micro_PL24` engine is responsible to perform a rule modification operation by changing the next-hop value of the rule present in location `HW_FIB24[t1][b1]`, from 100 to 200. In the meantime, `micro_PL24` has also received valid destination address, which has to be compared against the rules present in `HW_FIB24`. If it performs both the operations in parallel without any other logic, destination address will be matched to rule 255.240.60.7, before the next-hop value is updated to 200, lookup fetches 100 as next-hop and sends the value to `IP_FORWARD`, which results in incorrect routing. If it waits for the rule_update operation to finish, one clock cycle delay will be there for lookup algorithm.

Few IP Lookup techniques, wait for the updates to finish, halt the packet forwarding, continue the packet forwarding only after updates are finished on FIB tables, but the line rate is suffered a lot due to high rate of updates happening every second. This method is not recommended because, it has to stop packet routing multiple times, resulting in high lookup delays. Some other techniques, maintain two copies of FIB tables to handle routing updates. Thus increasing the memory requirement.

In this design, a parallel co-existing technique is incorporated to simultaneously handle rule updates and also facilitate packet routing at the same time. As mentioned previously, `micro_PL` engine has two major operations - (1) perform hardware write into `HW_FIB` sub-tables (`store-bit` is set), and (2) compare the destination address with the stored prefix rules (`lookup_bit` is set). Both of these operations need to access HW_FIB sub-tables to complete the process. Routing error occurs, when both these operations wish to access `HW_FIB` sub-tables simultaneously. Thus, extra functionality is added to `micro_PL engines`, to facilitate correct packet routing, while carrying out rule_store and rule_comparison operations concurrently.

Figure 3.10 indicates the mechanism of `micro_pl` engine. In the first stage, path-1 and path-2 shown in Figure 3.10, conduct the computations parallely to output next-hop

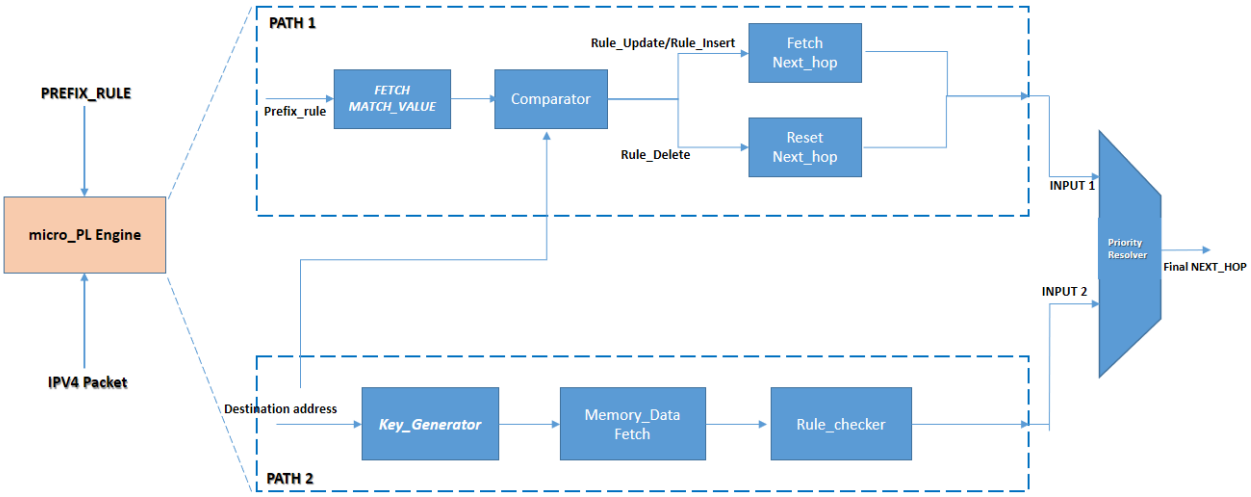


Figure 3.10: Structure of `micro_PL` engine to handle parallel store and lookup mechanisms

value. In the second stage, priority selector selects the input based on `store_bit` and `lookup_bit` value. When, `store_bit` is zero and `lookup_bit` is one, this implies `micro_PL` engine should carry out conventional packet routing mechanism (`HW_FIB` table structure is not modified), thus priority resolver selects Input-2 shown in Figure 3.10 (path-1 is not accessed). When both, `store_bit` and `lookup_bit` is high, this implies `micro_PL` engine has valid rule_data and destination address input. Hence the destination address should be compared with the incoming prefix rule data. Path-2, extracts the match_value from input rule_data and compares it with the destination address. If the destination address matches the prefix value, rule_update bit is made one and priority selector selects the Input-1, otherwise it selects Input-2 (the destination address doesn't match the incoming rule). Path-1 sets the value of Input-1 to next-hop of the rule_data, if the operation value encompassed in the rule_data is either rule_update or rule_insert. Input-1 value is set to zero, if the operation value of the rule_data is rule_delete. `micro_PL` engine sends this next-hop value to `IP_FORWARD`. In the meantime, `micro_PL` engine would have also stored the input prefix rule in `HW_FIB[table_id][bucket_id]` (`table_id` and `bucket_id` is given by rule_data input). Thus, this mechanism guarantees parallel store and lookup operation, without any clock cycle delay.

Chapter 4

Results and Discussion

This chapter summarizes the key contributions of the proposed hash engine by comprehensively analyzing various routing table data. As mentioned in the previous sections, the IP lookup engine must be capable of achieving a high line rate regardless of the exponential growth of the Internet traffic. The proposed micro-Cuckoo hash-based IP lookup is capable of delivering a high data line rate by forwarding over 280 million packets per second; also it alleviates the space complexity requirements needed to store the FIB tables. Additionally, the micro-Cuckoo hash engine supports concurrent complex operations like prefix rule update and next-hop data fetch. Further, the lookup engine (`micro_CHE`) is also capable of maintaining constant worst-case query time irrespective of the size complexity of the FIB tables.

In order to obtain accurate estimates of the performance metrics for the micro-cuckoo hash engine, real-time routing table entries are considered for packet processing. The CAIDA (Center for Applied Internet Data Analysis) organization maintains a database for routing entries of both IPV4 and IPV6 autonomous systems [45]. These data entries are collected from five different Internet backbone routers, namely: as286 (KPN Internet Backbone), as513 (CERN, European Organization for Nuclear Research), as1103 (SURFnet, the Netherlands), as4608 (Asia Pacific Network Information Center), and as4777 (Asia Pacific

Table 4.1: Memory requirement for five different data sets of prefixes

	Total number of prefix rules	Memory Required (kB)
Dataset 1	23081	277
Dataset 2	25000	300
Dataset 3	25500	306
Dataset 4	25300	304
Dataset 5	26400	317

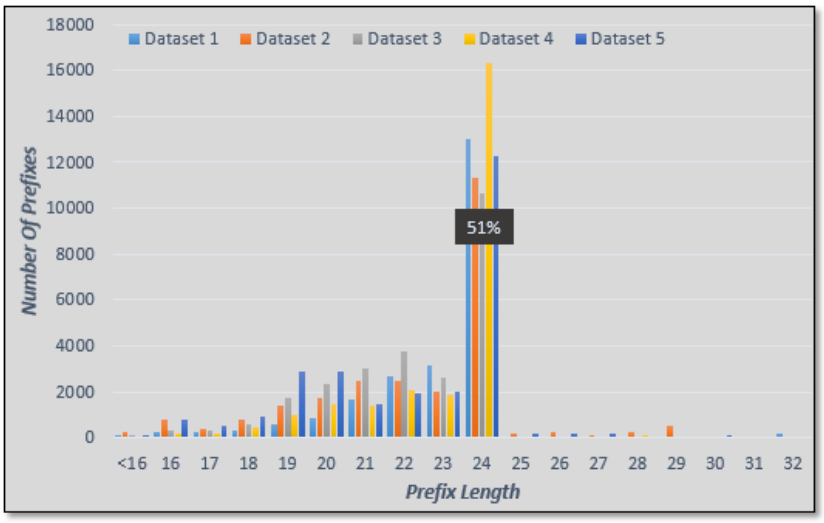


Figure 4.1: Prefix density graph of five data sets

Network Information Center). In this section, five different data sets of prefix rules are examined to evaluate the efficiency of the proposed algorithm. Table 4.1 shows five data sets considered for the analysis. The first column of the table indicates the number of routing entries present in each data set; each routing entry consists of three different data fields: prefix value, prefix length and next-hop information. The next-hop value associated with each prefix rule decides the autonomous network, to which router forwards the incoming packet. The second column in the Table 4.1 indicates the memory required to store each of these five data sets. Thus, on average to store 25k prefix rules; the hash engine must support a FIB table of 300kB (kilobytes) memory.

Table 4.2: Table allocation for prefix functions

Prefix Function	Number of Sub tables	Locations per Sub table
micro_PL16	4	256
micro_PL17	4	128
micro_PL18	4	256
micro_PL19	3	1024
micro_PL20	3	1024
micro_PL21	3	1024
micro_PL22	4	1024
micro_PL23	4	1024
micro_PL24	4	4096
micro_PL25	3	64
micro_PL26	4	64
micro_PL27	3	64
micro_PL28	4	64
micro_PL29	4	128
micro_PL30	3	32
micro_PL31	1	4
micro_PL32	3	64

The graph shown in Figure 4.1 outlines the number of prefix rules present in five data sets pertaining to each prefix length. Two interesting observations can be obtained from the graph - (1) 98% of the rules have prefix lengths between 16 and 24 bits, and (2) 24-bit prefixes comprise about 51% of the routing table entries. To achieve high-speed packet forwarding, the design of IP lookup engine must review these anomalies present in the distribution of prefix rules. In Section 3.2, the detailed architecture of the `micro_CHE` and `micro_PL` engines is explained, which highlights the fact that `micro_PL` engine stores the actual FIB table. In this design, 17 `micro_PL` engines are deployed to store prefix rules of length 16 to 32, to favor the first remark. In all five datasets, the number of rules having a prefix length between 1 and 15 is trivial, in the range of 100 to 200. Henceforth, these prefixes are not examined while forwarding the packets. The second remark is taken care by the granular tables assigned to each prefix function (`micro_PL` engine). Table 4.2 provides a detailed insight into the FIB table organization corresponding to each prefix function.

In this design, the number of FIB sub-tables and buckets present per sub-table is made parameterizable to facilitate the efficient utilization of the memory. For example, the `micro_PL24` engine is designated with four sub-tables, where each sub-table can store up

to 4k prefix rules. On the contrary, the engine `micro_PL30` contains only three sub-tables; while individual sub-table can store only 32 entries. The memory is allocated to prefix functions, based on the analysis of the density of the prefix rules associated with each prefix length. This implies density graph shown in Figure 4.1 act as the primary factor in making memory allocation decisions. Each `micro_PL` engine encompasses a CRC32 hash function to calculate the bucket_id values (Section 3.1.2). There are four different ways available to implement a 32-bit CRC function - (1) Modulo-2 Binary Division, (2) bit by bit processing, (3) byte by byte processing, and (4) lookup table-based. Among the four techniques, LUT based CRC32 provides faster results when implemented on hardware. Generally, the LUT-based scheme uses the slicing-by-16 algorithm, which generates 16 individual LUTS. All the lookup tables contain 256 entries, where a single entry is 32-bits long. The input to the CRC32 hash function is a 128-bit number. This 128-bit number is divided into chunks of 8-bits, which is further used to as an address bus to pick a particular 32-bit number from each lookup table. The sixteen 4-byte numbers obtained are XORed, to produce a final 32-bit output.

The CRC32 hash function was designed and synthesized in Vivado HLS tool, to estimate its memory requirements. Figure 4.2 shows the memory needed by the CRC32 function. As mentioned, each prefix function comprises one CRC32 hash engine; henceforth, this original CRC32 alone will consume 680 BRAMs. Thus, it is necessary to alleviate this space complexity. For this purpose, a reduced CRC function is designed that operates on only 4 LUTs. This enhanced version reduces the memory utilization factor by 75%, which is shown in Figure 4.3. One CRC function requires only 10 BRAMs, ergo the entire lookup system (17 prefix functions) utilizes only 170 BRAMs for CRC functions.

The memory space required to implement the entire micro-Cuckoo hash IP lookup engine is depicted in Figure 4.4. As discussed in the previous chapter, the hardware module of the system is comprised of `micro_PL` engines; further, each `micro_PL` engine is associated with one CRC32 hash function. Hence, both `micro_PL` engines and CRC32 hash functions are synthesized by the Vivado HLS tool. The proposed parallel IP lookup architecture

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	612
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	40	-	0	0
Multiplexer	-	-	-	1
Register	-	-	99	-
Total	40	0	99	613
Available	2160	2760	663360	331680
Utilization (%)	1	0	~0	~0

Figure 4.2: Memory utilized by original CRC32 hash function

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	180
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	10	-	0	0
Multiplexer	-	-	-	1
Register	-	-	2	-
Total	10	0	2	181
Available	2160	2760	663360	331680
Utilization (%)	~0	0	~0	~0

Figure 4.3: Memory utilized by reduced CRC32 hash function

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	381
FIFO	-	-	-	-
Instance	571	-	10404	6451
Memory	16	-	0	0
Multiplexer	-	-	-	1251
Register	-	-	4175	-
Total	587	0	14579	8083
Available	2160	2760	663360	331680
Utilization (%)	27	0	2	2

Figure 4.4: Memory utilization of the entire micro-Cuckoo hash system

utilizes only 27% of the total available memory. The Figure also indicates the total number of LUTs and FFs required to implement the micro-Cuckoo hash system. Further, Figure 4.5 depicts the intricate memory structure of all prefix functions separately. For example, `micro_PL24` engine utilizes 86 BRAMs, whereas `micro_PL30` requires only 28 BRAMs. This dissimilarity is present due to the variation in the FIB sub-table structures linked to them. Also, it summarizes the amount of FFs and LUTs encompassed in each prefix function. Therefore, with the inclusion of granular FIB sub-table structures and reduced CRC32 function, the space overhead complexity of the system is substantially minimized.

The next step is to validate the performance efficiency of this memory efficient hash-based IP lookup system. The performance proficiency of the system depends on two main aspects - (1) the ability of the hash engine to resolve the data collisions to make space for the new entry, and (2) the time required by the hash engine to conduct LPM (longest prefix match) on the prefix database and select a specific prefix rule for forwarding the incoming packet. As discussed in the previous sections, the cuckoo-hash engine `micro_CHE` is responsible for the store and collision resolve operations, whereas the `micro_PL` engine handles the query operations.

Five different data-sets of prefix rules were given as input to the micro-hash system,

Instance	Module	BRAM_18K	DSP48E	FF	LUT
grp_Forward_prefix_16_fu_1121	Forward_prefix_16	34	0	719	422
grp_Forward_prefix_17_fu_1171	Forward_prefix_17	34	0	718	422
grp_Forward_prefix_18_fu_1096	Forward_prefix_18	34	0	719	422
grp_Forward_prefix_19_fu_1292	Forward_prefix_19	31	0	571	379
grp_Forward_prefix_20_fu_1269	Forward_prefix_20	31	0	571	379
grp_Forward_prefix_21_fu_1246	Forward_prefix_21	31	0	571	379
grp_Forward_prefix_22_fu_1071	Forward_prefix_22	38	0	711	422
grp_Forward_prefix_23_fu_1046	Forward_prefix_23	38	0	711	422
grp_Forward_prefix_24_fu_1021	Forward_prefix_24	86	0	711	422
grp_Forward_prefix_25_fu_1361	Forward_prefix_25	28	0	563	379
grp_Forward_prefix_26_fu_1221	Forward_prefix_26	34	0	717	422
grp_Forward_prefix_27_fu_1338	Forward_prefix_27	28	0	563	379
grp_Forward_prefix_28_fu_1196	Forward_prefix_28	34	0	717	422
grp_Forward_prefix_29_fu_1146	Forward_prefix_29	34	0	718	422
grp_Forward_prefix_30_fu_1384	Forward_prefix_30	28	0	561	379
grp_Forward_prefix_32_fu_1315	Forward_prefix_32	28	0	563	379
Total		16	571	0	10404	6451

Figure 4.5: Summary of memory utilization factor for each micro-hash engine

Table 4.3: Result of insertion data operation carried out by IP lookup engine

	Total number of prefix rules	Number of prefix rules < 16	Number of prefix rules > 16	Number of saved entries	Number of Insertion Failures
Dataset 1	23081	135	22946	22159	787
Dataset 2	25000	252	24749	23990	759
Dataset 3	25500	112	25389	24227	1162
Dataset 4	25300	25	25276	23970	1306
Dataset 5	26400	155	26246	24754	1492

to measure the performance of micro_CHE and micro_PL engines. Table 4.3 outlines the summary of the store operations carried out by all the micro_CHE engines (17 engines). The first column of the table lists the total number of rules present in each data-set, second and third column indicates the quantity of rules having prefix length less than or greater than 16 respectively. It can be observed that the density of the rules with prefix length less than 16-bit is negligible; thus, these rules are not considered for further processing. Column four represents the total number of rules saved by the micro-engines in the FIB sub-tables. Further, column five summarizes the number of insertion failures pertaining to each data set. The insertion failure occurs when the micro-Cuckoo hash engine cannot store the prefix rule in the respective FIB tables. The more detailed performance analysis of micro_CHE engines is provided by the five graphs shown in Figure 4.6 to 4.10. These graphs portray the computation results of the 16 micro_CHE engines pertaining to each of the five data sets. The results of the micro_CHE24 engine are omitted in these graphs, and it is examined thoroughly in the next Section. The x-axis of the graph represents the different micro_CHE engines, and the y-axis shows the range of prefix rules. Four important factors are depicted in the graph - (1) total number of prefix rules given as input to a particular engine, (2) number of entries saved by each engine, (3) rate of the data collision occurrences handled by each micro engine, and (4) probability of data insertion failures for a given data set. It can also be observed from the five graphs, that the amount of prefix rules handled by each micro engine varies with each data set. For example, consider micro_CHE19 engine. In the first data set, micro_CHE19 examines only 563 rules, whereas, for the rest of the four data sets, it analyzes 1390, 1753, 1021 and 2851 prefix rules respectively. For five prefix data-sets, the micro-hash IP lookup system accomplishes an average load factor of 90%, despite the anomalies present in the density of prefix rules. This indicates micro_CHE engines accommodates 90% of the incoming rules in the appropriate FIB tables irrespective of the size complexity of the input dataset.

The load factor of the system can be increased by implementing Run-Count mechanism explained in the Section 3.1.5. The value of run_count decides the number of iterations

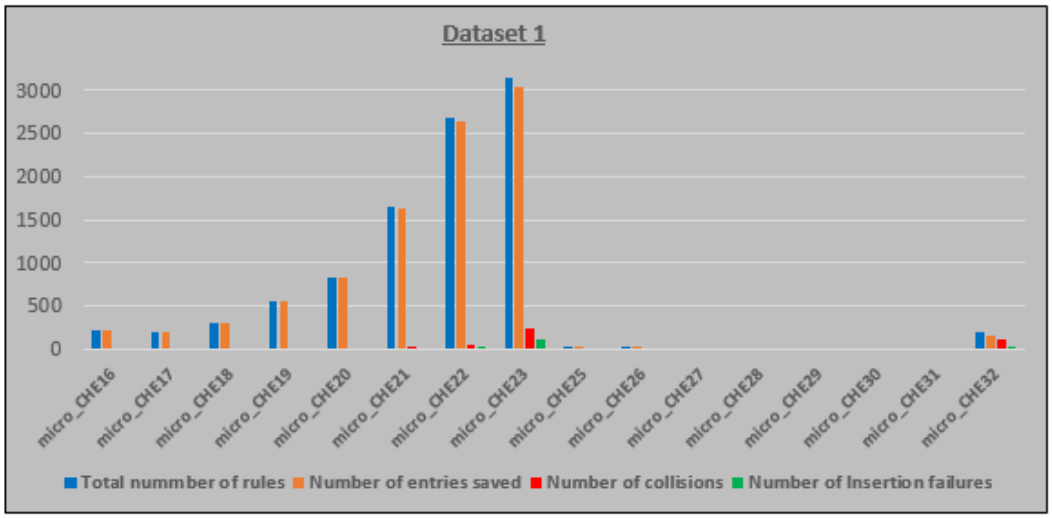


Figure 4.6: Data computations handled by each prefix function with respect to Dataset 1

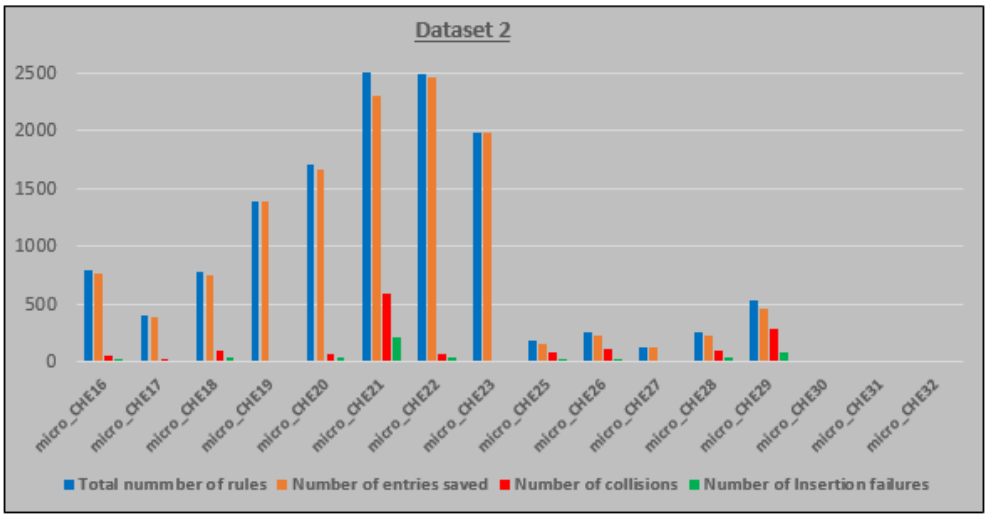


Figure 4.7: Data computations handled by each prefix function with respect to Dataset 2

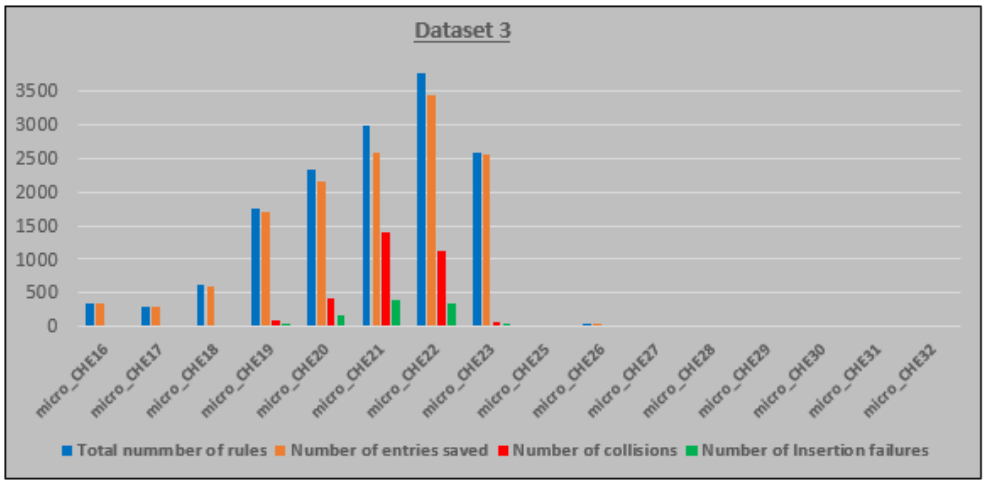


Figure 4.8: Data computations handled by each prefix function with respect to Dataset 3

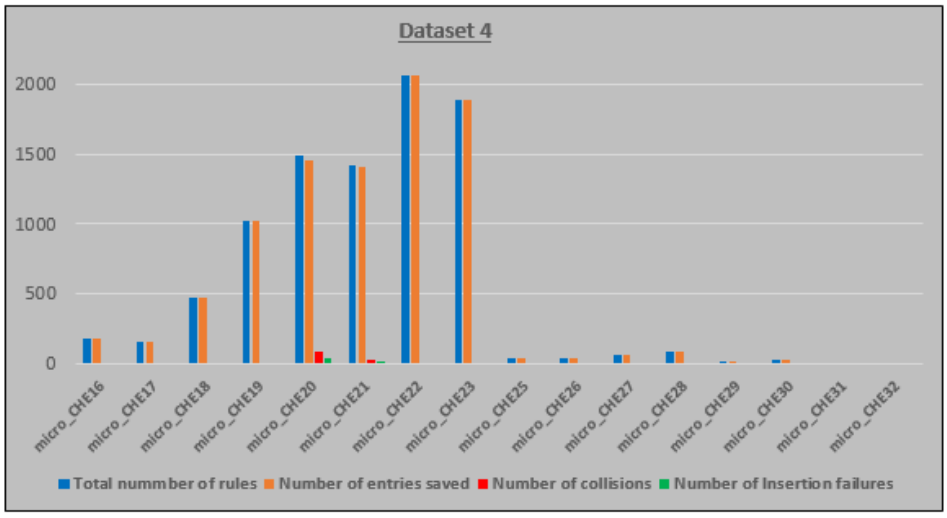


Figure 4.9: Data computations handled by each prefix function with respect to Dataset 4

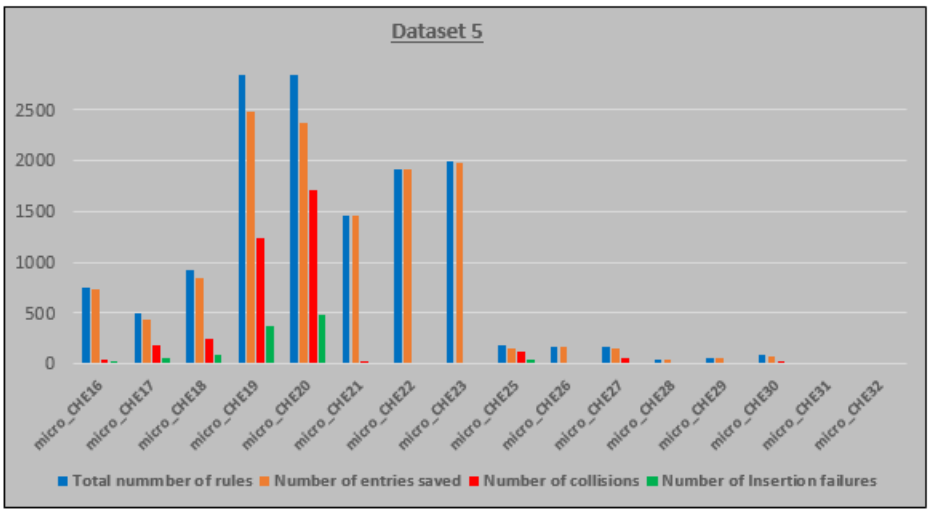


Figure 4.10: Data computations handled by each prefix function with respect to Dataset 5

carried out by the `micro_CHE` engines, before declaring an insertion failure. Run_count value doesn't depend on the number of FIB sub-tables, the designer is free to fix this threshold value. Each iteration conducted by the `micro_CHE` engine alters the data organization structure of the FIB sub-tables, i.e., the frequency of collision occurrence will expand. In summary, the implementation of the run_count technique results in two main outcomes - (1) the load factor of the system increases significantly, i.e., the frequency of rule drop is alleviated substantially, and (2) the rate of data collision occurrence is elevated. The first result is indicated by the line graph shown in Figure 4.11. It can be observed from the graph, the number of packets dropped by the `micro_CHE` engines reduces as the number of iterations increase. For example, consider dataset-5. Initially, when run_count is one, 1492 prefix rules cannot be accommodated in the FIB tables. On the contrary, when the run_count value is increased to 500; the `micro_CHE` engines fails to store only 146 prefix rules. Henceforth, with the inclusion of the run_count mechanism, the load factor of the IP lookup system can be elevated from 90% to 99%. However, the adverse effect is the inflated occurrences of the data collisions. To explain the negative effects of this mechanism, the computations of the engine `micro_CHE24` is studied in detail. The `micro_CHE24` engine is reviewed as it operates on 51% of the incoming prefix rules. The data shown in Table 4.4 represents the

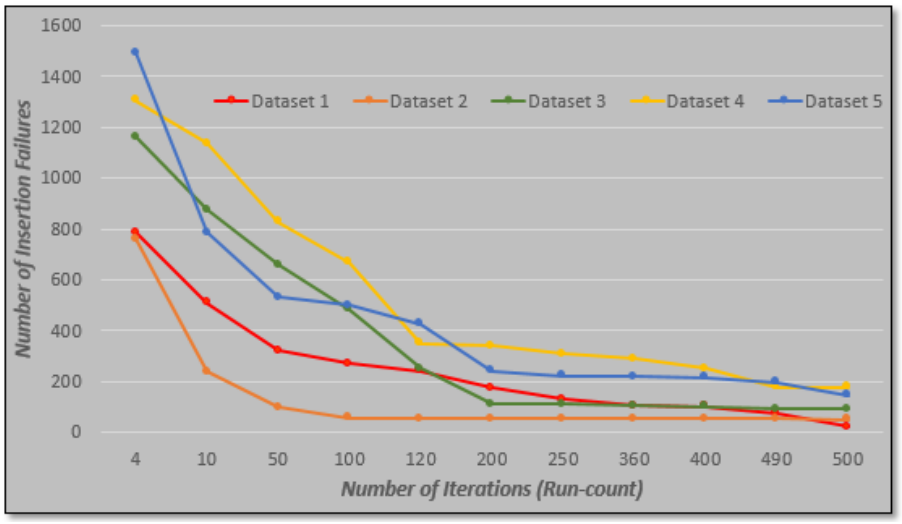


Figure 4.11: Analysis of Data Storage Capacity using Run_Count mechanism

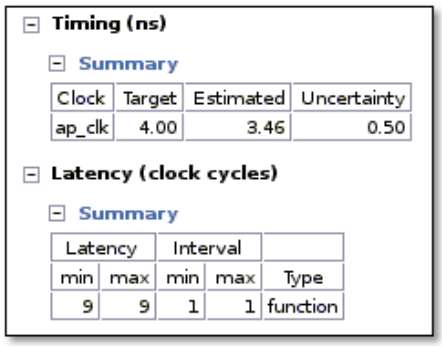
number of collisions handled by the `micro_CHE24` engine, when the run_count value is extended from 1 to 500. Thus, the value of run_count should be set to an optimal value, to gain notable improvement without suffering from a high data insertion latency. The optimal value of run_count is specific to input key data set; thus, it cannot be generalized. The time complexity of the `micro_PL` engines is indicated by the figures 4.12 and 4.13. The Figure 4.12 represents the overall time required by the system to process a single packet. Figure 4.13 presents the time complexity of each `micro_PL` engine.

The next important performance metric of an IP lookup system is the power required by the query engine to route the incoming packets. An 18MB TCAM, consisting of 16 transistors per cell, operating at a frequency of 266 MHz consumes 12 to 15 watts of power [33]. Thus, TCAM-based lookup engine is a power hungry approach. The proposed hash-based LPM technique deploys BRAMs to perform packet routing. [46] indicates the power consumption of a BRAM can be estimated using Xilinx XPower tool. Henceforth, it can be derived that a single 18-Kb BRAM running at a clock frequency of 289MHz utilize 17.03 mW of power [47]. With this result, the total power consumed by the entire micro-hash system can be estimated to be 9.49 watts.

Table 4.4: Computations carried out by `micro_CHE24` engine

RUN_COUNT	DATASET 1		DATASET 2		DATASET 3		DATASET 4		DATASET 5	
	NC	NE	NC	NE	NC	NE	NC	NE	NC	NE
4	483	596	245	236	140	166	1297	1985	1175	426
10	1343	314	504	121	492	88	2086	1130	1965	231
50	1563	186	548	50	1192	39	3208	668	4004	129
100	1735	47	638	0	1710	12	5719	348	5528	98
200	2143	3	638	0	2644	6	6662	308	7591	46
500	3116	2	638	0	5713	5	8934	288	9982	38

NC – Number of Collisions NE – Number of Entries Missed



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.46	0.50

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
9	9	1	1	function

Figure 4.12: Time complexity of the IP lookup system

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_Forward_prefix_24_fu_1021	Forward_prefix_24	5	5	1	1	function
grp_Forward_prefix_23_fu_1046	Forward_prefix_23	5	5	1	1	function
grp_Forward_prefix_22_fu_1071	Forward_prefix_22	5	5	1	1	function
grp_Forward_prefix_18_fu_1096	Forward_prefix_18	5	5	1	1	function
grp_Forward_prefix_16_fu_1121	Forward_prefix_16	5	5	1	1	function
grp_Forward_prefix_29_fu_1146	Forward_prefix_29	5	5	1	1	function
grp_Forward_prefix_17_fu_1171	Forward_prefix_17	5	5	1	1	function
grp_Forward_prefix_28_fu_1196	Forward_prefix_28	5	5	1	1	function
grp_Forward_prefix_26_fu_1221	Forward_prefix_26	5	5	1	1	function
grp_Forward_prefix_21_fu_1246	Forward_prefix_21	4	4	1	1	function
grp_Forward_prefix_20_fu_1269	Forward_prefix_20	4	4	1	1	function
grp_Forward_prefix_19_fu_1292	Forward_prefix_19	4	4	1	1	function
grp_Forward_prefix_32_fu_1315	Forward_prefix_32	4	4	1	1	function
grp_Forward_prefix_27_fu_1338	Forward_prefix_27	4	4	1	1	function
grp_Forward_prefix_25_fu_1361	Forward_prefix_25	4	4	1	1	function
grp_Forward_prefix_30_fu_1384	Forward_prefix_30	4	4	1	1	function

Figure 4.13: Summary of time complexity for each `micro_PL` engine

The performance estimates of the micro-Cuckoo hash IP system is compared against the existing IP lookup designs. Due to lack of accessibility to the data sets and the algorithms encompassed in other designs; the result comparison is based on the utilization numbers provided in the papers and the metrics are made scalable to match the data set sizes considered in this evaluation process. Zhou et al. proposed a fast and efficient Cuckoo switch based lookup engine capable of forwarding 350 million packets per second (MLPS); however, it required 2.16MB of memory to store the prefix rules data [48]. Zsolt et al. developed a hash table query engine to match the line-rate of data processing. It utilized only 295KB of memory but provided a data speed of 10Gbps [40]. Michel et al. designed a packet processing scheme based on progressive open hashing technique and set associative memory array [32]. Although by using progressive hashing mechanism an escalated line rate of 320Gbps was achieved; it requires 7x more memory than proposed micro-Cuckoo system. Similar to this approach, Stefanos et al. proposed IPStash lookup engine based on set associative memory, which is capable of processing an incoming packet within three clock cycles (three levels); yet suffers from space overhead complexity [33]. Yun et al. established a tree based query engine for high lookup rates [25]. The system comprises of 18-level BRtree (balanced tree). It is capable of processing over 400 million packets per second and consumes only 600KB of memory; however, it fails to maintain constant worst-case lookup time. Hoang et al. developed a power-efficient IP lookup engine, with the inbuilt capacity to sustain constant query time [46]. This approach is built on SRAM and tree structures, consuming only 912 BRAMs; yet it operates on a frequency of 170 MHz, which slows down the packet processing time. Arvind et al. designed a novel IP lookup engine using a high-level synthesis tool known as Bluespec [49][50]. The query engine is capable of operating on a high clock frequency of about 303MHz. However, the lookup engine entails a high memory consumption utility factor of about 63.5% to 99.9%, depending on throughput cycles three and one respectively. Henceforth, the micro-Cuckoo hash IP lookup system provides an efficient query engine capable of forwarding over 280 million packets per second, without injecting large memory overhead. The entire system requires only 587 BRAMs (27% of the available memory).

Chapter 5

Conclusion and Future Work

This thesis presented a new high-performance and memory efficient pipelined architecture to perform Layer 3 route lookups. The proposed micro-Cuckoo hash IP lookup engine achieves an excellent load factor utility by reducing the classical hashing overflow problem. The IP lookup engine incorporates a granular and parameterizable approach to store the prefix rules, which makes the design compact and efficient. The main contributions of this work are - (1) the query engine supports a high data line rate by processing over 280 million packets per second, (2) the space complexity of the FIB tables is alleviated substantially, (3) the micro-hash based IP lookup engine supports concurrent complex operations of rule update and LPM data fetch, and (4) the lookup engine is also capable of maintaining constant worst-case query time irrespective of the size complexity of the FIB tables. Furthermore, micro-Cuckoo is specified fully in C++, making it portable and easy to maintain. The micro-engines deployed in the system make the design user-friendly for porting to other applications.

In this thesis, the power consumption metric was not given a higher priority; further investigations can be done to explore the design space. The proposed IP lookup engine is developed and tested on IPV4 packets, the architecture of the system can be enhanced to deal with IPV6 packet forwarding by including additional micro-engines. Also, the micro-hash

engine scheme can be deployed to perform other network layer functionalities like security, data traffic monitoring, firewalling, multimedia communication and load balancing.

Bibliography

[1] L. G. Roberts. "Beyond Moore's law: Internet growth trends". In: *Computer* 33.1 (Jan. 2000), pp. 117–119.

[2] M. J. Akhbarizadeh and M. Nourani. "Hardware-Based IP Routing Using Partitioned Lookup Table". In: *IEEE/ACM Transactions on Networking* (Aug. 2005).

[3] Ewaryst Tkacz. *Internet - Technical Development and Applications*. Springer Berlin Heidelberg, 2009.

[4] D. Cavendish. "Evolution of optical transport technologies: from SONET/SDH to WDM". In: *IEEE Communications Magazine* 38.6 (June 2000), pp. 164–172.

[5] Ramaswami, Sivarajan, and Kumar N. *Optical networks. a practical perspective*. Morgan Kaufmann Publishers, 1998.

[6] P. Gupta and N. McKeown. "Algorithms for packet classification". In: *IEEE Network* 15.2 (Mar. 2001), pp. 24–32.

[7] R. J. Essiambre et al. "Capacity Limits of Optical Fiber Networks". In: *Journal of Lightwave Technology* 28.4 (Feb. 2010), pp. 662–701.

[8] Marcel Waldvogel. "Fast Longest Prefix Matching: Algorithms, Analysis, and Applications". MA thesis. Zurich: Swiss Federal Institute Of Technology, 1968.

[9] Valter Popeskic. *TCAM and CAM memory usage inside networking devices*. Feb. 2015.

[10] S. Demetriades et al. "An Efficient Hardware-Based Multi-hash Scheme for High Speed IP Lookup". In: *2008 16th IEEE Symposium on High Performance Interconnects*. Aug. 2008, pp. 103–110.

[11] V. Srinivasan and G. Varghese. "Fast Address Lookups Using Controlled Prefix Expansion". In: *ACM Trans. Comput. Syst.* 17.1 (Feb. 1999), pp. 1–40.

[12] Xuehong Sun. "IP Address Lookup and Packet Classification Algorithms". PhD thesis. Ontario, Canada: Carleton University, Nov. 2003.

[13] Alex Zinin. *Cisco IP Routing: Packet Forwarding and Intra-domain Routing Protocols*. Addison-Wesley Professional, Oct. 2001.

[14] F. Baker, ed. *Requirements for IP Version 4 Routers*. United States, 1995.

[15] Hoang Le, Thilan Ganegedara, and Viktor K. Prasanna. "Memory-efficient and Scalable Virtual Routers Using FPGA". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. ACM, 2011, pp. 257–266.

[16] Karthik Venkatesh. "Implementation of Memory Efficient IP lookup Architecture". MA thesis. Northridge: California State University, May 2014.

[17] P. Gupta, S. Lin, and N. McKeown. "Routing lookups in hardware at memory access speeds". In: *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Mar. 1998.

[18] F. Zane, Girija Narlikar, and A. Basu. "Coolcams: power-efficient TCAMs for forwarding engines". In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 1. Mar. 2003, 42–52 vol.1.

[19] A. X. Liu, C. R. Meiners, and E. Torng. "TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs". In: *IEEE/ACM Transactions on Networking* (Apr. 2010).

[20] D. Y. Chang and P. C. Wang. "TCAM-Based Multi-Match Packet Classification Using Multidimensional Rule Layering". In: *IEEE/ACM Transactions on Networking* (Apr. 2016).

[21] Edward Fredkin. "Trie Memory". In: *Commun. ACM* 3.9 (Sept. 1960).

[22] Yeim-Kuan Chang. "Fast binary and multiway prefix searches for packet forwarding". In: *Computer Networks* 51.3 (2007).

[23] Priyank Warkhede and Subhash Suri. "Multiway range trees: scalable {IP} lookup with fast updates". In: *Computer Networks* (2004).

[24] J. Lee and H. Lim. "A new name prefix trie with path compression". In: *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. Oct. 2016.

[25] Y. Qu and V. K. Prasanna. "High-Performance Pipelined Architecture for Tree-Based IP Lookup Engine on FPGA". In: *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. May 2013, pp. 114–123.

[26] Marcel Waldvogel and Varghese. "Scalable High Speed IP Routing Lookups". In: *SIGCOMM Comput. Commun. Rev.* (Oct. 1997).

[27] J. Hasan, S. Cadambi, and S. Chakradhar. "Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture". In: *33rd International Symposium on Computer Architecture (ISCA'06)*. 2006, pp. 203–215.

[28] S. Dharmapurikar. "Longest prefix matching using bloom filters". In: *IEEE/ACM Transactions on Networking* (Apr. 2006).

[29] Haoyu Song and Dharmapurikar. "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing". In: *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '05. 2005.

[30] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. "BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations". In: *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. 2009.

[31] Wei Liang, Wenbo Yin, and Ping Kang. "Memory efficient and high performance key-value store on FPGA using Cuckoo hashing". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016, pp. 1–4.

[32] Michel Hanna and Demetriades. "Progressive Hashing for Packet Processing Using Set Associative Memory". In: *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '09. 2009.

[33] S. Kaxiras and G. Keramidas. "IPStash: a set-associative memory approach for efficient IP-lookup". In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Mar. 2005.

[34] M. Kwon, P. Reviriego, and S. Pontarelli. "A length-aware cuckoo filter for faster IP lookup". In: *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Apr. 2016.

[35] Michel Hanna and Demetriades. "CHAP: Enabling Efficient Hardware-Based Multiple Hash Schemes for IP Lookup". In: *NETWORKING 2009: 8th International IFIP-TC 6 Networking Conference, Aachen, Germany, May 11-15, 2009. Proceedings*. 2009.

[36] Philippe Coussy and Michael Meredith. *An Introduction to High-Level Synthesis*. 2009.

[37] Rainer Kress. "High-level synthesis for dynamically reconfigurable hardware/software systems". In: *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm: 8th International Workshop*. 1998.

[38] S. Pontarelli and P. Reviriego. "Cuckoo Cache: A Technique to Improve Flow Monitoring Throughput". In: *IEEE Internet Computing* 20.4 (July 2016).

[39] Bin Fan, Andersen, and Dave G. "Cuckoo Filter: Practically Better Than Bloom". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. 2014.

[40] Zsolt István et al. "A Hash Table for Line-Rate Data Processing". In: *ACM Trans. Reconfigurable Technol. Syst.* 8.2 (Mar. 2015), 13:1–13:15.

[41] R. S. Meurer and A. Fronhlich. "An Implementation of the AES Cipher Using HLS". In: *2013 III Brazilian Symposium on Computing Systems Engineering*. Dec. 2013.

[42] Y. Sun et al. "MinCounter: An efficient cuckoo hashing scheme for cloud storage systems". In: *Symposium on Mass Storage Systems and Technologies (MSST)*. May 2015.

[43] Xiaozhou Li, Andersen, and David G. "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing". In: *Proceedings of the Ninth European Conference on Computer Systems*. 2014.

[44] N. Nguyen and P. Tsigas. "Lock-Free Cuckoo Hashing". In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. June 2014.

[45] *The Center for Applied Internet Data Analysis (CAIDA)*. URL: `http://data.caida.org/datasets/routing/routeviews-prefix2as/`.

[46] H. Le and V. K. Prasanna. "Scalable High Throughput and Power Efficient IP-Lookup on FPGA". In: *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. Apr. 2009, pp. 167–174.

[47] *Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics*. DS892. Rev. 1.14. Xilinx, Inc. Feb. 2017.

[48] Dong Zhou et al. "Scalable, High Performance Ethernet Forwarding with CuckooSwitch". In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. 2013.

[49] Rishiyur S. Nikhil and Arvind. "What is Bluespec?" In: *SIGDA Newsl.* 39.1 (Jan. 2009).

[50] Arvind et al. "High-level synthesis: an essential ingredient for designing complex ASICs". In: *IEEE/ACM International Conference on Computer Aided Design*. Nov. 2004.