Report Number: WUCS-2006-43

2006-08-01

Algorithms and Architectures for Network Search Processors

Sarang Dharmapurikar

The continuous growth in the Internet's size, the amount of data traffic, and the complexity of processing this traffic gives rise to new challenges in building high-performance network devices. One of the most fundamental tasks performed by these devices is searching the network data for predefined keys. Address lookup, packet classification, and deep packet inspection are some of the operations which involve table lookups and searching. These operations are typically part of the packet forwarding mechanism, and can create a performance bottleneck. Therefore, fast and resource efficient algorithms are required. One of the most commonly used techniques for such... **Read complete abstract on page 2.**

Algorithms and Architectures for Network Search Processors

Sarang Dharmapurikar

Complete Abstract:

The continuous growth in the Internet's size, the amount of data traffic, and the complexity of processing this traffic gives rise to new challenges in building high-performance network devices. One of the most fundamental tasks performed by these devices is searching the network data for predefined keys. Address lookup, packet classification, and deep packet inspection are some of the operations which involve table lookups and searching. These operations are typically part of the packet forwarding mechanism, and can create a performance bottleneck. Therefore, fast and resource efficient algorithms are required. One of the most commonly used techniques for such searching operations is the Ternary Content Addressable Memory (TCAM). While TCAM can offer very fast search speeds, it is costly and consumes a large amount of power. Hence, designing cost-effective, power-efficient, and high-speed search techniques has received a great deal of attention in the research and industrial community. In this thesis, we propose a generic search technique based on Bloom filters. A Bloom filter is a randomized data structure used to represent a set of bit-strings compactly and support set membership queries. We demonstrate techniques to convert the search process into table lookups. The resulting table data structures are kept in the off-chip memory and their Bloom filter representations are kept in the on-chip memory. An item needs to be looked up in the off-chip table only when it is found in the on-chip Bloom filters. By filtering the off-chip memory accesses in this fashion, the search operations can be significantly accelerated. Our approach involves a unique combination of algorithmic and architectural techniques that outperform some of the current techniques in terms of cost-effectiveness, speed, and power-efficiency.

2006-43

Algorithms and Architectures for Network Search Processors, Doctoral Dissertation, August 2006

Authors: Sarang Dharmapurikar

Corresponding Author: sarang@arl.wustl.edu

Web Page: http://www.arl.wustl.edu/projects/fpx/reconfig.htm

Abstract: The continuous growth in the Internet's size, the amount of data traffic, and the complexity of processing this traffic gives rise to new challenges in building highperformance network devices. One of the most fundamental tasks performed by these devices is searching the network data for predefined keys. Address lookup, packet classification, and deep packet inspection are some of the operations which involve table lookups and searching. These operations are typically part of the packet forwarding mechanism, and can create a performance bottleneck. Therefore, fast and resource efficient algorithms are required. One of the most commonly used techniques for such searching operations is the Ternary Content Addressable Memory (TCAM). While TCAM can offer very fast search speeds, it is costly and consumes a large amount of power. Hence, designing cost-effective, power-efficient, and high-speed search techniques has received a great deal of attention in the research and industrial community.

In this thesis, we propose a generic search technique based on Bloom filters. A Bloom filter is a randomized data structure used to represent a set of bit-strings compactly and support set membership queries. We demonstrate techniques to convert the search process into table lookups. The resulting table data structures are

Type of Report: Other

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ALGORITHMS AND ARCHITECTURES FOR NETWORK SEARCH PROCESSORS

by

Sarang Dharmapurikar, B.E.

Prepared under the direction of Professor John W. Lockwood

A dissertation presented to the Henry Edwin Sever Graduate School of Washington University in partial fulfillment of the requirements for the degree of

DOCTOR OF SCIENCE

August 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

ALGORITHMS AND ARCHITECTURES FOR NETWORK SEARCH PROCESSORS

by

Sarang Dharmapurikar

ADVISOR: Professor John W. Lockwood

August 2006

Saint Louis, Missouri

The continuous growth in the Internet's size, the amount of data traffic, and the complexity of processing this traffic gives rise to new challenges in building high-performance network devices. One of the most fundamental tasks performed by these devices is searching the network data for predefined keys. Address lookup, packet classification, and deep packet inspection are some of the operations which involve table lookups and searching. These operations are typically part of the packet forwarding mechanism, and can create a performance bottleneck. Therefore, fast and resource efficient algorithms are required. One of the most commonly used techniques for such searching operations is the Ternary Content Addressable Memory (TCAM). While TCAM can offer very fast search speeds, it is costly and consumes a large amount of power. Hence, designing cost-effective, power-efficient, and high-speed search techniques has received a great deal of attention in the research and industrial community.

In this thesis, we propose a generic search technique based on Bloom filters. A Bloom filter is a randomized data structure used to represent a set of bit-strings compactly and support set membership queries. We demonstrate techniques to convert the search process into table lookups. The resulting table data structures are kept in the off-chip memory and their Bloom filter representations are kept in the on-chip memory. An item needs to be looked up in the off-chip table only when it is found in the on-chip Bloom filters. By filtering the off-chip memory accesses in this fashion, the search operations can be significantly accelerated. Our approach involves a unique combination of algorithmic and architectural techniques that outperform some of the current techniques in terms of cost-effectiveness, speed, and power-efficiency.

To *Aai*, *Baba*, and *Ajji*

Contents

List of Tables

List of Figures

Acknowledgments

My inexpressible thanks to my research advisor, Dr. John W. Lockwood, for providing me an opportunity to pursue a D.Sc. under his mentorship. John provided me the freedom and support to explore the research topics of my interest. His passion for solving the *real* problems and making a difference in the world has always been inspiring. His emphasis on producing actual working systems has made me think about my research problems from a completely practical perspective. He has been very patient with me and took out time for discussions without fail even while managing a team of more than fifteen graduate students. I've always thoroughly enjoyed our non-academic, lengthy discussions and benefited from both his academic and personal advice.

Many thanks to Dr. Jonathan Turner for his invaluable guidance. It is indeed an honor to have worked with such a renowned scientist. I have learned much while closely working with him on various projects at the Applied Research Laboratory (ARL). His high-quality standards for research at ARL made me work harder to produce an interesting dissertation. His dedication to his work and unbeatable enthusiasm for his goals have been a continuous source of inspiration to me.

I also wish to thank Dr. Vern Paxson with whom I worked during the summer of 2004 at ICSI Center for Internet Research (ICIR). He introduced me to a wealth of new problems in network security and set a course for long-term collaboration. I am grateful to him for including me in the project investigating the hardware support for network security and analysis systems.

Thanks to Dr. Patrick Crowley who agreed to collaborate on exploring the packet buffer optimization problem which resulted in our Globecom'05 paper. I would also like to thank my committee members Dr. Jeremy Buhler and Dr. Ronald Indeck for their important suggestions on the technical content.

I am thankful to all my coauthors whose contributions assisted me in producing interesting publications. Particularly, I wish to thank Praveen Krishnamurthy, David

Taylor, Todd Sproull, Haoyu Song, and Sailesh Kumar for their significant contributions to the publications we authored. Brainstorming sessions with them were not only great fun but also helped me shape my research ideas. Thanks to Ramaprabhu Janakiraman for introducing me to the Bloom filters.

I would like to thank Jean Grothe, Tom Evola and Jason Marqart who helped me with the issues related to my immigration status. Thanks to Myrna Harbison, Sharon Matlock, Peggy Fuller and Stella Sung for their efforts to make life easy for me and my fellow graduate students. Many thanks to Robyn Brinks for her help in editing this manuscript.

I am grateful to my research sponsors, NSF (Grant 0096052), Global Velocity, SAIC, and Agilent Labs for supporting my work.

My tenure at Washington University has been fun-filled because of my friends, roommates, and lab-mates. Many thanks to them for spicing up my stay in St. Louis.

Finally, I would like to thank my parents, grandmother, brother, sister, and my fiancée, Rashmi, for their unconditional love, constant support, and encouragement. What they have done for me can not be expressed in words. I admire their patience while I have been away from home.

Sarang Dharmapurikar

Washington University in Saint Louis
August 2006

Chapter 1

Introduction

The Internet is one of the most important inventions of the twentieth century. Starting from a small network of computers developed for the defense purposes, the Internet rapidly evolved into a very large and complex network that is now a critical infrastructure for global communication, information exchange, and commerce. Today's Internet has millions of hosts connected to it. The layered architecture of the Internet allows independent development of services that can be deployed on top of the physical network. Day by day, innovative services are introduced which demand more complex data processing and faster data routing. The performance pressure on the network has led to the development of very high-speed routers consisting of several specialized hardware components for sophisticated packet processing.

A fundamental operation performed in the modern routers and the advanced networking devices is *searching*. In the context of packet processing, searching is the process of checking to see if network packets contain the information of interest. A simple example is the address lookup for packet forwarding. Given a routing table consisting of several addresses, the router searches the table for the destination address of each packet. Once this address is found, the packet can be forwarded to the next hop associated with this destination. Another example is searching for predefined keywords in the packet payload. The occurrence of such keywords in the payload can allow the router to give a special treatment to the packet.

Search operations are performed by key components in the data path of routers and their performance directly affects the packet forwarding speed. Slow search speeds

can create performance bottlenecks and degrade the router throughput. Therefore, sophisticated and high-speed techniques are required to design these components.

The focus of this dissertation is developing high-speed network search techniques. Specifically, we address three important search problems: the longest prefix matching for Internet Protocol (IP) route lookup, packet classification, and multi-pattern matching for deep packet inspection. We propose a new and generic search technique that uniquely combines some architectural and algorithmic techniques. We demonstrate how our proposed solution is superior to some of the existing solutions for the same problems.

1.1 Searching in the Context of Networks

In this chapter, we introduce network search operations and discuss the commonly employed techniques. We highlight their inadequacies and introduce our new approach.

Longest Prefix Matching (LPM): In LPM, we wish to search the longest matching prefix of an input key in a table consisting of a large number of prefixes. This is a fundamental process performed in the IP address lookup and forwarding.

The Internet consists of millions of networked computers. The data packets are routed from one host to another using IP. A host is connected to the Internet through an *interface* that has been assigned an IP address. For the Internet Protocol version 4 (IPv4), the addresses are 32 bits wide and allow a capacity of 2^{32} interfaces. Before 1993, the IP addresses were allocated to organizations in chunks of $2^8 = 256$ (Class C network), $2^{16} = 65,536$ (Class B network), and $2^{24} = 16,777,216$ (Class A network). This could be accomplished by assigning the most specific 24 bits, 16 bits, and 8 bits of an IP address to Class C, B, and A networks respectively. For instance, the IP address 128.252.153.00/24 specifies a network of 256 or fewer interfaces, each having an address with the first 24 bits being 100000001111110010011001, the binary representation of 128.252.153. However, such an assignment became problematic. If an organization wanted to support a network with interfaces significantly more than

256 but significantly less than 65,634, it would still be allocated space for 65,634 addresses which would result in an overallocation. This would cause a rapid shortage of IP addresses. Therefore, to prolong the life of IPv4 addresses, Classless Interdomain Routing (CIDR) was adopted by the Internet Engineering Task Force (IETF). With CIDR, the classes were abolished. Today, the network addresses can be allocated as a.b.c.d/p where /p indicates how many bits of the address prefix should be considered for the network. Since p can be any arbitrary number of prefix bits between 0 to 32, the assignment is more flexible. For instance, if an organization wants an address space worth 3,000 interfaces, then an address a.b.c.d/20 can be assigned to the network which allows up to 4096 interfaces and reduces overallocation. All of these interfaces will share the same 20 prefix bits.

To route a packet from one interface to another, the router responsible for the network containing the source host sends the packet to the router handling the destination network. Each router maintains a routing table consisting of thousands of network addresses, each specified in *address/prefix* form, and the associated next hop information. To send the packet to the correct network, the router searches the table for the longest matching prefix of the destination address in the packet. When one is found, the packet is forwarded to the associated next hop. The more matching prefix bits, the more specific the network address. It is possible that a packet matches two prefixes, one shorter than the other. The networks specified by these two prefixes can be handled by completely different routers and service providers. In order to resolve the next hop, the router looks for the most specific network address by matching the longest prefix. Address lookup with LPM needs to be performed for each incoming packet. The speed of address lookup directly impacts the forwarding rate of the router. Hence, high-speed solutions for address lookup are important. Comparing the destination address with all prefix entries in the routing table, one-by-one, is impractical when the table consists of several thousand prefixes. Sophisticated algorithmic techniques are required. LPM is a well researched problems in computer networking. The need of an efficient LPM algorithm is even more pronounced for IPv6 where the IP address is 128 bits long, four times that of an IPv4 address. A long IP address implies longer prefixes and a larger set of possible prefix lengths, both of which lead to increased complexity.

Conceptually, an l-bit prefix of a W-bit address defines a region of 2^{W-l} contiguous addresses in the space of 2^W addresses. We wish to search the smallest region in which a given destination address falls.

Packet Classification: While the IP route lookup operates on just a single field of the packet header, a more sophisticated search in which a key is specified over all the header fields, is required for applications such as firewalls. A firewall has predefined policies to control the flow of data in and out of the protected network. Consider the following rule.

if SIP $\in$ **[a:b] & DIP** $\in$ **[c:d] & SP** $\in$ **[e:f] & DP** $\in$ **[g:h] & Protocol = X**
then Action = drop

This rule states that a packet should be dropped when the source IP address (SIP) falls in the range a to b, the destination IP address (DIP) in the range c to d, the source port (SP) in the range e to f, the destination port (DP) in the range g to h, and the protocol field is exactly X. In general, there can be any number of header fields involved in the rule specification and the associated actions can be different. This rule specification is based on five fields and is useful in several applications. The classification based on such rules is commonly known as 5-tuple packet classification. Note that if the range of the values specified is a single value (e.g. $a = b$), then it is essentially an exact match for that field. More flexible rule specifications are also possible. Consider the following rule.

if SIP $\in S_1$ **& DIP** $\in S_2$ **& SP** $\in S_3$ **& DP** $\in S_4$ **& Protocol** $\in S_5$
then Action = drop

In this case, instead of defining ranges of values, a set of distinct values is specified. Thus, if SIP belongs to a set of predefined addresses, S_1, DIP belongs to S_2 and so on, then the rule matches and the action is executed. Such a rule can be converted into a form of the first type of rule by simply creating a separate rule for each value contained in S_1, S_2, and so on. In this way, a single rule can expand into several rules, each pointing to the same original rule.

Given a set of several thousand 5-tuple rules, the process of packet classification returns a set of matching rules for each arriving packet. From one perspective, this process can be viewed as a key searching process. A 5-tuple of the packet header is essentially a 104-bit key (32 bits of each address, 16 bits of each port, and 8 bits of protocol). This key can take any value in the space of 2^{104}. Each rule defines a region within this space. We wish to find if the given key (header of a packet) belongs to any region defined by the rules.

Pattern matching for deep packet inspection: With the growth of network security threats, network operators have started to control the flow of data depending on the packet content. Therefore, firewall policies which operate on just the packet header are no longer sufficient. Tighter policies that look deeper in the packet payload to decide if the data is safe are required. This gives rise to another class of searching processes: searching for a set of predefined keywords or patterns in the streaming network data. These keywords or patterns can be signature strings for detecting SPAM, Internet worms, and viruses. This problem, commonly known as multi-pattern matching, has several applications beyond network security. For example, in content-based routing, a packet is routed to the appropriate destination based on the payload which requires detection of predefined payload signatures. Copyright protection is another example in which deep packet inspection is required to protect copyrighted or sensitive data from being transferred over the network illegally.

The patterns can be more expressive than just simple strings. Consider a regular expression "abc$^+$d?e" which matches all the keys with the first three characters "abc," followed by zero or more occurrences of 'c,' followed by character 'd,' followed by any single character followed by 'e'. Unlike LPM or packet classification, the number of *keys* represented by this expression is infinite since there is no limit on the number of times 'c' occurs after "abc". Hence, searching such keys (regular expressions) would require more sophisticated algorithmic techniques.

In summary, the network search problem deals with searching a set of predefined keys in network data that includes packet headers and payload. Most of these applications are in-line components in the data path. Therefore, the search speed directly affects the network speed and emphasizes the need for developing high-speed search techniques. For such applications, scalability of the search table is another important

issue. Increasing network speeds and data traffic, growing size of the routing tables, larger firewall rule sets, and larger pattern sets, all demand scalable search techniques. Some commonly used techniques are described below.

1.2 Common Search Techniques

One of the most naïve ways to look up a key is walking through all the keys present in the table and looking for a match. This becomes impractically slow for large tables. Another way is to use direct indexing. If the universe of keys is small, then a table can have an entry for each possible key within that universe so that the key to be searched can be used to index the table entry. However, this is often not the case. The universe of the keys is orders of magnitude larger than the size of the key set kept in the table. Therefore, direct indexing needs prohibitively large storage.

To search such a table, the keys need to be arranged in certain data structures and algorithmic techniques must be employed. Most of the literature on searching deals with efficient representation of keys in suitable data structures and an efficient algorithm to search this data structure for a given key. The two most important criteria to evaluate any such algorithm are the memory requirement to maintain the data structure and the time required to execute a search algorithm on a key.

Among the most common techniques for such table lookups is the use of set-associative memory. In order to look up a key in a set-associative memory, the key is split into two parts, *index* and *tag*. The index is used to select a block of fixed number of contiguous memory locations. Each memory location contains a tag and associated data. The tag of the search key is compared with the tags of all the memory locations within the selected block. If a tag matches, then it indicates a complete match for the key and the associated data can be used. If the number of memory locations within a block is n, then this memory is called n-way set-associative. If n is as large as the memory size, then it is called fully-associative memory, in which case the entire key acts as a tag and there is no index. The fully-associative memory is also known as Content Addressable Memory (CAM).

A fundamental characteristic of a set-associative memory is that it does not guarantee that a specific key will always be placed in one of the memory locations. When the index part of a key to be inserted selects a completely filled block of memory locations, then the key can not be inserted without replacing a key already present in the block. While other blocks might have empty locations, the key can not be placed in them for the sake of correctness of the search. To guarantee that a given set of keys will always be placed in a given sized set-associative memory, it must be made fully-associative. Therefore, CAM is one of the most popular choices for implementing table lookup.

The main problem with CAM, however, is the excessive power consumption from large parallel comparisons. A given search key must be compared with each memory location and that consumes a significant amount of power. The power consumption gives rise to additional problems such as requirement of power dissipation mechanisms and extra space on the circuit board to accommodate them. To develop compact devices, either power reduction methods for CAM or alternative algorithmic approaches for table searching are needed. While progress is being made on both fronts by the research community and the industry, this dissertation focuses on the algorithmic techniques for table searches.

Some of the most commonly used algorithmic techniques are based on *decision trees.* In a decision tree, a search starts from the root of the tree and progresses hierarchically towards the leaves. At each level of hierarchy, the scope of the search is narrowed. After traversing the tree, if the key is not found, then it is absent from the set. Several decision tree data structures exist. B-trees, Red-Black trees, binary search trees, interval trees, and radix trees are examples [38]. Often, the tree data structure can be optimized for specific search problem. A radix tree, or *trie*, can be used for LPM. Trie-based LPM techniques have been significantly improved upon in terms of memory and lookup time requirement [15, 18]. Decision-tree based packet classification algorithms have also been explored [21, 27]. The well-known Aho-Corasick automaton for multi-pattern matching is also similar to a decision tree algorithm [5].

Decision-tree based data structures and algorithms are natural choices for algorithmic searches. A decision tree involves multiple sequential dependent memory accesses since the next level of trie can be reached only after the node at the previous level has been read and a child node to follow has been determined. Such dependent memory

accesses slow down the overall search. The speed can be improved by pipelining the accesses where each stage in the trie is busy processing a different search key. However, pipelining requires multiple memory chips. Moreover, it will also result in pipelines with several stages that can make the memory management difficult.

A simple algorithmic alternative to CAM is a hash table. A hash table consumes an *average* $O(n)$ space for n keys and an *average* $O(1)$ memory accesses to search for an *exact* key. In a hash table, a given set of keys is mapped to the table entries using a hash function. Hash function $h()$ takes the key, x, from the universe $1, 2, ..., U$ as input, computes the address of the key, $h(x)$, between the range 0 to $m - 1$, and inserts the key at this address. If two keys get mapped to the same location, they *collide*. One method to resolve collisions is chaining in which all the keys mapped to the same location are linked in a list. When a key is to be searched, the location of the key is calculated using the hash function and the linked list of keys mapped to this location is traversed sequentially to locate the input key. If n keys are to be stored in m memory locations, then a hash table requires an average of $1 + (n - 1)/2m$ memory accesses for a successful search and $1 + n/m$ accesses for an unsuccessful search [38]. With a sufficiently larger number of slots in the table than the number of keys (i.e. small n/m), the memory accesses for both successful and unsuccessful searches approach one. Therefore, a hash table is a memory efficient alternative to CAMs with an almost equally effective search time.

Although CAM and hash tables serve as effective solutions for searching *exact* keys, they are not suitable for searching tables with keys containing ranges and wild cards. Consider a universe of keys between 0 to 1023. Assume that a set of all the keys from 17 to 32 needs to be represented and put in a table. While each key in the range can be stored separately, it is impractical for large ranges. Clearly, a straightforward hash table application does not help until either the range is represented differently or the search key is used differently so that the searching boils down to exact searching. A variant of CAM known as Ternary CAM (TCAM), however, provides a simple solution to such searches. In a TCAM, a bit can take a value 0, 1, or don't care (denoted by *). Each range can be represented as a set of prefixes. For instance, the range [17 : 32], which is [0000010001 : 0000100000] in binary, can be represented as a set of the following prefixes:

$$
\begin{aligned}
0000010001 &\equiv [17] \\
000001001* &\equiv [18:19] \\
00000101** &\equiv [20:23] \\
0000011*** &\equiv [24:31] \\
0000100000 &\equiv [32]
\end{aligned}
$$

These prefixes can be stored in the TCAM. When the key 0000010110 is to be searched, it is compared with all keys, ignoring the * bits. Since the key matches 00000101**, the search is successful. Due to the ternary representation, the range key could be expressed with a small number of TCAM compatible keys. It is worth mentioning that the method presented here is not the only way to represent ranges in TCAM. There are more space-efficient ways that have been proposed recently [24].

While a TCAM provides a more sophisticated search technology, it suffers similar problems as CAM: high power consumption and high costs. The cost of a TCAM is higher than a CAM since each TCAM cell must implement a ternary comparison. Again, the research community has responded with two types of solutions to such complex search problems: ones that try to minimize the power consumption of TCAM and the others based on algorithmic techniques that use commodity memory chips. The algorithmic techniques are of particular interest because they cost less. So, the question arises: how can we devise algorithmic search techniques that match the performance of TCAM? To date, most of the algorithms have not been able to match the performance and the generality of TCAM.

Developing algorithmic search techniques as versatile and efficient as TCAM is challenging. However, by taking advantage of particular application characteristics, such techniques or even more sophisticated ones can be developed. This thesis explores such techniques and shows how a limited TCAM-like functionality can be achieved for certain search applications using parallel and embedded memory blocks on a chip in combination with commodity memory chips. Particularly, we first explore ways to express the keys and queries in a form suitable for exact searches. Then we use the hash tables for executing these searches. Most importantly, we illustrate a technique based on Bloom filters to suppress unnecessary or potentially unsuccessful exact searches in the hash table and speed up the overall searching process.

1.3 Thesis Organization

In the next chapter, we review the related work on Bloom filters since it is central to our algorithms. A Bloom filter is a randomized data structure which uses hashing to store a set of strings compactly but *approximately*. Several variants of Bloom filters and their applications have been proposed. We will review some of them.

In Chapter 3, we address the LPM problem. The existing solutions to the address lookup problem include both algorithmic (trie-based, hashing) and architectural (TCAM) solutions. We describe our Bloom filter based solution which is a mix of algorithmic and architectural techniques. We show how Bloom filters can be used effectively in combination with existing VLSI hardware technology to realize a LPM solution that outperforms the TCAM in terms of cost and speed.

In Chapter 4, we address the packet classification problem. We propose an approach that uses Bloom filters combined with a modified version of the Crossproducting algorithm [33]. The Crossproducting algorithm classifies a packet by performing single field lookups for each field and combining the results to get the best matching rule. It is fast but requires exorbitant memory space due to the extra rules introduced in the process of the crossproduct. We demonstrate a technique that modifies the algorithm and reduces the memory overhead by splitting the rule set into smaller subsets. More importantly, we show how we can preserve the speed of the original algorithm by reducing unnecessary memory accesses using Bloom filters.

In Chapter 5, we introduce a simple algorithm for multi-pattern matching. Like LPM, multi-pattern matching is a well researched problem with a family of algorithmic solutions. However, the existing solutions are inadequate to execute this task at very high speeds and tend to create a bottleneck. We leverage the hardware-based Bloom filters to perform multi-pattern matching at multi-gigabit per second rates.

While the algorithm introduced in Chapter 5 can process packets at line speed, it is only efficient for short strings (less than 16 characters). In Chapter 6, we extend this algorithm to handle strings of arbitrary lengths at the cost of more memory. The resulting solution is an enhancement of the classic Aho-Corasick algorithm for multi-pattern matching. Our solution takes advantage of the parallelism provided by the

hardware and the efficiency of Bloom filters to accelerate the Aho-Corasick algorithm by a constant factor dependent on the amount of available memory resources.

Chapter 7 summarizes our contributions.

Chapter 2

Bloom Filters and Variants

A Bloom filter is a data structure used to represent a set of items compactly and approximately. It is based on the concept of multiple hashing. Formulated by Burton H. Bloom in 1970 [8], it has found several applications including in the area of databases and computer networks [9].

In this chapter, the theory behind basic Bloom filters and their variants is reviewed.

2.1 Bloom Filters

A Bloom filter is essentially a bit-vector (which we call *Vector*) of length m used to efficiently represent a set of bit-strings. Given a set of bit-strings, S, with n members, a Bloom filter is "programmed" as follows. For each bit-string, x, in S, k hash functions, $h_1()...h_k()$, are computed on x producing k values each ranging from 1 to m. Each of these values addresses a single bit in the m-bit vector, hence each bit-string x causes k bits in the m-bit vector to be set to 1. Note that if one of the k hash values addresses a bit that is already set to 1, then that bit is not changed. The following pseudo-code describes adding a bit-string, x, to a Bloom filter.

BFAdd (x)
1. for (i=1 to k)
2. Vector[$h_i(x)$]$\leftarrow$ 1

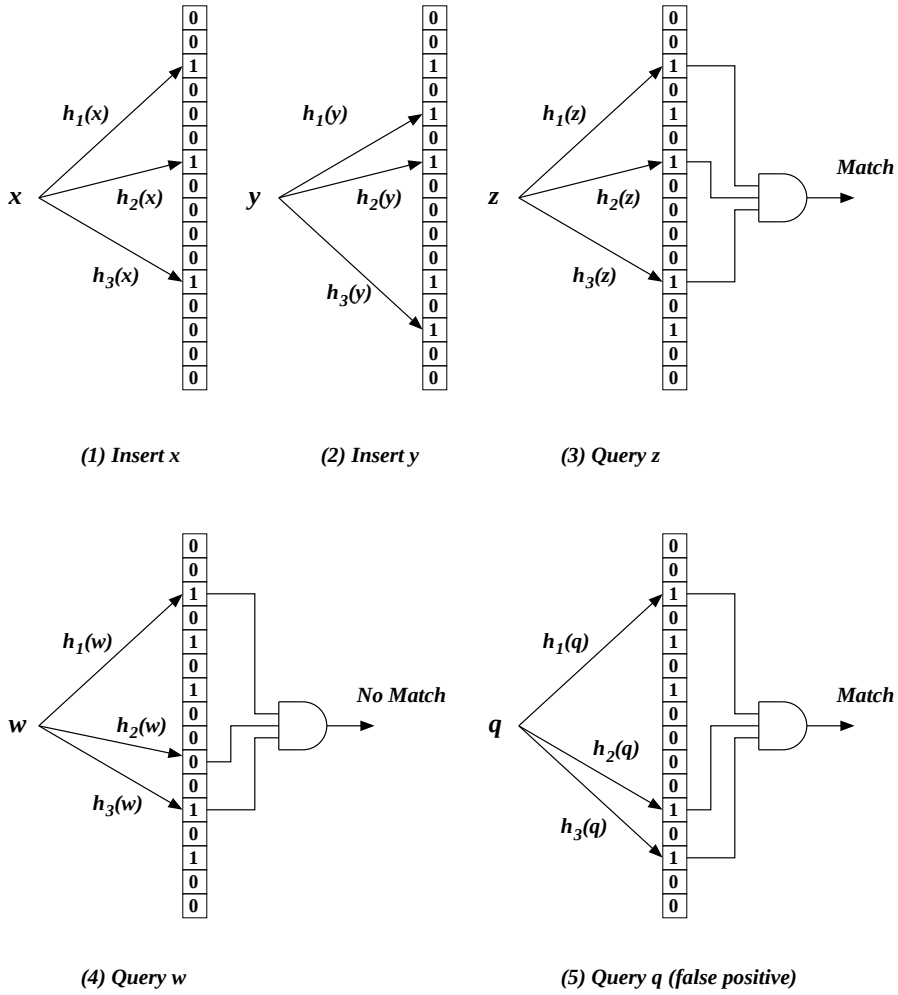


Figure 2.1: **Illustration of a Bloom filter**

Figure 2.1(1) and (2) illustrate Bloom filter programming. Two bit-strings, x and y are programmed in the Bloom filter with $k = 3$ hash functions and $m = 16$ bits in the array. Note that different strings can have overlapping bit patterns.

Querying the filter for set membership of a given bit-string, x, is similar to the programming process. Given bit-string x, k hash values are generated using the same hash functions used to program the filter. The bits in the m-bit vector at the locations corresponding to the k hash values are checked. If at least one of the k bits is 0, then the bit-string is declared to be a non-member of the set (Figure 2.1(4)). If all the bits are found to be 1, then the bit-string is said to belong to the set with a certain probability (Figure 2.1(3)). If all the k bits are found to be set and x is

not a member of S, then it is said to be a false positive. The following pseudo-code describes the query process:

BFQuery (x)
1. for (i=1 to k)
2. if (Vector[$h_i(x)$]=0) return false
3. return true

The ambiguity in membership comes from the fact that the k bits in the m-bit vector can be set by any of the n members of S. For instance, in Figure 2.1(5), q maps to all the bits which were set by x and y. Although $q \notin S$, the filter shows a match. Thus, finding a bit set does not necessarily imply that it was set by the particular bit-string being queried. However, finding a 0 bit certainly implies that the bit-string does not belong to the set; if it were a member, then all k-bits would have been set when the Bloom filter was programmed.

2.1.1 False Positive Probability

Now we look at the step-by-step derivation of the false positive probability i.e., the probability of finding all the k lookup bits set for a bit-string that is not programmed. The probability that a random bit of the m-bit vector is set to 1 by a hash function is simply $\frac{1}{m}$. The probability that it is not set is $1-\frac{1}{m}$. The probability that it is not set by any of the n members of X is $(1-\frac{1}{m})^n$. Since each of the bit-strings sets k bits in the vector, the probability becomes $(1-\frac{1}{m})^{nk}$. The probability that this bit is 1 becomes $1-(1-\frac{1}{m})^{nk}$. For a bit-string to be detected as a possible member of the set, all k bit locations generated by the hash functions need to be 1. The probability that this happens, f, is given by

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \tag{2.1}$$

For the large values of m the previous equation reduces to

$$f \approx \left(1 - e^{\frac{-nk}{m}}\right)^k \tag{2.2}$$

This probability is independent of the input bit-string and is termed the *false positive* probability. The false positive probability can be reduced by choosing appropriate values for m and k for a given size of the member set, n. It is clear that the size of the bit-vector, m, needs to be much larger than the size of the bit-string set, n. For the given ratio $\frac{m}{n}$, the false positive probability can be reduced by increasing the number of hash functions, k. In the optimal case, when false positive probability is minimized with respect to k, we get the following relationship

$$k = \frac{m}{n} \ln 2 \tag{2.3}$$

The false positive probability at this optimal point is given by

$$f = \left(\frac{1}{2}\right)^k \tag{2.4}$$

It should be noted that if the false positive probability is to be fixed, then the size of the filter, m, needs to scale linearly with the size of the bit-string set, n. In the optimally configured Bloom filter, the probability of finding a bit set is 0.5.

2.2 Counting Bloom Filters

Deleting a bit-string stored in the filter is impossible without introducing false negatives. Deleting a particular entry requires the corresponding k hashed bits in the bit vector be set to 0 which could disturb other bit-strings programmed into the filter that hash to any of these bits. In order to solve this problem, *Counting Bloom Filters* were proposed in [20]. A counting Bloom Filter maintains a vector of counters (which we call *Counter*) instead of a bit-vector. Whenever a bit-string is added to or deleted from the filter, the counters corresponding to the k hash values are incremented or decremented, respectively.

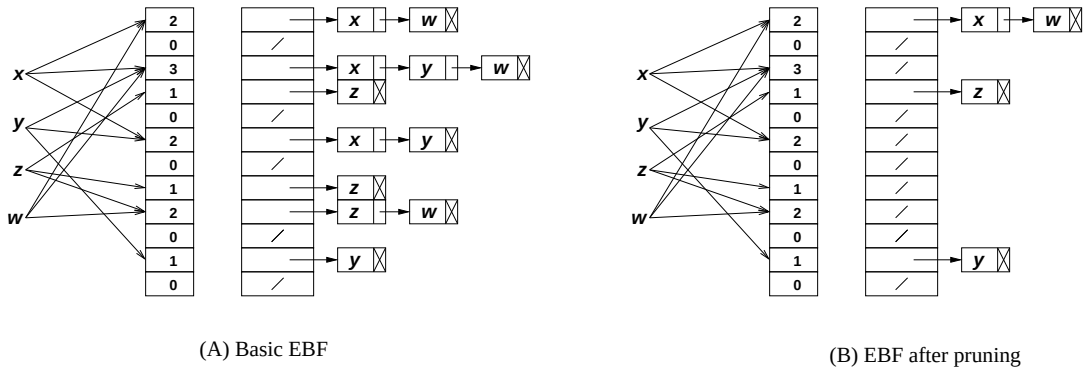


Figure 2.2: **Illustration of an Extended Bloom filter**

As will be clear later, we use a composite data structure consisting of a counting Bloom filter and a regular Bloom filter to support incremental updates to the database of items stored in Bloom filters.

2.3 Extended Bloom Filters

Extended Bloom Filters (EBF) were introduced by Song et. al. in [29]. An EBF is essentially a Counting Bloom filter coupled with a hash table. This combination enables not only a quick check for a query item in the set but also helps us retrieve the item and its associated information from the hash table. While some of the algorithms introduced in this work have a similar architectural flavor, there is a clear architectural separation between a Bloom filter and a hash table in our algorithms. An EBF on the other hand tightly couples the counting Bloom filter with the hash table. A one-to-one correspondence exists between the m counters of the counting Bloom filter and the m buckets of a hash table.

Song et. al. presented different flavors of the EBF. Only the basic EBF architecture is discussed here. An EBF works as follows. It maintains a counting Bloom filter with m counters and as many buckets in a hash table. An item is first inserted in the counting Bloom filter with the k hash functions and the same item is put in the k corresponding buckets of the hash table. This is illustrated in Figure 2.3(A).

Items x, y, z, and w are inserted in the data structure. Each item is replicated in k different places. The counter essentially indicates the number of items stored in the corresponding bucket. When an item is searched, the counting Bloom filter is queried. If the item is present in the Bloom filter, then the **bucket with the smallest counter is searched**. For instance, if y is to be searched, it will be searched in bucket 11 which has a count of 1 (and only one item in the corresponding bucket). A tie can be broken by choosing the least loaded bucket with the smallest index. The Bloom filter then serves a dual purpose: eliminating the unsuccessful searches and when a query is successful, pointing to the least loaded bucket to enable a quicker search. The authors argue that optimal parameters almost always cause an item to hash to a bucket where no other item is stored and a counter with a value of 1 can be found with a very high probability. If the counting Bloom filter is maintained in a fast *on-chip* memory and the hash table is kept in the slower but bigger *off-chip* memory, then unnecessary and expensive off-chip memory accesses can be avoided by first querying the Bloom filter.

Several optimizations to the basic EBF data structure were presented. Although there can be k copies of an item, only the copy in the least loaded bucket is searched. Hence, all the other copies of the item can be removed resulting in a pruned data structure where there is just one copy of each item. While removing the items, the counters are left unchanged to avoid incorrect searches. A pruned data structure is shown in Figure 2.3(B). Pruning makes incremental insertions and deletions more difficult. Incremental update-friendly algorithms that also maintain the same performance as a pruned EBF were presented in [29].

2.4 Spectral Bloom Filters

Cohen et. al. proposed Spectral Bloom Filters (SBF) to detect items having multiplicities greater than a particular threshold [13]. An SBF is essentially a CBF. Counters corresponding to an item are incremented each time the item is stored in a SBF. The main concept is that the number of times an item is stored can not exceed the smallest counter it hashes to. Thus, the smallest counter value selected by the hash functions is an estimate of the multiplicity of the item. The same idea was

proposed by Estan et. al. independently to count the number of packets belonging to a TCP flow approximately [19]. Both Cohen and Estan proposed the optimization of *conservative updates*. Essentially, since the minimum among all the selected counters represents the multiplicity of an item, the other counters don't need to be incremented until the minimum counter catches up. For instance, let an item, x, map to counters $\{12, 17, 23\}$ having values $C_{12} = 4$, $C_{17} = 5$, and $C_{23} = 2$. When the next instance of x is stored to the filter, instead of incrementing all the counters only C_{23} needs to be incremented. When the minimum counter catches up with other counters (when $C_{23} = C_{12} = 4$) then any next addition of x will increment both since both are now minimum counters until some other item hashing to one of these counters changes the value. When these two counters catch up to C_{17}, then all will be incremented. With this heuristic of minimal increase, the number of updates needed per item is reduced significantly. A drawback of the minimal increase heuristic is that an incremental deletion of items from the filter can introduce false negatives.

There are two false positives associated with the SBF data structure. The first is the regular false positive probability of the Bloom filters where an item, although not present in the filter, shows a match since all the counters it maps to are greater than 0. The second happens when the multiplicity of an item is overestimated, i.e. the multiplicity of item x is actually f_x but the filter's minimum counter corresponding to x shows it to be $f'_x > f_x$. Both false positive probabilities have the same value as Equation 2.2.

2.5 Bloomier Filters

Chazelle et. al. proposed a *Bloomier* filter which can encode arbitrary functions on a small subset of items from a large universe [10]. Formally, let f be the function that maps a finite arbitrary set of items, S, chosen from the domain $D = \{0, 1, 2, ..., N-1\}$. Let $R = \{\phi, 1, 2, ..., 2^r - 1\}$ be the range of this function so that $f(x)$ is ϕ for all $x \in D \backslash S$. Given an x, a Bloomier filter can produce $f(x)$ correctly for all $x \in S$ and almost correctly (with some small error probability) for all $x \in D$
S. A limitation, however, is that S needs to be static.

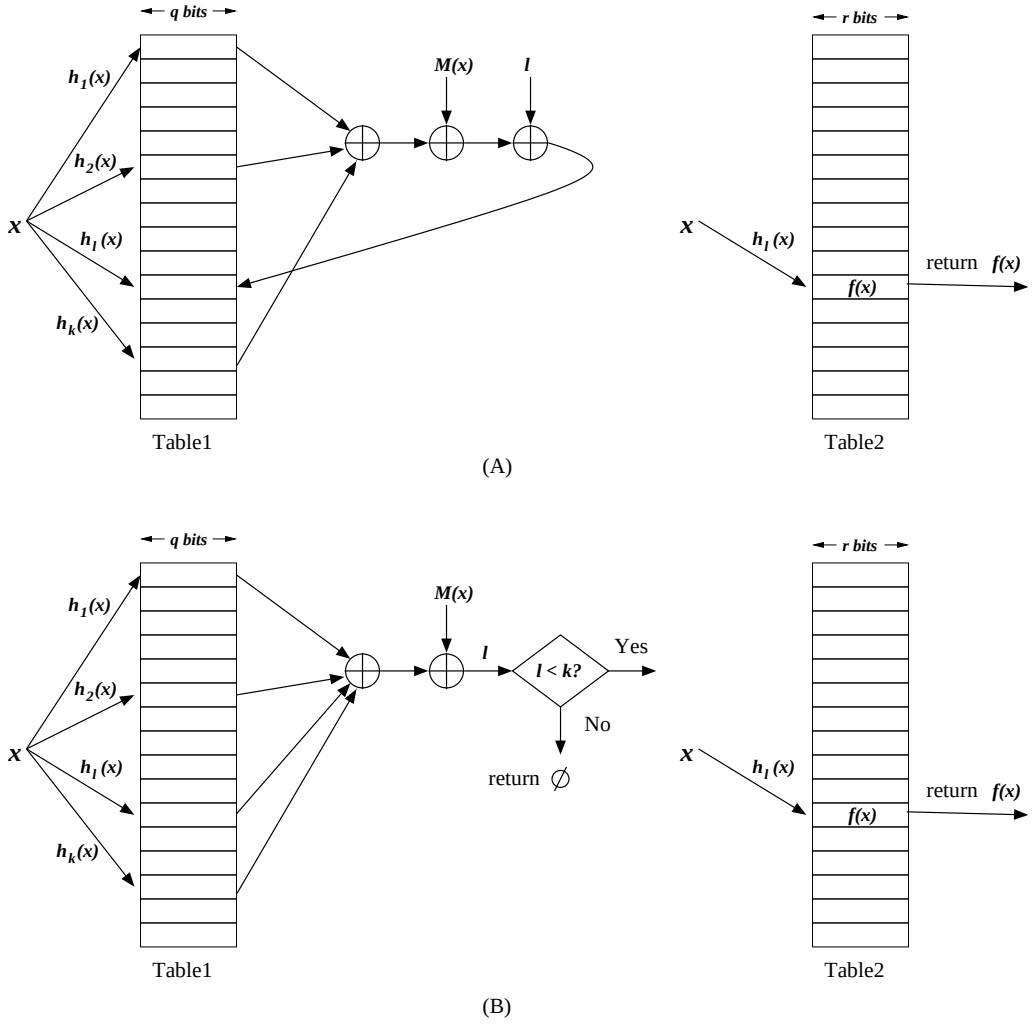


Figure 2.3: **Illustration of Bloomier filter**

Let $x \in D$ and let $N(x) = \{h_1(x), h_2(x), ..., h_k(x)\}$ be the set of hash values calculated on x by the k hash functions. $N(x)$ is called the neighborhood of x. Given set S, let Π denote a permutation on S. Let $x >_\Pi y$ denote y precedes x in Π. The hash functions h_i need to satisfy the following property: for any x and y having $x >_\Pi y$, there should be at least one h_i so that $h_i(x) \notin N(y)$. If there are multiple such hash function indices i, then the the smallest index is chosen. Let $l(x)$ denote that index and $L(x)$ denote the corresponding hash value for x, $h_{l(x)}(x)$. It follows that for any $x \neq y$ and $x, y \in S$, $L(x) \neq L(y)$. In other words, there are unique locations in the array of m locations corresponding to each of the items in S.

The insertion and query operation of a Bloomier filter are illustrated in Figure 2.3. The *mutable* form of a Bloomier filter contains two tables: $Table_1$ in which each entry is q bits wide and $Table_2$ in which each entry is r bits wide. $Table_2$ contains the actual value associated with an item in S. The values in $Table_2$ can be changed, however, the set S can not be changed dynamically. Items are inserted in the order specified by Π described before. To insert an item, x, all the values at the hashed locations of $Table_1$ corresponding to x except the one at the unique location, $L(x)$, are read and XORed. Then, this result is XORed with a random q-bit value, $M(x)$, specific to x. Finally, this result is XORed with $l(x)$ and the final result is kept in the location $L(x)$. Note that the final result is essentially an encoding of $l(x)$. The actual value associated with x, $f(x)$, is kept in $Table_2$ at $L(x)$. Thus,

$$\begin{aligned} Table_1[L(x)] &= M(x) \oplus l(x) \left(\bigoplus_{\substack{i=1\ldots k \\ i \neq l(x)}} Table_1[h_i(x)] \right) \\ Table_2[L(x)] &= f(x) \end{aligned} \tag{2.5}$$

Note that writing a value at $L(x)$ does not disturb any values already written for any y that precedes x in the permutation because $L(x) \notin N(y)$ for all such y. Hence, this value would never contribute to the values set for earlier items.

When the item is to be looked up, all the values kept in the hash locations of $Table_1$ corresponding to the query item are XORed. The result is then XORed with $M(x)$. If the item was stored, then the final result should be a decoded $l(x)$.

$$\begin{aligned} result &= \left(\bigoplus_{i=1\ldots k} Table_1[h_i(x)] \right) \oplus M(x) \\ &= \left(\bigoplus_{\substack{i=1\ldots k \\ i \neq l(x)}} Table_1[h_i(x)] \right) \oplus Table_1[L(x)] \oplus M(x) \end{aligned}$$

$$= \left(\bigoplus_{\substack{i=1\ldots k \\ i \neq l(x)}} Table_1[h_i(x)] \right) \oplus M(x) \oplus l(x) \left(\bigoplus_{\substack{i=1\ldots k \\ i \neq l(x)}} Table_1[h_i(x)] \oplus M(x) \right)$$
$$= l(x) \quad (2.6)$$

If the item is not in the table, then with a very high probability the result will be a $q-$bit random number since $M(x)$ is a q-bit random number which randomizes the entire result after XORing. In the true positive case, it gets cancelled. We know that each $l(x)$ for all x is a number between 1 and k. If the result of the lookup is $> k$, then it is clearly a mismatch. Therefore, if $x \notin S$ and $l(x) \leq k$, then we have a false positive. The probability of this false positive is simply $\frac{k}{2^q}$. Thus, choosing a larger q, the probability can be made very small. Once $l(x)$ is obtained, we simply look up $Table_2$ to retrieve the associated value, $Table_2[L(x)]$ (remember that $h_{l(x)}(x) = L(x)$).

The challenge in constructing a Bloomier filter is to find the permutation Π and the hash functions $h_i()$ which satisfy the required properties. Challez et. al. give a greedy algorithm for the same and analyze its complexity.

2.6 Compressed Bloom Filters

A compressed Bloom filter was proposed by Mitzenmachar [25]. If a Bloom filter is used to encode a set of items and transmitted over the network to another entity, then reducing the size of the transmitted data is useful. A compressed Bloom filter is designed with this motivation. Given a set of n items and a size of z bits *after* compression, what is the best way to choose m, the original size of the Bloom filter and k, the number of hash functions, to minimize the false positive probability f? Moreover, z must respect the information theoretic bounds $z \geq mH(p)$ where $H(p)$ is the entropy function and p is the probability of a bit being set in the filter.

A traditional Bloom filter with optimal parameters gives the worst compression since it has maximum entropy. On the other hand, a filter with a large m and just one hash function can give a significant compression as well as a very small false positive probability. However, we can not have a very large m because the uncompressed form

of the filter needs to be practical. Therefore, a compressed Bloom filter needs to be designed with some practical limitations on m. Theoretical analysis and examples are given in [25]. An example indicates that if the original Bloom filter bits per item, m/n, is 92, and the number of hash functions is $k = 1$, then this filter can be compressed to $z/n = 8$ bits per item. This filter shows a false positive probability of 0.0108 in its original form.

Chapter 3

Longest Prefix Matching

3.1 Introduction

In the Longest Prefix Matching (LPM) problem, we are given an input key I with length W bits and a table that contains keys with variable lengths from 1 to W. We are required to find the longest matching prefix of I in the table. LPM has received significant attention in the networking community due to its important role in routing packets in the Internet. A router maintains a table of Internet Protocol (IP) address prefixes. Each of these prefixes represents a set of IP addresses sharing the same prefix. Therefore, each prefix represents a sub network in the Internet. When an IP packet arrives, a router searches for the longest matching prefix of the destination address of the packet. When such a prefix is found, the router forwards the packet to the next hop associated with this prefix. The rate at which a router can forward packets directly depends on how fast it can perform LPM. Hence, efficient lookup techniques are important for high-speed packet processing.

Several LPM algorithms have been developed by the research community. A naïve way to perform LPM is by constructing a radix trie with the given set of prefixes and traversing it with the help of the IP address bits in a packet. This simple algorithm is slow because it can require W memory accesses in the worst case. Several improvements to this naïve data structure have been proposed. We will review the prominent algorithmic techniques in the next section.

3.2 Related Work

The Lulea scheme [15] first converts the trie into a *complete* prefix trie so that each node has either two children or no children. This is achieved with a technique known as *leaf pushing* in which the information associated with a node is pushed to the children of that node. The resulting trie is *expanded* with stride lengths of 16, 8, and 8. The expanded trie is represented using three levels of data structures containing bitmaps which can be indexed using the first 16 bits, next 8 bits and the last 8 bits of an IP address prefix. At each level, a set bit indicates that the search either terminates there in which case the next hop information is retrieved or the search continues further so that the pointer to the next level bitmap is retrieved. If the bit is not set then the search terminates there and the next hop associated with the nearest ancestor of the current node is used to forward the packet. The authors use several techniques for reducing the memory requirement. This algorithm, although faster than the basic radix tree, does not match the performance of TCAM. It is primarily geared towards a software implementation. The incremental updates to the prefix set are difficult due to the trie expansion involved in the algorithm.

Waldvogel et. al. introduced a technique based on hash tables that performs a binary search on prefix lengths [40]. Assume that the prefixes of length i are kept in hash table i. For IPv4, we would have 32 hash tables in the worst case. Given the range of prefix lengths, the algorithm looks up the hash table of the middle prefix length (Hash table 16 for IPv4) with the corresponding number of IP prefix bits. If the match is found, the binary search continues in the hash tables on the right half, otherwise, it continues in the left half. Thus, if a match is found in hash table 16, then hash table 24 is probed with 24 prefix bits. Otherwise, hash table 8 is probed with 8 prefix bits. However, this scheme won't work without modification to the prefix set. For instance, let's assume that the longest matching prefix of an IP address is 9-bits long, 100110100*. When the search is conducted on the left half of the prefix lengths, the hash table 8 would be looked up. To get the match for our 9-bit prefix, we must get a match for the 8-bit prefix so that the search is guided to the hash tables between 8 and 15. If there is no prefix 10011010* in hash table 8, then it will be a mismatch and the result will be incorrect. Hence, a prefix 10011010*, called *marker*, is artificially added to the hash table 8 in order to preserve the correctness. By adding markers

to the hash tables whenever necessary, the correct longest matching can always be found. Binary search on prefix length requires $\log_2 W$ hash table accesses in the worst case which is 5 for IPv4 and 7 for IPv6. Although attractive from the worst case performance perspective, this scheme makes the incremental updates to the data structure difficult due to the requirement of markers. The average performance does not match the performance of a TCAM.

Srinivasan et. al. introduced another technique called Controlled Prefix Expansion which is similar to the Lulea scheme [32]. All the prefixes are converted into a set of fixed length prefixes by expanding a shorter prefix to multiple longer prefixes of a predetermined length. Consider a 2-bit prefix 10* which can be expanded to a set of 4-bit prefixes 1000*, 1001*, 1010*, 1011*. Specific prefix lengths can be picked and any prefix with shorter length can be expanded to multiple prefixes of the next predetermined longer length. The authors give an algorithm to pick the most optimal prefix lengths to expand the prefix sets. After the expansion, various LPM algorithms can be used. The authors illustrate how a multi-bit trie can be used, similar to the one used in Lulea. Binary search on prefix length algorithm can also be used. Due to the reduction in the distinct prefix lengths, the search speed improves.

Using a multi-bit trie, multiple bits of an IP address can be matched at a time. Each node in a multi-bit trie represents a set of nodes in the radix trie. The tree bitmap technique proposed by Eatherton et. al. uses a compressed representation of multi-bit trie [18]. A radix trie of depth i has 2^i-1 nodes and 2^i children. Eatherton's technique uses two bit maps to encode the 2^i-1 nodes of a radix trie in a single *super-node*: an "internal bitmap" that contains 2^i-1 bits and an "external bitmap" containing 2^i bits. A bit in the internal bitmap indicates if the corresponding node in the radix trie represents a valid prefix. A bit in the external bitmap indicates if the corresponding child exists. If a child exists then the search continues. All the children super-nodes of each super-node are arranged in consecutive memory slots and only the pointer to the first child is maintained. Using the knowledge from the external bitmap, the exact child super-node can be accessed by adding the corresponding offset in the base pointer. This technique saves a significant amount of memory but complicates the memory management and incremental updates. Overall, the compressed multi-bit trie is a fast and memory efficient technique but can not beat the performance of TCAM due to multiple dependent memory accesses for each lookup.

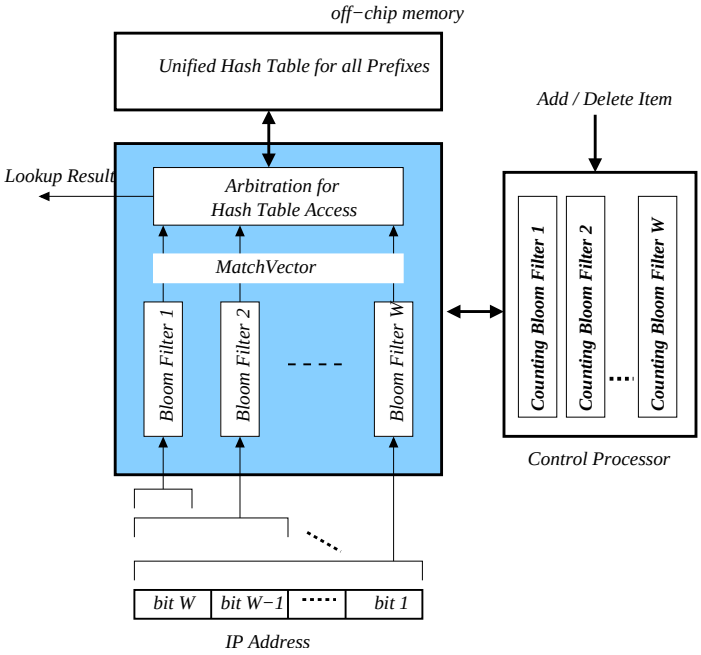


Figure 3.1: **Basic configuration of Longest Prefix Matching using Bloom filters.**

3.3 Our Approach

A hash table can be used to find the longest matching prefix of an IP address. We can construct a hash table from all the prefixes and with each prefix keep the associated next hop information. Given an IP address, we can test for the presence of each possible prefix in the hash table, starting from the longest. We stop when a matching prefix is found and the next hop information is retrieved. Constructing hash tables to minimize collisions with reasonable amounts of memory is well-studied. For our purpose, we assume that probing a hash table stored in an off-chip memory requires one memory access [40]. Therefore, this naïve approach would require W hash table lookups (and as many off-chip memory accesses) in the worst case, where W is the length of the IP address. However, we can reduce the memory accesses to almost a single access by using Bloom filters implemented in *on-chip* memory. The basic configuration of our system is shown in Figure 3.1.

We group prefixes according to length and define a Bloom filter for each set. We maintain W parallel Bloom filters in the embedded memory with each filter corresponding to a unique prefix length. There is no need to maintain a Bloom filter

corresponding to a prefix length for which there are no prefixes. Before querying a prefix in the hash table, we check to see if it is present in the corresponding Bloom filter. Only if the Bloom filter identifies a match do we proceed to query the hash table. A Bloom filter can produce a false positive match but never a false negative. If the match was a false positive, then it would be discovered after the hash table probe results in an unsuccessful search. In case of a false positive, we simply perform an extra memory access and proceed to check the next prefix showing a match in the corresponding Bloom filter[1].

Due to the parallelism offered by embedded memories, all the Bloom filter lookups can be performed concurrently. A carefully designed Bloom filter gives a lookup result in a single cycle of the system clock. We create a *MatchVector* with W bits in which each bit is set by a Bloom filter if the corresponding prefix query shows a match. Then, we use a priority encoder to walk through the MatchVector from the longest to the shortest prefix and execute the hash table lookups for the matching prefixes. The pseudo-code in Figure 3.2 formally describes the Longest Prefix Matching on an IP address I:

```
LPM (I)
1.    for (j = W downto 1)
2.        MatchVector[j]← BFQuery_j(I_j)
3.    for (j = W downto 1)
4.        if (MatchVector[j] = true)
5.            {prefix, NextHop} ← HashTableLookup(I_j)
6.            if (prefix = I_j) return {prefix, NextHop}
7.    return {NULL, DefaultHop}
```

Figure 3.2: **Pseudo-code for the LPM algorithm**

I_j denotes the j bit prefix of I and BFQuery_j denotes the process of querying the Bloom filter j. The loop of lines 1-2 is executed in parallel. By designing Bloom filters to exhibit a very small false positive probability (explained in section 3.3.2), we can ensure that only the hash table lookup corresponding to the longest matching prefix is performed with a very high probability.

[1]The technique of using a hash table to resolve Bloom filter false positives was jointly developed by Praveen Krishnamurthy and Sarang Dharmapurikar

3.3.1 Supporting Incremental Updates

We can support efficient addition and deletion of prefixes to the prefix table using an augmented data structure. As mentioned, ordinary Bloom filters are not sufficient to support deletion of prefixes. Hence, corresponding to each Bloom filter we maintain a counting Bloom filter. Because a counting Bloom filter consumes more memory than an ordinary Bloom filter, keeping it in the on-chip memory is not cost-effective. At the same time we note that addition and deletion of routes is relatively infrequent compared to actual lookup process and need not be performed at the same speed. Therefore, we keep the counting Bloom filters in another off-chip memory with the control processor responsible for updates (See Figure 3.1).

When adding a new prefix to the set, it is first inserted in the counting Bloom filter corresponding to its length. During the insertion, if any counter value changes from zero to one, then we set the corresponding bit of the associated on-chip Bloom filter. When a modified counter is non-zero before addition, then the corresponding bit of the associated Bloom filter would already be set and no modification is required. After modifying Bloom filters, we need to add the {prefix, NextHop} pair in the hash table.

Likewise, when we want to delete an existing prefix from the set, we first delete it from counting Bloom filters (i.e. decrement the corresponding counters). If a counter changes its value from one to zero, then the corresponding on-chip Bloom filter is updated by resetting the bit. Also, the prefix is deleted from the hash table which incurs another memory access. The pseudo-code in Figure 3.3 describes addition and deletion of prefixes.

AddPrefix (I_j, NextHop)
1. **for** ($i = 1$ **to** k)
2. Counter$_j[h_i(I_j)]$++
3. **if** (Counter$_j[h_i(I_j)]$= 1) Vector$_j[h_i(I_j)] \leftarrow 1$
4. InsertHashTable($\{I_j$, NextHop$\}$)

Figure 3.3a: **Pseudo-code for adding a prefix**

DeletePrefix(I_j)
1. **for** ($i = 1$ **to** k)
2. Counter$_j[h_i(I_j)]$−−
3. **if** (Counter$_j[h_i(I_j)]$= 0) Vector$_j[h_i(I_j)] \leftarrow 0$
4. DeleteHashTable(I_j)

Figure 3.3b: **Pseudo-code for deleting a prefix**

The memory operations involved in an add or a delete operation are: (1) k memory accesses to increment/decrement the counter, (2) at the most k *parallel* memory accesses to update the associated Bloom filter, and (3) one more memory accesses to insert/delete the prefix in the hash table.

3.3.2 Analysis

We can show the relationship between false positive probability of Bloom filters and its effect on the throughput of the system. We measure the throughput of the system as a function of the average number of memory accesses per lookup. As described in the LPM algorithm, we perform a hash table probe for prefix i only if Bloom filter i shows a match. The following notations will be used:

- W : number of Bloom filters in the system; we assume it to be equal to the length of the IP address.
- n_i : number of prefixes of length i in the table (items in Bloom filter i)
- N : total number of prefixes in the table, $\sum n_i$ for all i
- m_i : number of bits allocated to Bloom filter i
- M : total on-chip memory available to construct Bloom filters, $\sum m_i$ for all i
- f_i : false positive probability of Bloom filter i
- k_i : number of hash functions used in Bloom filter i

- p_i : probability that the prefix of length i being inspected indeed belongs to the set of prefixes of length i (in other words successful search probability of hash table i or true positive probability of Bloom filter i)
- t_s : average memory accesses required for a successful search in any hash table
- t_u : average memory accesses required for an unsuccessful search in any hash table
- T_i : cumulative memory accesses required from Bloom filter i down to Bloom filter 1

We can express the cumulative memory accesses T_i from Bloom filter i using the recursion:

$$T_i = p_i t_s + (1 - p_i)(f_i t_u + T_{i-1}) \text{ for } i = W \text{ downto } 1 \tag{3.1}$$

This equation essentially illustrates that starting from i^{th} Bloom filter, we execute a successful search in a hash table with probability p_i and the memory accesses required are $p_i t_s$. Otherwise, with probability $(1 - p_i)$, we get a false positive match requiring $f_i t_u$ memory accesses. In case of a false positive, we proceed to the next filter result and repeat the procedure which will result in accesses T_{i-1}. This equation can be simplified by making certain assumptions. To begin, we assume that $t_s = t_u = 1$. This can be achieved in a carefully constructed hash table. Hence,

$$T_i = p_i + (1 - p_i)(f_i + T_{i-1}) \text{ for } i = W \text{ downto } 1 \tag{3.2}$$

It is difficult to assume the true match probabilities for any particular prefix length. However, by profiling and observing, we can determine the values of p_i for each prefix length i. An interesting question arises: Given p_i, what values of f_i would minimize T_i subject to the constraint $\sum m_i \leq M$? Intuitively, if the prefixes of length i are the longest matching prefixes most often (i.e. high p_i), then our search ends at the i^{th} Bloom filter and it is acceptable to have Bloom filters with large false positive probability beyond filter i since they are rarely used. Therefore, we can allocate less

memory to those Bloom filters. On the other hand, since the search ends at i^{th} filter, we should not have any false memory accesses and can allocate more memory to the Bloom filters before the i^{th} to reduce the false positive probability. Therefore, given the true match probabilities of the filters, a fixed amount of memory can be allocated optimally to the filters to reduce overall memory accesses. We have discussed this trade-off since it can be useful when true match probabilities are known. However, we do not explore it for the IP lookup application since no assumptions can be made regarding true match probabilities.

To simplify the equation further, we prove the following theorem.

Theorem 1: $T_i \leq \sum_{j=2}^{i} f_j + T_1$ for $i \geq 2$

proof: The proof is by induction on i using Equation 3.2.

For $i = 2$,

$$T_2 = p_2 + (1 - p_2)(f_2 + T_1) \leq f_2 + T_1$$

For $i = 3$,

$$T_3 = p_3 + (1 - p_3)(f_3 + T_2) \leq p_3 + (1 - p_3)(f_3 + f_2 + T_1) \leq f_3 + f_2 + T_1$$

Now assume that the result holds for j,

$$T_j \leq \sum_{l=2}^{j} f_l + T_1$$

Hence,

$$\begin{aligned} T_{j+1} &= p_{j+1} + (1 - p_{j+1})(f_{j+1} + T_j) \leq p_{j+1} + (1 - p_{j+1})(f_{j+1} + (\sum_{l=2}^{j} f_l + T_1)) \\ &\leq (\sum_{l=2}^{j+1} f_l + T_1) \end{aligned}$$

Hence the proof. •

This theorem gives a pessimistic average bound on the performance. It essentially illustrates that the cumulative *average* search time required from the i^{th} filter onwards is the worst when there is no true match in the first $B-1$ filters and memory accesses must be executed due to the false positives of these filters each with probability f_i.

Furthermore, since $T_1 \leq 1$, we get the following equation.

$$T_W \leq \sum_{j=2}^{W} f_j + 1 \tag{3.3}$$

In order to simplify this analysis, we assume that all Bloom filters are optimally tuned and obey the equation

$$k_i = \frac{m_i \ln 2}{n_i} \tag{3.4}$$

Therefore,

$$f_i = \left(\frac{1}{2}\right)^{\left(\frac{m_i}{n_i}\right) \ln 2} \tag{3.5}$$

This assumption is optimistic since the value of k_i can be a fraction. However, in practice, k_i is always an integer and achieving optimum is not always possible. Given a m_i and n_i, k_i can be calculated with this equation and the value can be rounded to the nearest integer; larger or smaller depending on which results in a lower false positive probability for the filter. The resulting configuration does not deviate much from the optimal configuration.

Furthermore, we tune all the optimal Bloom filters to exhibit the same false positive rate, f. Therefore, all of the Bloom filters use the same number of hash functions.

$$f_i = f = \left(\frac{1}{2}\right)^{\left(\frac{m_i}{n_i}\right) \ln 2} \forall i \in [1 \ldots W] \tag{3.6}$$

This implies that

$$\frac{m_1}{n_1} = \frac{m_2}{n_2} = ... = \frac{m_B}{n_B} = \frac{\sum m_i}{\sum n_i} = \frac{M}{N} \tag{3.7}$$

In other words, each Bloom filter is allocated a memory segment from the available M bits that is proportional to its share of prefixes in the total. Ideally, it is possible

to allocate memory in this fashion but in practice it is difficult. Usually embedded memory is allocated in blocks of bits as opposed to a single bit. However, by slightly over-allocating the memory to meet the block size requirement, nearly the same performance can be maintained.

With this assumption, the false positive probability f_i for a given filter i may be expressed as

$$f_i = f = \left(\frac{1}{2}\right)^{\left(\frac{M}{N}\right)\ln 2} \tag{3.8}$$

Let τ_{avg1} denote the average number of memory accesses per address lookup for this basic configuration (1 in the subscript is for the first scheme). Finally, from Equation 3.3,

$$\tau_{avg1} = T_W = (W-1)f + 1 = 31\left(\frac{1}{2}\right)^{\left(\frac{M}{N}\right)\ln 2} + 1 \tag{3.9}$$

With a moderately large value of $M \ln 2/N$, the factor $31\left(\frac{1}{2}\right)^{\left(\frac{M}{N}\right)\ln 2}$ becomes very small when compared to 1. The average number of memory accesses approaches 1. While the average performance of our algorithm is appealing, the worst case performance is poor. Each Bloom filter could then show a false match and force the corresponding memory access to be performed. We denote the worst case memory accesses for this configuration as τ_{worst1}. Hence,

$$\tau_{worst1} = W = 32 \tag{3.10}$$

We now plot the average memory accesses as a function of the available on-chip memory M for different values of table sizes, N. The performance is shown in Figure 3.4.

Figure 3.4 shows that with more memory, the false positive probabilities of Bloom filters decrease exponentially. As a result, the average number of memory accesses per lookup decrease exponentially. With 2MB of on-chip memory, we can support 250,000 prefixes with each lookup requiring less than two memory accesses. If implemented using a commodity SRAM chip operating at 333 MHz frequency, then this system can

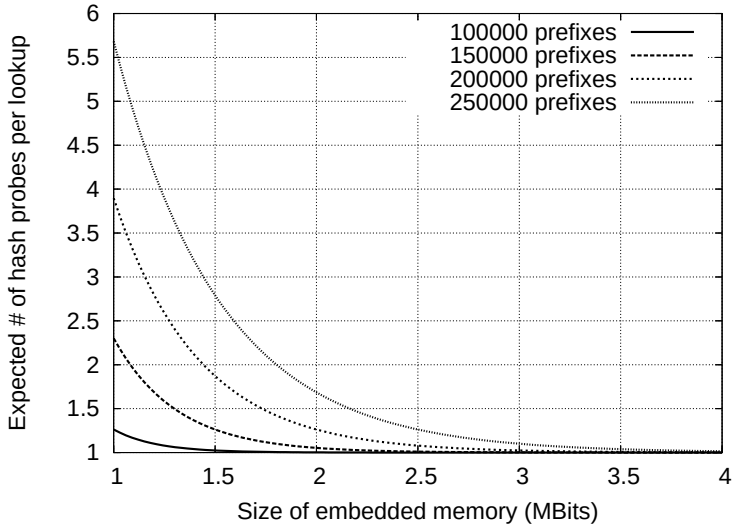


Figure 3.4: **The average number of hash probes per lookup, τ_{avg1}, versus the total embedded memory size, M, for various values of total prefixes, N, using a basic configuration for IPv4 with 32 Bloom filters.**

perform 166 million lookups per second. In the next section, we describe optimizations to this basic configuration that reduce the worst case memory accesses.

3.4 Optimizations

The worst case memory accesses depend on the number of Bloom filters. Therefore, the number of filters must be reduced to improve the performance. We describe two enhancements to this basic configuration to reduce the number of filters. The first enhancement uses a direct indexing array for prefixes of lengths below a certain length. The second enhancement uses Controlled Prefix Expansion (CPE) [32] to reduce the number of distinct prefix lengths and Bloom filters[2].

[2]These optimizations were suggested by David Taylor

3.4.1 Direct Indexing

Lookups for short prefix lengths can be easily done by using the prefix as the address to index into an array. The index location stores the next hop information. To lookup a prefix of length i bits, we need to maintain an array of 2^i bits. Indeed, if the memory were plentiful, then we could maintain an array for all possible values in an IP address, 2^W. This becomes expensive and impractical for large W such as 32 for IPv4 and beyond the technological reach for $W = 128$ for IPv6. However, direct indexing can still be used for prefixes of short lengths such as up to 20 bits since they can be accommodated by a million array locations. Supporting a million entries is certainly feasible with today's SRAM. A straightforward indexing scheme would require an array for each prefix length. For instance, we would require a 2-bit array for prefix length 1, 4-bit array for prefix length 2, 8-bit array for prefix length 3, and so on. This requires separate memory blocks to keep these arrays and allow parallel lookups. However, the requirement for individual arrays can be avoided easily if we expand all the shorter prefixes to a fixed prefix length and use a single array for it. For instance, if we decide to use a direct lookup array for prefixes of length up to 8, then given a prefix with a shorter length, say 101*, we need to expand it by enumerating all the prefixes from 10100000* to 10111111* and associate the same forwarding information to each of these prefixes. This certainly increases the number of prefixes in the table but does not require more memory since the direct lookup array would have space for all the 2^8 possible prefixes. Therefore, expanding any number of prefixes with any prefix length shorter than 8 bits will not require more than 2^8 entries. It should be noted that two prefixes of different lengths when expanded to this upper limit might need to share the same location. For instance, if with the prefix 101* there is another prefix 1010* in the table, then they will share the same set of entries ranging from 10100000 to 10101111. In that case, the forwarding information of the longer original prefix, 1010*, will be associated with all the resulting entries.

In practice, it is feasible to use a direct lookup array for prefixes up to length 20. An array of $2^{20} = 1M$ locations can be used with each location containing the next hop information. Assuming that the forwarding information is an IP address requiring four bytes, the direct lookup array needs 4MB of space. Commodity SRAM chips

can be used to implement this array. A drawback is that the incremental updates take longer since multiple array entries need to be populated or deleted for inserting or deleting a shorter prefix.

After using the direct lookup for prefix lengths 20 or less, we are left with only 12 Bloom filters corresponding to prefix lengths from 21 to 32. Hence, the average memory accesses per lookup are

$$\tau_{avg2} = 12f + 1 = 12\left(\frac{1}{2}\right)^{\left(\frac{M \ln 2}{N - \sum_{i=1}^{20} n_i}\right)} + 1 \tag{3.11}$$

and the worst case is reduced to

$$\tau_{worst2} = 13 \tag{3.12}$$

Thus, at the cost of using more off-chip memory for direct indexing array, we can reduce both the average and worst case lookup performance. For the same amount of on-chip memory, this hybrid scheme exhibits better performance since the memory released by the 20 Bloom filters can now be used for the remaining 12 Bloom filters to reduce their false positive rates. In order to evaluate the performance of our system after this optimization, a specific prefix length distribution must be taken into account. We collected 15 IPv4 BGP tables from [4]. The average prefix length distribution for these BGP tables is shown in Figure 3.5.

From the distribution analysis, it was discovered that 24.6% of the prefixes range from 1 to 20 bits. With the direct lookup array for these prefixes, a significant portion of the table is eliminated from the Bloom filters. By substituting $\sum_{i=1}^{20} n_i = 0.246N$ in Equation 3.11, the resulting performance can be evaluated numerically as plotted in Figure 3.6. A comparison of Figure 3.4 and Figure 3.6 shows that the average memory accesses are reduced significantly for the same amount of memory with the direct lookup array.

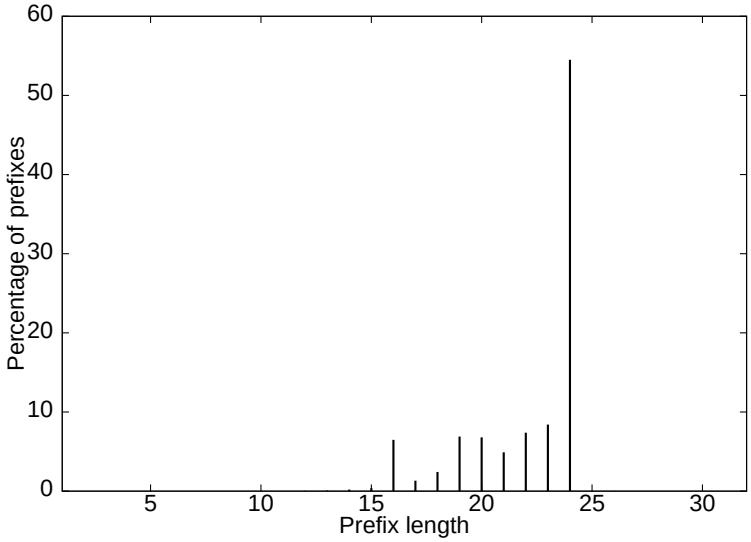


Figure 3.5: **Average prefix length distribution for IPv4 BGP table snapshots.**

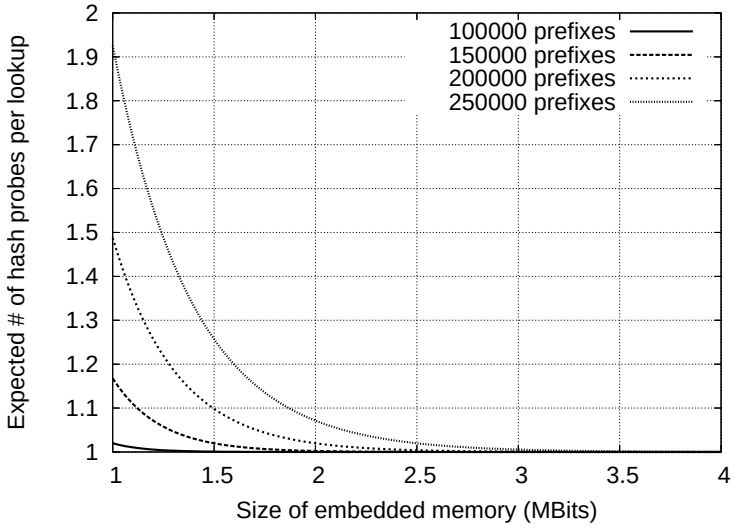


Figure 3.6: **Average number of hash probes per lookup, τ_{avg2}, versus total embedded memory size, M, for various values of total prefixes, N, using a direct lookup array for prefix lengths $1 \ldots 20$ and 12 Bloom filters for prefix lengths $21 \ldots 32$**

3.4.2 Controlled Prefix Expansion

More Bloom filters can be further eliminated by CPE. An analysis of the prefix tables shows that there are fewer prefixes of lengths 21- to 23-bits than 24-bit prefixes. Also, there are very few prefixes of lengths 25 to 31. Therefore, all the 21- to 23-bit prefixes

can be expanded to 24-bit prefixes and all the 25- to 31-bit prefixes to 32-bit prefixes. While this expansion results in more prefixes, the number of Bloom filters can now be reduced from 12 to 2, one corresponding to 24 bit prefixes and the other to 32-bit prefixes. Let α denote the expansion factor for prefixes of lengths 21 to 24: the ratio of total prefixes after expansion to the total prefixes before expansion. Likewise, let β denote the expansion factor for 25-32 bit prefixes. Our experiments with real prefix tables show that $\alpha \approx 1.8$ and $\beta \approx 50$. Although β is large, the overall number of prefixes of lengths 25 to 32 is very small and the expansion is tolerable. With CPE, the new equations are

$$\tau_{avg3} = 2f + 1 = 2\left(\frac{1}{2}\right)^{\left(\frac{M \ln 2}{N - \sum_{i=1}^{20} n_i + \alpha \sum_{i=21}^{23} n_i + \beta \sum_{i=25}^{31} n_i}\right)} + 1 \tag{3.13}$$

$$\tau_{worst3} = 3 \tag{3.14}$$

The performance is plotted in Figure 3.7. The expansion overhead increases the average memory accesses slightly since there are more prefixes to be handled using the same amount of memory. At the same time, great savings are achieved in the worst case memory accesses. At most, there are two Bloom filter matches and a final direct lookup access. Again, the CPE increases the cost of incremental updates. Any prefix with a length between 21 to 23 or 25 to 31 now needs to be expanded into multiple 24-bit or 32-bit prefixes and requires as many insertions in the table.

3.5 Performance Simulations

Thus far we have described three system configurations, each offering an improvement over the earlier in terms of worst case performance. In this section, we present simulation results for each configuration using forwarding tables constructed from real IPv4 BGP tables.

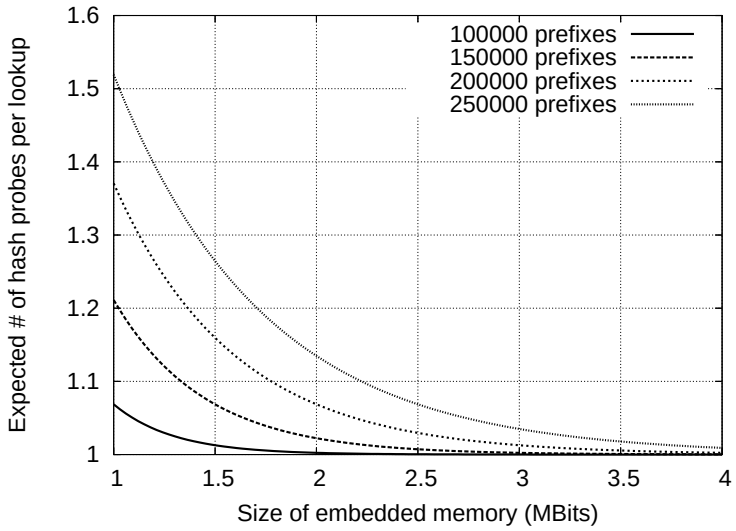


Figure 3.7: **Average number of hash probes per lookup, τ_{avg3}, versus total embedded memory size, M, for various values of total prefixes, N, using a direct lookup array for prefix lengths $1 \ldots 20$ and two Bloom filters for prefix lengths $21 \ldots 24$ and $25 \ldots 32$**

For all schemes we set M = 2 Mb and adjusted m_i for each asymmetric Bloom filter according to the distribution of prefixes in the database being tested. For each combination of database and system configuration, we computed the theoretical value of τ_{avg1}, τ_{avg2}, and τ_{avg3}.

A simulation was performed for every combination of database and system configuration. The ANSI C function *drand48()* was used to generate hash values for the Bloom filters as well as the prefix hash tables. The collisions in the prefix hash tables were approximately 0.8% which are negligibly small. In order to investigate the effects of input addresses on system performance, various traffic patterns were used. They varied from completely random addresses to only the addresses with a valid prefix in the database under test. In the latter case, the IP addresses were generated in proportion to the prefix distribution; thus, IP addresses corresponding to a 24-bit prefix in the database dominated the input traffic. One million IP addresses were applied for each test run. The average number of hash probes per lookup from the test runs with each database on the three system configurations and their corresponding theoretical values are shown in Table 3.1. The maximum number of memory accesses (hash probes and direct lookup) per lookup was recorded for each test run of all the

schemes. While the theoretical worst case memory accesses per lookup for Configuration 1 and Configuration 2 are 32 and 13, respectively, the worst observed lookups required less than four memory accesses in all test runs. For Configuration 3, the worst observed lookups required three memory accesses in most test runs.

Using Configuration 3, the average hash probes per lookup in all test databases was found to be 1.003 corresponding to a lookup rate of about 332 million lookups per second with a commodity SRAM device operating at 333 MHz. This is a speedup of 3.3x over state-of-the-art TCAM-based solutions. At the same time, the scheme has a worst case performance of 2 hash probes and one array access per lookup. Assuming that the array is stored in the same memory device as the tables, worst case performance is 110 million lookups per second, which exceeds current TCAM performance.

Note that the average hash probes per lookup in simulations generally agree with the values predicted by the equations. We now provide a direct comparison between theoretical performance and observed performance. To see the effect of total embedded memory resources, M, for Bloom filters, Configuration 3 was simulated on the first database with $N = 116189$ prefixes for values of M between 500kb and 4Mb. Figure 3.8 shows theoretical and observed values for the average number of hash probes per lookup for each value of M. Simulation results show slightly better performance than the corresponding theoretical prediction since theoretical values are **pessimistic** average values.

3.6 Implementation Considerations

There are relevant implementation issues when targeting our approach to hardware. The two most important issues are: (1) supporting variable prefix length distributions and (2) supporting multiple hash probes to the embedded memory.

From previous discussions, it is clear that Bloom filters designed to suit a particular prefix length distribution perform better. However, an ASIC design optimized for a particular prefix length distribution will have sub-optimal performance if the distribution varies drastically. This can happen even if the new distribution requires the

Table 3.1: **Observed average number of hash probes per lookup for 15 IPv4 BGP tables on various system configurations dimensioned with M = 2Mb.**

		τ_{avg1}		τ_{avg2}		τ_{avg3}	
set	Prefixes	Theoretical	Observed	Theoretical	Observed	Theoretical	Observed
1	116,819	1.008567	1.008047	1.000226	1.000950	1.005404	1.003227
2	101,707	1.002524	1.005545	1.000025	1.000777	1.002246	1.001573
3	102,135	1.002626	1.005826	1.000026	1.000793	1.002298	1.001684
4	104,968	1.003385	1.006840	1.000089	1.000734	1.004443	1.003020
5	110,678	1.005428	1.004978	1.000100	1.000687	1.003104	1.000651
6	116,757	1.008529	1.006792	1.000231	1.000797	1.004334	1.000831
7	117,058	1.008712	1.007347	1.000237	1.000854	1.008014	1.004946
8	119,326	1.010183	1.009998	1.000297	1.001173	1.012303	1.007333
9	119,503	1.010305	1.009138	1.000303	1.001079	1.008529	1.005397
10	120,082	1.010712	1.009560	1.000329	1.001099	1.016904	1.010076
11	117,211	1.008806	1.007218	1.000239	1.000819	1.004494	1.002730
12	117,062	1.008714	1.006885	1.000235	1.000803	1.004439	1.000837
13	117,346	1.008889	1.006843	1.000244	1.000844	1.004515	1.000835
14	117,322	1.008874	1.008430	1.000240	1.001117	1.004525	1.003111
15	117,199	1.008798	1.007415	1.000239	1.000956	1.004526	1.002730
Avg	**114,344**	**1.007670**	**1.007390**	**1.000204**	**1.000898**	**1.006005**	**1.003265**

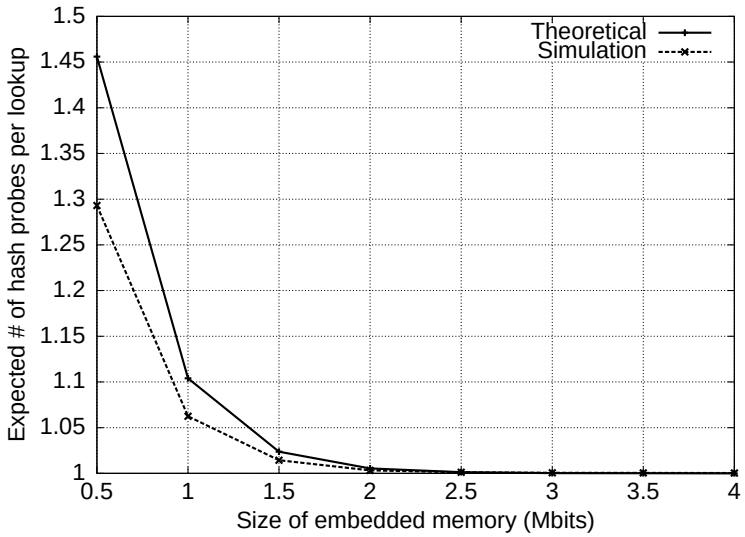


Figure 3.8: **Average number of hash probes per lookup for Configuration 3 programmed with database 1, $N = 116,819$, for three embedded memory sizes, M.**

same aggregate embedded memory resources as before the change in the distribution. In spite of the availability of the embedded memory resources, the inflexibility in allocating them to different Bloom filters can lead to poor system performance. The ability to support a lookup table of certain size, independent of the prefix length distribution, is a desirable feature of this system.

If we could allocate memory to Bloom filters in the granularity of a single bit then ideal performance could be achieved. However, such allocation is impractical in hardware. Typically, the embedded memory in VLSI chips is implemented as some fixed size blocks such as 2 Kbits or 4 Kbits. Hence, we are constrained to allocate memory to Bloom filters in the granularity of the block sizes. Alternatively, memory blocks with different sizes can be fabricated. For instance, the Altera FPGAs have a mix of different memory blocks ranging from 512 bits to 512 Kbits.

Also, memory blocks don't always support the number of access ports to lookup the k hash values. The number of hash functions, k, is essentially the lookup capacity of the memory that stores a Bloom filter. $k = 6$ implies that 6 random locations must be accessed in the time alloted for a Bloom filter query. In single cycle Bloom filter queries, on-chip memories need to support at least k reading ports. In reality, the number of ports supported by embedded memory blocks is fewer. The most common

memory blocks are dual-ported or at the most quad-ported. A Bloom filter with a given number of access ports needs to be constructed with the help of embedded memory blocks with fewer access ports.

We assert that fabrication of 6 to 8 *read ports* for an on-chip Random Access Memory is attainable with today's embedded memory technology [17]. Moreover, it should be noted that small on-chip memory blocks are always faster than the off-chip memory. Therefore, in the time that it takes to perform one off-chip lookup, multiple lookups can be performed in the on-chip memory using the same access ports. The time multiplexing on the memory ports in this fashion is equivalent to having multiple ports. This speedup depends on the size of the block — the smaller the block, the faster it is. A speed up of 2 for the embedded memory is reasonable to assume. Modern FPGAs, such as Virtex II Pro from Xilinx, contain embedded memories that can be operated at 500 MHz whereas the current SRAM chips run at about 250 MHz.

Hence, the embedded memory constructed with x ports effectively has $x\rho$ ports if there is a speedup of ρ. After taking into account the speedup, let's abstract a single embedded memory as a t port and m' bit block. We will assume that we have a pool of b such memory blocks to construct our Bloom filters.

We can use all of the b such memory blocks to construct a group of t Bloom filters each having b hash functions as follows. We connect the i^{th} hash function of each of the t Bloom filters to one of the t access ports of the i^{th} memory block as shown in Figure 3.9(A). The figure shows three memory blocks ($b = 3$); each has four ports ($t = 4$). With these blocks, four Bloom filters can be constructed. The first hash function of each Bloom filter is connected to one of the t ports of the first memory block, the second hash function of each Bloom filter is connected to one of the ports of the second memory block, and so on.

It is important to note that by connecting a hash function to one of the memory ports of a block, we are restricting the range of addresses the hash function can access. The hash function tied to the block can address the bits only in that memory block. Since the original Bloom filter allows any hash function to address any bit, the new constraint on the hash function affects the false positive probability. We will now evaluate the false positive probability of this group of t Bloom filters and

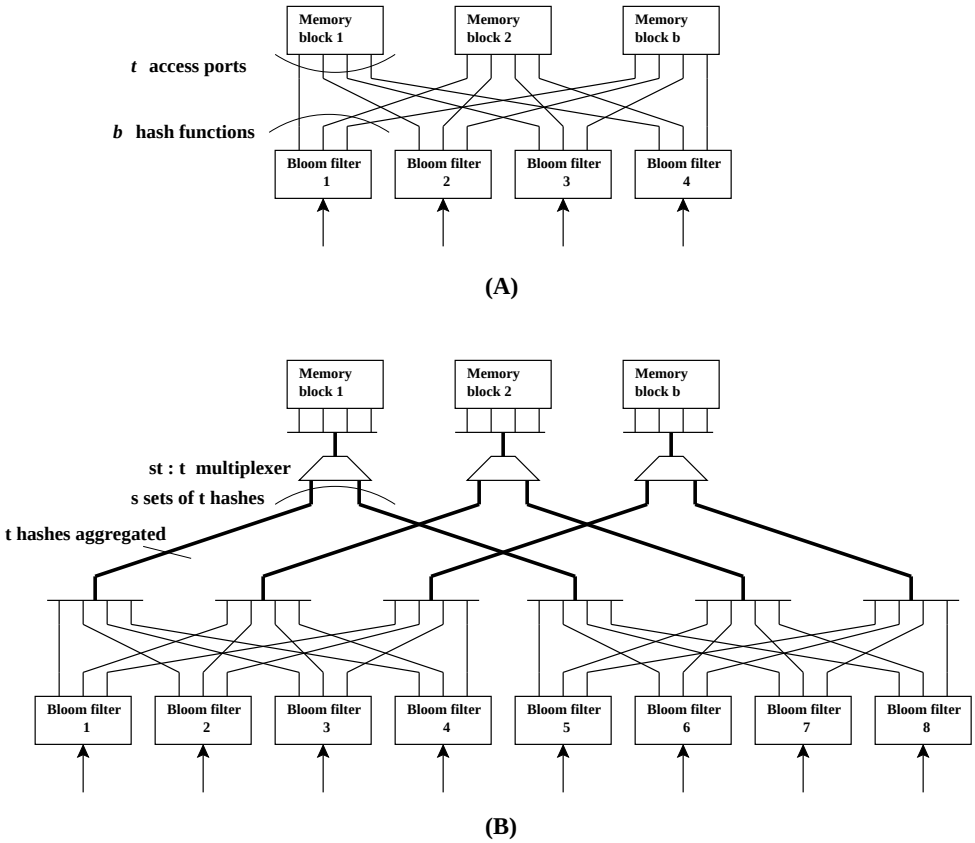


Figure 3.9: **Using fixed size memory blocks with fixed number of ports to construct multiple Bloom filters. (A) A group of t Bloom filter can be constructed using b memory blocks each with t ports. (B) Multiple sets of t Bloom filters can share the same set of b memory blocks by using the multiplexers.**

show that the increase in the false positive probability is negligibly small. Let the i^{th} Bloom filter in the group of these t Bloom filters contain n_i items. Note that each of the $\sum_{i=1}^{t} n_i$ items sets only one bit in each of the memory blocks. Therefore, using similar arguments for the derivation of the false positive probability of the original Bloom filter, the new false positive probability can be expressed as:

$$\begin{aligned} f' &= \left(1 - \left(1 - \frac{1}{m'}\right)^{(\sum_{i=1}^{t} n_i)}\right)^b \\ &\approx \left(1 - e^{-\frac{\sum_{i=1}^{t} n_i}{m'}}\right)^b \end{aligned} \tag{3.15}$$

If each of the b hash functions could access any bit in any block, like the original Bloom filter, then the false positive probability would have been:

$$\begin{aligned} f &= \left(1 - \left(1 - \frac{1}{bm'}\right)^{(b\sum_{i=1}^{t} n_i)}\right)^b \\ &\approx \left(1 - e^{-\frac{\sum_{i=1}^{t} n_i}{m'}}\right)^b \end{aligned} \tag{3.16}$$

First, the two Bloom filters have approximately the same false positive probability. By distributing the hash functions of Bloom filters across multiple memory blocks in this fashion, the false positive probability becomes oblivious to the distribution of items across individual Bloom filters and is dependent on only the total number of items. Second, the false positive probability expression involves the term b only once. As a result, there is no optimal value of f' with respect to b; the larger b is, the smaller the false positive probability. Thus, the number of ports on a memory block decides how many Bloom filters can be grouped together whereas the number of memory blocks decides the false positive probability. Third, we don't need to use all the memory blocks for a set of Bloom filter even if it is connected to all of them. We can use fewer memory blocks by simply ignoring the bit lookup result obtained from the unwanted memory blocks. For instance, in Figure 3.9(A), if the set of Bloom filters needs just the first two memory blocks to bound the false positive probability, then the result from the third can be ignored. An appropriate amount of memory can be assigned depending on the number of items. If the false positive probability equation dictates that k hash functions are required, then the first k blocks are chosen.

Thus far we have described how to construct just one group of t Bloom filters and allocate memory to them. When we have multiple such groups, say s, then the available memory needs to be shared among all of them. This can be achieved by using the architecture shown in Figure 3.9(B). We can use a $s : 1$ multiplexer to allow one of the s groups access a given memory block. For instance, if the first group has more items than the second group and needs two memory blocks, then the multiplexers can be configured to allow the first group to access the first two memory blocks and the second group to access the third memory block. When we configure

the system in this way, we are also wasting the extra hash computation of each group that would have accessed a memory block. In this example, we would waste the third hash function of each Bloom filter in the first group and the first two hash functions of each Bloom filter in the second group. However, as we will show in the next section, the hash computation is inexpensive in hardware and hence the price to be paid for the memory allocation flexibility is trivial.

3.6.1 Hash Functions

A class of universal hash functions described in [26] were found to be suitable for hardware implementation. Recall that k hash functions are generated. Following is a description of how this hash matrix is calculated. Represent a bit string X with W bits as

$$X = \langle x_0, x_1, x_2, x_3, ..., x_{W-1} \rangle$$

The hash function over the first j bits denoted as $h(x[0...j-1])$ is calculated as follows,

$$h(x[0...j-1]) = d_0 \cdot x_0 \oplus d_1 \cdot x_1 \oplus d_2 \cdot x_2 \oplus d_3 \cdot x_3 \oplus ..... \oplus d_{j-1} \cdot x_{j-1} \qquad (3.17)$$

where '$\cdot$' is a bitwise AND operator, i.e.,

$$\begin{aligned} d_i \cdot x_i &= d_i \;\; if \;\; x_i = 1 \\ &= 0 \;\; otherwise \end{aligned}$$

and '$\oplus$' is a bitwise XOR operator. d_i is a predetermined random number in the range $[0 \ldots m'-1]$ where m' is the number of bits in a block. *Note that the hash value can be out of the range $[0 \ldots m'-1]$ if m is not a power of 2. Hence, m must be a power of 2.* It can be observed that the hash functions are calculated cumulatively and the results calculated over the first i bits can be used for calculating the hash function over the first $i+1$ bits. This property of the hash functions results in a

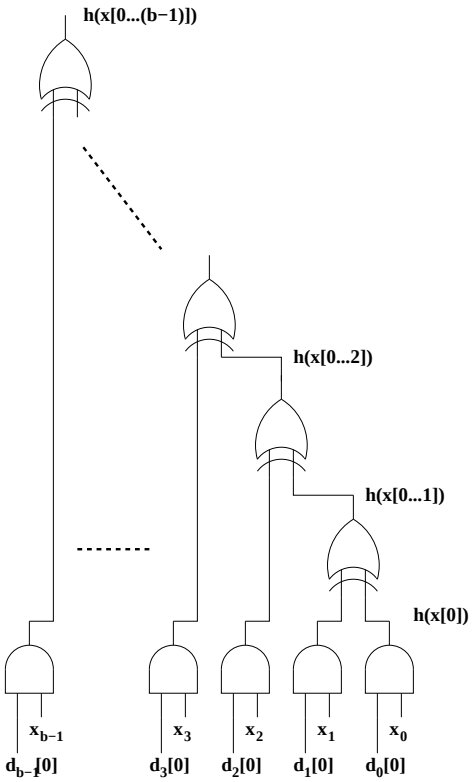


Figure 3.10: **Computation of a single bit in a hash function. If the hash value is represented as l bits then l such circuits are required for a complete hash function.**

regular and less resource-consuming hash function matrix. Each d_i has $\log_2 m'$ bits so the hash result will be as many bits. Let's consider only one bit of the final hash result and construct a circuit to produce it (Figure 3.10). We would need $\log_2 m'$ of these circuits to compute a complete hash function, each of which would produce an individual bit. Finally, if we are using b such hash functions for b memory blocks then b such copies of the complete hash function are required.

Now we can easily compute the number of "2-input" gates required to compute hash functions. Since each bit in the input consumes one AND gate and one XOR gate to produce a single bit of a single hash value, we need $G = 2 \times W \times \log_2 m' \times b$ 2-input logic gates. For IPv4, $W = 32$, and assuming that we are using at the most $m' = 64$ Kbit memory blocks, we need $G = 1024 \times b$ logic gates. For a small number of memory blocks such as $b = 128$, this is a moderate logic resource consumption.

3.7 Summary

We have introduced a Longest Prefix Matching (LPM) algorithm that employs Bloom filters to efficiently narrow the scope of the search. We abstract the LPM as a hash table access problem and eliminate potentially unsuccessful hash table searches by performing a quick check in Bloom filters. Since Bloom filters can be implemented using highly parallel and fast embedded memory blocks, the filtering process can be performed quickly. With a very high probability, the first off-chip hash table access succeeds. Through theoretical analysis and simulations we have shown that the technique requires close to one memory access per lookup with only a moderate amount of on-chip memory. If implemented in current semiconductor technology and coupled with a commodity SRAM device operating at 333 MHz, our algorithm could achieve an average performance of over 300 million lookups per second and a worst case performance of over 100 million lookups per second. By comparison, the state-of-the-art TCAM-based solutions for LPM provide 100 million lookups per second. The power consumption per prefix bit of our scheme is significantly less than TCAM-based solutions.

In the subsequent chapters, we will show how this LPM technique can be used as a basic building block for a variety of other algorithms. The essential idea remains the same: abstract the search problems as table lookups and eliminate the potentially unsuccessful lookups using on-chip Bloom filters.

Chapter 4

Packet Classification

4.1 Introduction

The general packet classification problem has received a great deal of attention over the last decade. The ability to classify packets into flows based on their packet headers is important for QoS, security, virtual private networks (VPN), and packet filtering applications. Conceptually, a packet classification system must compare each packet header received on a link against a large set of rules and return the identity of the highest priority rule in the set that matches the packet header (or in some cases, all matching rules). Each rule can match a large number of packet headers since the rule specification supports address prefixes, wild cards, and port number ranges. Much of the research to date has concentrated on algorithmic techniques which use hardware or software lookup engines that access data structures stored in commodity memory. However, none of the algorithms developed to date have been able to displace TCAMs in practical applications.

TCAMs offer consistently high performance, which is largely independent of the characteristics of the rule set. Unfortunately, they are relatively expensive and consume large amounts of power. A TCAM requires a deterministic time for each lookup. Recent TCAM devices can classify more than 100 million packets per second. Although TCAMs are a favorite choice of network equipment vendors, alternative solutions are still being sought, to alleviate the high cost and high power consumption of TCAM devices. The cost per bit of a high performance TCAM is about 15 times larger than a comparable SRAM [3], [2]. TCAMs consume more than 50 times as much power

per access compared to SRAM [44],[36]. This gap in costs and power consumption makes the exploration of better algorithmic solutions worthwhile.

We introduce an algorithmic solution which is both fast and highly memory efficient. Our solution is based on the well-known "Crossproducting Algorithm" [33]. The crossproducting algorithm decomposes the packet classification problem into a set of single field lookup problems and combines the results to form a key to retrieve the best matched rule from a direct lookup table. From the throughput perspective, the single field lookups are the only real performance bottleneck. However, the major problem with this algorithm is its prohibitively high memory consumption due to the large number of additional "crossproduct rules" that must be added to the rule set. Even small rule sets can require impractically large amounts of memory.

Leveraging recent advances in algorithms and architectures, we introduce some new ideas to address these problems. In particular, our Multi-Subset Crossproducting Algorithm significantly reduces this memory overhead while preserving the overall speed of the algorithm. First, we perform the single field lookup by longest prefix matching (LPM) on each field using the fast and memory efficient Bloom filter-based algorithm introduced in Chapter 3. Using this algorithm, only one off-chip memory access is needed on average for each single field lookup. Therefore, with very high probability, the longest prefix matching can be performed on the source and destination addresses and ports in just four memory accesses.

To reduce memory consumption, we divide the rules into multiple subsets and then construct a crossproduct table for each subset. This drastically reduces the overall crossproduct overhead. In addition, instead of a direct lookup table, a hash table is used to further reduce the size of the crossproduct lookup table. Since the rules are divided into multiple subsets, we need to perform a lookup in each subset. However, we can use Bloom filters to avoid lookups in subsets that contain no matching rules which makes it possible to sustain high throughput. In particular, we show that the highest priority matching rule can be found using only p more memory accesses, where p is the number of rules a packet can match. In summary, we demonstrate a method, based on Bloom filters and hash tables, that can classify a packet in $4+p+\epsilon$ memory accesses where ϵ is a small constant $\ll 1$ determined by the false positive probability of the Bloom filters. With two memory chips, one for the LPM tables and the other

for rule tables, the LPM phase of four memory accesses and rule lookup phase of p memory accesses can be pipelined. With pipelining, the memory accesses per packet can be reduced to $max\{4, p\}$. We also show how a special case of our algorithm is a highly optimized variant of the well-known Tuple Space Search algorithm proposed by Srinivasan et. al. [31]. We also discuss the underlying architectural issues in implementing this method in hardware. The results show that our architecture can handle large rule sets that contain hundreds of thousands of rules, efficiently and with an average memory consumption of only 30 to 45 bytes per rule.

The rest of the chapter is organized as follows. In the next section, we discuss related work. We describe the naïve crossproducting algorithm in more detail in Section 4.3. Section 4.4 discusses our Multi-Subset Crossproducting Algorithm. In Section 4.5, we describe our heuristics for intelligent partitioning of the rules into subsets that reduce the overall crossproducts. Finally, in Section 4.6, we discuss the architectural issues involved in implementing our algorithm in hardware. Section 4.7 concludes the chapter.

4.2 Related Work

A vast body of literature exists on packet classification. An excellent survey and taxonomy of the existing packet classification algorithms and architectures can be found in [36]. Here, we discuss only the algorithms that are closely related to our work.

Algorithms that can provide deterministic lookup throughput are similar to the basic crossproducting algorithm [33]. The basic idea of the crossproducting algorithm is to perform a lookup on each field first and then combine the results to form a key to index a crossproduct table. The best-matched rule can be retrieved from the crossproduct table in only one memory access. The single-field lookup can be performed by direct table lookup as in the Recursive Flow Classification (RFC) algorithm [22] or by using any range searching or LPM algorithm. The Bit Vector (BV) [23] and Aggregate Bit Vector (ABV) [6] algorithms use bit vector intersections to replace the crossproduct table lookup. However, the width of a bit vector needs to be equal to the number of

rules and each unique value of each field needs to store such a bit vector. Therefore, the storage requirement is significant and the scalability is limited.

Using a similar reduction tree, the Distributed Crossproduct of Field Labels (DCFL) [35] algorithm uses hash tables rather than direct lookup tables to implement the crossproduct tables at each tree level. However, depending on the lookup results from the previous level, each hash table needs to be queried multiple times and multiple results are retrieved. For example, at the first level, if a packet matches m nested source IP address prefixes and n nested destination IP address prefixes, we need $m \times n$ hash table queries with the keys that combine these two fields. The lookups typically result in multiple valid outputs that require further lookups. For a multi-dimensional packet classification, this incurs a large performance penalty.

TCAMs are widely used for packet classification. The latest TCAM devices also include a banking mechanism to reduce the power consumption by selectively turning off the unused banks. Traditionally, TCAM devices needed to expand the range values into prefixes for storing a rule with range specifications. The recently introduced algorithm, Database Independent Range PreEncoding (DIRPE) [24], uses a clever technique to encode ranges differently. The technique results in less overall rule expansion than the traditional method. The authors also recognized that in modern security applications, it is not sufficient to stop the matching process after the first match is found but report all the matching rules for a packet. They devised a multi-match scheme with TCAMs that involves multiple TCAM accesses.

Yu et. al. described a different algorithm for multi-match packet classification based on geometric intersection of rules [43]. A packet can match multiple rules because the rules overlap. However, if the rules are broken into smaller sub-rules such that all rules are mutually exclusive, then the packet can match only one rule at a time. This overlap-free rule set is obtained through geometric intersection. Unfortunately, the rule set expansion due to the newly introduced rules by the intersection can be very large. In [44], they describe a modified algorithm called Set Splitting Algorithm (SSA) which reduces the overall expansion. They observe that if the rules are partitioned into multiple subsets in order to reduce the overlap then the resulting expansion will be small. At the same time one would need to probe each subset independently to search a matching rule. In a way, our algorithm is similar to SSA in that we also try to

reduce the overlap between the rules by partitioning them into multiple subsets and thus reduce the overall expansion. While SSA only cares about an overlap in all the dimensions, our algorithm considers the overlap in any dimension so the partitioning techniques are different. Moreover, SSA is a TCAM based algorithm whereas ours is memory based. Finally, SSA requires a one-by-one probe of all the subsets formed which requires as many TCAM accesses as there are subsets. Our algorithm needs only p memory accesses, only as many as the number of matching rules per packet.

4.3 Naive Crossproducting Algorithm

The naïve crossproducting algorithm works as follows. Let a 5-tuple rule be specified as $r = [v_1, v_2, v_3, v_4, v_5]$ where each v_i is a prefix of field i. Let $V_i = \cup v_i$ i.e. V_i be a set of all the distinct prefixes of the field i present in the rule set. The crossproducting algorithm creates all the possible rules of the form $r' = [v'_1, v'_2, v'_3, v'_4, v'_5]$ where $v'_i \in V_i$. In other words, the algorithm simply produces the crossproduct set $V_1 \times V_2 \times V_3 \times V_4 \times V_5$. Given a five-tuple of the packet header, a matching rule can be searched as follows. First, we perform an independent search on each field and find the most specific prefix i.e. the longest matching prefix. After having obtained these longest matching prefixes for each field u_i, we create a unique key $u = [u_1, u_2, u_3, u_4, u_5]$ and use it to directly index the crossproduct rule table. Each rule in the crossproduct table is either the original rule or an artificial rule generated in the process of crossproducting. Moreover, each extra rule either corresponds to an original rule or none at all. Upon a match, we either get the ID of an original rule or we don't get any ID. When there is a match, the correct matching rule can always be found. This is illustrated with the example shown in Figure 4.3. We show only a two dimensional rule set where each field is four bits wide for the purpose of illustration. Figure 4.3(A) shows the original rule set. Figure 4.3(C) shows the crossproduct table for this rule set. Figure 4.3(B) shows the representation of the rules using a trie.

In this rule set, the first field contains four unique prefixes {1*, 00*, 01*, 101*} which can be labeled as $\{1, 2, 3, 4\}$ respectively. Likewise, the second field contains 4 unique prefixes {*, 00*, 11*, 100*}, which can also labeled as $\{1, 2, 3, 4\}$ respectively. A straightforward crossproduct table would contain $4 \times 4 = 16$ entries. Among

these 16 entries are the six original rules (white colored rows) and the remaining rules are generated by the crossproduct. There are crossproduct rules that correspond to an original rule, i.e., a match of any of these crossproduct rules implies a match for one or more of the original rules. We call these "pseudo-rules" (blue colored rows). Take for instance the rule p_7 = [101*, 00*]. If there is a match for this rule, then it implies a match for original rules r_1 and r_2 since p_7 is more specific than both r_1 and r_2. There are also some entries which do not map to any original rule, e.g. [01*, 1*], which we call "empty rules" (green colored rows). To illustrate the rule matching process, assume that we get a packet with the header value [1011, 0011]. We perform the longest prefix matching on each field and find that the prefixes are 101 with a label of 4 and 00 with a label of 2. The entry at the location $4 \times 2 = 8$ in the table can be looked up for a match. Since there is a matching rule, p_7, we can declare a match for the original matching rules, r_1 and r_2.

This algorithm has two problems: 1) A large number of empty rules and 2) A very large number of pseudo-rules. The first problem can be mitigated by using a hash table instead of a direct lookup table. The crossproduct table maintains all possible entries generated from the crossproduct so that it can be directly indexed. Since there are several empty rules, the sparsity can be utilized to compress the table further by using a hash table. This is a trivial modification to the crossproduct table. Henceforth we assume that the crossproduct table contains only the rules that correspond to at least one of the original rules, i.e. we have only pseudo-rules and the original rules but no empty rules. A trie-based representation of the pseudo-rules along with the original rules is shown in Figure 4.3(D). We will use this trie-based representation to further illustrate our algorithm. We build a trie for each field. We mark the nodes corresponding to the prefixes involved in the rules. A connection between the marked nodes of each field represents a rule.

With this representation, it is easy to see that after the removal of empty rules from the crossproduct table, the remaining pseudo-rules satisfy a particular property as follows. If we keep following the *marked* ancestors of the nodes of each field then there is at least one combination of marked ancestors that represents one of the original rules. This observation also allows us to create the crossproduct rule set with an alternative and more efficient procedure described below. First, we introduce the following notations.

- Let R denote the set of original rules and C the set of rules after the crossproduct.
- Let $u \prec v$ denote that u is a prefix of v. (Note that $v \prec v$ always holds). Since each prefix corresponds to a marked node in the trie, we will use the terms prefix and marked node interchangeably. Hence, $u \prec v$ also denotes that the node u is the marked ancestor of marked node v.
- Let r denote a rule. Let $r.v_i$ denote the prefix of field i in the rule. Let $r.Id$ denote the set of rule IDs associated with this rule.
- Let T_i denote the trie built from the prefixes of field i.

The pseudo-code for the crossproduct algorithm is described in Figure 4.3.

BuildCrossproductTable(S)
1. **for each** $r \in R$
2. **for each** field i
3. InsertInTrie($r.v_i, T_i$)

4. **for each** $r \in R$
5. **for each** field i
6. $V_i \leftarrow V_i \cup r.v_i \cup GetAllMarkedDescendants(r.v_i, T_i)$
7. **for each** node $v_1 \in V_1$
8. **for each** node $v_2 \in V_2$
9. **for each** node $v_3 \in V_3$
10. **for each** node $v_k \in V_k$
11. $c.v_1 \leftarrow v_1$, $c.v_2 \leftarrow v_2$,. . . , $c.v_k \leftarrow v_k$
12. if $c \in C$
13. $c.Id \leftarrow c.Id \cup r.Id$
14. else
15. $c.Id \leftarrow r.Id$
16. $C \leftarrow C \cup c$

Figure 4.1: **The pseudo-code for building a crossproduct table.**

To build a crossproduct table, we first build a trie for each field with the prefixes of that field in all the rules (lines 1-3). Then we pick rules one-by-one. For each, we locate the node corresponding to each field prefix in the trie and get the set of all the corresponding descendants (line 5-6) including the node under consideration. A

set of such descendants for field i, including the given node itself, is denoted by V_i. We then take the crossproduct of these sets and insert the resulting rules into the crossproduct set. Note that this crossproduct set will also include the original rule since we are also including the nodes of the original rule in the crossproduct. Each of the crossproduct rules points to the original rule under consideration for which the crossproduct is being generated. During the process, we see if the rule is already inserted into the table while considering any other original rule. If the rule is already inserted, then we just need to append the ID of the original rule under consideration to the set of rule IDs associated with this pseudo-rule (lines 12-13). A match for this pseudo-rule will mean a match for all the rule IDs of the original rules associated with it. If the rule is not present in the table, then it is added and the associated rule ID is set to the original rule ID (lines 14-16). The resulting rule set C consists of both the original rules and the crossproduct rules.

The packet classification process is simple:

ClassifyPacket(P)
1. **for each** field i
2. $r.v_i \leftarrow LPM(P.f_i)$
3. $\{match, \{Id\}\} \leftarrow HashTableLookup(r)$

Figure 4.2: **The pseudo-code classifying a packet.**

As the algorithm describes, we first execute LPM on each field with a value of f_i of packet P and assign the longest matching prefix to a rule r. Then we look up this rule in the hash table. The result of this lookup indicates if the rule matched and also outputs a set of associated matching rule IDs.

It is evident that the crossproduct algorithm is efficient in terms of memory accesses. The memory accesses are required for only LPM on each field and the final hash table lookup to search the rule in the crossproduct table. For 5-tuple classification, we don't need to perform the LPM for the port field. The port lookup can be done with a direct lookup array in a small on-chip table. Moreover, if we use the Bloom filter based LPM technique described in Chapter 3, we would need approximately one memory access per LPM. Therefore, the entire classification process takes five memory accesses with very high probability.

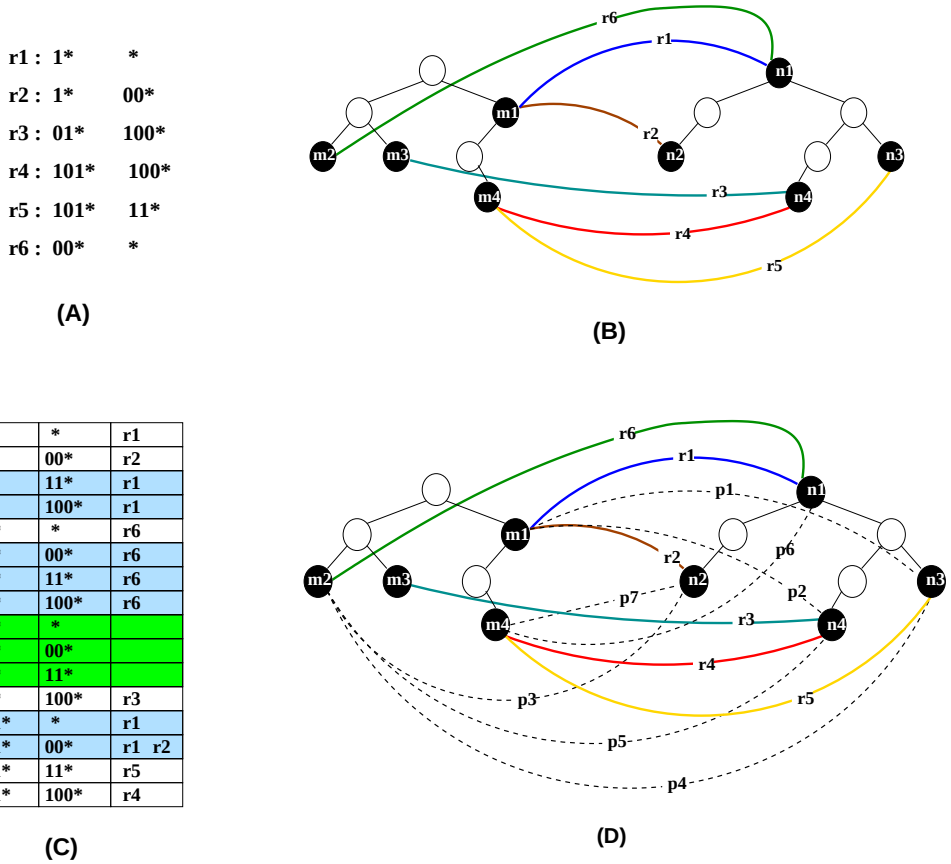


Figure 4.3: **Illustration of basic ideas. (A) Rule set (B) Rule representation using trie (C) Crossproduct table (D) Representation of original rules and pseudo-rules using trie**

However, the overhead of pseudo-rules can be very large. If each field has 100 unique values in the rule set (excluding the protocol field), then the expanded rule set can be potentially as large as 100^4 making it impractical to scale for larger rule sets.

In order to get a sense of the amount of expansion the naïve crossproducting algorithm can cause, we experimented with several real life rule sets as well as synthetic rule sets which preserved the structure of the real rule sets. We used the synthetic rule set generator ClassBench [37]. The real life rule sets obtained from access control lists (ACL), firewalls (FW), and IP chains (IPC) were used as seeds to generate larger rule sets with approximately ten thousand rules (all the rule sets with names ending in 's' in Table 4.1). Note that our algorithm needs the ranges to be expanded into prefixes. Due to this expansion, the size of the rule set increases. The reported number of

rules in each set is the number after the range to prefix expansion. The number of rules in each set and the expansion factor, δ, after the naïve crossproduct is shown in Table 4.1. As the table shows, the expansion factor can be very large. The smallest expansion was observed to be 200 times the original rule set size and the largest was 5.7×10^6 times! Clearly, the naïve crossproducting algorithm is impractical for large rule sets.

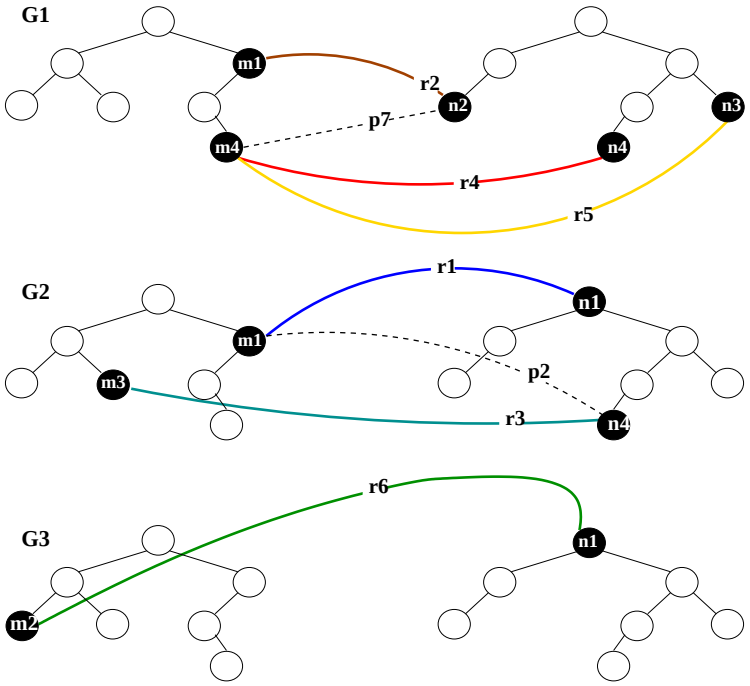
So how can we reduce the overhead of the pseudo-rules and also preserve the fast speed of the algorithm? We present our Multi-subset Crossproducting Algorithm that achieves this objective.

4.4 Multi-Subset Crossproducting Algorithm

In the naïve scheme, we require just one hash table access to get the list of matching rules. However, if we allow ourselves to use multiple hash table accesses then we can split the rule set into multiple smaller subsets and take the crossproduct within each of them. With this arrangement, the total number of pseudo-rules can be reduced significantly compared to the naïve scheme. This is illustrated in Figure 4.4. We divide the rule set into three subsets. Within each subset, we take a crossproduct and retain only the rules that correspond to one of the original rules within that subset. This results in inserting pseudo-rules p_7 in subset 1 (G_1) and p_2 in subset 2 (G_2). All the other pseudo-rules vanish and the overhead is significantly reduced. Why does the number of pseudo-rules reduce drastically? This is because the crossproduct is inherently multiplicative in nature. When the number of overlapping prefixes of a field i gets reduced by a factor of x_i due to partitioning, the resulting reduction in the crossproduct rules is proportional to Πx_i and can therefore be quite large.

After having reduced the crossproduct memory overhead, an independent hash table can be maintained for each rule subset and an independent rule lookup can be performed in each. The splitting introduces two extra memory access overheads: 1) The entire LPM process on all the fields needs to be repeated for each subset 2) a separate hash table access per subset is needed to lookup the final rule. We now describe how to avoid the first overhead and reduce the second overhead.

In Figure 4.4, due to the partitioning of rules into subsets G_1, G_2 and G_3, the sets of valid prefixes of the first field are $\{m_1, m_4\}$ for G_1, $\{m_1, m_3\}$ for G_2 and $\{m_2\}$ for G_3. Hence, the longest prefix for one subset might not be the longest prefix in other subset requiring a separate LPM for each subset.



(A)

	G1	G2	G3
1*	1	1	–
00*	–	–	2
01*	–	2	–
101*	3	1	–

LPM Table for field 1

	G1	G2	G3
*	–	0	0
00*	2	0	0
11*	2	0	0
100*	3	3	0

LPM Table for field 2

(B)

Figure 4.4: **Dividing rules in separate subsets to reduce overlap. The corresponding LPM tables.**

However, this can be easily avoided by modifying the LPM data structure. For each field, we maintain only one global data structure that contains the unique prefixes of that field from all the subsets. When we perform LPM on a field, the matching prefix is the longest one across all subsets. Therefore, the longest prefix for individual

subsets is either the prefix that matches or its sub-prefix. With each prefix in the LPM table, we can maintain a list of sub-prefixes, one for each subset, such that, each prefix in the list is the longest prefix for that subset.

Conceptually, the LPM table for field i consists of entries where each entry t_i consists of a prefix $t_i.v$ which is the lookup key portion of that entry and the associated information consists of g entries, $t_i.u[1] \ldots t_i.u[g]$, where g is the number of subsets formed. Each $t_i.u[j]$ is either $NULL$ or has a value such that $t_i.u[j]$ is the longest matching prefix of field i in subset j and obeys $t_i.u[j] \prec t_i.v$. If $t_i.u[j] == NULL$ then there isn't any prefix of $t_i.v$ that is the longest prefix in subset j.

After a global LPM on the field, we have all the information we need regarding the matching prefixes in individual subsets. Since $t_i.u[j]$ is a prefix of $t_i.v$, we do not need to maintain the complete prefix $t_i.u[j]$ but just its length. The prefix $t_i.u[j]$ can always be obtained by considering the correct number of bits of $t_i.v$.

The LPM table for the example shown in Figure 4.4(A) is shown in Figure 4.4(B). Since we have three subsets, each prefix has three entries that correspond to three subsets. For instance, the table for field 1 tells us that if the longest matching prefix on this field in the packet is 101, then there is a sub-prefix of `101` of length 3 (which is `101`=m_4 itself) that is the longest prefix in G_1. There is a sub-prefix of length 1 (which is `1` =m_1) that is the longest prefix in G_2 and there is no sub prefix (indicated by —) of `101` that is the longest prefix in G_3.

Likewise, the table for field 2 says that if the longest matching prefix for this field in the packet header is `100`, then there is a sub-prefix of `100` of length 3 (which is `100`=n_4) that is the longest prefix in G_1. There is a sub-prefix of length 3 (hence again `100`=n_4) that is the longest prefix in G_2. Finally, there is a sub-prefix of length 0 (hence $* = n_1$) that is the longest prefix in G_3. Thus, after finding the longest prefix of a field, we can read the list of longest prefixes for all the subsets and use it to probe the hash tables. For example, if `101` is the longest matching prefix for field 1 and `100` is for the field 2, then we will probe the G_1 rule hash table with the key $\langle$`101, 100`$\rangle$ and the G_2 rule hash table with the key$\langle$`1, 100`$\rangle$. We don't need to probe the G_3 hash table.

The classification algorithm is described in Figure 4.5.

ClassifyPacket(P)
1. **for each** field i
2. $t_i \leftarrow LPM(P.f_i)$
3. **for each** subset j
4. **for each** field i
5. **if**$(t_i.u[j] \neq NULL)$ $r.v_i = t_i.u[j]$
6. **else** break
7. $\{match, \{Id\}\} \leftarrow HashTableLookup_j(r)$

Figure 4.5: **The pseudo-code classifying a packet using multiple crossproduct tables.**

Even after splitting the rule set into multiple subsets, only one LPM is required for each field (lines 1-2). We maintain a similar LPM performance to the naïve crossproduct algorithm. After the LPM phase, individual rule subset tables are probed one by one with the keys formed from the longest matching prefixes within that subset (lines 3-7). However, a probe is not required for a subset if there is no sub-prefix corresponding to at least one field within that subset. In this case, we simply move to the next subset (lines 5-6). Hence, the number of rule subset tables probed can be less than the actual number of subsets, depending on the actual prefix values in the rule set. For the purpose of analysis, we will stick to the conservative assumption that all fields have some sub-prefix available for each subset requiring all the g subsets to be probed.

We will now explain how we can avoid probing all these subsets by using Bloom filters. If a packet can match at the most p rules and if all these rules reside in distinct hash tables, then at most p of these g hash table probes will be successful and will return matching rules. Other memory accesses are unnecessary, and can be filtered out using on-chip Bloom filters. We maintain one Bloom filter in the on-chip memory corresponding to each off-chip rule subset in a hash table. We first query the Bloom filters with the keys to be looked up in the subsets. If the filter shows a match, we look up the key in the off-chip hash table. Figure 4.6 illustrates the flow of the algorithm.

From equation 3.2, the average number of hash table accesses, t_i, for the LPM on field i, with length W_i is $t_i = 1 + \sum_{j=1}^{W_i - 1} f_j$, where f_j is the false positive probability of Bloom filter j. If we tune the Bloom filters to exhibit the same false positive

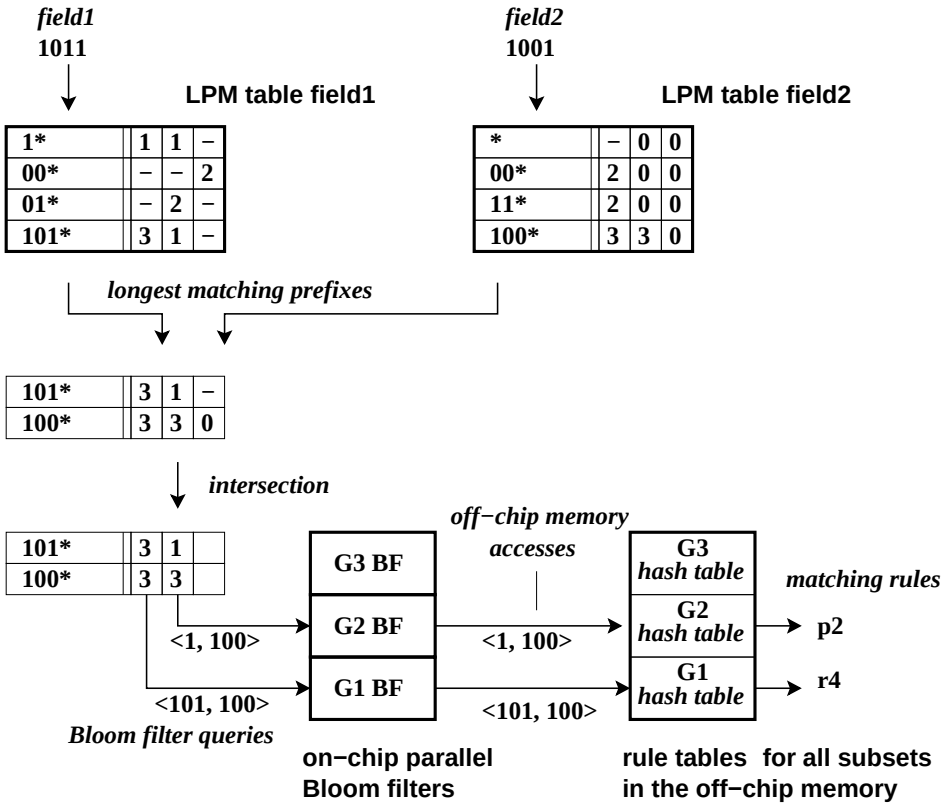


Figure 4.6: **Illustration of the flow of algorithm. First, LPM is performed on each field. The result is used to form a set of g tuples, each of which indicates how many prefix bits to use for constructing keys corresponding to that subset. The keys are looked up in Bloom filters first. Only the keys matched in Bloom filters are used to query the corresponding rule subset hash table kept in the off-chip memory.**

probability, f, by allocating the appropriate amount of memory and number of hash functions, then the average hash table accesses on field i can be expressed as:

$$t_i = 1 + (W_i - 1)f \tag{4.1}$$

For IPv4, we need to perform LPM on the source and destination IP addresses (32 bits each) and ports (16 bits each). The protocol field can be looked up in a 256 entry direct lookup array kept in the on-chip registers. We don't need memory accesses for protocol field lookup. We can use a set of 32 Bloom filters to store the source and destination IP address prefixes of different lengths. While storing a prefix, we tag it

with its type to create a unique key (for instance, source IP type = 1, destination IP type = 2). While querying a Bloom filter with a prefix, we create the key by combining the prefix with its type. Similarly, the same set of Bloom filters can be used to store the source and destination port prefixes. Bloom filters 1 to 16 can be used to store the source port prefixes and 17 to 32 can be used for destination port prefixes. Hence, the total number of hash table accesses required for LPM on these four fields can be expressed as

$$\begin{aligned} T_{lpm} &= (1+31f)+(1+31f)+(1+15f)+(1+15f) \\ &= 4+92f \end{aligned} \tag{4.2}$$

We need g more Bloom filters for storing the rules of each subset. During the rule lookup phase, when we query the Bloom filters of all the g subsets, we will have up to p true matches and the remaining $g-p$ Bloom filters can show a match, each with false positive probability of f. Hence the hash probes required in the rule matching are

$$T_g = p + (g-p)f \tag{4.3}$$

The total number of hash table probes required in the entire process of packet classification is

$$T = T_g + T_{lpm} = 4 + p + (92+g-p)f = 4 + p + \epsilon \tag{4.4}$$

where $\epsilon = (92+g-p)f$. By keeping the value of f small (e.g. 0.0005), the ϵ can be made negligibly small, giving us total accesses equal to $\approx 4+p$. It should be noted that we have so far only dealt with the number of hash table accesses and not the memory accesses. A carefully constructed hash table requires close to one memory access for each single hash table lookup.

Furthermore, our algorithm is a "multi-match" algorithm as opposed to the priority rule match. For our algorithm, priorities associated with all the matching rules need to be explicitly compared to pick the highest priority match.

As Equation 4.4 shows, the efficiency of the algorithm depends on how small g and f are. In the next section, we explore the trade-off involved in minimizing the values of these two system parameters.

4.5 Intelligent Grouping

When we try to create very few subsets with a given rule set, then each subset can still produce a significant number of crossproducts. On the other hand, we do not want a very large number of subsets because to avoid using a large number of hardware resource-consuming Bloom filters. We would like to limit g to a moderately small value. The key to reducing overhead of pseudo-rules is to intelligently divide the rule set into subsets to minimize the crossproducts. The following questions arise. How can we reduce the number of subsets as well as the pseudo-rules since these appear to be conflicting goals? The pseudo-rules are required only when there are overlapping prefixes of different rules. So, is there an overlap-free decomposition into subsets so that we don't need to insert any pseudo-rules? Alternatively, we would also like to know: given a fixed number of subsets, how can we create them with minimum number of pseudo-rules? We address these questions in this section.

4.5.1 A Problem Formulation

The problem of constructing subsets of overlap-free rules from a given rule set can be modeled as graph coloring problem. We represent the rule set with a graph $G = (V, E)$ in which each vertex in V represents a rule. We add an edge between two vertices if the two rules overlap in at least one dimension, i.e. the two rules create extra crossproduct rules if they are kept in the same subset. Now, we want to color all the vertices with a minimum number of colors such that no two vertices connected by an

edge have the same color. A color is equivalent to a subset. Graph coloring is known to be an NP-complete problem.

With the graph theoretic problem formulation and heuristic solutions, potentially a tight bound can be found on the number of such subsets. However, we avoid the graph theoretic solution and seek a simpler heuristic solution that is specific to this problem. Our heuristic of forming subsets is based on the concept of Nested Level Tuple (NLT) explained in the next section. Our solution is simple and provides a loose yet practical upper bound on the number of subsets. Moreover, it requires very little computation.

Although obtaining subsets of overlap-free rules is our objective, such a partitioning potentially can result in a large number of subsets. Instead, we fix a particular number of subsets and try to partition the rules in them so the overall number of pseudo-rules is minimized. How can we create such subsets? We provide an approximate model of this problem by extending the graph model described above. We create a graph $G = (V, E)$ as described above and assign weights to each edge, where the weight equals the number of crossproduct rules due to the overlap of the two rules corresponding to the vertices connected by the edge (i.e. pairwise crossproduct rules). Given this weighted graph, we wish to color the vertices with g colors such that the sum of the weights on the edges connecting vertices of the same color is minimum. Since the rules with the same color belong to the same subset, we wish to minimize the sum of pairwise crossproduct rules between all of them, hence the sum of the weights should be minimum. This is a standard MIN K-PARTITION problem which is also NP-complete.

This is an approximate model of the problem because in the context of our problem, the total number of crossproduct rules can be less than the sum of the pairwise crossproduct rules. This is because, some of the crossproduct rules can be common to multiple sets of pairwise crossproduct rules and thus redundant. However, the sum of the pairwise crossproduct rules is an upper bound on the amount of expansion within a subset.

Again, although a graph theoretic solution is possible for this problem, we avoid this approach and seek a simpler solution by taking advantage of the nature of the

problem. We describe a simple heuristic solution in Section 4.5.3 which modifies the NLT based solution for the first problem. Specifically, we use the first heuristic to produce an overlap-free grouping. Given a fixed number of subsets (colors), we pick the same number of most populated subsets. Then, we merge the remaining subsets to the fixed subsets with the objective of reducing the overall crossproduct rules generated by merging.

4.5.2 Overlap-Free Grouping

A loose bound on the number of overlap-free subsets is the number of prefix length tuples. We now describe the Tuple Space Search (TSS) algorithm. While TSS provides one loose bound, we seek a much tighter bound by modifying TSS. We describe our modifications at the end of this section.

Tuple Space Search (TSS)

A Prefix Length Tuple (PLT) is the combination of prefix lengths of different fields. For instance, the PLT [32, 24, 16, 7, 0] implies that the source IP prefix length is 32, the destination IP prefix length is 24, the source port prefix length is 16, the destination port prefix length is 7 and the protocol prefix length is 0 (wild-card). Each rule is contained within a tuple. For IPv4 5-tuple packet classification, the PLT space consists of $33 \times 33 \times 17 \times 17 \times 2 = 629442$ PLTs. In the worst case, each rule can represent a unique tuple and the number of PLTs will be the number of rules. For instance, the tuples associated with the rules in our example rule set are [1, 0], [1, 2], [2, 3], [3, 3], [3, 2], and [2, 0] each containing a single rule. However, in practice, the number of PLTs usually is much smaller than the number of rules.

The TSS algorithm maintains all the rules that belong to a PLT in an independent hash table. Upon receiving a packet, it simply looks up all the hash tables by probing them with the keys formed by considering the appropriate number of bits of each field corresponding to that PLT. This naïve approach requires several hash lookups which can be significantly reduced by a tuple pruning technique. The TSS algorithm first gets the longest matching prefix of each field. With each longest matching prefix of a

given field, a list of PLTs corresponding to the given prefix and any shorter prefix is maintained. After reading the lists corresponding to all fields, only the PLTs in the intersection of these lists need to be looked up. We can illustrate the process with our example rule set. The algorithm is illustrated in Figure 4.7 which uses our example rule set. As the figure shows, each LPM table contains prefix entries and a list of PLTs that the prefix as well as its sub-prefixes are associated with. For instance, the prefix 1* of the first field is associated with rules $r_1 = [1*, *]$ and $r_2 = [1*, 00*]$ contained in the PLTs [1, 0] and [1, 2] respectively. Hence, the LPM entry 1* of the first field contains these two PLTs in the list. Likewise, the prefix 101* of the first field is associated with PLTs [3, 2] and [3, 3]. Since, 1* is a sub-prefix of 101*, the PLTs [1, 0] and [1, 2] are also contained in the list associated with 101*.

If the matching prefixes of the two fields were 101* and 100*, then the common PLT list will contain [3, 3] and [1, 0]. Hence, it implies that it is likely that the rules {101*, 100*} and {1*, *} are contained in the table. These keys are used to probe the respective hash tables and find matching entries.

Before we elaborate on the relevance of this algorithm to ours, it is important to mention that the actual TSS algorithm as proposed in [31] does not perform a LPM for source and destination ports. Instead, a different technique based on *Range ID* is used. The authors observed that when the port ranges are converted into prefixes, the resulting expansion could be large. To avoid this expansion, a unique ID is assigned to each range. LPM is performed only on source and destination addresses. The hash key is constructed by taking the appropriate bits from these addresses and combining the range IDs associated with source and destination ports. The Range ID can be obtained from the port number using different techniques, including a search tree or a direct lookup. While the search tree based lookup requires more memory accesses, the direct lookup array requires more memory, potentially two arrays containing 64K entries each. Secondly, for assigning a unique ID to a given range, all the ranges must be non-overlapping. If they overlap, then the overlap must be removed by breaking a single range into multiple, mutually exclusive smaller ranges. This division of overlapping ranges into smaller non-overlapping ranges essentially results in geometric intersection on each port and consequently, some rule expansion.

Instead of using the range ID approach, we will use the range to prefix conversion approach for our algorithm. This will allow us to use the Bloom filter based LPM technique for port matching as opposed to the search tree based technique for Range ID matching. Moreover, it will potentially consume less memory compared to the direct lookup array for Range ID matching. Finally, it will not restrict us to using non-overlapping ranges and allow flexible specification of ranges. Considering these factors, we will use the version of TSS algorithm that deals with the prefix representation of port ranges and performs LPM for each field.

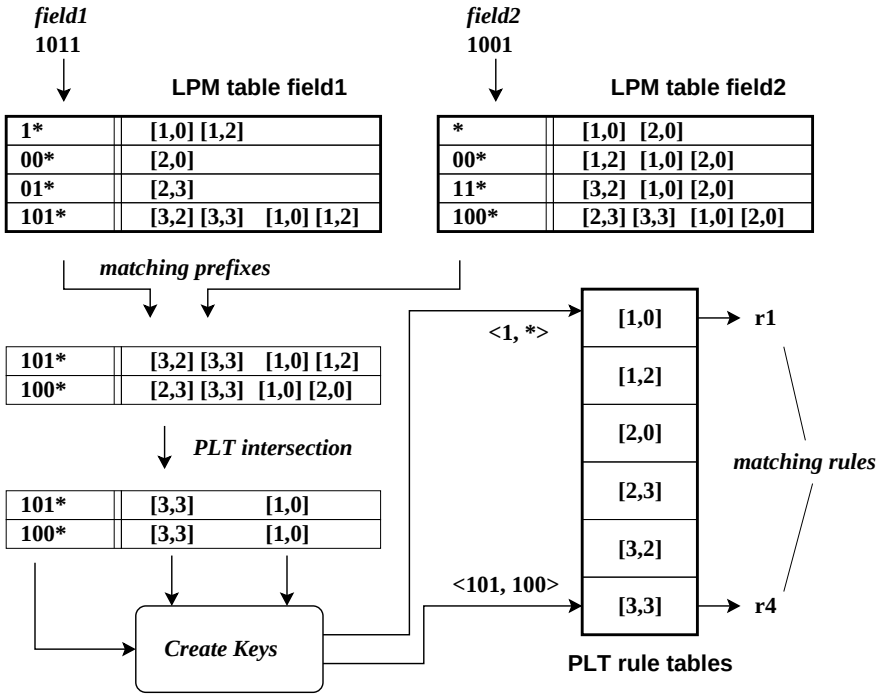


Figure 4.7: **Illustration of the Tuple Space Search algorithm**

We can now draw a parallel between TSS and our algorithm. Note that the rules contained in the same PLT share the same prefix lengths of each field. Therefore, between any two prefixes of the same prefix length, neither is the ancestor of the other. Due to this property, the rules contained within the same PLT do not need crossproducts. Indeed, the number of distinct PLTs in the rule set is essentially one loose upper bound on the number of overlap-free subsets. In fact, when we use the

PLTs as the subsets, our algorithm is the same as TSS except for a few differences in the data structure arrangements. With each prefix in the LPM tables, TSS maintains a list of PLTs. We instead maintain an array with the number of entries equal to the total number of PLTs; each entry contains the length of the prefix for that PLT. This is illustrated in Figure 4.8. For instance, consider the prefix 101* of the first field. There are six entries next to it, each corresponding to a subset (or a PLT). The PLTs are ordered and indexed. As shown, PLT [1,0] is first, [1,2] is second, and so on. The first entry among the six is 1 which implies that the given prefix has a sub-prefix that corresponds to a rule contained in the first PLT ([1,0]) and the length of this sub-prefix is 1. Likewise, the fifth entry,3, implies that the given prefix has a sub-prefix that corresponds to a rule contained in the fifth PLT ([3,2]) and the length of this sub-prefix is 3. When the entry is '-', it means that there is no sub-prefix of the given prefix belonging to any rule in that PLT. In other words, the prefix and its sub-prefixes have nothing to do with that PLT. When we perform LPM on each field and read the array, the intersection becomes easy. We need to consider only those PLTs for which the prefix length in each field is specified. If at least one prefix has '-' for a given PLT then it can be ignored. As the figure shows, after LPM on 1011 and 1001 respectively, the only remaining PLTs are the first and the sixth. The prefix lengths of the individual fields are [1,0] and [3,3]. Now the appropriate number of bits can be considered to construct the keys and the PLT rule sets can be queried.

Note that there is a bit of redundancy in the LPM data structure which can be used to further simplify it, as proposed in original the TSS algorithm. Instead of maintaining the prefix length in each entry, we can simply set a bit to indicate that the given prefix or its sub-prefix belongs to that PLT. Thus, the array can be replaced by a bit map with the number of bits equal to the number of PLTs. To perform the intersection, we just perform a bit-wise AND. Finally, for all remaining PLTs after intersection, we lookup a table to get the associated prefix lengths for each field and after having obtained those, we can construct the keys as before to probe the appropriate PLT rule tables(Figure 4.9). Note that this optimization is possible only because we know that there is a unique prefix length of each field associated with a PLT. In the context of a generic crossproduct, it is not true that a given subset of rules contains prefixes of a specific length for each field; there can be multiple prefixes with different lengths

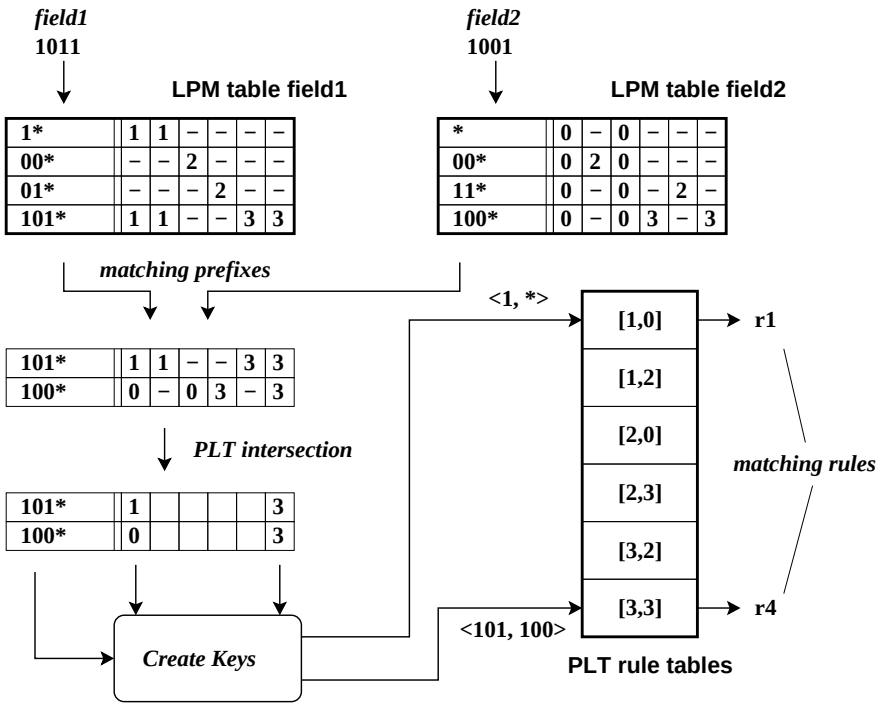


Figure 4.8: **Illustration of the Tuple Space Search algorithm with an alternative LPM table structure.**

within the same subset. Hence, this bitmap data structure can be used only in this special case.

We now discuss the differences between our approach and TSS.

- The original TSS algorithm used conventional trie based techniques for LPM. We use the Bloom filter based LPM algorithm which is faster.
- The original TSS algorithm probes all the PLTs obtained after pruning whereas we use one more stage of filtering using on-chip Bloom filters. Thus, all the PLT queries after pruning can be passed through Bloom filters so that only the potentially successful ones (approximately 'p') will be executed.

By using Bloom filters for memory access filtering, the algorithm performance can be accelerated significantly. However, one important drawback of the system combining

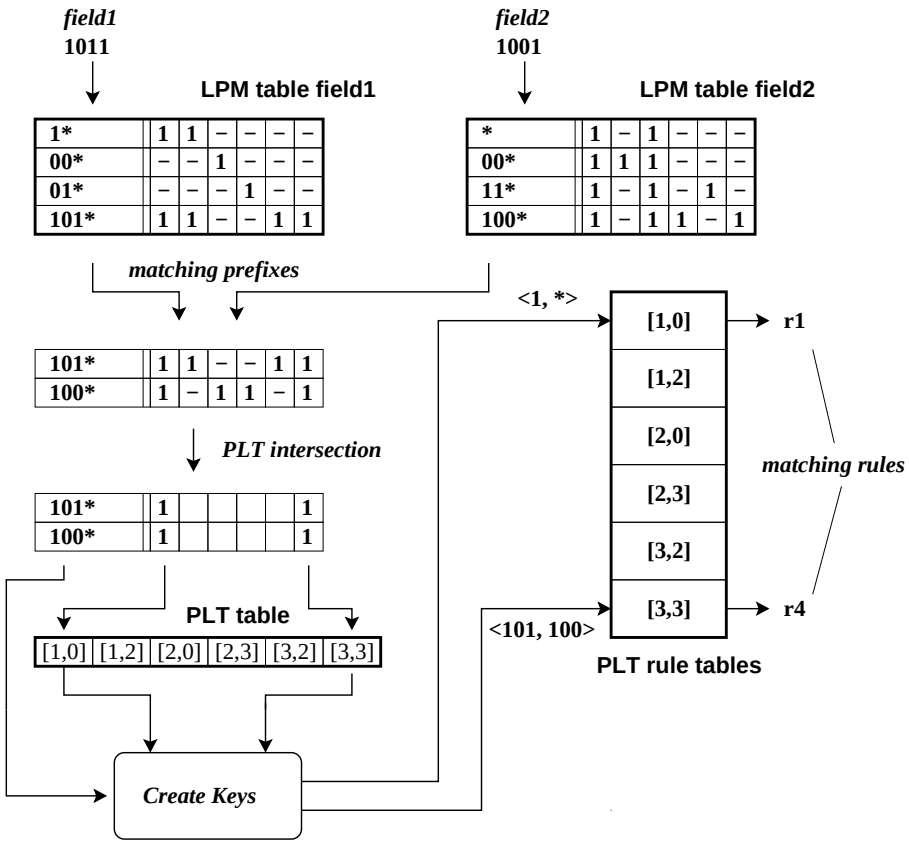


Figure 4.9: **The LPM table can be compressed further by using a bit map.**

Bloom filters and TSS is that the number of PLTs and the number of Bloom filters can be very high. We experimented with our rule sets and found that the number of PLTs can be as high as 11,000 for just 25,000 rules as indicated in the Table 4.1. It is impractical to support such a large number of Bloom filters. However, this problem can possibly be mitigated by using the same set of *physical* Bloom filters to host a large set of *virtual* Bloom filters. When we store an item in a Bloom filter, we can combine the *type* of the item along with the actual item in order to create a unique key, as discussed before. Items of different types can reside in the same physical Bloom filter. When we query the filter with an item, we can combine the type with the key. Therefore, only the item belonging to the correct type will match. This is equivalent to having as many Bloom filters as key types superimposed on the same physical substrate Bloom filter. We call them virtual Bloom filters. In this fashion, if we have b physical Bloom filters to support B PLTs, we can design a mapping that will allow each physical Bloom filter to host $\leq \lceil B/b \rceil$ PLTs. We can time multiplex

the probing of all PLT Bloom filters by probing b of them at a time and covering all B probes in $\lceil B/b \rceil$ iterations (or clock cycles). Moreover, after pruning the PLTs, only a few remain to be checked and the actual number of probes can be much less than the worst case of B. Regardless, the number of PLTs to be checked can still be high and variable.

Secondly, it is impractical to maintain an array with each prefix having 11,000 entries. Even the bitmap technique is not practical for the same reason. Hence, we must use the original TSS technique which maintains a list of PLTs along with each prefix entry. Unfortunately, this will make intersecting of the PLT lists very difficult. The problem can be formulated as follows. We are given t sets of numbers S_1,..,S_t, set i containing n_i numbers. Each number is taken from a large universe U. How can we intersect all sets S_i in hardware? The intersection would have been very easy if the U was small. In that case, we could have maintained a bitmap of $|U|$ bits for each S_i and set the bits indexed by the numbers present in that set. Intersection is just the bit-wise AND.

In light of these drawbacks, we now describe a technique that will substantially reduce the number of subsets. When the number of subsets is substantially reduced, the bitmap technique can be used which makes the intersection process easier. This reduction in the number of subsets is based on the concept of *Nested Level Tuple* which is explained next.

Nested Level Tuple Space Search (NLTSS) Algorithm

We begin by constructing an independent binary prefix-trie with the prefixes of each field in the given rule set just as shown in Figure 4.3(B). We will use some formal definitions given below.

Nested Level: *The nested level of a marked node in a binary trie is the number of proper ancestors of this node which are also marked. We treat the root node as if it were marked.* For example, the nested level of node m_2 and m_3 is 1 and the nested

level of node m_4 is 2.

Nested Level Tree: *Given a binary trie with marked nodes, we construct a Nested Level tree by removing the unmarked nodes and connecting each marked node to its nearest ancestor.* Figure 4.10 illustrates a nested level tree for field f_1 in our example rule set.

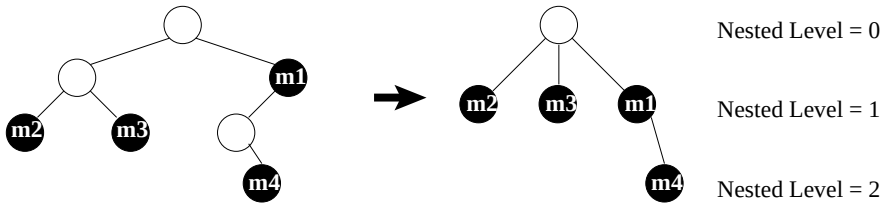


Figure 4.10: **Illustration of Nested Level Tree**

Nested Level Tuple (NLT): *For each field involved in the rule set, we create a Nested Level Tree (See Figure 4.11). The Nested Level Tuple (NLT) associated with a rule r is the tuple of nested levels associated with each field prefix of that rule.* For instance, the NLT for r_6 is [1,0] and for r_4 is [2,1].

From the definition of the nested level, it is clear that among the nodes at the same nested level, no one is the ancestor of the other. Therefore, the prefixes represented by the nodes at the same nested level in a tree do not overlap. **Since there is no overlap between the prefixes contained in the same nested level of the tree, the rules contained in the same Nested Level Tuple do not create any crossproducts.**(See Figure 4.11). This gives us one bound on the number of subsets such that each subset contains overlap-free rules.

We experimented with our rule sets to obtain the number of NLTs in each. The numbers are presented in Table 4.1. While a consistent relationship can't be derived between the number of rules and the number of NLTs from the observations, it is clear that even a large rule set containing several thousand rules can map to less than 200 NLTs. The maximum NLTs were found to be 151 for approximately 25,000 rules. Given that there are very few NLTs compared to the PLTs, it becomes feasible to use the bitmap to indicate the subsets to which a prefix belongs. Therefore, it also

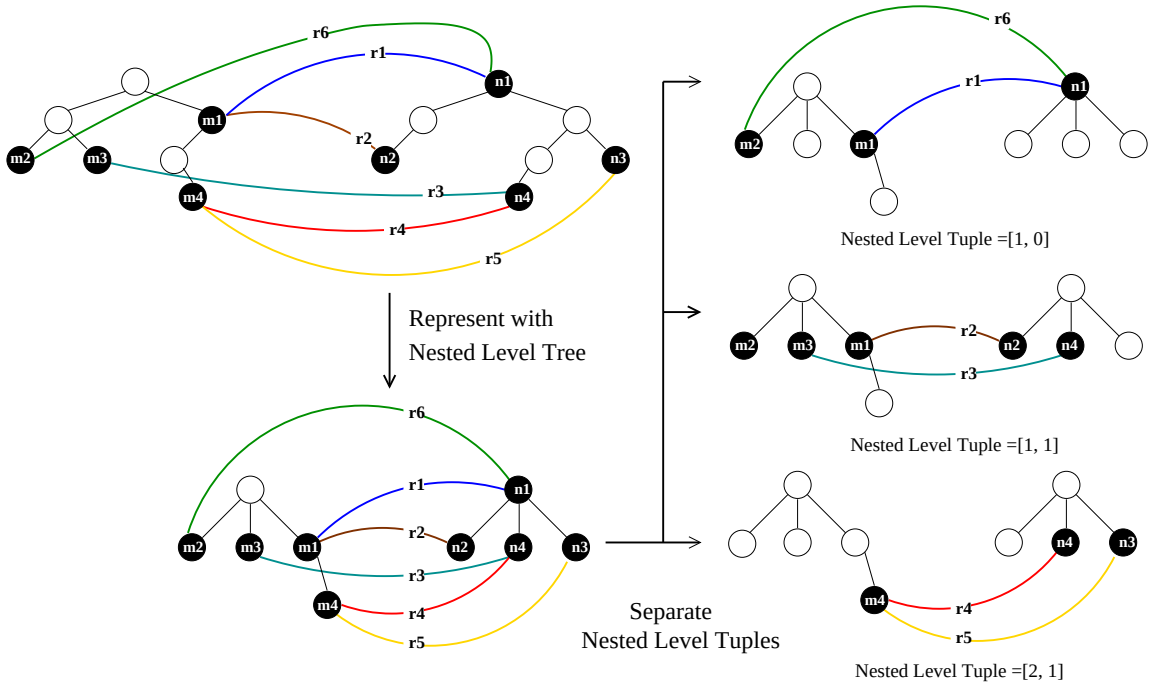


Figure 4.11: **Overlap free grouping of rules**

becomes feasible to intersect the bitmaps associated with the longest matching prefix of each field for pruning the rule subsets to lookup.

However, given an NLT, we only know the nested level associated with each prefix. We don't know the exact prefix length to use to form our query key for that NLT rule set. Therefore, we need to maintain another bitmap with each prefix which gives a prefix length to nested level mapping. We call this bitmap a PL/NL bitmap. For instance, for an IP address prefix, we would maintain a PL/NL bitmap of 32 bits in which a bit set at a position indicates that the prefix of the corresponding length is present in the rule set. Given a particular bit that is set in the PL/NL bitmap, we can calculate the nested level of the corresponding prefix by summing up the number of bits set before the given bit. Consider an 8 bit IP address and the PL/NL bitmap associated with it:

IP address	:	10110110
PL/NL bitmap	:	10010101

The prefixes of this IP address available in the rule set are: 1* (nested level 1), 1011* (nested level 2), 101101* (nested level 3,) and 10110110 (nested level 4). To get the nested level of 101101*, we need to sum all the bits set in the bitmap up to the bit corresponding to the prefix. If we are interested in knowing the prefix length at a particular nested level, then we can keep adding the bits in the PL/NL bitmap until it matches the specified nested level and return the bit position of the set bit as the prefix length. Thus, we can construct the PLT from a NLT using the PL/NL bitmaps associated with the involved prefixes. The PLT tells us which bits to use to construct the key while probing the associated rule set (or Bloom filter). The modified data structures and the flow of the algorithm is shown in Figure 4.12. Each prefix entry in the LPM tables has a PL/NL bitmap and a NLT bitmap. For instance, prefix 101* of field 1 has a PL/NL bitmap of 1010 which indicates that the associated sub-prefixes are of length 1 (i.e. prefix 1*) and 3 (i.e. prefix 101* itself). Therefore, the nested level associated with prefix 1* is 1 and with 101* is 2. Another bitmap, NLT bitmap, contains as many bits as the number of NLTs. The bits corresponding to the NLTs to which the prefix and sub-prefixes belong are set. Thus, 101* belongs to all the three NLTs whereas 1* belongs to only NLT 1 and 2. After the longest matching prefixes are read, the associated NLT bitmaps are intersected to find the common set of NLTs for all prefixes. As the figure shows, since the prefixes belong to all the NLTs, the intersection contains all the NLTs. From this intersection bitmap, we obtain the indices of the NLTs to check. From the NLT table, we obtain the actual NLTs. Combining the knowledge from the PL/NL bit maps of each field, we convert the nested level to the prefix length and obtain the list of PLTs. This list tells us how many bits to consider in order to form the probe key. The probe is first filtered through the on-chip Bloom filters and only the successful ones are used to query the off-chip rule tables. As the example shows, the key $\langle 1, 100 \rangle$ gets filtered out and doesn't need the off-chip memory access.

Note that the bitmap technique can be used instead of the prefix length array only because there is a unique nested level or prefix length associated with a subset for a particular field. For a generic multi-subset crossproduct, we can use the bitmap technique since there can be multiple sub-prefixes of the same prefix associated with the same subset. Therefore, we need to list the individual prefix lengths, as shown in Figure 4.6 or 4.8.

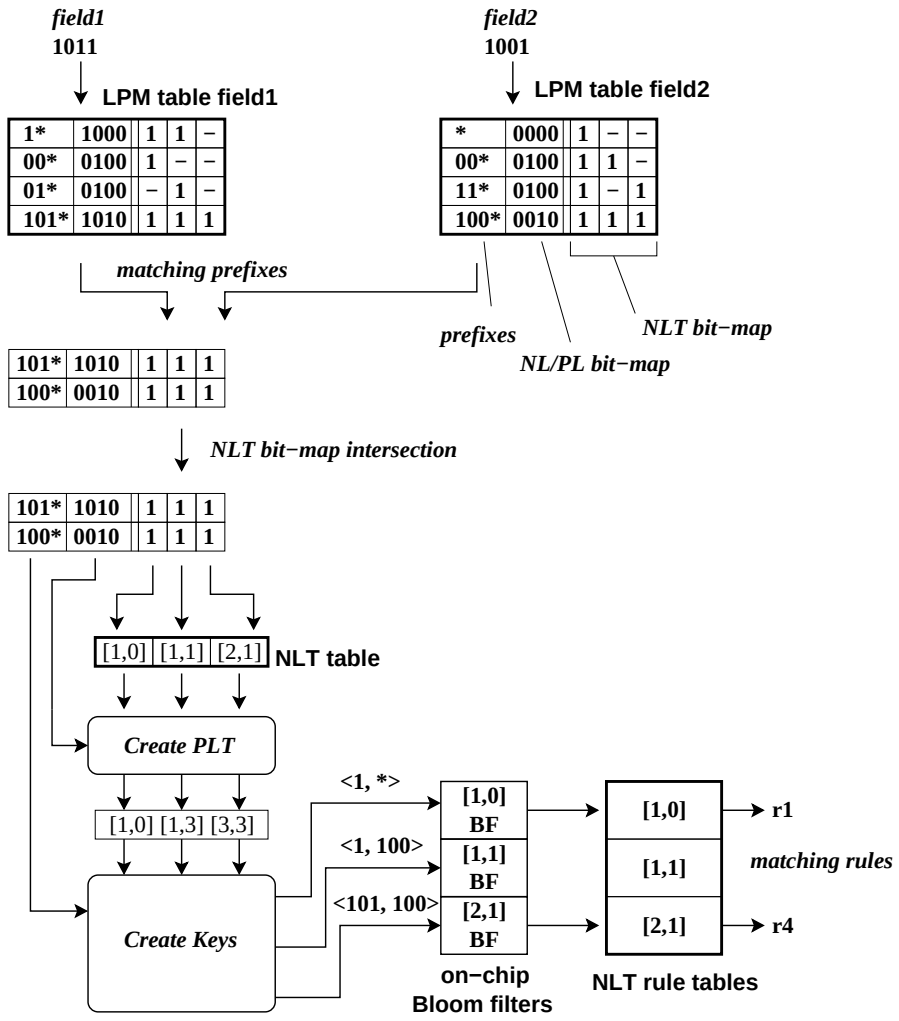


Figure 4.12: **Using NLT based grouping to form the subsets. Each prefix entry in LPM table needs a NL/PL bitmap and another bitmap indicating the NLTs to which the prefix or its sub-prefixes belong.**

4.5.3 Limiting the Number of Subsets

While the NLT based grouping works fine in practice, we might ask, is there still room for improvement? Can the number of subsets be further reduced? This brings us back to our earlier question: how can we limit the number of subsets to a desired value? While the NLT technique gives us crossproduct-free subsets of rules, we can still improve upon it by merging some of the NLTs and applying the crossproduct technique to them in order to limit the number of subsets. Fewer subsets also means fewer Bloom filters and a more-resource efficient architecture. In the next subsection, we

describe our NLT merging technique and the results after applying the crossproduct algorithm.

NLT Merging and Crossproduct (NLTMC) Algorithm

In order to reduce the subsets to a given threshold, we need to find the NLTs that can be merged. We exploit an observation that holds for all the rule sets we analyzed: the distribution of rules across NLTs is highly skewed. Most of the rules are contained within just a few NLTs. In Figure 4.13, on the x-axis, we plot the number of NLTs and on the y-axis, the fraction of rules that are left out of the corresponding number of NLTs. For instance, the curve corresponding to fw3s indicates that only 10% rules are not covered by the first 40 most dense NLTs.

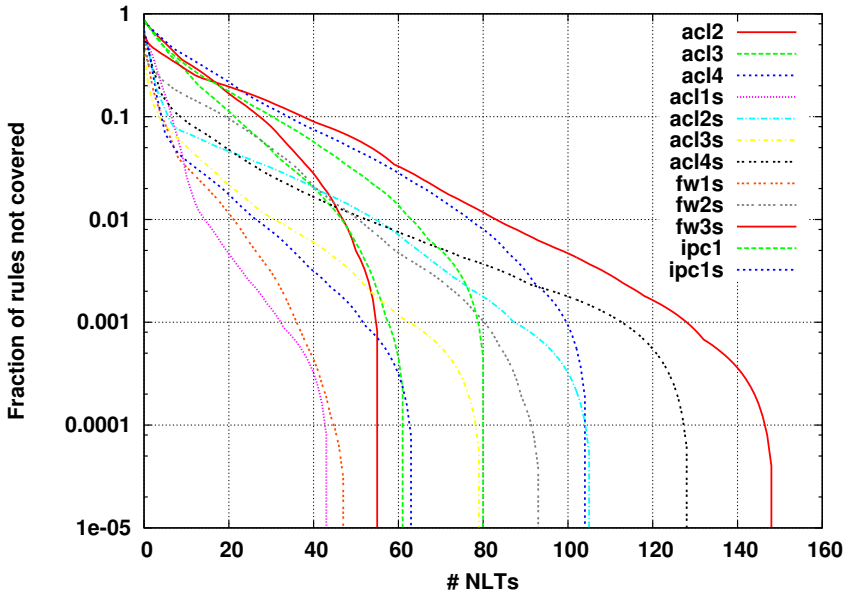


Figure 4.13: **The distribution of the left over rules across NLTs. A large number of rules is covered by a few NLTs. Less than 10% the rules are outside the first 40 NLTs. Only the rule sets with more than 40 NLTs are represented for the purpose of clarity.**

This distribution indicates that we can take care of a large fraction of rules with just a few subsets. We need an NLT-merging algorithm whereby we start with the overlap-free NLT set, retain the densest NLTs equal to the specified subset limit, and then merge the rules in the remaining NLTs to the fixed subsets with the objective

of minimizing the pseudo-rule overhead. It is possible to devise clever heuristics to meet this objective. We provide a simple heuristic that proves very effective in our experiments. Our NLT merging algorithm works as follows.

- Sort the NLTs according to the number of rules in them.
- Pick the densest g NLTs where g is the given limit on the number of subsets. Merge the remaining NLTs to these g NLTs.
- While any of the remaining NLTs can be merged with any one among the fixed g NLTs, a blind merging will not be effective. To optimize the merging process, we choose the most appropriate NLT to merge with as follows. Take the "distance" between the NLT i and each of the fixed g NLTs. We merge the NLT i with an NLT having minimum distance. In case of a tie, we merge with the NLT having minimum rules. We define the distance between the two NLTs to be the sum of differences between individual field nested levels. For instance, the NLTs [4, 3, 1, 2, 1] and [4, 1, 0, 2, 1] have a distance of $|3-1|+|1-0|=3$. The intuition behind the concept of distance is that when the distance between the NLTs is large, it is likely that one NLT will have several descendant nodes corresponding to the nodes in another NLT thereby potentially creating a large crossproduct. A shorter distance will potentially generate fewer crossproducts.
- Although, merging helps us reduce the number of NLTs, it can still result in a large number of crossproducts. At this point, while merging a NLT with another, we try to insert a rule and see how many pseudo-rules it generates. If the number exceeds a threshold, then we don't insert it. We consider it to be a "spoiler." We denote by t this threshold on pseudo-rules to consider a rule spoiler. All the spoilers can be taken care of by some other efficient technique such as a tiny on-chip TCAM. We emphasize that such an architecture will be significantly cheaper and more power-efficient than a TCAM for all the rules. As we will see, our experiments show that the spoilers are typically less than 1% to 2% and the required TCAM overhead is not significant.

In summary, given a fixed number of subsets, we begin by forming NLTs. If the the NLTs are greater than the subset limit, we pick the densest NLTs equal to the

number of subsets and merge the remaining NLTs to the fixed NLTs. While merging, we isolate the spoilers. This proves to be an effective technique to meet the objective of containing the tuples as well as reducing the spoilers, as indicated by the results presented in Table 4.1. We denote by α the ratio of the size of the new rule set after executing our algorithm to the size of the original rule set (after range to prefix expansion). We experimented with different values of g, i.e. the desired limit on NLTs. The pseudo-rule threshold was arbitrarily fixed to $t = 20$.

From the results, it is clear that even with the number of subsets as small as 16, the rule set can be partitioned without much expansion overhead. The average expansion factor for $g = 16$ is just 1.43. Among the 20 rule sets considered, the maximum expansion was observed to be almost four times (acl3s) for 16 subsets. For all other rule sets, the expansion was less than twice. Furthermore, it can also be observed that as we increase the number of subsets, the expansion decreases, as expected. However this trend has an exception for fw3s where both $g = 24$ and $g = 32$ show larger expansion than $g = 16$. This is because $g = 16$ configuration throws out more spoilers than $g = 24$ or $g = 32$. Thus, our algorithm, in this particular case, trades off more spoilers for less expansion. Overall, it can also be observed that the spoilers are very few, on average, $\beta < 2\%$. As we increase the number of subsets, the spoilers is significantly reduced. Clearly, $g = 32$ is the most attractive choice for the number of subsets due to the small number of spoilers and the small expansion factor.

4.6 Architecture

In this section, we describe the architecture of the entire system and discuss some engineering considerations in a hardware implementation of our algorithm.

4.6.1 Hash Table Architecture

An important issue in any hash table-based algorithm is hash collision reduction. Song et. al. proposed a Fast Hash Table (FHT) architecture [29]. This architecture is explained in chapter 2. We borrowed the example from [29] to show how FHT

Table 4.1: **Results with different rule sets. δ denotes the expansion factor on the original rule set after naïve crossproduct. α denotes the expansion factor on the original rule set after Multi-subset Crossproduct. β denotes the *percentage* of the original rules which are treated as spoilers.**

						g=16		g=24		g=32	
rule set	rules	δ	PLT	NLT	prefixes	α	β	α	β	α	β
acl1	1247	2.4e+4	79	31	610	1.03	0.00	1.03	0.00	1.00	0.00
acl2	1216	7.6e+3	195	57	437	1.93	4.19	1.24	1.40	1.17	0.00
acl3	4405	2.3e+5	367	63	1211	1.29	4.45	1.16	0.75	1.14	0.25
acl4	5358	4.3e+5	397	107	1445	1.74	7.95	1.52	2.24	1.20	0.62
acl5	4668	7.0e+2	69	14	304	1.00	0.00	1.00	0.00	1.00	0.00
acl1s	12507	3.2e+4	1349	45	1524	1.03	0.28	1.00	0.10	1.00	0.00
acl2s	18589	1.0e+3	6131	107	626	1.12	2.32	1.14	0.56	1.14	0.39
acl3s	17395	2.5e+4	4136	81	947	3.99	0.71	2.27	0.54	2.26	0.21
acl4s	16291	4.4e+4	4003	130	1090	1.46	2.22	1.45	0.53	1.42	0.42
acl5s	13545	2.3e+4	1197	31	2401	1.03	0.00	1.00	0.00	1.00	0.00
fw1	914	3.0e+5	221	37	205	1.37	0.11	1.10	0.11	1.03	0.00
fw2	543	7.4e+3	159	21	132	1.06	0.00	1.00	0.00	1.00	0.00
fw3	409	1.6e+4	169	29	147	1.25	0.00	1.03	0.00	1.00	0.00
fw1s	32135	5.7e+6	237	50	337	1.92	0.80	1.15	0.012	1.09	0.006
fw2s	26234	1.5e+3	11016	95	271	1.60	2.81	1.46	1.47	1.46	0.42
fw3s	24990	6.7e+3	11296	151	460	1.53	6.45	2.05	1.45	1.80	0.94
ipc1	2179	1.9e+5	244	83	396	1.73	5.69	2.10	1.19	1.41	0.73
ipc2	134	3.1e+2	8	8	72	1.00	0.00	1.00	0.00	1.00	0.00
ipc1s	12725	6.0e+4	3433	65	519	1.86	1.09	1.12	0.26	1.03	0.09
ipc2s	9529	1.7e+4	782	11	4596	1.00	0.00	1.00	0.00	1.00	0.00
avg						1.43	1.95	1.28	0.70	1.20	0.34

functions using Figure 4.14(A). Four items, x, y, z, and w are being inserted in the hash table and the counters of the buckets to which they hash are incremented. After the hash table is pruned by removing the unnecessary copies of the items, it looks as shown in Figure 4.14(B). This hash table significantly reduces the collisions which makes it suitable for our purpose. In fact, all we need to do is convert our ordinary Bloom filter into a counting Bloom filter and associate a hash bucket with it. Each hash bucket keeps the pointer to the list of items hashed to it. As will be explained in the next subsection, we use the ratio of 16 hash buckets per item. With this ratio and using the results from [29], it can be shown that among 128K items, there are only fewer than 75 items that collide. This is a small and acceptable number of collisions. The colliding items can be kept in the on-chip memory. Therefore, it is reasonable to assume that with an FHT, we need only one memory access to read an item from the hash table.

We modify FHT to further reduce the memory consumption by compressing the pointer array.[1] Note that the bucket associated with a non-empty item list is sparse in an FHT. This sparsity can be exploited to compress the pointer array. Figure 4.14(C) illustrates the compression of the pointer array. Let L be the number of items stored in a m-bucket array where $L < m$. We divide the array into smaller segments of s buckets. For each bucket, we maintain a bit indicating if it is occupied or not. Then we keep a pointer to the first item falling in that segment. The first item of all the other lists in that segment are kept in successive memory locations in the item memory. Each of these items can be accessed with reference to the pointer to the first item. When an item in a bucket is to be accessed, we check to see if the bucket is occupied or not. If it is, then we count the number of bits set to 1 within that segment up to the given bucket and add this offset to the base pointer to get the required item. For instance, consider bucket number 4 in the figure which contains item z. To access this item, we first see if the bit corresponding to the bucket is set. Then we count the number of bits set to 1 before the given bit within that segment. There is just one bit set before the bit corresponding to z. Hence, we add the offset 1 to the base pointer associated with that segment and access the required item. The base pointer points to x and z is right next to it. Hence, we retrieve z.

[1] This compression scheme was jointly developed by Haoyu Song and Sarang Dharmapurikar.

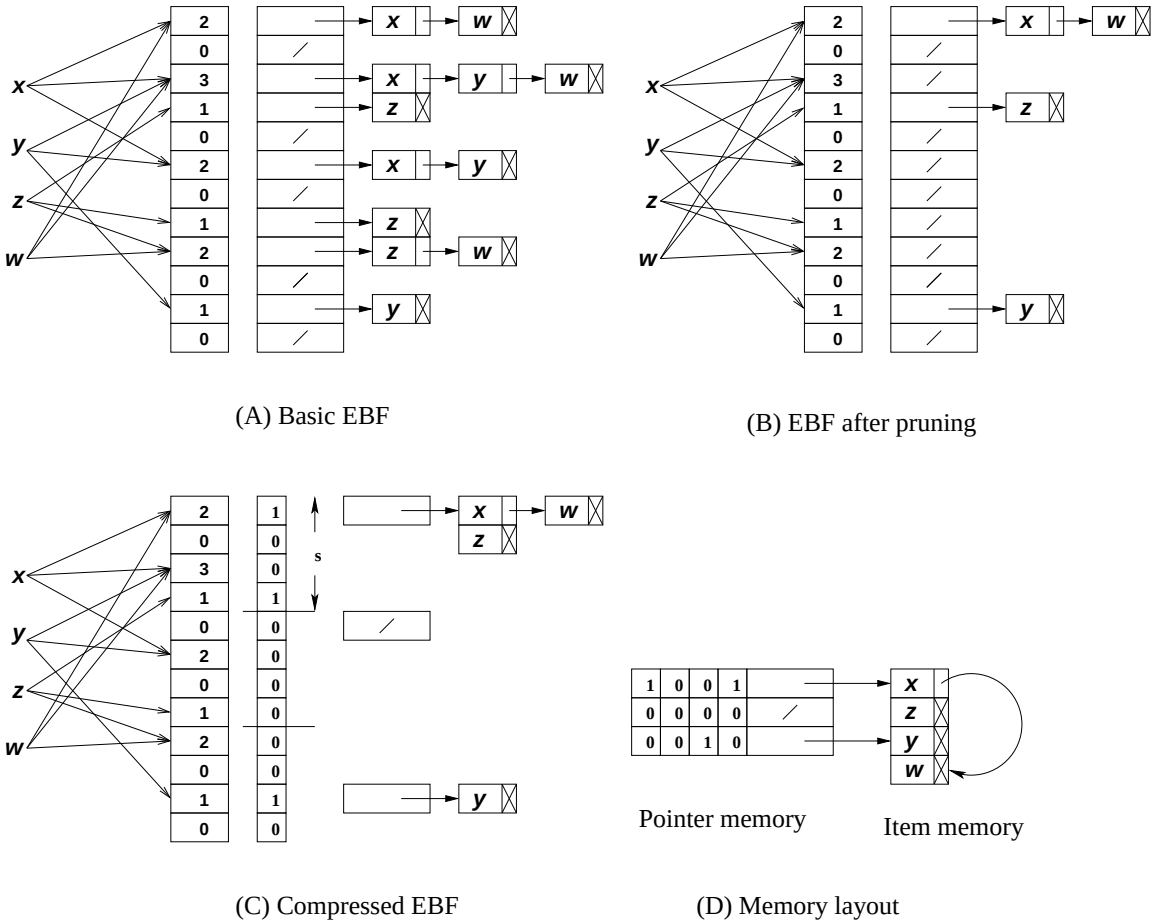


Figure 4.14: **Illustration of Fast Hash Table and its compression. The example is borrowed from [29]. (A) The basic FHT (B) FHT after pruning (C) Compressing pointer array (D) Arranging pointer array compactly in memory**

With this technique, we need just a s-bit vector and a pointer to the first item within that segment as opposed to s pointers. If the length of a pointer is t bits, it results in a space reduction from st bits to $s + t$ bits. The bit-vector and the base pointer can be arranged compactly in the SRAM as shown in Figure 4.14(D). The compression technique using a bit-vector is not new. It has been used in various data structures previously including the encoding of the multibit-trie [18]. However, its application in the context of hash table compression is new. Any hash table can be compressed with this bit-vector technique. In the next subsection, we will evaluate the amount of memory required for the counting Bloom filters and the pointer array to achieve a desired performance.

4.6.2 Memory Requirement

There are three data structures in our algorithm that consume memory. The first is the actual item memory, the second is the counting Bloom filter, the third is the pointer array.

Item Memory: The item memory consists of two types of items: (1) prefixes for all four fields and (2) rules. The prefix entries in the LPM table depend on the algorithm we choose. When we implement the NLTSS algorithm explained in Section 4.5.2, the optimized prefix entry shown in Figure 4.12 contains the 32-bit IP prefix, 32 bits for the PL/NL bitmap, and g bits for the NLT bitmap for as many NLTs. For a port prefix entry, we need 16 bits to specify a prefix and 16 bits for PL/NL bitmap, hence 32 bits less. However, for the sake of uniformity, we use the same amount of space for port prefixes as used for IP address prefixes. Therefore, a prefix entry needs $64+g+2$ bits, the last 2 bits used for specifying the particular field out of source/destination IP and source/destination port. We round it up to the nearest multiple of 36 since SRAM memory is available with this word size. Thus, a single prefix entry requires b_{NLTSS} bits given as follows.

$$b_{NLTSS} = \lceil (66 + g)/36 \rceil \times 36 \tag{4.5}$$

For the NLTMC$_g$ algorithm, the LPM data structure is as shown in Figure 4.4(B). With each prefix, we maintain a word which contains the prefix length information of all the g subsets. Each entry in this array takes a value between 0 to W or NULL where W is the maximum length of the prefix. Therefore, there are $W + 2$ possible values requiring $\lceil log_2(W + 2) \rceil$ bits per entry which is 6 bits for the IP addresses and 5 bits for the ports. For an IP prefix, we need 33 bits to specify a prefix of arbitrary length, $g \times 6$ bits to maintain the sub-prefix information for the g subsets, and 2 more bits to indicate the field to which the prefix belongs. Totally, we need $6g + 34$ bits to store a prefix item for this algorithm. Rounding it up to the nearest multiple of 36 gives us b_{NLTMC_g} bits per entry given as follows.

$$b_{NLTMC_g} = \lceil (6g + 34)/36 \rceil \times 36 \tag{4.6}$$

The actual rule can be specified by using 33 bits for each source and destination IP, 17 bits for each source and destination port, and 9 bits for protocol. If we use 17 bits for the next pointer, a rule item requires 126 bits. Again, we round it up to the next multiple of 36, 144.

To compute the average number of *item bytes* per original rule with all of the above parameters, we use the following formula:

$$M_{of} = \frac{\#rules \times \alpha_g \times 144 + \#prefixes \times b}{\#rules \times 8} \tag{4.7}$$

b is equal to b_{NLTSS} or b_{NLTMC_g} depending on the algorithm. α_g is 1 for NLTSS and as specified in Table 4.1 for the NLTMC with $g = 16$, 24, and 32 subsets.

Bloom filters and pointer array: We next compute the memory required for Bloom filters and pointer array.

We use $k = 12$ hash functions and set buckets per item to 16 (i.e. $m/n = 16$) which give a false positive probability of 0.00046, low enough for our purpose. Keeping the ratio of m/n fixed, we experiment with different values of m. Since FHT needs a counting Bloom filter, each bucket of the Bloom filter is a counter of 2 bits. Moreover, associated with each bucket of the Bloom filter is a hash table bucket containing the pointer to the actual items. Therefore, we have m pointers for n items. If we restrict the maximum number of items in each Bloom filter to 64K, then we can use a 16-bit pointer. We compress the array as described earlier using $s = 16$ as the segment length. Thus, for every 16 entries of the array, we have a 16-bit vector and 16-bit pointer. Therefore, the memory consumption per bucket due to the pointer array is $((m/16) \times (16 + 16))/m = 2$ bits. The total memory consumption *per bucket* due to the pointer array and the counting Bloom filter together is now $2 + 2 = 4$ bits. Since there are 16 buckets per item ($m/n = 16$), the number of bits per item is $4 \times 16 = 64$. The total number of items in the system is simply the number of rules after expansion (#rules $\times\alpha$) plus the unique prefixes of all the fields. Hence, the memory consumption per original rule in *bytes* due to the Bloom filters and the pointer array is

$$M_{on} = \frac{64 \times (\alpha_g \times \#rules + \#prefixes)}{\#rules \times 8} \tag{4.8}$$

Again, $\alpha_g = 1$ for the NLTSS algorithm. The average bytes required per original rule is the sum of the two components:

$$M = M_{of} + M_{on} \tag{4.9}$$

We evaluated this memory requirement for each of our rule sets. The numbers are shown in Table 4.2.

As the table shows, the NLTSS algorithm requires fewer bytes compared to all the configurations of the NLTMC algorithm. This is due to two reasons. First, there is rule set expansion due to crossproducts in NLTMC which is absent from the NLTSS algorithm. Second, NLTMC requires a wider word for each prefix entry in the LPM table. As we increase the number of subsets from 16 to 32, some interesting observations can be made about memory requirement for different rule sets. Consider the acl1 rule set. With an increase in the number of subsets, the LPM entry becomes wider and requires more off-chip memory per rule. On the other hand, acl4 shows exactly the opposite trend because, with fewer subsets, acl4 shows a higher factor of rule set expansion due to crossproducts. Hence, with fewer subsets, the overall memory required per rule is larger. A combination of both of these factors can be seen in acl2 where the memory requirement is highest for $g = 16$ subsets, lowest for $g = 24$ subsets and between these two values for $g = 32$ configuration. This is because, with 16 subsets, there is too much rule set expansion that dwarfs the effect of shorter LPM entry requiring a larger amount of memory per rule. With 24 subsets, the expansion gets reduced and its effect dominates the increase in the LPM entry size. With 32 subsets, the LPM entry becomes wider and hence results in more memory per rule while the effect of reduced expansion is not as significant. Thus, different NLTMC configurations are suitable for different rule sets. However, it is clear that NLTSS always beats all configurations of NLTMC in terms of memory efficiency.

On the other hand, NLTMC requires a fewer and fixed number of subsets of rules whereas the NLTSS requires many more, potentially up to 151 (see Table 4.1). Fewer

Table 4.2: **The performance of different algorithms with different parameters. M_{on} and M_{of} denote the average on-chip and off-chip memory in bytes per rule. The throughput is in Million Packets per second. Throughput was computed for different number of matching rules per packets, $p \leq 4$, $p = 6$, $p = 8$. When $p \leq 4$, LPM is the bottleneck and throughput is decided by how wide the LPM entry is.**

rule set	NLTSS					NLTMC														
						$g = 16$					$g = 24$					$g = 32$				
	Memory		Throughput			Memory		Throughput			Memory		Throughput			Memory		Throughput		
	M_{on}	M_{of}	≤ 4	6	8	M_{on}	M_{of}	≤ 4	6	8	M_{on}	M_{of}	≤ 4	6	8	M_{on}	M_{of}	≤ 4	6	8
acl1	12	25	38	25	19	12	28	38	25	19	12	30	25	25	19	12	34	19	19	19
acl2	11	25	38	25	19	18	42	38	25	19	13	31	25	25	19	12	33	19	19	19
acl3	10	23	38	25	19	12	29	38	25	19	11	28	25	25	19	11	30	19	19	19
acl4	10	25	25	25	19	16	37	38	25	19	14	34	25	25	19	12	31	19	19	19
acl5	8	19	38	25	19	8	20	38	25	19	8	20	25	25	19	8	21	19	19	19
acl1s	9	21	38	25	19	9	21	38	25	19	9	21	25	25	19	9	22	19	19	19
acl2s	8	19	25	25	19	9	21	38	25	19	9	22	25	25	19	9	22	19	19	19
acl3s	8	20	25	25	19	33	73	38	25	19	19	43	25	25	19	19	43	19	19	19
acl4s	8	20	25	25	19	12	28	38	25	19	12	28	25	25	19	12	28	19	19	19
acl5s	9	21	38	25	19	9	22	38	25	19	9	22	25	25	19	9	24	19	19	19
fw1	10	22	38	25	19	13	29	38	25	19	10	25	25	25	19	10	26	19	19	19
fw2	10	22	38	25	19	10	24	38	25	19	10	24	25	25	19	10	26	19	19	19
fw3	11	23	38	25	19	13	29	38	25	19	11	27	25	25	19	11	30	19	19	19
fw1s	8	19	38	25	19	15	35	38	25	19	9	21	25	25	19	9	20	19	19	19
fw2s	8	19	25	25	19	13	29	38	25	19	12	27	25	25	19	12	27	19	19	19
fw3s	8	19	19	19	19	12	28	38	25	19	17	38	25	25	19	15	33	19	19	19
ipc1	9	23	25	25	19	15	35	38	25	19	18	42	25	25	19	13	32	19	19	19
ipc2	12	26	38	25	19	12	28	38	25	19	12	31	25	25	19	12	35	19	19	19
ipc1s	8	19	38	25	19	15	35	38	25	19	9	22	25	25	19	8	20	19	19	19
ipc2s	12	25	38	25	19	12	27	38	25	19	12	29	25	25	19	12	34	19	19	19
avg	10	22	34	25	19	14	31	38	25	19	12	29	25	25	19	12	29	19	19	19

subsets also implies fewer Bloom filters and a more resource efficient architecture. Potentially we can save a significant amount of logic gates resources required to implement Bloom filters if we choose NLTMC, but at the cost of more memory.

4.6.3 Classification Throughput

The speed of the classification depends on multiple parameters, including the implementation choice (pipelined/non-pipelined), the number of memory chips used for off-chip tables, the memory technology used, and the number of matching rules per packet (i.e. the value of p).

Memory technology: We will assume the availability of 300 MHz DDR SRAM chips with 36-bit wide data bus. Such SRAM can allow reading two 36-bit words in each clock cycle of a 300 MHz clock. The smallest burst length is two words (72 bits).

Pipelining: We will use a pipelined implementation of the algorithm. The first stage of pipeline executes the LPM on all the fields and the second stage executes the rule lookup. In order to pipeline them, we will need two separate memory chips, the first containing the LPM tables and the second containing rules. Here, we will also need two separate sets of Bloom filters, the first for LPM and the second for rule lookup. Let τ_{lpm} denote the time needed to perform a single LPM lookup in the off-chip memory in terms of the number of clock cycles of the system clock. Likewise, let τ_{rule} be the time required for a single rule lookup. If a packet matches p rules in a rule set, then, with a pipelined implementation, a packet can be classified in time $max\{4\tau_{lpm}, p\tau_{rule}\}$. Typically, p is ≤ 6 as noted in [24] [22]. We will evaluate the throughput for different values of p.

Choice of algorithm: As before, we have a choice between NLTSS and NLTMC. It should be recalled that depending on the algorithm and the configuration used, the width of an LPM entry can be different. Therefore, LPM lookup time (τ_{lpm}) is different for these two algorithms and different configurations of NLTMC. For the NLTSS, the LPM entry width differs with the rule set under consideration whereas it is constant with a specific configuration for NLTMC. We evaluate the throughput for each rule set. Finally, we always need to read the data in bursts of 72 bits (2 words,

36 bits each) due to which we might need to read more words than we actually need. This too will affect the throughput. Let $\tau_{(lpm,NLTSS)}$ and $\tau_{(lpm,NLTMC_g)}$ denote the time in clock ticks to read a LPM entry for NLTSS and NLTMC$_g$ respectively. These can be expressed as follows.

$$\tau_{(lpm,NLTSS)} = \lceil b_{NLTSS}/72 \rceil \tag{4.10}$$

and

$$\tau_{(lpm,NLTMC_g)} = \lceil b_{NLTMC_g}/72 \rceil \tag{4.11}$$

Recall that each rule can fit 144 bits and needs exactly two clock cycles to read. Hence, $\tau_{rule} = 2$. The throughput can be given as

$$R_{NLTSS} = \frac{300 \times 10^6}{max\{4\tau_{(lpm,NLTSS)}, 2p\}} \ \ packets/second \tag{4.12}$$

and

$$R_{NLTMC_g} = \frac{300 \times 10^6}{max\{4\tau_{(lpm,NLTMC_g)}, 2p\}} \ \ packets/second \tag{4.13}$$

The throughput is shown in Table 4.2. Let's consider NLTSS. When $p \leq 4$, the $max\{4\tau_{(lpm,NLTSS)}, 2p\} = 4\tau_{(lpm,NLTSS)}$ and the LPM phase becomes the bottleneck in the pipeline. Hence, the throughput depends on how wide the LPM entry is. A throughput of 38 million packets per second (Mpps) can be achieved for some rule sets having fewer NLTs and hence shorter LPM entry. When the matching rules per packet increases, the rule matching phase becomes the bottleneck and limits the throughput. With $p = 6$, the throughput is 25 Mpps and with $p = 8$, it is 19 Mpps. In some cases, such as fw3s, the LPM entry is so wide that the LPM phase continues to be the bottleneck and limits the throughput to 19 Mpps even if the matching rules per packet is 8.

Now, let's consider the NLTMC$_g$ algorithm. As mentioned before, for each value of g the LPM entry has a fixed width across all the rule sets. Therefore throughput is constant for all the rule sets. As can be seen from the table, just like

For $g = 16$ and $p \leq 4$, LPM is bottleneck but since the LPM word is short due to smaller number of subsets, the throughput can be as high as 38 Mpps. As p increases, throughput decreases since rule matching becomes the bottleneck. Likewise, for $g = 24$, LPM is the bottleneck up to $p = 6$ and throughput is limited to 25 Mpps. With $p = 8$, rule matching is the bottleneck and throughput reduces to 19 Mpps. For $g = 32$, when $p \leq 4$, the throughput is 19 Mpps because LPM is the bottleneck due to wide entry and it continues to be the bottleneck even if $p = 8$.

With NLTMC, it is clear that the configuration with fewer subsets gives better throughput due to shorter LPM words. On the other hand, fewer subsets can also cause more memory consumption due to more crossproducts as discussed before. Hence, there is a trade-off between throughput and memory requirement. Another interesting point to note is that in some cases, NLTSS shows a better throughput than NLTMC_{16} but in other cases it is the opposite. For fw3s, all the NLTMC configurations offer a consistently high throughput because there are 151 bits in the NLT bitmap of the LPM entries of NLTSS which slows this configuration down. Hence, in such cases, restricting the number of subsets to a smaller value through merging and crossproducts makes sense. Overall, the throughput depends on the nature of the rule set and an appropriate configuration can be chosen to suit the requirements.

4.7 Summary

TCAM is widely used for high-speed packet classification. However, due to the excessive power consumption and the high cost of TCAM devices, algorithmic solutions that are cost-effective, fast, and power-efficient are still of great interest. We have proposed an efficient solution that meets all of the above criteria. Our solution combines Bloom filters implemented in high-speed on-chip memories with our Multi-Subset Crossproducting Algorithm. Our algorithm can classify a single packet in only $4 + p$ memory accesses on an average where p is the number of rules a given packet can match. The classification reports all the p matching rules. Hence, our solution is naturally a multi-match algorithm. Furthermore, the pipelined implementation of our algorithm can classify packets in $max\{4, p\}$ memory accesses.

Due to its primary reliance on memory, our algorithm is power-efficient. It consumes about an average 30 to 45 bytes per rule of memory (on-chip and off-chip combined). Rule sets as large as 128K can be easily supported in less than 5MB of SRAM. Using two 300 MHz 36-bit wide SRAM chips, packets can be classified at OC-192 speed.

Chapter 5

A Simple Multi-String Matching Algorithm

5.1 Introduction

Modern packet processing applications such as Network Intrusion Detection/Prevention Systems (NID(P)S), Layer-7 switches, packet filtering and transformation systems perform deep packet inspection. Fundamentally, all these systems need to make sense of the application layer data in the packet. One of the most frequently performed operations in such applications is a search for predefined patterns in the packet payload. A web server load balancer, for instance, may direct a HTTP request to a particular server based on a certain predefined keyword in the request. Signature-based NID(P)S looks for the presence of the predefined signature strings deemed harmful to the network such as an Internet worm or a computer virus in the payload. In some cases, the starting location of such predefined strings can be deterministic. For instance, the URI in a HTTP request can be spotted by parsing the HTTP header and this precise string can be compared against the predefined strings to switch the packet. In certain cases, one doesn't know where the string of interest can start in the data stream making it imperative for the system to scan every byte of the payload. This is typically true of the signature-based intrusion detection systems such as Snort [1].

Snort is a light-weight NIDS which can filter packets based on predefined rules. Each Snort rule first operates on the packet header to check if the packet is from a source

or to a destination network address and/or port of interest. If the packet matches a certain header rule, then its payload is scanned against a set of predefined patterns associated with the header rule. Matching one or multiple patterns implies a complete match of a rule and further action can be taken on either the packet or the TCP flow. The number of patterns can be in the order of a few thousand. Snort version 2.2 contains over 2000 strings.

In all these applications, the speed of pattern matching critically affects the system throughput. Efficient and high-speed algorithmic techniques that can match multiple patterns simultaneously are needed. Ideally, we would like to use techniques which are scalable with number of strings as well as network speed. Software-based NIDS suffer from speed limitations. They cannot sustain a wire-speed of more than a few hundred mega bits per second. This has led the networking research community to explore hardware-based techniques for pattern matching. Several interesting pattern matching techniques for network intrusion detection have been developed, a majority of them produced by the FPGA community. These techniques use the reconfigurable and highly parallel logic resources on an FPGA to contrive a high-speed search engine. However, these techniques suffer from scalability issues, either in terms of speed or the number of patterns to be searched, primarily due to the limited and expensive logic resources.

We present a fast and scalable pattern matching algorithms using Bloom filters. We store the given set of strings in Bloom filters constructed in high speed and parallel embedded memory blocks in an FPGA. Using these on-chip Bloom filters, a quick check is performed on the packet payload strings to see if any is likely to match a string in the set. Upon a Bloom filter match, the presence of the string is verified by using a hash table in the off-chip memory. Since the strings of interest are rarely found in the packets, the quick check in a Bloom filter reduces expensive memory accesses in a hash table search and greatly improves the overall throughput. However, since the algorithm involves hashing over a maximum length pattern-sized text window, it works well for shorter strings (up to 16 bytes long). In the next chapter, we show how the algorithm can be extended to handle arbitrarily long strings. The basic algorithm presented in this chapter can scan about 1500 strings with 220 Kbits of memory at more than 12 Gbps speed.

The rest of the chapter is organized as follows. In the next section, we summarize related work in hardware-based multi-pattern matching. In Section 5.3 we describe a simple algorithm based on a Bloom filter to match a large number of short strings at very high speeds. We provide the analysis of this algorithm in Section 5.4. We present the evaluation of this algorithm on the Snort string set in Section 5.5.

5.2 Related Work

Multi-pattern matching is one of the well studied classical problems in computer science. The most notable algorithms include Aho-Corasick and Commentz-Walter algorithms which can be considered extensions of the well-known KMP and Boyer-Moore single pattern matching algorithms respectively [14]. Both algorithms are suitable only for software implementation and suffer throughput limitations. The Aho-Corasick algorithm and its drawbacks are detailed in the next chapter. The current version of Snort uses an optimized Aho-Corasick algorithm.

Recently, several interesting algorithms and techniques have been proposed for multi-pattern matching in the context of network intrusion detection. The hardware-based techniques make use of commodity search technologies such as TCAM [42] or FPGAs [12, 34, 7, 30]. Some of the FPGA-based techniques make use of the on-chip logic resources to compile patterns into parallel state-machines or combinatorial logic. Although very fast, these techniques are known to exhaust most of the chip resources with just a few thousand patterns and require bigger and more expensive chips. Therefore, scalability with pattern set size is the primary concern of purely FPGA-based approaches.

An approach presented in [11] uses FPGA logic with embedded memories to implement parallel Pattern Detection Modules (PDMs). PDMs can match arbitrarily long strings by segmenting them into smaller substrings and matching them sequentially.

The technique proposed by Suguwara et. al. in [34] seeks to accelerate the Aho-Corasick automaton by considering multiple characters at a time. Our approach is similar, however, our underlying implementation is completely different. While they use suffix matching, we use prefix matching with multiple machine instances.

Moreover, their implementation uses FPGA lookup tables and is limited in the pattern set size whereas our implementation is based on Bloom filters, which are memory efficient data structures.

From the scalability perspective, memory-based algorithms are attractive since memory chips are inexpensive. Unfortunately, while using memory-based algorithms, the memory access speed becomes a bottleneck. A highly-optimized, hardware-based Aho-Corasick algorithm was proposed in [39]. The algorithm uses a bitmap for compressed representation of a state node in Aho-Corasick automaton. Although very fast even in the worst case (8 Gbps scanning rate), the algorithm assumes the availability of an excessively large memory bus such as 128 bytes to eliminate the memory access bottleneck and would suffer power consumption issues.

A TCAM-based solution for pattern matching proposed in [42] breaks long patterns into shorter segments and keeps them in TCAM. A window of characters from the text is looked up in TCAM and upon a partial match, the result is stored in a temporary table. The window is moved forward by a character and the lookup is executed again. At every stage, the appropriate partial match table entry is taken into account to verify if a complete string has matched. The authors deal with the issue of deciding a suitable TCAM width for efficient utilization of TCAM cells. They also use TCAM cleverly to support wild card patterns, patterns with negations, and correlated patterns. Although this technique is very fast, being TCAM-based, it suffers from other well known problems such as excessive power consumption and high costs. Further, the throughput of this algorithm is limited to a single character per clock tick. Scanning multiple characters at a time would require multiple TCAM chips.

5.3 A Simple Multi-Pattern Matching Algorithm

In a multi-pattern matching problem, we are given a set of strings, $S = \{s_1, s_2, s_3, ..., s_n\}$, and *streaming* data T (which is alternatively called *text*). We would like to find all the occurrences of any strings in S in T. We pre-process the strings in S and build a

machine. We feed the streaming data to this machine which then reports matching strings.

Basically, string matching can be abstracted as a LPM problem (See Figure 5.3. Let's denote $T[i...j]$ as the substring of T starting at location i and ending at j. Consider the set of strings, S_l, in which each string has a length of l bytes. In order to check if any l byte string in T starting at location i, i.e. $T[i..(i+l-1)]$ matches any of the string in S_l, we simply need to *look up* $T[i...(i+l-1)]$ in S_l. A hash table can be used to perform this lookup quickly with an average of $O(1)$ complexity. However, such a table must be maintained for each set of strings with different lengths. Let L be the largest length of any string in S. When we consider a window of L bytes starting at location i, i.e., $T[i...(i+L-1)]$, we have L possible prefix strings $T[i...(i+L-1)], T[i...(i+L-2)], ..., T[i...i]$, each of which must be looked up individually in the corresponding hash table. After these prefixes have been looked up, we can move the window forward by a single byte so that we consider bytes from location $i+1$ to $i+1+L$ that form the window $T[(i+1)...(i+L)]$. The same hash table lookup procedure can be repeated for each prefix in this window. Thus, by looking up all the prefixes of a window and forwarding this window by a byte in each iteration, we ensure that we scan the entire input to look for the predefined strings with fixed lengths. The hash table accesses can possibly be reduced if one of the prefixes is found to match. With each longer prefix string in the hash table, we maintain the list of shorter prefixes that should match. As a result, when we start our hash table probes from the longest prefix, we do not need to continue once we find a matching prefix.

For applications such as network intrusion detection systems, most of the network data does not contain any string of interest. When we perform all the L hash table lookups per byte, most of these accesses result in unsuccessful searches. This gives us the opportunity to use Bloom filters to reduce the unsuccessful hash table accesses just as we did for the IP lookup problem. In the case of string matching, the use of Bloom filters is particularly attractive because the strings of interest are typically several bytes long but consume very few bits in the Bloom filters. Due to this compact representation, it is feasible to maintain a large number of strings in the fast on-chip memory to enable quick lookup.

Similar to our IP lookup architecture, we maintain a separate Bloom filter corresponding to each hash table kept in the off-chip memory. This Bloom filter contains the set of all the strings in the corresponding hash table. Before we execute a hash table query, we query the corresponding on-chip Bloom filter to check if the string under inspection is present in the off-chip hash table. We proceed to check the hash table only if the filter shows a match. Note that verification in the off-chip hash table is necessary since a Bloom filter can show a false match identifiable only after hash table access. However, false positives are rare and most off-chip accesses result only from true matches. Figure 5.3 shows the block diagram of the system. An array of parallel Bloom filters, each corresponding to a hash table, is shown. Each filter is queried with a prefix of the text window under inspection. The Bloom filters can be engineered to execute one query every clock cycle using the embedded memory blocks in FPGAs. If some Bloom filters show a match for the corresponding prefix, we search those prefixes in the off-chip hash table serially starting from the longest matching prefix. We stop when a successful match is found.

The string matching algorithm is shown in pseudo-code of Figure 5.2.

As described in this pseudo-code, to detect strings, we perform a LPM over a L byte window (line 3) and then advance this window by a byte (line 4). The Bloom Filter Lookups (BFL) of line 1-2 in the LPM algorithm are performed in parallel (in a single clock cycle). Since most of the time the match vector of line 3 is empty, the more expensive Hash Table Lookups (HTL) in the off-chip memory are avoided. Therefore, with rare true occurrences of strings in the data stream, our scheme effectively processes one byte per clock cycle. With a system clock of 250 MHz, this results in a throughput of 2 Gbps. We will consider the true and false positives and quantify their effects on the throughput.

The throughput can be improved further by simply deploying multiple identical machines in parallel, each scans the input window with a byte-offset. This is illustrated in Figure 5.3 in which we use $r = 4$ parallel machines. While the first machine scans the input window, $T[i...(i + L - 1)]$, the remaining $r - 1$ machines scan windows $T[(i + 1)...(i + L)]$ to $T[(i + r - 1)...(i + r + L - 2)]$. Each machine can be equipped with its own off-chip hash table or can share a single hash table. In the first case, we can resolve the matches independently which certainly improves the throughput

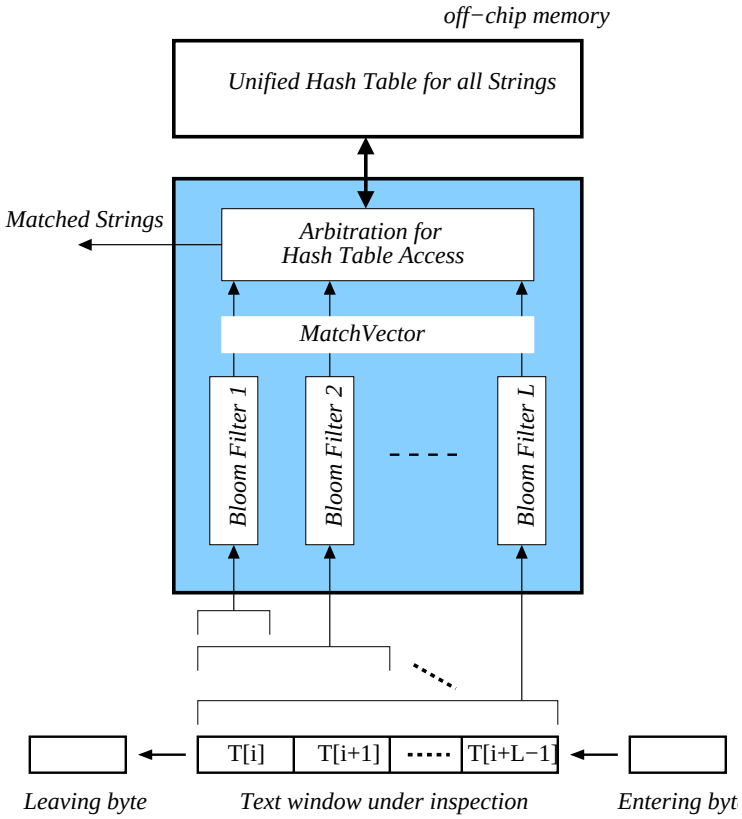


Figure 5.1: **A string matching machine consisting of multiple Bloom filters each of which detects strings of a unique length. The longest string is of length L. Upon a Bloom filter match, the string is looked up in the corresponding hash table. After inspecting a window it is moved by a byte and the lookup procedure is repeated.**

due to more off-chip memory bandwidth. In the latter case, although all matches are checked in the same hash table, a single table suffices since the matches are rare.

5.4 Analysis

We will use the same notations from Chapter 3 for the purpose of analysis.

A hash table is accessed for each string that shows a match. Clearly, an off-chip access is made either when the string being queried is truly present in the hash table or when it is not present but Bloom filter shows a false match. We now revisit Equation 3.2

DetectStrings
1. **while** (text available)
2. $x = T[i...(i + L - 1)]$
3. LPM(x)
4. $i++$

LPM (x)
1. **for** ($i = k$ **downto** 1)
2. $match[i] \leftarrow \text{BFL}_i(x[1...i])$
3. **for** ($i = k$ **downto** 1)
4. **if** ($match[i] = 1$)
5. $\{y[1...i], info\} \leftarrow \text{HTL}(x[1...i])$
6. **if** ($y[1...i] = x[1...i]$)
7. **return** $\{info\}$

Figure 5.2: **String matching algorithm which essentially performs the Longest Prefix Matching (LPM) over the text window.**

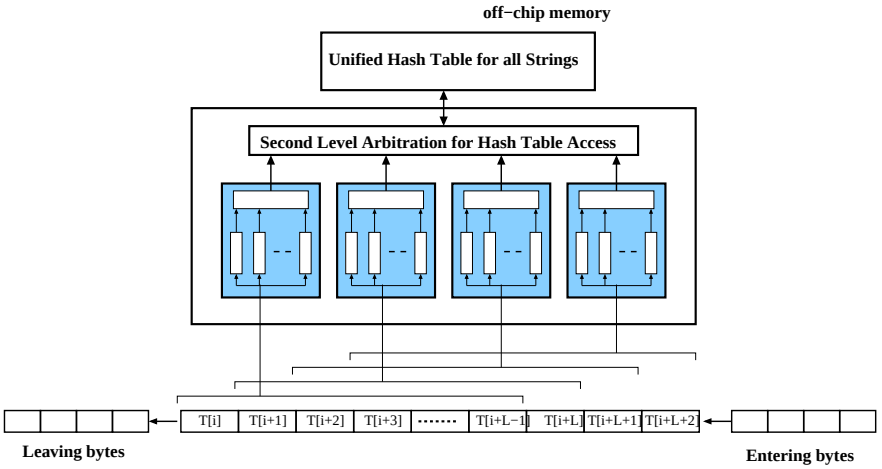


Figure 5.3: **Using multiple parallel engines for better throughput**

to represent the cumulative number of hash table accesses in this LPM process.

$$T_i = p_i + (1 - p_i)(f_i + T_{i-1}) \tag{5.1}$$

with $T_0 = 0$. Starting from i^{th} Bloom filter, we execute a successful search in the hash table with a probability of p_i and accesses required are $p_i t_s$. Otherwise, with

probability $(1 - p_i)$, we get a false positive match and thereby require accesses $f_i t_u$. Additionally, we proceed to the next filter result and repeat the procedure resulting in accesses T_{i-1}.

The memory accesses due to true positives can not be avoided. If each window of input data under inspection contains a true string then it will result in a degraded throughput. This, in general, is not true for the purpose of NIDS string matching; the strings to be searched rarely appear in the streaming data.

We can reuse Theorem 1 in Chapter 3 to simplify the expression 5.1 by deriving a pessimistic upper bound on T_i as follows:

$$T_i \leq \sum_{j=2}^{i} f_j + T_1 \quad for \quad i \geq 2 \tag{5.2}$$

With this pessimistic assumption, the average memory accesses depend on the false positive probabilities of the Bloom filters (which can be tuned by allocating appropriate amount of memory and hash functions) and the true positive probability of just the last Bloom filter, p_1.

When we have L distinct Bloom filters in an engine, the cumulative accesses spent in scanning L byte window is simply T_L. Therefore,

$$T_L \leq \sum_{j=2}^{L} f_j + T_1 = \sum_{j=2}^{L} f_j + p_1 + (1 - p_1) f_1 \tag{5.3}$$

With r parallel engines, the accesses required is rT_L and consume a total time of $\tau r T_L$ where τ is the time required for one memory access while manipulating a hash entry. Apart from time $\tau r T_L$, we need to spend a clock cycle to move the window itself. The total time required to inspect a L byte window is $1/F + \tau r T_L$ where F is the system clock frequency. We will assume that the system clock frequency is the same as the off-chip SRAM memory used to store the hash tables. Since we shift r bytes per $\tau r T_L + 1/F$ seconds, the throughput of the system can be expressed as

$$R_r = \frac{8r}{\tau r T_L + 1/F} = \frac{8}{\tau T_L + 1/rF} \;\; bits/s \tag{5.4}$$

The factor of 8 in this equation is due to byte to bit conversion. We will use this equation to compute the throughput of our system for various configurations. In the next section, we evaluate the performance of our algorithm using the Snort rule set.

5.5 Evaluation with Snort

We used Snort version 2.0 for our experiments. It has 2259 content filtering rules. The total number of strings associated with these rules is 2412. The string length distribution is shown in Figure 5.4.

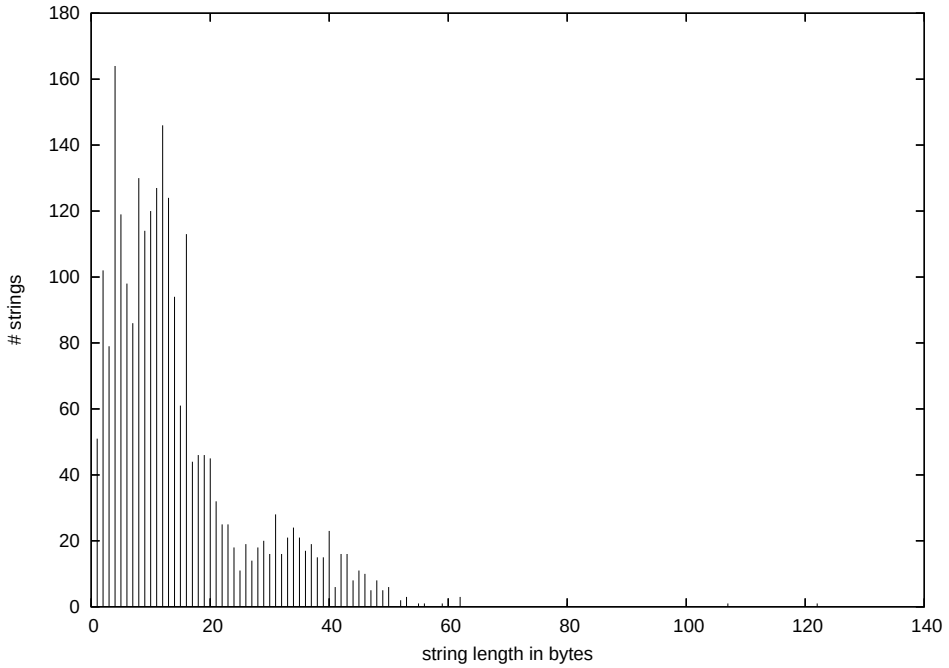


Figure 5.4: **String length distribution. Maximum string length is 122 bytes.**

As the figure shows, Snort rule set contains strings with lengths up to 122 bytes. Since our algorithm requires maintaining a Bloom filter for each string length, we would require several Bloom filters as well as calculation of hash functions over the longest string. Both issues degrade the performance of this algorithm from the practical

implementation perspective. Therefore, the algorithm works well only when we have a small number of unique string lengths and short strings. Our implementation experience indicates that strings of lengths up to 16 bytes can be handled efficiently in FPGA hardware. We will use only the 1576 strings with lengths of up to 16 bytes for the purpose of our evaluation. This number constitutes almost 70% of the strings. In the next section, we extend our algorithm to handle arbitrarily long strings at the cost of a small amount of extra memory.

We begin by constructing Bloom filters of various configurations for the given set of strings. For this experiment, we used eight or sixteen hash functions per Bloom filter. For a Bloom filter, i, with number of hash functions, h, and number of strings, n_i, the optimal amount of memory, m_i, to be allocated is given by the formula [9]:

$$m_i = 1.44hn_i \tag{5.5}$$

We round this number to the next power of 2 since the embedded memories are usually available in power of 2 sizes. The false positive probability of the Bloom filter constructed in this way is given by the formula [9]:

$$f_i = \left(1 - e^{-n_i h/m_i}\right)^h \tag{5.6}$$

To store the off-chip table, we will assume the use of a 250MHz , 64-bit wide, QDRII-SRAM which is available commercially. We also assume $F = 250MHz$, since the latest FPGAs, such as Virtex-4 from Xilinx, can operate at this frequency and in synchronization with the off-chip memory. To store strings of up to 16 bytes in one hash table entry, we would require two words of this SRAM. With some additional information, such as the matching string identifier(s), and the next entry pointer (assuming that hash collisions are resolved by chaining), we would require a few more bytes in the record. We round up the number of bytes to a total of four words of SRAM (32 bytes) per hash table entry. In a carefully constructed hash table, we require approximately one memory access to read one table entry, i.e. two clock cycles

at dual data rate of a 250 MHz, 64-bit wide data bus. Hence, $\tau = 8ns$ in equation 5.3. Furthermore, in equation 5.3, we use $p_1 = 1/100$. For every 100 characters scanned, we have a match for one of the shortest strings in our set. This value may seem rather arbitrary; however, it is quite conservative since our experiments with Snort indicate that the string concentration in the network data is as small as 1/20,000.

Using Equations 5.6, 5.3, and 5.4, we can obtain the theoretical pessimistic average system throughput. In order to see how this system performs practically, we use it to scan a synthetic text composed of random characters interspersed between the strings from the set. We pick up each string from the set and insert random characters between them to ensure that there is only one matching string every 100 characters scanned. We term this value as *string concentration*. The results for various configurations are shown in Table 5.1.

Table 5.1: **The evaluation of our algorithm with Snort string set. A synthetic text was generated with a string concentration of one true string in every 100 characters. The system operates at a speed of $F = 250MHz$. An off-chip QDRII-SRAM operating at the same frequency was assumed to be available for storing the hash tables. The total number of strings considered was 1576.**

# Engines (r)	# hash functions (h)	Total on-chip memory bits $\sum m_i$	Theoretical Throughput (Gbps)	Observed Throughput (Gbps)
1	8	27648	1.89	1.92
2	8	55296	3.60	3.71
4	8	110592	6.57	6.94
8	8	221184	11.6	12.8
1	16	55296	1.96	1.99
2	16	110592	3.84	3.96
4	16	221184	7.39	7.87
8	16	442368	13.7	15.4

From the table, it is evident that the observed throughput is typically greater than the corresponding theoretical throughput since the theoretical throughput was computed with some pessimistic assumptions (such as only the shortest prefix Bloom filter shows the match). It is clear that we can construct a system to scan data at as much as 12 Gbps speed with as little as 220 Kbits of on-chip memory. Also, using more hash functions per filter shows diminishing returns. Although the throughput is better

with more hash functions, the extra memory needed offsets this gain. For instance, with 220 Kbits of memory, we can construct eight engines each with 8 hash functions per filter to give 12.8 Gbps throughput. With the same memory, we can construct only four engines having 16 hash functions per filter and get a throughput of 7.8 Gbps. Therefore, in this case, with the same amount of on-chip memory, it is more efficient to construct a larger number of engines each of which shows more false positives than to construct a smaller number of engines which show fewer false positives.

5.6 Summary

We have shown that the LPM algorithm explained in Chapter 3 can be used to perform multi-string matching on streaming data. Our system maintains one hash table for a set of strings having a particular length. All the hash tables are kept in the off-chip commodity memory. A Bloom filter is maintained corresponding to each hash table in the on-chip memory. Given a window of streaming data, we perform an LPM to see if any prefix is a matching string. Then we move the window by a byte and repeat the procedure. By scanning the data in this fashion, we can discover the matching patterns at any arbitrary location. In network intrusion detection, the true strings are rarely found in the network data, so the window can be shifted quickly without having to perform any off-chip memory accesses other than when Bloom filters show false positives.

Through theoretical analysis and simulations, we have shown that with a true string occurrence rate of once per hundred bytes, a commodity SRAM chip running at 250 MHz, 220 Kbits of on-chip memory, and with 1500 strings to search, we can scan data at the rate of 12 Gbps. Although performance is appealing, a limitation of this technique is that it will work well for short strings (up to 16 bytes long) and fewer unique string lengths since hash computation over long strings becomes computationally intensive and impractical. In the next chapter, we will see how we can get rid of the string length limitation and make this technique scalable to arbitrary string lengths at the cost of more memory.

Chapter 6

Accelerated Aho-Corasick Algorithm

As mentioned earlier, the LPM-based string matching algorithm works well for short strings and strings with fewer unique lengths. Hash computation over long strings (100 characters or more) consumes more resources and introduces more latency. In this chapter, we extend the algorithm presented in the previous chapter to handle arbitrarily large strings. The basic idea behind this new algorithm is to split longer strings into multiple fixed-length shorter segments and use our first algorithm to match the individual segments. For instance, if we can detect strings of four characters in length using Bloom filter techniques and if we want to detect a string "technically," then we cut this string into three pieces "tech," "nica," and "lly." These segments can be stitched in a chain and represented as a small state machine with four states q_0, q_1, q_2, and q_3 as shown in Figure 6.1.

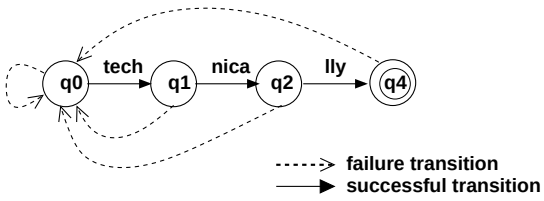


Figure 6.1: **Illustration of basic technique for handling long strings.**

We start from q_0 and check to see if we get a match for "tech" in the text stream. Upon a match, we proceed to state q_1. From q_1, we jump to q_2 if the next four text characters match "nica." Otherwise, we return to q_0. Similarly, from q_2, we go to q_3 if we see a match for "lly," otherwise, we make a failure transition to q_0. Upon

reaching q_3 we declare a complete match for the string "technically." At each state, the matching for the associated segments can be performed using the Bloom filter technique.

When this technique is generalized to multiple strings, it essentially results in an Aho-Corasick automaton in which the underlying alphabet consists of symbols formed by a group of characters instead of a single character. We show how a regular Aho-Corasick automaton can be transformed into an automaton which is suitable for matching multiple characters at a time as opposed to a single character. Moreover, we show how this transformation allows us to *parallelize* the Aho-Corasick algorithm. Once we parallelize the Aho-Corasick automaton, we can use multiple instances to achieve a required speedup by advancing the text stream, by multiple characters at a time. Most importantly, we show how each automaton can be implemented using Bloom filters which help us not only reduce the off-chip memory requirement but also suppress off-chip memory access to a great extent. When a string appears in the text stream, our machine performs memory accesses. This slows down the string matching process. However, since the typical text stream in the context of NIDS rarely contains strings of interest, our algorithm can maintain the desired speedup for such a text stream.

The rest of the chapter is organized as follows. We discuss the basic Aho-Corasick algorithm and highlight its drawbacks from the perspective of hardware implementation. We then present our algorithm in Section 6.2. The numerical analysis of the performance is presented in Section 6.3 and finally the simulation results with a Snort string set are presented in 6.4. Section 6.5 summarizes this chapter.

6.1 Aho-Corasick Algorithm

For a given set of strings, the Aho-Corasick algorithm constructs a finite automaton (FA). Figure 6.2 shows an example of a constructed FA for a set of strings.

Pattern matching is easy: given a current state in the automaton and the next input character, the machine checks to see if the character causes a failure transition. If not, then it makes a transition to the state corresponding to the character. Otherwise, it

s1 : technical
s2 : technically
s3 : tel
s4 : telephone
s5 : phone
s6 : elephant

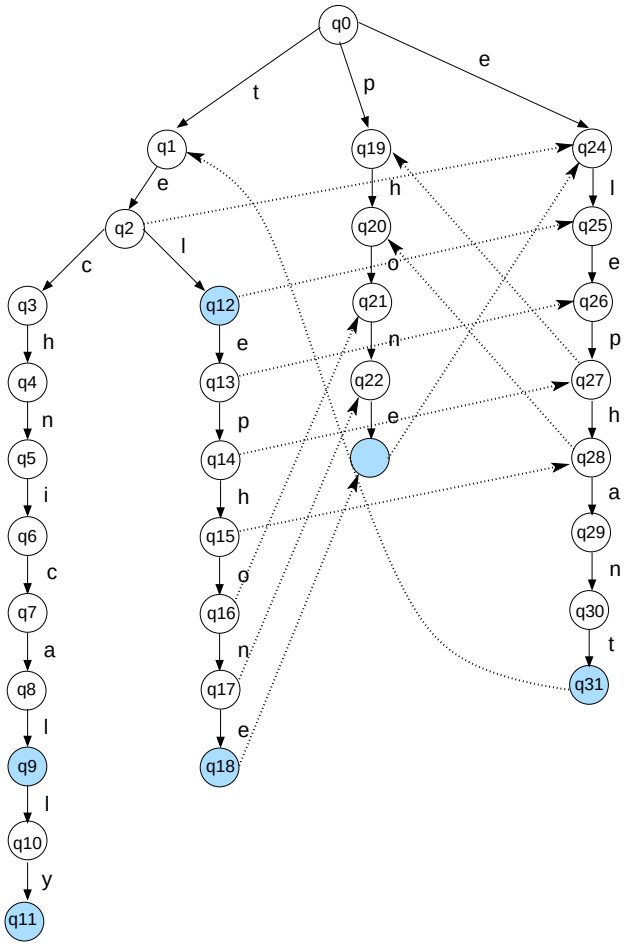


Figure 6.2: **Building an Aho-Corasick FA for a set of strings. Failure transitions to only non q_0 states are shown for the purpose of clarity. All other states make a failure transition to q_0**

makes a *failure* transition. In case of a failure transition, the machine must reconsider the character causing the failure for the next transition and the process is repeated recursively until the given character leads to a non-failure transition. Note that eventually there will be one non-failure transition since all the transitions from q_0 are always non-failure transitions. By falling down the chain of failure transitions, the machine can reach state q_0 in the worst case. Only after the transition settles to a particular state can the next character from the text be considered.

The first fundamental problem the Aho-Corasick algorithm suffers is a high memory access requirement. At least one memory access is needed to read the state node on each input character. Furthermore, the sequential failure transitions can cause more

memory accesses. In the worst case, the average number of memory accesses required per input character is two. Therefore, the throughput of the system can severely degrade if the high-latency and slow-speed commodity memory chips are used.

The second problem with the regular Aho-Corasick algorithm is that it can not be readily parallelized. Hence, we are forced to consider only one character at a time from the text stream regardless of how much logic or memory resources are available. The processing of one character per clock cycle of the system clock can create a bottleneck for high speed networks.

One approach to improve the throughput by scanning multiple characters at a time is to simply use multiple instances of the automata working in parallel consuming a single character at a time. While this is possible, due to the memory requirement of each automaton, we would need separate external memory chips for each automaton to achieve the desired speedup. Using multiple memory chips is not an attractive solution due to the high costs, a larger number of pins to interface the chips, and the resulting power consumption. Moreover, with such an implementation, a single TCP flow needs to be handled by only one of the automata which can potentially lead to an imbalanced throughput across flows.

We describe a hardware-based implementation of our modified version of the Aho-Corasick algorithm which addresses the two problems mentioned. First, we parallelize the Aho-Corasick algorithm and then use on-chip Bloom filters to suppress the memory accesses to off-chip memory.

6.2 A Scalable and High-Speed Algorithm

6.2.1 Basic Ideas

We reorganize the Aho-Corasick automaton to consider k characters at a time. While executing this FA, given the current state of the FA, we directly jump to a state to which we would eventually go by considering one character at a time in the next k characters.

With respect to Figure 6.2, and assuming that $k = 4$ characters at a time, let the machine be in the state q_0 and the string "tech" be given as an input. The machine then jumps to state q_4 and continues its execution from this state by looking at the next k characters. If the next four characters are "nica," then it jumps to q_8. While in q_4, if any other string (e.g. "nice") is seen, then we simply make a failure transition to the failure state associated with q_4 since we know that this results in a failure transition eventually. Thus, we do not need to track the states sequentially all the way to q_7 by considering the characters 'n', 'i' and 'c' and make a failure transition due to 'e'. From the failure state of q_4, we can reconsider the same k characters again.

By comparing the next k characters with all valid k character segments associated with a given state, we can determine if the machine eventually goes to a non-failure or failure state. For instance, the valid 4-character strings associated with state q_0 are "tech", "tele", "phon," and "elep" which lead the machine to states q_4, q_{13}, q_{22}, and q_{27} respectively. Likewise, "nica" is a valid 4-character segment associated with q_4 which leads the automaton to q_8.

Thus, in our new FA, we treat a group of k characters as a *single symbol*. The new FA essentially jumps k characters ahead. We call this variation of the algorithm Jump-ahead Aho-CorasicK FA (JACK-FA). Tracking the states in this fashion, however, eventually requires us to match fewer than k characters to detect a string completely. For instance, after matching "tech" and "nica," we go to q_8. From this state, we need a match for either "l" or "lly" to detect an entire valid string and these segments contain less than four characters. We refer to these segments as *tails* associated with strings. "l" and "lly" are tails of "technical" and "technically." The JACK-FA for the example string set is illustrated in Figure 6.3.

Briefly, our algorithm first matches the longest prefix of strings having a length of multiple of k characters using JACK-FA. After this prefix is found to match, we check to see if the tails associated with that particular string match any prefix of the next k character text. When a matching tail is found in the next k characters, a string matches completely. When we scan k characters to look for a matching tail, we can start from the longest prefix of these k characters and move towards the shortest prefix. We stop when we find the longest matching prefix. By keeping the information regarding any shorter prefix that should match along with this longer

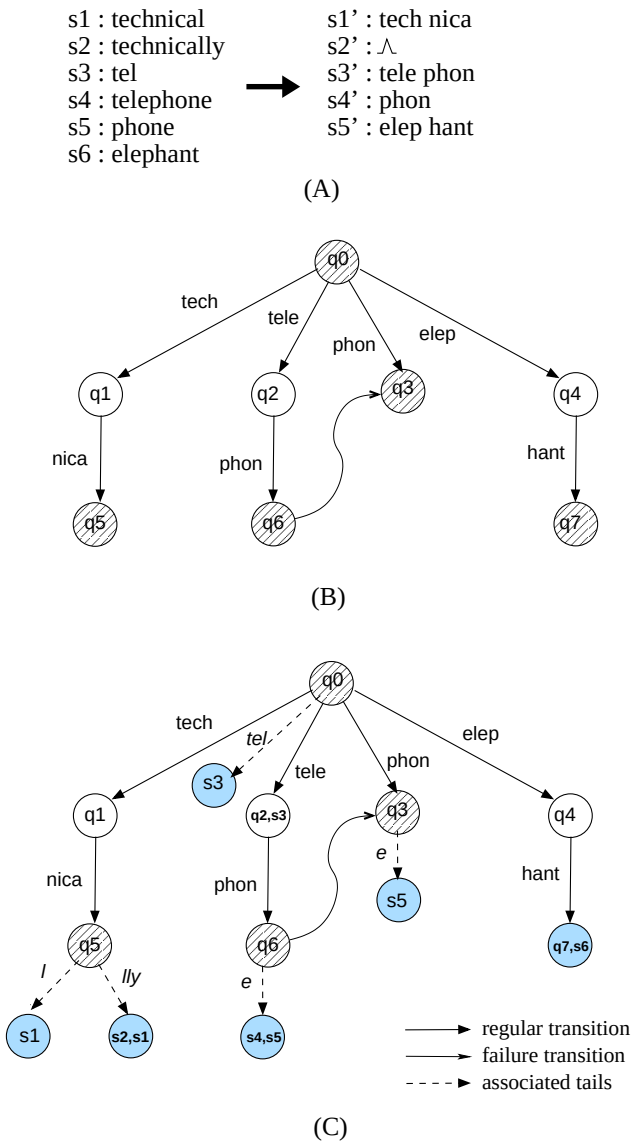


Figure 6.3: **(A) The original string set and modified string set. The space between the k character boundary is shown as a demarcation (B) Jump-ahead Aho-Corasick (JACK) FA. The nodes with a dashed pattern indicate a state corresponding to a matching substring. Failure transition to only non q_0 states are shown. Failure transitions of the remaining states are to q_0 (C) JACK-FA with tails associated with states.**

prefix, the algorithm can correctly match all the patterns. For instance, when the machine is in state q_5 and inspects the next four characters, if it finds a match for "lly," then it can stop and report a match for strings "technical" and "technically". It doesn't need to match the shorter prefix "l." Likewise, when the machine is in q_0

and finds a match for the four-character segment "tele," then it can directly jump to state q_2 and also report a match for the string "tel."

Thus, in any state, the machine looks at the next k characters for a match of all of them (in which case it makes a transition to another state) or looks for a matching prefix of k characters (for tail matching). Therefore, it is the same as looking for the longest matching prefix of a $k-$character substring where the prefixes to be searched are specific to a state and they change when the state changes.

To match the string correctly, our machine must capture it at the correct $k-$character boundary. If a string were to appear in the middle of the k character group, it will not be detected. For instance, if we begin scanning text "xytechnical...," then the machine will view the text as "xyte chni cal.." and since "xyte" is not the beginning of any string and is not a valid 4-character segment associated with state q_0, the automaton will never jump to a valid state causing the machine to miss the detection. Thus, we must ensure that the strings of interest appear at the correct byte boundary. To do this, we deploy k machines which scan the text with one byte offset. In our example, if we deploy four machines, the first machine scans the text as "xyte chni cal.." The second machine scans it as "ytec hnic al..," the third machine as "tech nica l..," and the fourth machine as "echn ical ...". Since the string appears at the correct boundary for the third machine, it will be detected by it. Therefore, by using k machines in parallel, we will never miss detection.

These k machines need not be *physical machines*. They can be four *virtual machines* emulated by a single physical machine (See Figure 6.4). We use $k = 4$ virtual machines that each scan the text by a character offset with respect to the neighboring machines.

To implement these virtual machines, we maintain k independent states, $state_1$ to $state_k$ and initialize each to q_0. When we start scanning the text, we consider first k characters $T[1...k]$, and compute a state transition of $state_1$ and assign the next state to $state_1$. Then, we move a character ahead and consider characters $T[2...k+1]$ and update $state_2$ with these characters as the next symbol. In this way, we keep moving a byte ahead and update the state for each virtual machine. After considering the first $k-1$ bytes, bytes $T[k...2k-1]$ will update the last virtual state machine to complete a cycle of updates. Now we consider characters $T[k+1...2k]$ and update

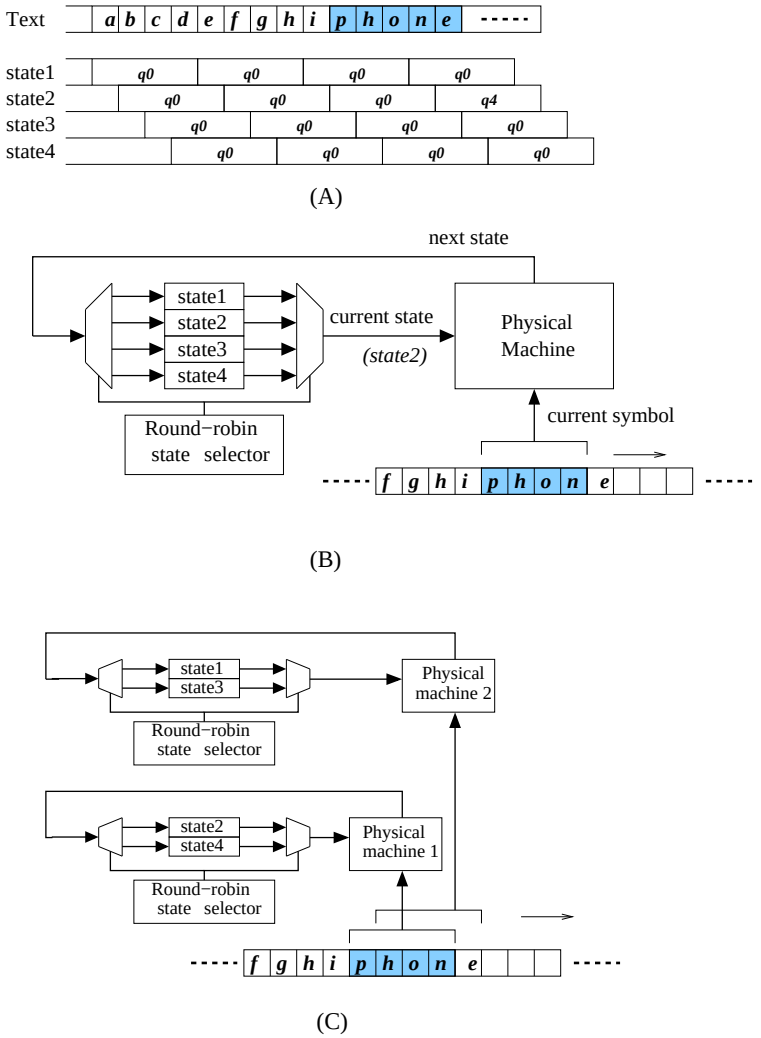


Figure 6.4: **Illustration of Virtual Machines. (A) Each virtual machine i maintains its $state_i$. This state is updated every k iterations. In each iteration we update k states corresponding to k virtual machines. (B) Machines are virtual since only the *state* variable of each machine is independent. The component that updates the state is the same for all. We call it a physical machine. (C) Multiple virtual machines can be implemented using the same physical machine. The figure shows that machine 1 and 3 are implemented by one physical machine and machine 2 and 4 by another. This gives a speedup of two**

machine 1. We repeat this round-robin of updates to each virtual machine. This ensures that each character is considered only once by a virtual machine and each machine will always look at k characters at a time. Clearly, one of the machines will always capture and detect the strings at the correct boundaries. Note how machine 2 makes a transition from q_0 to q_4 after receiving the symbol *phon* at the correct

boundary. After reaching in q_4 and considering the next four characters, the machine outputs *phone*.

With virtual machines, we still consume only a character at a time. We gain the speedup by using multiple physical machines to implement the virtual machines. Note that the virtual machine operations are independent and can be implemented in parallel. Moreover, the number of physical machines need not be equal to the number of virtual machines; they can be less than the number of virtual machines where each physical machine implements multiple virtual machines. To shift the text stream by r characters at a time, we use r physical machines to emulate k virtual machines where $r \leq k$. This can be illustrated with Figure 6.4(C). If we consider $k = 4$ virtual machines which keep $state_1$ to $state_4$ and would like a speed up of just two then two machines can be used. In the first clock cycle, two machines update $state_1$ and $state_2$ and shift the stream by two characters. In the next clock cycle, they update $state_3$ and $state_4$ and shift the stream again by two characters. The procedure repeats.

To reduce memory accesses in each cycle, we use on-chip Bloom filters containing compressed transition tables of the machine. When we look up the transition table to get the next state from the current state and symbol, we first check the on-chip Bloom filters to see if the state transition should succeed or fail. Only for successful transitions do we need to look up the input key in the off-chip table.

The exact details of the algorithm and data structures are explained in the next subsection.

6.2.2 Data Structures and Algorithm

The challenge is to implement each "physical machine" shown in Figure 6.4 such that the machine requires very few memory accesses to execute a state transition. We begin by representing the JACK-FA using a hash table. Each table entry consists of a pair $\langle state, substring \rangle$ which corresponds to a transition edge of this FA. For instance, $\langle q_0, tel \rangle$, $\langle q_0, phon \rangle$ are the transition edges in the FA. This pair is used as a key for the hash table. Associated with this key, we keep the tuple $\{NextState, MatchingStrings, FailureChain\}$ where $FailureChain$ is the set of states

the machine will fall through if it fails on $NextState$. For instance, the tuple associated with the key $\langle q_0, tele\rangle$ is $\{q_2, s_3, q_0\}$ which means that the machine goes from state q_0 to q_2 with substring $tele$ and from q_2 it can fail to q_0. When it is in q_2, it implies a match for string s_3. Likewise, the tuple associated with $\langle q_2, phon\rangle$ is $\{q_6, NULL, q_3, q_0\}$ implying that there are no matching strings at q_6 and upon a failure from q_6, the machine goes to q_3 from where it can fail to q_0. The entire transition table for the example JACK-FA is shown in Table 6.1. It should be noted that the final state of each FailureChain is always q_0. The keys in which the substrings are tails don't lead to any actual transition and neither have any failure states.

Table 6.1: **Transition table of JACK-FA. This table can be implemented in the off-chip memory as a hash table.**

$\langle state, substring\rangle$	Next State	Matching Strings	failure chain
$\langle q_0, tech\rangle$	q_1	NULL	q_0
$\langle q_0, tele\rangle$	q_2	s_3	q_0
$\langle q_0, phon\rangle$	q_3	NULL	q_0
$\langle q_0, elep\rangle$	q_4	NULL	q_0
$\langle q_1, nica\rangle$	q_5	NULL	q_0
$\langle q_2, phon\rangle$	q_6	NULL	q_3, q_0
$\langle q_4, hant\rangle$	q_7	s_6	q_0
$\langle q_0, tel\rangle$	NULL	s_3	NULL
$\langle q_3, e\rangle$	NULL	s_5	NULL
$\langle q_5, l\rangle$	NULL	s_1	NULL
$\langle q_5, ly\rangle$	NULL	s_2, s_1	NULL
$\langle q_6, e\rangle$	NULL	s_4, s_5	NULL

This table can be kept in the off-chip commodity memory. We now express our algorithm for *just a single physical machine emulating k virtual machines* (Figure 6.4(B)) with the pseudo-code shown in Figure 6.5.

j denotes the virtual machine being updated and i denotes the current position in the text. All virtual machines are modified in the round robin fashion (expressed by the mod k counter in line 12). All states corresponding to the virtual machines are initialized to q_0 (line 2). To execute JACK-FA, we consider the next k characters from the text (line 4) and search for the longest matching prefix associated with the current state of the virtual machine being updated, $state_i$ (line 5). Assume that the

DetectStrings

```
1.   j ← 1,i ← 1
2.   for (l = 1 to k) state_l ← q_0
3.   while (text available)
4.     x = T[i...i + k − 1]
5.     {state, strings} ← LPM(state_j, x)
6.     report strings
7.     l ← 0
8.     while (state = NULL)
9.       {state, strings} ← LPM(state_j.f[l++], x)
10.      report strings
11.    state_j ← state
12.    i ← i + 1,j ← (j + 1 mod k)
```

Figure 6.5: **Algorithm for detecting strings.**

LPM process returns a set of matching strings, the next state of the machine, and the failure chain associated with the next state. If the next state is valid, we update the state of the current virtual machine (line 11). If the next state is NULL, then we execute the same procedure with each of the states in the failure chain (lines 9-11) starting from the first (line 8). We stop falling through the failure chain once we find a successful transition from one of them. When we perform LPM for failure states, we report the matching strings at each failure node as we trickle down the failure chain (line 10).

Now, consider the $LPM(q, x)$ process to find the longest matching prefix of x which is associated with q. This could be easily performed by probing the hash table with the keys $\langle q, x[1...i]\rangle$ starting from the longest prefix, $x[1...k]$. We could continue probing until we find a matching prefix. If none was found then we could return NULL. However, this naïve process will require k memory accesses for as many hash probes in the worst case. For applications such as NIDS, the true matches are rare and most of the time the machine results in a failure transition. We can use Bloom filters to filter out unsuccessful searches in the off-chip table. We first group all the keys containing the prefixes of the same length. For instance, the keys $\langle q_5, l\rangle$, $\langle q_3, e\rangle$, and $\langle q_6, e\rangle$ form one group. $\langle q_0, lly\rangle$ forms another group. Then we store all the keys of a group in one Bloom filter which is implemented using a small amount of on-chip memory. Since k groups are possible with k unique prefix lengths, we maintain as

many parallel Bloom filters, each corresponding to a unique prefix length as shown in Figure 6.6.

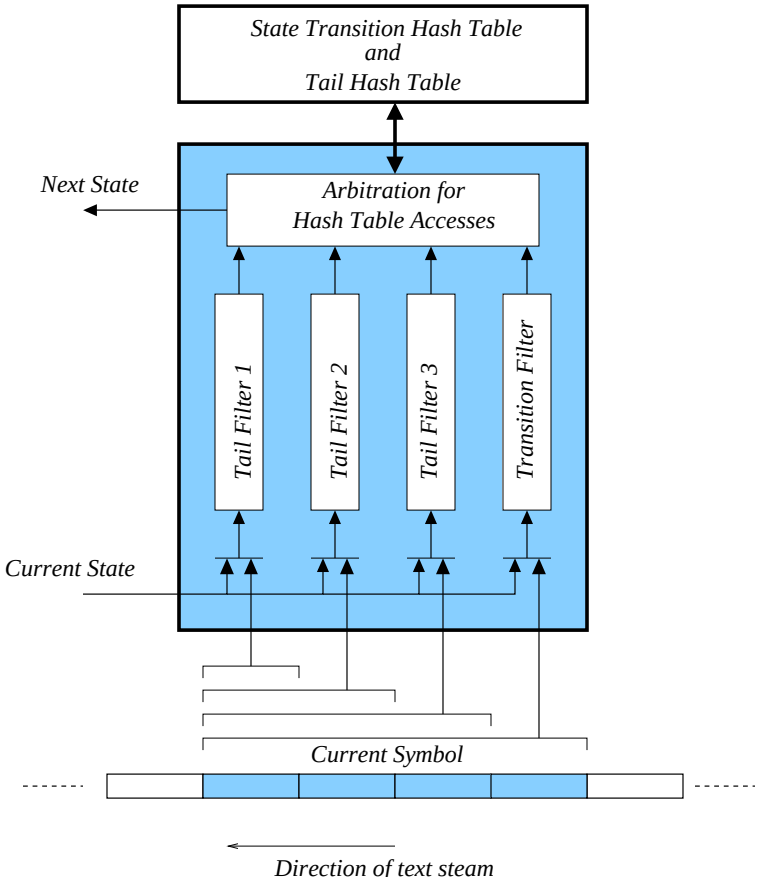


Figure 6.6: **Implementation of a physical machine. For this figure, $k = 4$. The pair $\langle state, prefix \rangle$ is looked up in the associated Bloom filter before off-chip table accesses.**

Before, we look up the pair $\langle q, x[1...i] \rangle$ in the off-chip table, we probe the on-chip Bloom filter corresponding to length i. In total, k parallel queries are performed to Bloom filters and the bit array of their results, *match* vector, is obtained. Since we can engineer a Bloom filter to give the result in a single clock cycle (see [16]), we immediately know which of the k keys are a possible match by looking at the bits in *match* vector. We then walk through the *match* vector from the longest to the shortest prefix and execute the hash table lookup, which we refer to as HTL($\langle q, x[1...i] \rangle$), for a prefix showing a match in the filter. These operations are described through the pseudo-code in Figure 6.7.

LPM (q, x)
1. **for** $(i = k$ **downto** $1)$
2. $\quad match[i] \leftarrow \text{BFL}_i(\langle q, x[1...i]\rangle)$
3. **for** $(i = k$ **downto** $1)$
4. $\quad$ **if** $(match[i] = 1)$
5. $\quad\quad \{\langle q', y[1...i]\rangle, next, strings\} \leftarrow \text{HTL}(\langle q, x[1...i]\rangle)$
6. $\quad\quad$ **if** $(\langle q', y[1...i]\rangle = \langle q, x[1...i]\rangle)$
7. $\quad\quad\quad$ **return** $\{next, strings\}$
8. **if** $(q = q_0)$ $next \leftarrow q_0$
9. **else** $next \leftarrow NULL$
10. **return** $\{next, NULL\}$

Figure 6.7: **Algorithm for the Longest Prefix Matching.**

The speed advantage comes from the fact that the operations of lines 1-2 can be executed in parallel and a result can be obtained in a single clock cycle. Furthermore, *match* vector is usually empty with the exception of rare false positives and true positives. Therefore, we rarely need to access the hash table. Note that if no matching prefix is found and if the current state is q_0, then we return q_0 as the next state since it is a failure but there are no failure states for q_0.

6.3 Analysis

Now we analyze the performance of our algorithm in terms of the number of memory accesses performed after processing t characters of the text. The number of $k-$character symbols seen by the first machine is $\lceil t/k \rceil$. Since the second machine scans the text with a byte offset, the number of symbols it sees is $\lceil (t-1)/k \rceil$. Likewise, the number of symbols, y_i seen by the i^{th} machine is

$$y_i = \begin{cases} \left\lceil \frac{t-(i-1)}{k} \right\rceil & \text{for } t \geq i \\ \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

Before we analyze the algorithm, we will derive an expression for the number of memory accesses required to execute an LPM. Recall that in the LPM algorithm, we start inspecting the *match* vector from the longest to the shortest prefix and execute a hash table search for any bit that is set. If the bit was set due to false positive of a Bloom filter, we waste a memory access and proceed to search the next matching prefix. If the bit is set due to a true match, then we stop after the hash table search. We use our results from Chapter 3 for the analysis of the LPM subroutine. T_i denotes the cumulative number of memory accesses spent in hash table accesses from bit i down to bit 1. Let p_i denote the probability that the input to filter i was a true string (also known as true positive probability or successful search probability). Finally, let f_i denote the false positive probability of Bloom i.

$$T_i = p_i + (1 - p_i)(f_i + T_{i-1}) \tag{6.2}$$

with boundary condition $T_0 = 0$. A special case of this equation in which $p_k = 0$ will be used later. In this case, we know that the filter k did not have a true input. This is the case in which a machine fails from a given state and starts evaluating failure states. We will denote the cumulative number of memory accesses for this particular case as T'_k which is

$$T'_k = f_k + T_{k-1} \tag{6.3}$$

where T_{k-1} is given by the recursive relation of Equation 6.2. From Theorem 1 in Chapter 3, we have, $T_i \leq 1 + \sum_{j=2}^{i} f_j$ for $i \geq 2$ As a result, the following holds

$$T'_k \leq 1 + \sum_{i=2}^{k} f_i \tag{6.4}$$

We will analyze the performance for three different cases:

- Worst case text: triggers the most pathological memory accesses pattern
- Random text: composed of uniformly randomly chosen characters
- Synthetic text: randomly chosen characters but with some concentration of the strings of interest

6.3.1 Worst Case Text

To evaluate the worst case behavior, let's assume that the JACK-FA has gone down to a state at depth d after which it falls through the failure chain by consuming the next symbol (the depth of state q_0 is 0). Thus, at this state, when it consumes a symbol, it needs T'_k memory accesses for LPM. Further, the worst case length of a failure chain associated with a state having a depth d is d, including the state q_0. We execute a LPM on each of these states requiring $T'_k d$ memory accesses. Thus, after a failure from depth d state, in the worst case, we require $T'_k d + T'_k = T'_k(d+1)$ memory accesses.

To reach a state with depth d, the machine must consume at least d symbols. Moreover, for each of these symbols, it executes an LPM which stops at the first memory access returning a match for a $k-$character symbol (and not less than k) since only a successful $k-$character symbol match pushes the machine to the next state. This implies that to reach a depth d state, exactly d memory accesses are executed after consuming d symbols. Finally, $d+1^{th}$ symbol causes failure and subsequently $T'_k(d+1)$ memory accesses. The worst case memory accesses for machine i after consuming y_i characters can be expressed as

$$w_i = y_i(T'_k + 1) - 1 \tag{6.5}$$

The total number of worst case memory accesses by all k machines after processing t character text can be expressed as

$$\begin{aligned} W &= \sum_{i=1}^{k} w_i = \sum_{i=1}^{k} (y_i(1+T'_k)-1) \\ &= \sum_{i=1}^{k} \left(\left\lceil \frac{t-(i-1)}{k} \right\rceil (1+T'_k) - 1 \right) \end{aligned}$$

If $t >> i$, i.e., the number of characters consumed is more than the number of machines then we have $t-(i-1) \approx t$. If $t >> k$, i.e., the number of characters consumed is greater than the symbol size then $\lceil t/k \rceil \approx t/k$. Therefore,

$$W \approx \sum_{i=1}^{k} \left(\frac{t}{k}(1+T'_k) - 1 \right) = t(1+T'_k) - k$$

Using Equation 6.4, we have

$$W \leq t(2+\sum_{i=2}^{k} f_i) - k < t(2+\sum_{i=2}^{k} f_i)$$

and the worst case memory accesses per character, M_w,

$$M_w = 2 + \sum_{i=2}^{k} f_i \tag{6.6}$$

By keeping the false positive probabilities f_i moderately low, we can reduce the factor $\sum_{i=2}^{k} f_i$. It should be recalled that the worst case memory accesses for the original Aho-Corasick is $2t$. Thus, our technique doesn't provide any gain over the original Aho-Corasick in the worst case scenario. However, as we will see in the next two

subsections, the average case text, which is what is expected to be seen in typical network traffic, can be processed quickly.

6.3.2 Random Text

We would like to know how our algorithm performs when the text being scanned is composed of random characters. It should be recalled that our state machine makes a memory access whenever a filter shows a match either due to a false or a true positive. Hence, the performance depends on how frequently the true positives are seen. Let b_l^q denote the number of l character branches sprouting from state q. In Figure 6.3, $b_4^{q_0} = 4$ and $b_3^{q_0} = 1$. While the machine is in state q, the probability of spotting one of its l character branches in next k-character symbol from the text is $p_l^q = b_l^q/256^l$. Since, the average case evaluation of the algorithm depends on the true positive probability, we must make some assumptions regarding the value of branching factor for states. We assume that $b_l^q << 256^l$. This is clearly justifiable for values of $l \geq 3$ where $256^3 = 16M$ and the practical values of b_l^q are less than a few thousand. For instance, when we constructed the JACK-FA with Snort string set we found that $b_4^{q_0} = 1253$ which was also the maximum. The remaining states had $b_4^q \leq 4$. Hence, for Bloom filters corresponding to length $l \geq 3$ the probability of a true positive, $b_l^q/256^l$, can be considered negligibly small for practical purposes. Further, we assume that we have very few $2-$character strings in our set and there are no single character strings. With these assumptions, $p_2^{q_0} \approx 0$ and $b_1^{q_0} = 0$. Since for all i, $p_i^{q_0} \approx 0$, the JACK-FA rarely leaves state q_0. Therefore, all the memory accesses performed while scanning random text are due to the false positives shown by Bloom filters. Formally, by substituting $p_i = 0$ in Equation 6.2, we obtain the average number of memory accesses each machine performs while scanning a single symbol as $T_k = \sum_{i=1}^{k} f_i$. Since there are k characters in one symbol, the average memory accesses per machine per character is T_k/k. Since there are k machines, the average memory access per character are $M_a = T_k$:

$$M_a = \sum_{i=1}^{k} f_i \tag{6.7}$$

By keeping the false positive probability of each Bloom filter moderately low, the overall memory accesses can be reduced greatly. Therefore, the random text can be scanned quickly with very few, if any, memory accesses. We will consider Bloom filter design for low false positives and evaluate it with Snort strings.

6.3.3 Synthetic Text

In subsection 6.3.1, we considered the most pathological text which causes two memory accesses per character and in subsection 6.3.2, we assumed that the text was composed of random characters. However, these two cases are extremes of what is seen in reality. Typically, the strings being searched occur with some finite frequency. In Snort, the strings are searched in the packet payload only when the packet header matches a certain "header rule". Therefore, although certain strings are commonly seen in the data stream, they are of interest or are said to occur truly only when they appear within a particular context. To quantify the frequency of string appearance in mostly random text, we define a new parameter called *concentration*, c, as the ratio of the number of string characters in the text to the total number of text characters. For instance, if we spot a 10-character string of our set in a 1000 character text, then the concentration is $c = 10/1000 = 0.01$. We use this value of concentration to model our text input for further analysis. This value may seem rather arbitrary; however, our experiments with Snort indicate that typical concentration is approximately 1/20,000, quite smaller than what we have assumed. To be conservative, we assume that when the string appears in the text (with concentration c), it triggers the most pathological memory accesses: two memory accesses per character (M_w). Otherwise, for random characters (with concentration $1 - c$), the memory accesses are given by M_a. We can represent the average number of memory accesses, M_s, for a synthetic text as approximately

$$\begin{aligned} M_s &= M_w c + M_a(1-c) \\ &= (2 + \sum_{i=2}^{k} f_i)c + (1-c)\sum_{i=1}^{k} f_i \end{aligned} \tag{6.8}$$

The assumption of two memory accesses per string character tends to overestimate the value of M_s. On the other hand, practical Bloom filters with a given configuration of memory and hash function show a higher false positive rate which causes M_s to be underestimated. In order to get a sense of the actual value of M_s we simulate our algorithm over Snort strings and synthetic text with a given concentration of strings.

6.4 Evaluation with Snort

We used the Snort version 2.0 for our experiments which has 2280 content filtering rules. The total number of strings associated with all these rules is 2259. The string length distribution is shown in Figure 5.4.

We simulated our algorithm with $k = 16$. Remember that the larger the k, the smaller the total number of segments and tails. Ideally, we would like the k to be the largest string length. However, since calculation of hash over such a long string is not practical and it would require several Bloom filters to operate in parallel, we resort to the Aho-Corasick, the motivation behind our work. The experimental results in [16] indicate that $k = 16$ is a feasible value. Around 70% strings are within 16 characters and don't need to be broken up (these are the tails associated with q_0) if $k = 16$. With a JACK-FA constructed from Snort strings, the resulting number of items, n_i, in i^{th} Bloom filter within an engine of k Bloom filters is shown in Table 6.2.

It is noteworthy that the number of items in k^{th} Bloom filter is larger than the items in each of the other filters. This is expected since all the strings are first broken up into segments of k characters and thus create maximum number of items of this length whereas segments of any shorter lengths are leftover tails. The amount of memory to be allocated to each filter depends on the number of hash functions, h, used per Bloom filter. We used two values of hash functions, $h = 8$ and 16. For a Bloom filter i, with number of hash functions h, and number of strings n_i, the optimal amount of memory, m_i, to be allocated is given by the formula:

$$m_i = 1.44hn_i \tag{6.9}$$

Table 6.2: **Results of Bloom filter construction.**

		hash functions $h = 8$			hash functions $h = 16$		
i	n_i	m_i (bits)	observed f_i	theoretical f_i	m_i (bits)	observed f_i	theoretical f_i
1	70	1024	0.000000	0.000991	2048	0.005719	0.000001
2	76	1024	0.000972	0.001614	2048	0.000000	0.000003
3	146	2048	0.001092	0.001273	4096	0.000000	0.000002
4	228	4096	0.000290	0.000278	8192	0.000000	0.000000
5	173	2048	0.003331	0.003389	4096	0.000010	0.000011
6	138	2048	0.001240	0.000909	4096	0.000000	0.000001
7	127	2048	0.000490	0.000547	4096	0.000000	0.000000
8	172	2048	0.002490	0.003281	4096	0.000010	0.000011
9	131	2048	0.000620	0.000662	4096	0.000000	0.000000
10	156	2048	0.001720	0.001879	4096	0.000120	0.000004
11	159	2048	0.002700	0.002098	4096	0.000000	0.000004
12	172	2048	0.003859	0.003281	4096	0.000010	0.000011
13	155	2048	0.001600	0.001810	4096	0.000000	0.000003
14	123	2048	0.000490	0.000448	4096	0.000020	0.000000
15	94	2048	0.000070	0.000080	4096	0.000000	0.000000
16	968	16384	0.000180	0.000405	32768	0.000000	0.000000

We round this number to the next power of 2 since usually the embedded memories are available in power of 2 sizes. The memory allocated to all Bloom filters for each configuration of hash functions is also shown in Table 6.2. The false positive probability of the Bloom filter constructed in this way is given by Equation 5.6.

These theoretical false positive probabilities are shown in the table. We created a synthetic text with a string concentration of $c = 0.01$. To achieve this, we scanned all strings with which we created the JACK-FA, one-by-one, with random characters interspersed in between two strings. The total number of random characters inserted were 1000 times the total characters in the string set. The total number of memory accesses and accesses per character were noted. The observed false positive probability of each Bloom filter was also obtained. The values are shown in Table 6.2. The table clearly indicates that the observed false positive rate of each Bloom filter is very close to the one predicted theoretically.

In scanning a large text with t characters, we spend $tM_s\tau$ time in external memory accesses where τ is the average time for one memory access. Moreover, with r physical engines, we spent t/r clock ticks of the system clock in just shifting the text. If F is the system clock frequency, then the total average time spent in scanning t character text is $tM_s\tau + t/rF$ giving us an average throughput of

$$R = \frac{8t}{tM_s\tau + \frac{t}{rF}} = \frac{8}{M_s\tau + \frac{1}{rF}} \ bps \tag{6.10}$$

In order to store the off-chip table, we will assume the use of a 250MHz , 64bit wide, QDRII-SRAM which is available commercially. We also assume $F = 250MHz$, since the latest FPGAs, such as Virtex-4 from Xilinx, can operate at this frequency and in synchronization with the off-chip memory. In order to calculate τ, we must know the size of the off-chip table entry. We assume 4 bytes for state representation. Also, instead of keeping the list of all the matching string IDs, we will keep a pointer to such list which can be implemented in the same or a different memory. A pointer will make the table entry more compact. With $k = 16$ character symbols, the resulting JACK-FA exhibits a failure chain length of just one, i.e. failure to state q_0 for more than 97% of the states. Hence, we will keep space for only one failure state or alternatively a pointer to a chain of states if there is more than one state in the failure chain. Therefore, each table entry can be represented in a 32 byte data structure (4+16 bytes for $\langle state, symbol \rangle$ + 4 bytes for NextState + 4 bytes for Matching Strings Ptr + 4 bytes for FailureChain/Ptr). In a carefully constructed hash table, we require approximately one memory access to read one table entry, i.e. two clock cycles of dual data rate 250 MHz 64 bit wide data bus. Hence, $\tau = 8ns$.

We use the values of the theoretical false positive probabilities of all the Bloom filters from Table 6.2 to compute the theoretical pessimistic average memory accesses per character using Equation 6.8. Substituting it in Equation 6.10, we obtain the theoretical throughput value. At the same time, we observe the average memory accesses per character during the simulation and use it to compute the observed throughput. These results are shown in Table 6.3.

Table 6.3: **The evaluation of DetectString with a Snort string set. A synthetic text was generated with string concentration of one true string in every 100 characters. The system operates at a speed of $F = 250MHz$. An off-chip QDRII-SRAM operating at the same frequency was assumed to be available for storing the hash tables. The total number of strings considered were 2259.**

# Engines (r)	# hash functions (h)	Total on-chip memory bits $\sum m_i$	Theoretical Throughput (Gbps)	Observed Throughput (Gbps)
1	8	47104	1.84	1.88
2	8	94208	3.41	3.57
4	8	188416	5.95	6.45
8	8	376832	9.48	10.8
1	16	94208	1.92	1.96
2	16	188416	3.70	3.87
4	16	376832	6.89	7.49
8	16	753664	12.1	14.1

We see from the table that the observed throughput has always been better than the theoretically predicted due to some pessimistic assumptions involved in deriving the theoretical throughput. The results also indicate that we can construct an architecture to scan data at rates as high as 10 Gbps with just 376 Kbits of on-chip memory. Moreover, increasing the number of hash functions gives diminishing returns since, with the same amount of memory, an architecture with 8 hash functions per filter can do a better job than the architecture with 16 hash functions per filter. Thus, in this particular case, it is feasible to build several less perfect engines (i.e. with high false positive rate) than to build fewer but near perfect engines (i.e. with very low false positive rate).

6.5 Summary

We have presented a hardware accelerated version of the Aho-Corasick multi-pattern matching algorithm for network content filtering and intrusion detection applications. We modified the original LPM-based string matching algorithm presented in Chapter 5 to handle arbitrarily long strings. A long string is broken into multiple

shorter fixed length segments and an Aho-Corasick automaton is built by treating each segment as a single symbol. Each smaller segment is matched using the previous algorithm. The segmentation introduces some insignificant overhead.

The *logic* resources required to implement our algorithm in hardware are independent of the number of patterns or pattern lengths since they primarily make use of embedded on-chip memory blocks in VLSI hardware. Due to its reliance on only embedded memory, we argue that our algorithm is more scalable in terms of speed and the size of the pattern set, when compared to other hardware approached based on FPGA and TCAM. With less than 50 KBytes of on-chip memory, our algorithm can scan more than 2000 Snort patterns at more than 10 Gbps.

Chapter 7

Conclusions

This thesis describes a set of algorithms for network search processing. The primary network search problems addressed in this work are longest prefix matching (LPM) for IP route lookup, 5-tuple packet classification, and pattern matching for deep packet inspection. All the algorithms are based on Bloom filters, the randomized data structures.

We proposed a simple technique that reduces the number of memory accesses involved in all aforementioned search processes. The reduction in memory accesses significantly accelerates these searches. First, we represent each search process as a set of table lookups. Then, the resulting tables are implemented as hash tables for quick lookups. These tables can be maintained in the off-chip commodity memory such as SRAM or SDRAM. However, if all the table lookups are executed, the required off-chip memory accesses would create a performance bottleneck. We observe that in these search processes, most of the table lookups are unsuccessful and if we could avoid them, the overall process could be greatly accelerated. We use Bloom filters to filter these potentially unsuccessful table lookups. Corresponding to each off-chip table, we maintain a Bloom filter in the on-chip memory that contains the same set of lookup keys as the off-chip table. A lookup is performed in the on-chip Bloom filter; only if the key shows a match do we execute the corresponding off-chip table lookup. Bloom filters can also produce false positive matches with a very small probability. A key that shows a match in a Bloom filter but not in the corresponding off-chip table is a false positive. These false positives cause very few additional memory accesses than required. Therefore, the overall memory accesses performed in the process are almost equal to the *necessary* memory accesses. This memory access filtering technique is

useful only if the process involves a large number of unsuccessful table lookups. If all the table lookups are successful then Bloom filters are redundant.

Since Bloom filters allow a very compact representation of the search keys, it is feasible to implement them using a small amount of on-chip memory. With the advanced VLSI technology, it is possible to fabricate very high speeds and multi-port independent memory blocks on a single chip. A Bloom filter can be realized using such multiple blocks and a lookup can be performed very quickly. Thus, our search algorithm uniquely combines algorithmic and architectural techniques.

Now we will summarize the details of each algorithm. In the LPM problem, we are given a set of IP address prefixes and we want to find the longest matching prefix of a given IP address. We form a set of hash tables. Each table contains prefixes of one particular length. Given an IP address, we look up its prefixes in the corresponding hash tables starting with the longest prefix. When a matching prefix is found, the search returns it as the longest matching prefix. In this process, the hash table lookups performed for the prefixes other than the longest matching prefix do not yield any useful result. By looking up the prefix in a Bloom filter first, the expensive hash table access can be avoided if the filter shows no match. A false positive match in a Bloom filter will be detected if the prefix lookup in the corresponding hash table is unsuccessful. With the exception of a few false positives, the longest prefix matching requires a single memory access.

Our packet classification technique is based on the longest prefix matching. For this purpose, a packet classification rule must be represented in the form of prefixes by converting ranges into prefixes. Therefore, a packet classification rule is basically a tuple consisting of prefixes of each field. A rule is said to match a packet when all the prefixes of a rule match the corresponding fields in the packet.

The main observation we exploit is that a match for a prefix also implies a match for its shorter prefixes. This observation can be extended for multiple dimensions. A match for a rule created by combining the matching prefixes of all the fields implies a match for any other rule consisting of the sub-prefixes. Based on this observation, we can perform packet classification using crossproduct. First, obtain the set of prefixes involved in the rules for each field. Then, construct a table of rules by using

all possible combinations of prefixes of each field. This is the crossproduct table. All original rules are contained in this table since each is just one of the possible combinations. The remaining rules are of two types: the rules which, if matched, indicate a match for one of the original rules and the rules that don't indicate a match for anything. The latter can be omitted. Thus, we have two kinds of rules in the resulting crossproduct table, the original rules and the *pseudo-rules* that indicate a match for one or more original rules. In fact, a pseudo-rule is nothing but a rule formed by longer prefixes that shows a match for an original rule consisting of the shorter prefixes. Geometrically, a pseudo-rule represents a region in the 5-dimensional space which is fully contained in the space represented by an original rule. In other words, the pseudo-rule is more specific than the corresponding original rules. Given this construction, matching rules can be found by first performing a longest prefix matching on each field and then looking up the combination of these prefixes in the rule table that can be implemented as a hash table. Using the Bloom filter based LPM technique, approximately four memory accesses are required for LPM on source and destination ports and IP addresses. The protocol field can be looked up in a small on-chip directly indexed table. Finally, the rule formed by combining the matching prefixes can be looked up in the hash table with approximately one memory access. This is a fast algorithm. Unfortunately, it suffers exorbitant memory overhead due to the pseudo-rules which can be very large in number.

To reduce this memory overhead, we proposed a technique whereby we partition the rules into multiple sets and build a crossproduct table for each of them. With this strategy, the resulting number of rules is the sum of the smaller number of crossproduct rules in the individual sets. That number is significantly smaller than the crossproduct with all the rules kept in a single set. Furthermore, we observed that a pseudo-rule is produced in the crossproduct only if there are two prefixes of the same field, one being a prefix of another. If the subsets are formed intelligently such that no prefix of any field has a sub-prefix in the same rule subset, then no crossproduct is required. Finding such a rule partition with minimum number of crossproduct-free subsets is an NP-hard problem. However, using the structure provided by the packet classification problem, a simple heuristic solution based on Nested Level Tuples can be devised. Some of the crossproduct-free subsets can be merged with others and a crossproduct can be performed within the merged groups. In this way, the number

of subsets can be reduced at the cost of a small amount of crossproducting. Due to this rule partitioning, a rule needs to be looked up in each subset. We exploit the observation that if a packet matches at the most p rules, which we assume to be stored in distinct subsets, then only p lookups are successful. The remaining unsuccessful lookups can be avoided using Bloom filters. Therefore, the resulting technique requires $4 + p$ memory accesses with very high probability.

Our string matching algorithm is similar to the LPM algorithm. We group the strings to be searched according to their length (specified in bytes) and store each group in an off-chip hash table. We then maintain an on-chip Bloom filter for each hash table. We consider a window of characters in the data stream to be examined and look up all the prefixes within this window in the corresponding Bloom filters. When a matching string is found in a Bloom filter, it is verified in the off-chip hash table. Since a true matching string rarely appears in the data stream, the text can be forwarded when the Bloom filters don't show any match. Therefore, the reduced matches allow fast text searching. In the absence of true positives, the text searching can progress with almost one character in a clock cycle. However this technique requires as many Bloom filters as there are distinct string lengths. Additionally, it requires computation of hash over the entire string at a time. Therefore, this technique works well for short strings. To extend this scheme to arbitrarily long strings, we chop long strings into smaller, fixed-length segments, the last segment potentially containing fewer characters. To detect a match for this string, it is matched part by part. This is done by constructing a state machine where each state indicates the match for one more segment after the previous state. A state combined with an outgoing character segment uniquely specifies a branch. To progress down the state machine, the current state can be combined with the next character segment from the data stream and a lookup key can be constructed. Only if there is a match for this key do we follow the branch to the next state and update the current state. In case of a mismatch, a failure state can be reached. The overall solution is essentially a variant of the classic Aho-Corasick algorithm which is accelerated by using Bloom filters.

In summary, we conclude that Bloom filters can be used effectively to avoid potentially unsuccessful table lookups and save memory accesses. This lookup filtering technique can be applied wherever the process can be represented as a set of table lookups. While some search problems naturally exhibit a structure for representation as table

lookups, for others it can be less evident and challenging (packet classification and Aho-Corasick acceleration). Once converted into table lookups, the resulting tables can be implemented as hash tables in the off-chip memory and corresponding to each table, an on-chip Bloom filter containing the same lookup keys can be maintained. The pre-filtering of off-chip lookups can significantly accelerate the overall algorithm.

Furthermore, since this is a unique underlying technique, a generic Bloom filter engine can be constructed that can aid a variety of search algorithms. The only algorithm-specific difference is the size of the key to look up. A small reconfigurable logic engine can be constructed to allow computations of hashes over different sizes of keys depending on the need while keeping the remaining Bloom filter structure fixed. Such a processor can be a power-efficient and cost-effective alternative to a TCAM.

References

[1] *http://www.snort.org.*

[2] IDT Generic Part: 71P72604. http://www.idt.com/?catID=58745&genID=71P72604.

[3] IDT Generic Part: 75K72100. http://www.idt.com/?catID=58523&genID=75K72100.

[4] BGP Table Data. `http://bgp.potaroo.net/`, February 2003.

[5] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[6] Florin Baboescu and George Varghese. Scalable Packet Classification. In *ACM SIGCOMM*, 2001.

[7] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on FPGAs. In *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 223–232. ACM Press, 2004.

[8] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM*, 13(7):422–426, May 1970.

[9] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey, 2002.

[10] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[11] Y. Cho and W. Mangione-Smith. Fast reconfiguring deep packet filter for 1+ Gigabit network. In *IEEE Symposium on Field-Programmable Custom, Computing Machines, (FCCM)*, Napa, CA, April 2005.

[12] Christopher R. Clark and David E. Schimmel. Scalable multi-pattern matching on high-speed networks. In *IEEE Symposium on Field-Programmable Custom, Computing Machines, (FCCM)*, Napa, CA, April 2004.

[13] Saar Cohen and Yossi Matias. Spectral bloom filters. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252, New York, NY, USA, 2003. ACM Press.

[14] D. L. Stephen. String Searching Algorithms. In *Lectures Notes Series on Computing*, volume 3, 1994.

[15] MiKael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small Forwarding Tables for Fast Routing Lookups. In *ACM SIGCOMM*, 1997.

[16] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel Bloom filters. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, August 2003.

[17] Brian Dipert. Special Purpose SRAMs smooth the ride. *EDN*, Jun 1999.

[18] Will Eatherton, George Varghese, and Zubin Dittia. Tree Bitmap: hardware/software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communication Review*, 2004.

[19] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *ACM SIGCOMM*, August 2002.

[20] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[21] P. Gupta and N. McKeown. Packet Classification using Hierarchical Intelligent Cuttings. In *IEEE Symposium on High Performance Interconnects (HotI)*, 1999.

[22] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.

[23] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM*, 1998.

[24] K. Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for Advanced Packet Classification using Ternary CAM. In *ACM SIGCOMM*, 2005.

[25] M. MITZENMACHER. Compressed bloom filters. In *Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing*, IEEE/ACM Trans. on Networking, pages 144–150, 2001.

[26] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In *Proc. of Int. Conf. on Computing and Information*, pages 1621–1636, 1994.

[27] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification using Multidimensional Cutting. In *ACM SIGCOMM*, 2003.

[28] H. Song, J. Turner, and J. Lockwood. Shape Shifting Tries for Faster IP Lookup. In *IEEE ICNP*, 2005.

[29] Haoyu Song, Sarang Dharmarpurikar, Jonathan Turner, and John Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *ACM SIGCOMM*, 2005.

[30] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *IEEE Symposium on Field-Programmable Custom, Computing Machines, (FCCM)*, Napa, CA, April 2004.

[31] V. Srinivasan, Subhash Suri, and George Varghese. Packet Classification Using Tuple Space Search. In *ACM SIGCOMM*, 1999.

[32] V. Srinivasan and G. Varghese. Faster IP Lookups using Controlled Prefix Expansion. In *SIGMETRICS*, 1998.

[33] V. Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and Scalable Layer Four Switching. In *ACM SIGCOMM*, 1998.

[34] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. Over 10gbps string matching mechanism for multi-stream packet scanning systems. In *Field Programmable Logic and Application: 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004. Proceedings*, pages 484–493, Antwerp, Belgium, August 2004. Springer-Verlag.

[35] David Taylor and Jon Turner. Scalable Packet Classification Using Distributed Crossproducting of Field Labels. In *IEEE INFOCOM*, July 2005.

[36] David E. Taylor. Survey and taxonomy of packet classification techniques. *Washington University Technical Report, WUCSE-2004*, 2004.

[37] David E. Taylor and Jonathan S. Turner. Classbench: A Packet Classification Benchmark. In *IEEE INFOCOM*, 2005.

[38] Thomas H. Corman, Charles E. Leiserson, Ronald D. Rivest, Clifford Stein. *Introduction to Algorithms*. Prentice Hall, 2002.

[39] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE Infocom*, Hong Kong, China, March 2004.

[40] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM '97*, pages 25–36, September 1997.

[41] T. Y. C. Woo. A Modular Approach to Packet Classification. In *IEEE INFOCOM*, 2000.

[42] Fang Yu, Randy Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *IEEE Internation Conference on Network Protocols (ICNP)*, Berlin, Germany, October 2004.

[43] Fang Yu and Randy H. Katz. Efficient Multi-Match Packet Classification with TCAM. In *IEEE Hot Interconnects*, August 2003.

[44] Fang Yu, T. V. Lakshman, Martin Austin Motoyama, and Randy H. Katz. Ssa: a power and memory efficient scheme to multi-match packet classification. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, 2005.

Vita

Sarang Dharmapurikar

Date of Birth July 7, 1977

Place of Birth Nanded, India

Degrees B.E. Honors, Electrical and Electronics Engineering, May 1999
Birla Institute of Technology and Science, Pilani, India.

Publications

Journal

- **Longest Prefix Matching Using Bloom Filters**, Sarang Dharmapurikar, Praveen Krishnamurthy, David E. Taylor, *IEEE Transaction on Networking, April 2006*
- **Fast and Scalable Pattern Matching for Network Intrusion Detection Systems**, Sarang Dharmapurikar, John Lockwood, *accepted for publication in IEEE Journal on Selected Areas in Communicaitons*

Conference

- **Rethinking Hardware Support for Network Analysis and Intrusion Prevention** , Vern Paxson, Krste Asanovic, Sarang Dharmapurikar, John Lockwood, Ruoming Pang, Robin Sommer and Nick Weaver *USENIX First Workshop on Hot Topics in Security (HotSec), July 31, 2006.*
- **Algorithms to Accelerate Multiple Regular Expression Matching for Deep Packet Inspection** , Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley and Jonathan Turner, *ACM SIGCOMM, Pisa, Italy, September 2006.*

- **Fast and Scalable Pattern Matching for Content Filtering** , Sarang Dharmapurikar, John Lockwood *Symposium on Architectures for Networking and Communications Systems (ANCS), Princeton, 2005.*
- **Optimizing Memory Bandwidth of a Multi-Channel Packet Buffer** , Sarang Dharmapurikar, Sailesh Kumar, John Lockwood and Patrick Crowley, *IEEE Globecom, St. Louis, 2005.*
- **Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing**, Haoyu Song, Sarang Dharmapurikar, Jonathan Turner and John Lockwood, *ACM SIGCOMM, Philadelphia, 2005.*
- **Robust TCP Stream Reassembly in the Presence of Adversaries**, Sarang Dharmapurikar, Vern Paxson, *14th USENIX Security Symposium, Baltimore, 2005.*
- **Longest Prefix Matching Using Bloom Filters**, Sarang Dharmapurikar, Praveen Krishnamurthy, David E. Taylor, *ACM SIGCOMM, Karlsruhe, Germany, 2003.*
- **Deep Packet Inspection Using Parallel Bloom Filters**, Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, John W. Lockwood *Symposium on High Performance Interconnects (HotI), Stanford, 2003.*
- **An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall**, John W. Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim; *Field Programmable Logic and Applications (FPL), Lisbon, Portugal, 2003.*
- **System-on-Chip Packet Processor for an Experimental Network Services Platform** , David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar, John Lockwood, Wenjing Tang, Jonathan Turner, *IEEE Globecom, San Francisco, 2003.*
- **Beyond Performance: Secure and Fair Memory Management for Multiple Systems on a Programmable Chip** , Carlos Macian, Sarang Dharmapurikar, John Lockwood *IEEE International Conference on Field-Programmable Technology (FPT'03), Tokyo, Japan, 2003*

Select Technical Reports

- **Fast Packet Classification Using Bloom Filters**, Sarang Dharmapurikar, Haoyu Song, Jonathan Turner, John Lockwood, *WUCS-2006-27, May 12, 2006.*
- **Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters**, Sarang Dharmapurikar, Michael Attig, John Lockwood, *WUCSE-2004-12, March 25, 2004.*
- **Synthesizable Design of a Multi-Module Memory Controller**, Sarang Dharmapurikar and John Lockwood, *WUCS-01-26, October, 2001.*

Patents

- **Method and Apparatus for Performing Longest Prefix Matching using Bloom Filters**, *Sarang Dharmapurikar, Praveen Krishnamurthy, David E. Taylor* (pending)
- **Method and Apparatus for Detecting Predefined Signatures In Packet Payload using Bloom Filters**, *Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, John Lockwood* (pending)

August 2006