Washington University in St. Louis
School of Engineering & Applied Science

SEVER INSTITUTE OF TECHNOLOGY

DOCTOR OF SCIENCE DEGREE

DISSERTATION ACCEPTANCE

(To be the first page of each copy of the dissertation)

DATE: September 8, 2006

STUDENT'S NAME: Haoyu Song

This student's dissertation, entitled Design and Evaluation of Packet Classification Systems has been examined by the undersigned committee of five faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Doctor of Science.

APPROVAL: ______________________________ Chairman

____

____

____

____

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DESIGN AND EVALUATION OF PACKET CLASSIFICATION SYSTEMS

by

Haoyu Song M.S.Co.E, B.E.E.E

Prepared under the direction of Professor Jonathan S. Turner

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

September 2006

Saint Louis, Missouri

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

ABSTRACT

DESIGN AND EVALUATION OF
PACKET CLASSIFICATION SYSTEMS

by

Haoyu Song

ADVISOR: Professor Jonathan S. Turner

September 2006

Saint Louis, Missouri

Although many algorithms and architectures have been proposed, the design of efficient packet classification systems remains a challenging problem. The diversity of filter specifications, the scale of filter sets, and the throughput requirements of high-speed networks all contribute to the difficulty. We need to review the algorithms from a high-level point-of-view in order to advance the study. This level of understanding can lead to significant performance improvements. In this dissertation, we evaluate several existing algorithms and present several new algorithms as well.

The previous evaluation results for existing algorithms are not convincing because they have not been done in a consistent way. To resolve this issue, an objective evaluation platform needs to be developed. We implement and evaluate several representative algorithms with uniform criteria. The source code and the evaluation results are both published on a web-site to provide the research community a benchmark for impartial and thorough algorithm evaluations.

We propose several new algorithms to deal with the different variations of the packet classification problem. They are: (1) the Shape Shifting Trie algorithm for longest prefix matching, used in IP lookups or as a building block for general packet classification algorithms; (2) the Fast Hash Table lookup algorithm used for exact flow match; (3) the longest prefix matching algorithm using hash tables and tries, used in IP lookups or packet classification algorithms;(4) the 2D coarse-grained tuple-space

search algorithm with controlled filter expansion, used for two-dimensional packet classification or as a building block for general packet classification algorithms; (5) the Adaptive Binary Cutting algorithm used for general multi-dimensional packet classification. In addition to the algorithmic solutions, we also consider the TCAM hardware solution. In particular, we address the TCAM filter update problem for general packet classification and provide an efficient algorithm. Building upon the previous work, these algorithms significantly improve the performance of packet classification systems and set a solid foundation for further study.

copyright by

Haoyu Song

2006

To Chuanlan, Grace, and Carol

Contents

List of Tables . . . . **ix**

List of Figures . . . . **x**

Acknowledgments . . . . **xiii**

1 Introduction **1**

1.1 Packet Classification 1

1.2 Algorithmic Solutions 2

1.2.1 High-level Review 3

1.3 TCAM Solution 6

2 Algorithm Evaluation **9**

2.1 Motivation 9

2.1.1 Incommensurable Evaluation Results 9

2.1.2 Irreproducible Implementations and Evaluation Results 10

2.1.3 Incomplete Evaluation and Unconvincing Results 10

2.1.4 Inadequate Insights 10

2.2 Approach 11

2.2.1 Documentation of Method 11

2.2.2 Documentation of Filter Set 12

2.2.3 Metrics for Evaluation 12

2.2.4 Sensitivity Study 13

2.3 Summary 14

2.3.1 Evaluated Algorithms 14

2.3.2 Filter Sets 15

2.3.3 Sample Results — the HiCuts Algorithm 16

2.3.4 Observations 21

3 Shape Shifting Tries . . . **23**
3.1 Introduction . . . 23
3.2 SST Representation . . . 25
3.3 Lookup in an SST . . . 26
3.4 Constructing Optimal SSTs . . . 29
3.4.1 Optimality of BFP . . . 31
3.4.2 Effectiveness of Shape Shifting Trie . . . 33
3.5 A Hybrid Algorithm . . . 34
3.6 Reference Implementations . . . 34
3.7 Performance Evaluation . . . 36
3.7.1 Performance on IPv4 Route Lookup . . . 36
3.7.2 Performance on IPv6 Route Lookup . . . 38
3.7.3 Scaling Characteristics of SST . . . 42
3.8 Updating an SST . . . 42
3.9 SST Optimizations . . . 43
3.9.1 Compressing the Underlying Binary Trie . . . 43
3.9.2 Increasing SST Node Capacity . . . 45
3.10 Related Work . . . 47
3.11 Conclusion . . . 49

4 Fast Hash Table . . . **51**
4.1 Introduction . . . 51
4.1.1 Hash Tables for Packet Processing . . . 51
4.1.2 Related Work . . . 55
4.1.3 Scope for Improvement . . . 56
4.2 Fast Hash Table and Lookup Algorithm . . . 57
4.2.1 Basic Fast Hash Table . . . 59
4.2.2 Pruned Fast Hash Table (PFHT) . . . 62
4.2.3 PFHT List-balancing Heuristic . . . 67
4.2.4 Shared-node Fast Hash Table (SFHT) . . . 68
4.2.5 Memory Compression . . . 72
4.3 Analysis . . . 74
4.3.1 Expected Linked List Length . . . 75
4.3.2 Effect of the Number of Hash Functions . . . 79

4.3.3 Average Access Time 82
4.3.4 Memory Usage 83
4.4 Simulations 85
4.5 Implementation 87
4.6 Conclusion 88

5 LPM using Hash Tables and Tries 89
5.1 Introduction 89
5.2 Related Work 91
5.3 Algorithm 92
5.4 Implementation 97
5.5 Evaluation 100
5.6 Conclusion 104

6 2D Coarse-grained Tuple Space Search 105
6.1 Introduction 105
6.2 Related Work 107
6.2.1 Tuple Space Search 107
6.2.2 Crossproducting 109
6.3 Combining Tuple Space Search and Cross Products 111
6.4 Tuple Partition 113
6.5 Evaluation 114
6.6 Conclusion 116

7 Adaptive Binary Cutting 118
7.1 Background 118
7.2 Related Work 120
7.3 Observations 121
7.4 Algorithm 125
7.4.1 ABC Variation I 127
7.4.2 ABC Variation II 131
7.4.3 ABC Variation III 133
7.4.4 Comparison 134
7.4.5 Implementation 135
7.5 Optimizations 137

7.5.1 Reduce Filters Using Hash Table 138
7.5.2 Filter Partition on the Protocol Field 139
7.5.3 Partition Filters Based on Duplication Factor 141
7.5.4 Hold Filters Internally and Reverse Search Order 142
7.6 Evaluation 143
7.6.1 Comparison of ABC Variations 144
7.6.2 Comparison with Other DT-based Algorithms 149
7.6.3 Incremental Updates 152
7.7 Conclusion 152

8 Fast TCAM Filter Updates 154
8.1 Introduction 154
8.2 Related Work 155
8.3 Algorithm 157
8.3.1 Real Filter Priority 157
8.3.2 Using Extended Filter 160
8.3.3 Lookup 162
8.3.4 Update 163
8.4 Evaluation 164
8.4.1 Filter Distribution 164
8.4.2 Lookup Throughput Performance 165
8.4.3 Update Performance 165
8.5 Conclusion 167

9 Summary 168
9.1 Contributions 168
9.2 Future Directions 170

References 172

Vita 178

List of Tables

3.1 Performance on IPv4 BGP Table 37
3.2 Performance on IPv6 BGP Table 39
3.3 Performance on Trimmed IPv6 BGP Table 39
3.4 Performance on the Synthetic IPv6 Table 41
3.5 Performance Optimization on IPv4 BGP Table 46
3.6 Performance Optimization on Synthetic IPv6 Table 46

4.1 Expected # of Items for Which All Buckets Have $> j$ Entries 86

5.1 Prefix Table 93
5.2 Expanded Prefix Table 93
5.3 BGP Table Results I 101
5.4 BGP Table Results II 101
5.5 IPv6 BGP Table Results (Using TBM only) 103
5.6 IPv6 BGP Table Results (Using SST and TBM) 103

6.1 Number of Nonempty Tuples in ACL Filter Sets 107
6.2 ACL Filter Set Expansion 110
6.3 Filter Set Expansion for Different Configurations 115
6.4 Number of Items in Bloom Filters for LPM 115
6.5 # Filters for Different Tuple Configurations 116

7.1 Bit Consumption of CSBs and EPB 134
7.2 ABC DT Node Encoding Scheme(# Bits) 138
7.3 DT Node Encoding Scheme for Other Algorithms(# Bits) 151

8.1 Real Priority Levels in Real Filter Sets 159
8.2 Lookup Throughput Performance 165
8.3 The Worst-Case Update Performance 166

List of Figures

1.1 Cutting and Projection 4

2.1 Presentation of Evaluation Results 13
2.2 Effect of Parameter Settings 13
2.3 HiCuts Illustration 17
2.4 Child Node Reuse Optimization 19
2.5 Redundancy Elimination Optimization 20
2.6 HiCuts Performance Evaluation 20
2.7 Sensitivity to Space Measure Factor 21
2.8 Sensitivity to Bucket Size 21

3.1 An SST and the Corresponding Data Structure 26
3.2 A Lookup Example Using SST 27
3.3 Minimum Size and Height Partitions of a Binary Tree 30
3.4 Data Structure Node Formats 36
3.5 Prefix Length Distribution of the IPv4 BGP Table 37
3.6 Prefix Length Distribution of the IPv6 BGP Table 38
3.7 Prefix Length Distribution of the Synthetic IPv6 Table 40
3.8 Bit Value Distribution of the Synthetic IPv6 Table 41
3.9 Effects of the Bit Assignment 42
3.10 Child Promotion Optimization 44
3.11 Nearest Ancestor Collapse Optimization 45
3.12 Up Down Counts Shape Encoding 50

4.1 A Naive Hash Table 58
4.2 Basic Fast Hash Table (BFHT) 61
4.3 Pruned Fast Hash Table (PFHT) 63
4.4 PFHT List-balancing 68
4.5 Shared-node Fast Hash Table (SFHT) 69

4.6 Pointer Array Compression 74
4.7 Probability Distribution of Searched Linked-list Length 78
4.8 Expected Items for Which the Searched Bucket Contains $> j$ Items . 80
4.9 The Effect of Optimal Configuration of FHT 81
4.10 The Effect of Non-optimal Configuration of FHT 82
4.11 Expected Search Time 84
4.12 Item Memory Usage of Different Schemes 85
4.13 Hierarchical Structure of Fast Hash Table 87

5.1 LPM Data Structure for the Example Table 94
5.2 The Worst-Case Performance vs the Number of Bloom Filters 98
5.3 Tree Partition Example Using TBM and SST 99
5.4 The Worst-Case Performance vs the Number of Items per Prefix . . . 102

6.1 2D Tuple Space Search 108
6.2 2D Filter Expansion 110
6.3 2D Coarse-grained Tuple Space Partition 112
6.4 Another Tuple Partition Scheme 114
6.5 Filter Set Expansion vs. Number of Hash Queries 115
6.6 Experiments on Irregular Tuples 116
6.7 Dynamic Tuple Partition 117

7.1 Filter Distribution on the Value of First Header Field Byte 122
7.2 Cuttings on a DT Node for Different Algorithms 124
7.3 ABC-I: Cuts and Lookup on a DT Node 128
7.4 ABC-II: a DT Node and Decoding Example 131
7.5 Decoding the DT Node to Find the Child DT Node Address 135
7.6 Data Structure of the Algorithm Implementations. 137
7.7 Effect of Filter Reduction by Using a Hash Table 139
7.8 Protocol Pointer Table Structure 140
7.9 Filter Duplication Factor Distribution I 141
7.10 Filter Duplication Factor Distribution II 141
7.11 Algorithm Scalability on Filter Set Size 144
7.12 The Tradeoff of the Storage and the Throughput 146
7.13 The Effect of Filter Reduction Using a Hash Table 147

7.14 The Effect of Looking Up on Protocol Field First 147
7.15 The Effect of Holding Filters Internally and Reversing Search Order . 148
7.16 The Effect of Removing Highly Duplicated Filters 148
7.17 The Effect of Changing DT Node Size 149
7.18 Compare ABC with the Other DT-based Algorithms 151

8.1 Grouping and Priority Value Assignment 158
8.2 Effect of Inserting a New Filter R 160
8.3 Searching for the Filter with the Minimum Priority Value 161
8.4 Priority Value Distribution for Real Filter Sets 164
8.5 The Worst-Case Distribution of TCAM Accesses 166

Acknowledgments

Many people's efforts and contributions have made this dissertation possible, to whom I cannot fully express my sincerely gratitude.

First and foremost, I want to thank my research advisor, Dr. Jonathan Turner. He taught me how to conduct research and come up with useful ideas and results. It is not only fun but also very rewarding to work with him. He has impressed me again and again with his insatiable curiosity to the unknowns, uncompromising posture to the academic integrity, sharp intuition and insights on the essence of the problems, and terse yet precise communication and writing. These are the most precious lessons I have learned from him during the course of my study.

It is to Dr. John Lockwood's credit that I was involved in this amazing research group and many exciting projects where real problems are attacked and solved. I was given the freedom to explore my interests and was often encouraged to achieve higher goals. I am deeply grateful for the time he spent on our papers. Accomplishments were never the effort of a sole researcher but of a team. I want to thank the other members in my thesis committee: Dr. Raj Jain, Dr. Weixiong Zhang, and Dr. Ronald Indeck. They all gave me valuable feedbacks from the early stage to the final output of this dissertation. I also like to extend my thanks to the ARL staff and my fellow graduate students, for their numerous and kind assistance in the lab, and their collaborations in projects and research.

Finally, I am eternally indebted to my wife, Chuanlan Kang, who has dedicated all her time and energy to take care of the family so that I can focus on my research with undivided attention.

Haoyu Song

Washington University in Saint Louis
September 2006

Chapter 1

Introduction

1.1 Packet Classification

Packet classification enables network routers to provide advanced network services, e.g. network security, QoS routing, and resource reservation. There is increasing industrial and academic interest in algorithms and systems for efficient packet classification. On the one hand, network security and QoS have become urgent driving factors requiring large-scale packet classification. Currently the largest packet filter sets in use contain thousands of filters and each filter involves five or more header fields. Tens of thousands of filters in a filter set are expected in the future. On the other hand, increasing network traffic poses greater challenges than ever for the application of large-scale packet classification. As of 2005, OC-192 (10 Gbps) connections have become common in backbone networks. Although the use of OC-768 (40 Gbps) connections is still rare, widespread adoption is expected. To support OC-192 wire-speed processing, more than 30 million packets need to be classified in a second in the worst case. i.e. the systems are required to provide a classification result every 32 ns. This daunting task is made more difficult when we think of the advent of terabit networks. For these reasons, packet classification is still an open and challenging problem demanding continuing investigations.

The function of the packet classification system is to match packet headers against a set of pre-defined filters. The relevant packet header fields usually include the source IP address, the destination IP address, the transport protocol, the source port, and the destination port. Other header fields, e.g. the TCP flags, can also be matched.

Formally, a filter set consists of a finite set of n filters, R_1, R_2 ... R_n. Each filter is a combination of k header field specifications, H_1, H_2 ... H_k. Each header field specifies one of four kinds of matches: exact match, prefix match, range match, or masked-bitmap match. A packet P is said to match a filter R_i if and only if the header fields, H_1, H_2 ... H_k, match the corresponding fields in R_i in the specified way. Each filter R_i has an associated action Act_i that determines how a packet P is handled if P matches R_i. Filters can overlap; hence, a packet can match multiple filters. In the single match variation of the problem, the one with the highest priority among all the matching filters is chosen as the best matching filter. Usually, the filter's position in an ordered list of filters defines its priority.

A linear search of the filter set, although simple and allowing the most compact storage, is too slow for large filter sets. Instead, more sophisticated algorithms or Ternary Content Addressable Memory (TCAM) hardware are used to attack the problem. Both of these approaches have their own advantages and disadvantages in terms of performance, economics, ease of implementation, and scalability. Hybrid architectures which leverage these two approaches are also possible.

1.2 Algorithmic Solutions

Algorithmic solutions use commodity memory to minimize the storage cost. We can map the packet classification problem to the point location problem in a multi-dimensional space where each header field is treated as a dimension. In the space, filters define hyper-cubes and a packet defines a point. The goal is to determine the highest priority hyper-cube covering a given point. Point location in computational geometry has proven to be difficult. Assuming there are n filters and F dimensions, [49] shows that in order to achieve $O(\log n)$ lookup time, the required storage can be as large as $O(n^F)$; if the storage is limited to $O(n)$, a lookup may take $O(\log^{F-1} n)$ time to finish. Both extremes are unacceptable in practice. Fortunately, real filter sets often exhibit structure that allows packets to be classified using more efficient heuristic algorithms.

1.2.1 High-level Review

An excellent survey of packet classification techniques can be found in [69]. In this section, we identify certain high level characteristics of algorithmic approaches, with the objective of developing insights that can lead to fundamental improvements.

One theme - space and time tradeoff

Well-designed algorithms exhibit a clear tradeoff between storage and throughput. Algorithms without tunable parameters often perform poorly. For example, the cross-producting algorithm [65] builds the direct cross-producting lookup table and suffers from poor storage efficiency, so it fails to scale to even moderate sized filter sets. The Recursive Flow Classification (RFC) algorithm [33], a variation of the cross-producting algorithm in essence, trades off throughput in order to reduce storage to some extent. However, the storage efficiency of RFC remains low and hence the scalability of RFC remains poor. This suggests that additional tradeoffs need to be considered to obtain better performance.

Two techniques - cutting and projection

The geometric view of packet classification reveals some basic ideas on how to construct the data structures and to represent packet filters. In the geometric view, many algorithms adopt either cutting or projection in multi-dimensional space to preprocess filter sets. Figure 1.1 illustrates both on a 2D plane. The cutting technique slices the space at selected vantage points into smaller subregions. Each subregion therefore contains fewer filters. This process helps narrow the search scope. The second technique projects the end-points of ranges to each dimensional axis. Two adjacent points define an elementary interval that is fully covered by a unique subset of the filters. Identifying the elementary intervals that a packet belongs to also helps narrow the search scope. Projection has finer granularity than cutting so it can better differentiate filters; however, locating an elementary interval from projection is more difficult than locating a subregion from cutting.

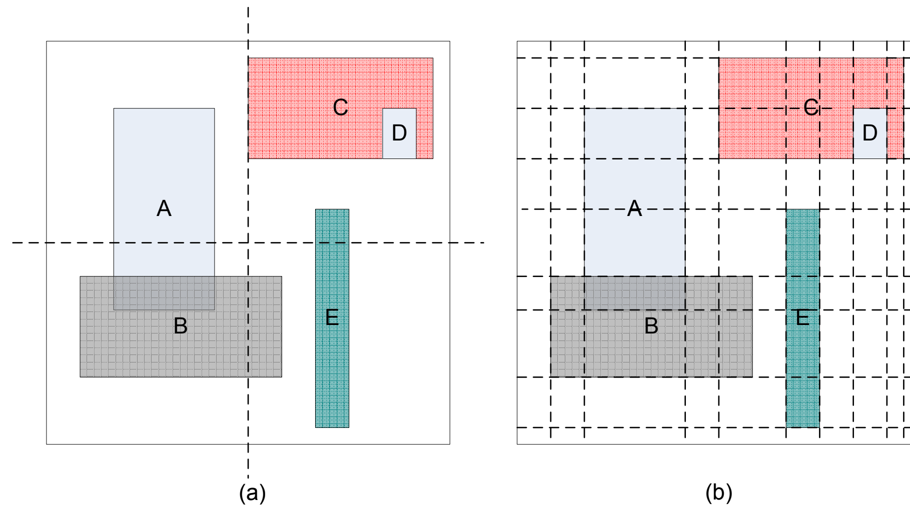

Figure 1.1: Cutting and Projection

The decision tree-based algorithms usually apply the cutting technique and the decomposition-based algorithms often apply the projection technique.

Three Approaches - splitting, intersecting, and grouping

The goal of packet classification is to find the best matching filter for a given packet header. The initial set is too large to be handled efficiently in terms of restricted space or time, so the basic strategy is nothing more than "divide and conquer." We achieve the filter set reduction by "eliminating" the filters that are not needed for identifying the final match.

The first approach is *filter set splitting*. It uses a few header bits to split the filter set into smaller subsets. Some other header bits are then used to continue splitting each subset. A decision tree is formed from this recursive process. Algorithms using this approach include Woo's modular packet classification [78], Hierarchical Intelligent Cuttings (HiCuts) [32], Multidimensional Cuttings (HyperCuts) [56], and our Adaptive Binary Cutting algorithm discussed in Chapter 7 .

The second approach is *filter set intersecting*. The key idea of this technique is that it is easier to match a partial filter than to match the entire filter at one time. If we split the packet header into a set of substrings, then each substring can match

a subset of filters. The intersection of these subsets is exactly the filters matching the entire packet header. Intersection can be implemented with different tradeoffs of storage and throughput. For example, we can intersect all subsets obtained from the substring lookups in a single step. The Bit Vector (BV) algorithm [39] and the Aggregated Bit Vector (ABV) algorithm [10] explicitly represent the subset of filters for each partial match by using bit vectors. On the other hand, the cross-producting algorithm [65] implicitly encodes the subsets of filters into indices that are used to form the keys to the cross-product table. These algorithms are fast and their throughput mainly depends on the partial header lookup speed; However, they can consume excessive amounts of memory. On the contrary, The RFC algorithm [33] and the Distributed Cross-producting of Field Labels (DCFL) algorithm [68] provide a nice tradeoff of storage and throughput by recursively performing parallel set intersections in multiple steps. The Fat Inverted Segment Trees (FIST) algorithm [30] uses a similar approach but performs the intersections in sequence. The partial header lookup can be done using different methods. The fastest method is to use a direct lookup table, yet it also consumes the most storage. We can also use any single field lookup technique, such as binary search and longest prefix matching. Our SST algorithm discussed in Chapter 3 is perfect for this purpose.

The last and least used approach is *filter set grouping*. Filters in a set are regrouped into disjoint subsets according to certain common features. Lookups can be performed on each of these smaller subsets in parallel. The best match is determined from the results of all the lookups. Tuple Space Search [63], in which filters are grouped based on a tuple specification, belongs in this category. Lookups in each tuple can be conducted through a simple hash table. Our 2D Compressed Tuple Space Search algorithm discussed in Chapter 6 also takes this basic approach.

Each of the approaches has its own limitations. It appears that to achieve consistently good performance, one needs to combine the best characteristics of different approaches and make good use of time-space tradeoffs. While attempts to find new algorithms from a totally different perspective seem unlikely, a systematic analysis of the existing algorithms can lead to significant improvements. For instance, one problem with *filter set splitting* is that some filters are too similar to be efficiently separated. But one can use *filter set grouping* to group the filters based on some sort of “similarity” measure so that the filters in a single subset exhibit maximum

dissimilarity. For each subset, *filter set splitting* may be more efficient. The problem of *filter set intersecting* is its excessive memory consumption which is partially due to a large number of elementary intervals [30] or equivalence classes [33]. We can reduce the number by aggregating elementary intervals. This coarser granularity leads to much smaller cross-product tables and can also speed up the single field lookups at the cost of a small linear search in the final intersection step. In the RFC algorithm, the first level lookup tables take very large amounts of space. By slightly sacrificing the throughput, one can construct a more efficient data structure to reduce the table size significantly. To summarize, there are still many opportunities to improve the algorithm performance once existing algorithms are fully understood.

1.3 TCAM Solution

TCAMs are widely deployed in high performance network routers for packet classification because of their unmatched lookup throughput and generality. A TCAM is a special memory device which can store ternary bit strings and perform parallel searches on all of its entries simultaneously. In TCAMs, packet filters are represented as ternary bit strings and stored in decreasing priority order. Given a packet header, the search for the best matching filter with the highest priority is performed on all the entries in parallel. The index of the first matching filter is then used to access a memory to retrieve the associated data for the matching filter. This elegant architecture allows classifying packets at very high throughput. A commercially available TCAM chip can store more than 100K ternary filters, which is more than enough for even the largest filter set applied today. It can classify 250 million packets per second, which satisfies the throughput demands of all the existing networks today [3]. The room for algorithmic solutions to compete with TCAMs seems very narrow.

While TCAMs remain the most popular choice for high performance packet classification in network routers, it is generally agreed that they are not preferred for IP lookups. At the same time, the research on algorithmic alternatives for general packet classification is still going on because of the following drawbacks of TCAM devices.

- *Low Density & High Cost.* A TCAM device requires up to 16 transistors for a bit while SRAM requires six and SDRAM just one. Consequently, the storage density of a TCAM is significantly lower than that of commodity memory technologies. Moreover, the relatively small market for TCAMs makes them relatively expensive, with a cost per bit that is roughly 20 times that of SRAM and hundreds of times that of SDRAM.

- *High Power Consumption.* Because they search all entries in parallel on every packet, TCAMs consume a lot of power. 25 Watts is a fairly typical power budget for a TCAM device in a high performance application. Modern TCAMs do allow entries to be grouped into segments, that can be selectively searched in order to reduce power usage. When filters are partitioned among segments appropriately, it can significantly reduce power consumption [82, 62, 83]. However, these hybrid solutions actually lower the system throughput and impair the generality of TCAMs.

- *Poor Arbitrary Range Support.* TCAMs naturally support searches on ternary bit strings. This is not ideal for packet filters that include arbitrary ranges for some of their fields. The standard way to solve this problem is to convert filters with ranges into sets of filters defined by bit strings. But this can lead to significant expansion in the space required to represent a filter (as much as 900x expansion in the worst case) [62]. This observation has triggered the development of new methods that combine single field searches with encoded range values [44, 40, 46] and proposals for direct hardware support of range lookups [62]. Again, these hybrid solutions tend to lower the system throughput and impair the generality of TCAMs.

- *Poor Multiple-Match Support.* Recent network security applications, such as network intrusion detection and prevention, require all the matching filters to be reported, not just the first one. Conventional TCAMs can only output the matching filter with the smallest index. It seems likely that future TCAMs, driven by new application requirements, will be designed to support multiple matches efficiently. Currently, the approaches for dealing with this problem are discussed in [80, 81, 60, 40]. On the other hand, almost all the algorithmic solutions naturally support multiple-match applications.

In addition to these issues, the problem of filter update in TCAM deserves more attention. A close examination of this issue shows that updates can have a significant performance impact on TCAM-based lookups. Chapter 8 studies this issue and proposes possible solutions.

Our research focuses primarily on algorithmic solutions to packet classification, aiming to promote better design and evaluation standards for packet classification systems. As the first step of this effort, in this chapter we survey the existing algorithms from a high-level perspective, trying to extract basic ideas and inherent links. Better algorithms can sometimes be obtained by relaxing restrictions or introducing more degrees of freedom. Indeed, this systematic analysis has led to the new algorithms discussed in Chapter 3 through Chapter 8. The literature survey also shows the status of algorithm evaluation is far from acceptable. The research community urgently needs a more systematic evaluation of existing algorithms to enable consistent performance comparisons. Our efforts include implementing some representative algorithms and evaluating them under uniform criteria. This project is described in Chapter 2. In Chapters 3, 4, 5, 6, and 7, we present several new algorithms which solve different variations of the packet classification problem, from one dimension to multiple dimensions, from exact match to general match. Chapter 9 summarizes our contributions and discusses the future work.

Chapter 2

Algorithm Evaluation

2.1 Motivation

Although many packet classification algorithms and architectures have been proposed and research is ongoing, researchers and technology adopters find it is difficult to choose an appropriate algorithm for application and to evaluate new algorithms objectively. Before exploring new possibilities, it is imperative to understand existing algorithms under uniform test conditions and a common set of benchmark criteria. Unfortunately, the existing algorithm evaluation is hardly persuasive for the following reasons:

2.1.1 Incommensurable Evaluation Results

First, evaluations by different authors are not based on the same filter sets. Researchers have limited accessibility to real-world filter sets. Sometimes they have to use randomly generated filter sets for evaluations. However, the performance of many packet classification algorithms is very sensitive to the structure of the filter sets. Second, the evaluations are not based on common implementation assumption. They do not share a common implementation model. Some algorithms assume a software-based implementation and the others assume a hardware-based implementation. Different assumptions on implementation architectures and platforms can lead to very different performance evaluation results. Third, there is an absence of evaluation tools, benchmarks, and publicly accepted measurements. Different people

have their own understanding of the evaluation criteria. Some criteria are unrealistic and make it hard to determine the actual performance one might expect in practice. This makes it difficult to understand and compare the evaluation results.

2.1.2 Irreproducible Implementations and Evaluation Results

Researchers rarely provide enough details to allow readers to exactly reproduce the work reported in their research papers. Either some key points of the algorithm description are missing or the test conditions, such as parameter settings and the filter sets used, are undisclosed. These situations cause confusion and create unnecessary hurdles for others trying to understand the algorithms and advance the research.

2.1.3 Incomplete Evaluation and Unconvincing Results

Packet classification algorithms often involve some tradeoffs, heuristics, and optimizations. Tunable parameters may have subtle effects on algorithm performance. It is important to isolate them and evaluate their behavior carefully in order to clarify their impact on the algorithm. However, some evaluations fail to identify the performance impact of individual parameters.

Moreover, some researchers boast about some aspects of their algorithms while underplaying their drawbacks. Researchers sometimes make claims without sufficient proof. Some assumptions and prerequisites are impractical or invalid. The ambiguities in algorithm descriptions and evaluations are too confusing to allow readers to make valid judgments

2.1.4 Inadequate Insights

Some research papers focus on the algorithm details and lack high-level insights which reveal the inherent and intrinsic principles that underlie the algorithms. Proposed algorithms become more and more complex without convincing benefits. Deeper

understanding of the problem is needed to enable more effective algorithm design efforts.

2.2 Approach

Ideally, the evaluation should cover the criteria of throughput, storage, incremental update support, preprocessing time, scalability to the size of filter sets, adaptability to the structure of filter sets, implementation cost, and power dissipation. All the evaluation results should be normalized in a directly comparable way. In different applications, some criteria may be more important than others, but the evaluation should provide information without preference and let readers make their own judgements. While asymptotic analysis of timing and storage complexity is a useful metric, the evaluation should not be limited by it. Because packet classification algorithms are mostly based on heuristics, different filter sets with different structures and sizes tend to give very different results. The performance of the algorithm on real filter sets is the decisive factor in any realistic evaluation.

By identifying the problems in algorithm evaluations for packet classification, we establish a standard procedure of algorithm description and evaluation. In particular, we provide the research community an objective and "advocacy-free" evaluation of a suite of packet classification algorithms. A summary of our approach follows:

2.2.1 Documentation of Method

First, we provide a complete description of the key data structures and all the tunable parameters. Second, we provide a detailed description of the algorithm preprocessing and lookup process along with step-by-step illustrations using an example. Third, we provide the source code for an actual implementation.

We assume that a simple hardware-based model or a network processor-based model is used in our implementation, which includes multiple on-chip lookup engines or

threads, a memory interface, and a commodity off-chip memory. All data are retrieved from the off-chip memory. The lookup for one packet is conducted by a sequence of dependent memory accesses. The memory bandwidth is shared by multiple independent lookup engines or threads. The on-chip resource usage is small relative to filter set size, so we ignore the cost of it in our evaluation. To save memory bandwidth and improve performance, the implementations are supposed to use efficient methods to compress data structure representation.

We also try to categorize the algorithms based on their high-level ideas and provide insights to help improve the algorithm performance or design better algorithms.

2.2.2 Documentation of Filter Set

The open-source *ClassBench* [70] is used to generate synthetic filter sets with different scales and structures. We provide the parameters used for filter set generation. We also generate a packet header trace using *ClassBench* for each filter set for implementation verification and algorithm evaluation. The size of a trace is about 10 times that of the corresponding filter set.

We provide the original filter sets that are used as seeds for the synthetic filter sets. The statistics files extracted from the original filter sets can be downloaded from the *ClassBench* website.

2.2.3 Metrics for Evaluation

For objective and meaningful algorithm evaluation, we measure the storage efficiency of an algorithm using the average number of bytes consumed per filter. We measure the throughput of an algorithm using the memory bandwidth consumption: the number of bytes per memory access and the number of dependent memory accesses per packet lookup. The memory bandwidth consumption is evaluated in both worst and average case.

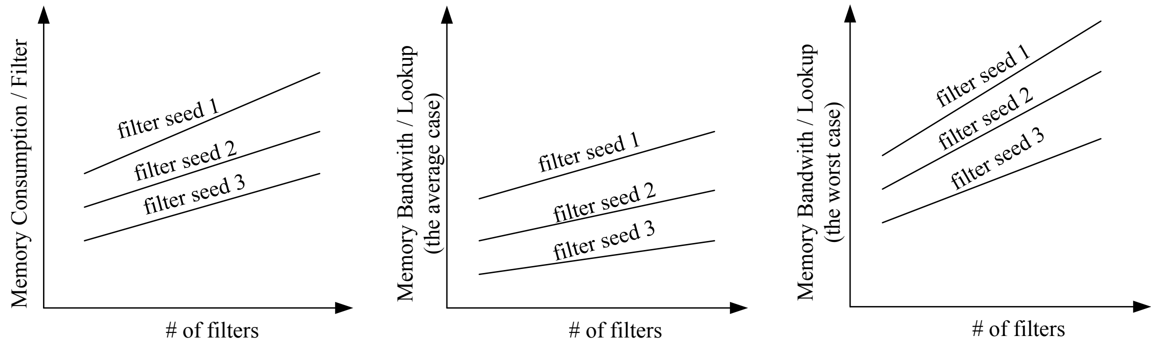

Figure 2.1: Presentation of Evaluation Results

We will use three figures like those shown in Figure 2.1 to present the results. The overall data structure size is the product of the memory consumption per filter and the number of filters. The overall throughput can be calculated by dividing the total memory bandwidth by the memory bandwidth consumed per packet lookup.

2.2.4 Sensitivity Study

We determine how each individual parameter influences the overall performance quantitatively in the algorithm evaluation. For each tunable parameter, we produce some figures like that shown in Figure 2.2. Each figure use a different scaled filter set. The sensitivity study will clarify issues often left unresolved in the original papers. It will also help users to determine the optimal design parameters for a given filter set.

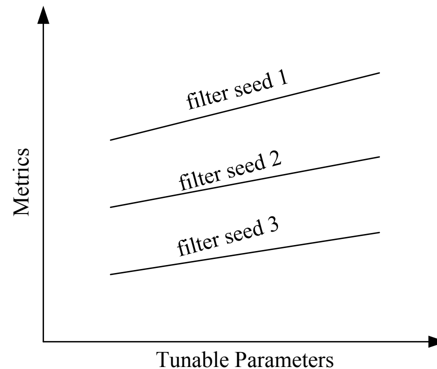

Figure 2.2: Effect of Parameter Settings

The documentation of the algorithm evaluation results are posted on a publicly accessible website:
`www.arl.wustl.edu/~hs1/PClassEval.html`

Note that our implementations are only for the purpose of simulation and evaluation, thus the source code is not optimized for software execution and the implementations do not directly map to either hardware or network processor. In addition, we do not consider preprocessing cost, incremental update cost or power dissipation. These factors are left for future studies. Our effort will help promote better understanding of some representative algorithms, promote the standard for algorithm evaluation, ease the research curve, and encourage contributions from the research community to make it better.

2.3 Summary

2.3.1 Evaluated Algorithms

Six representative algorithms have been evaluated. They are HiCuts [32], HyperCuts [56], Woo's Modular Packet Classification [78], RFC [33], BV [39], and the Tuple Space Search Algorithm [63]. They cover all the basic approaches we have discussed in Chapter 1.

The difficulty we encountered during the algorithm implementation mainly results from the ambiguous and incomplete algorithm descriptions in the original papers. The authors failed to provide deterministic algorithm descriptions. The likely reason for that is that the algorithms depend heavily on heuristics, and the performance of the heuristics is very sensitive to the filter set structure. Some algorithms include several options or only high level guidelines for implementing the algorithm details. It is up to the user to determine appropriate implementations so trial and error is needed. During implementation, we found that sometimes the details are actually very tricky and getting the details right is crucial for the overall performance. This unfortunate situation reveals serious problems with the algorithm design and demands more attention from the algorithm designers.

For example, when implementing the HiCuts algorithm [32], we were confused by the heuristic for choosing the dimension to cut. Although the authors listed four possible strategies, it is unclear which one is actually used in their performance evaluation. Our experiments shows this decision is crucial. A careless choice may cause poor performance or incorrect operation. For example, the option of choosing the dimension that maximizes the entropy of rule distribution does not work in the case when all the remaining rules span the entire region on some dimensions. No matter how many cuts are performed on these dimensions, they always return the maximum entropy, but by choosing one of these dimensions, the cuttings are useless: the only effect is to duplicate the same set of rules into multiple subregions. Similar problems arise in other papers. The algorithm description of the HyperCuts algorithm [56] is particularly confusing. While the dimension selection strategy is clear, the decisions about the number of cuts and their distribution among the chosen dimensions are very ambiguous. The algorithm description of RFC [33] does not tell how exactly the reduction tree should be organized, which can significantly bias the memory efficiency.

Such ambiguity and uncertainty in algorithm description can seriously damage an algorithm's credibility. We believe that in any case the algorithm description should be thorough and deterministic, without any ambiguity. Further investigation and thoughtful design can help avoid this situation, giving an algorithm a more forcible standing.

2.3.2 Filter Sets

In our evaluation, we use three parameter files to generate the synthetic rule sets with variable size of 100 to 10K. The parameter files are:

- *acl1*: extracted from an Access Control List (ACL) rule set with 733 rules. In this rule set, the source and destination IP prefix specifications are quite specific. The destination port specification can be exact value (in most cases), arbitrary range or wildcard. All the source port specifications are wildcard. So the filters can be seen as four dimensional.

- *ipc1*: extracted form an IP Chain (IPC) rule set with 1702 filters. The rule set structure is similar to the ACL rule set except that the source ports also have exact value or arbitrary range specifications.
- *fw1*: extracted from a Firewall rule set with 283 filters. In this rule set, all fields have many wildcard specifications.

The detailed characteristics of these parameter files can be found in [6].

2.3.3 Sample Results — the HiCuts Algorithm

Here we give the evaluation results of HiCuts [32] as an example. Please refer to the project website [7] for more information and results.

HiCuts is the first decision tree-based packet classification algorithm. It takes the geometric view of the packet classification problem. We consider a k-dimensional space where k is the number of header fields involved in packet filters. Each filter defines a hypercube in this space and each packet header defines a point. The decision-tree construction algorithm recursively cuts the space into smaller sub-regions, one dimension per step. Each sub-region contains fewer overlapped hypercubes than its parent sub-region. The construction algorithm stops dividing nodes when the number of contained hypercubes is small enough for a linear search to be acceptably fast. The lookup algorithm is straightforward. Based on the value of the packet header, the algorithm follows the cutting sequence to locate the target sub-region (i.e. a leaf node in the decision tree) and then performs a linear search on the hyper cubes that overlap this sub-region.

Figure 2.3 illustrates the decision-tree construction for a 2-dimensional filter set. On the plane are five rectangles, each representing a filter. At the first step, we cut along the x-axis to generate four sub-regions. At the following steps, we choose two of these sub-regions to cut along the y-axis and x-axis, respectively. Now each sub-region overlaps ≤ 2 rectangles. If we decide it is affordable to do a linear search on at most two filters, we can stop cutting the space further. The resulting decision tree is also shown in Figure 2.3.

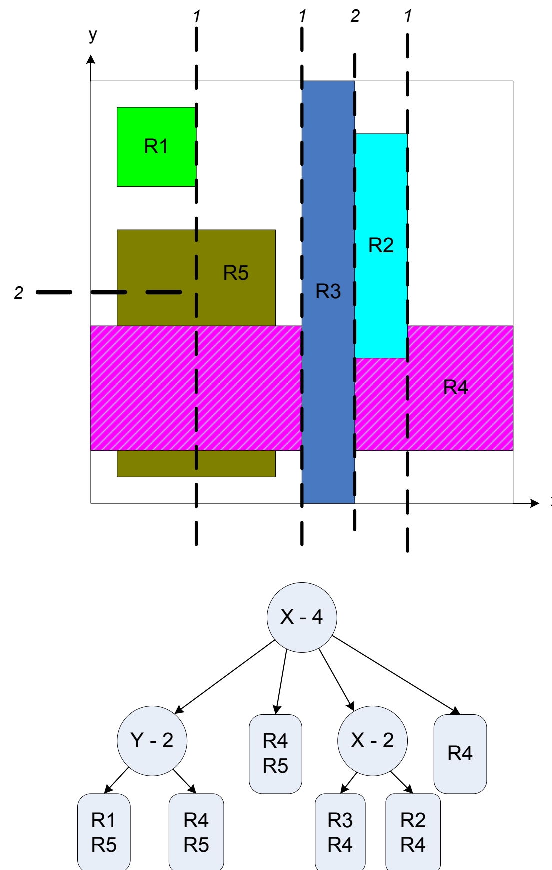

Figure 2.3: HiCuts Illustration

The number of decision tree nodes and the number of stored filters determine the space required by the algorithm data structure, and the depth of the decision tree and the number of filters in the leaf nodes determine the worst-case lookup throughput. It is difficult to find the globally optimal decision tree, so in practice the construction algorithm uses some heuristics to make optimal local decisions and trade off storage and throughput. Now we introduce the configurable parameters used to control the tradeoff.

Configurable Parameters

The Number of Cuts: Intuitively, the more cuts are made at each step, the fatter and shorter the resulting decision tree will be. However, a large number of cuts may lead to excessive duplication of filters. Therefore, we choose a suitable

number of cuts *np* at each intermediate decision tree node r. *np* is dynamically determined by the local cutting situation and a global configurable space measure factor, *spmf*. We choose the largest possible *np* as long as the following inequation is satisfied and it doesn't exceed the design limitation:

$$spmf * number\text{-}of\text{-}filters\text{-}at\text{-}r \geq \sum number\text{-}of\text{-}filters\text{-}at\text{-}each\text{-}child\text{-}of\text{-}r + np$$

For convenience of implementation, the chosen value of *np* is always a power of 2. Different configurations of *spmf* need to be tested to determine the best one.

The Dimension Chosen to Cut: Apart from the number of cuts, the dimension to cut along at each intermediate decision tree node r is also critical to the algorithm performance. The algorithm gives four options. Neither one is consistently better than the others for different filter sets. Experiments are needed to determine their effectiveness.

- *option 0*: Find the largest number of filters n_i in one child node for each field. Choose the dimension that gives the smallest n_i.
- *option 1*: Assume node r contain n filters and a child node of r contains n_i filters. Let n_i/n be a probability distribution of *np* elements. Choose the dimension that gives the largest entropy of this distribution.
- *option 2*: Choose the dimension that results in the smallest

$$\sum number\text{-}of\text{-}filters\text{-}at\text{-}each\text{-}child\text{-}of\text{-}r + np$$

- *option 3*: Choose the dimension that has the largest number of distinct range specifications of filters.

The Bucket Size: The maximum number of filters allowed in a leaf node. This is used to determine when we terminate the decision tree construction. A larger bucket size can help reduce the size and depth of a decision tree, but can yield a longer linear search time. A smaller bucket size has the opposite effects. Experiments are needed to select the proper bucket size.

Algorithm Optimizations

Child Node Reuse: After a number of cuts are performed along a dimension, many child nodes may contain an identical set of filters. In such a case, we can avoid storing these child nodes individually. Instead, we use a pointer to point to a common child node which is shared by all such child nodes. This optimization has two implications. First, a pointer must be maintained for each potential child node. Second, each node must keep its region boundary explicitly.

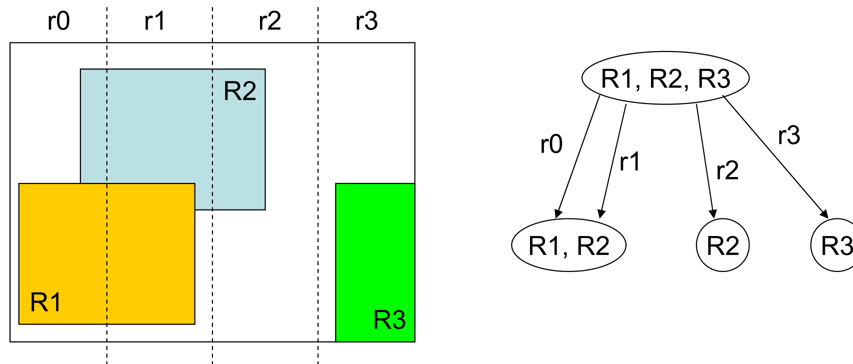

Figure 2.4: Child Node Reuse Optimization

In Figure 2.4, the first two sub-regions r_0 and r_1 have the same set of filters, $\{R_1, R_2\}$, so only one child node is generated.

Redundancy Elimination: After a sequence of cuttings is performed, the portion of a hypercube in a sub-region might be fully covered by another hypercube with a higher priority. The corresponding filter at this decision tree node is therefore redundant and can be removed to save storage.

In Figure 2.5, if the filter R_1 has higher priority than R_3, then in the sub-region r_1, R_3 becomes redundant and can be removed. However, R_1 and R_3 should coexist in the sub-region r_0 since they are only partially overlapped.

Storage Efficiency and Scalability of Storage and Throughput

In the simulation, we set the bucket size to 16, the space measure factor to 2, and the dimension selection option to 3. Figure 2.6 shows the results. The acl1 filter sets

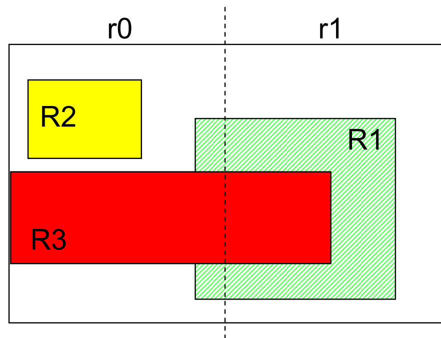

Figure 2.5: Redundancy Elimination Optimization

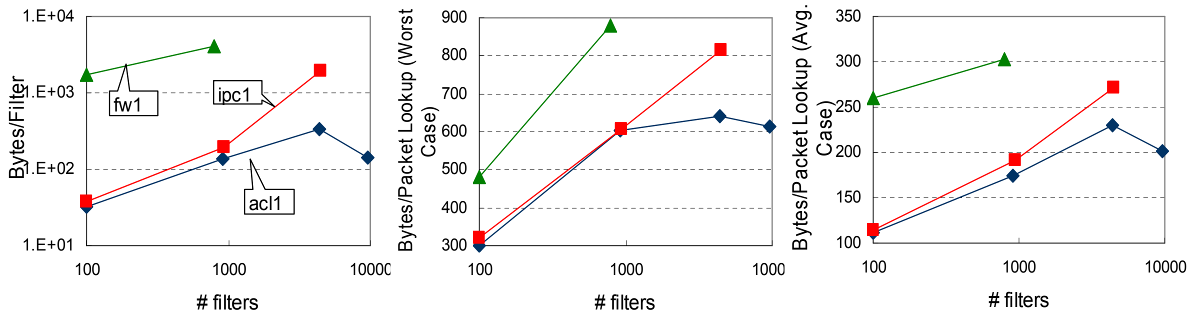

Figure 2.6: HiCuts Performance Evaluation

consistently demonstrate better performance and scalability. The fw1 filter sets give the poorest overall performance. When the number of filters exceeds 1K for fw1 or 5K for ipc1, the performance becomes unacceptable, so the data points are not shown in the figure.

Sensitivity Study

We use the acl1-10K, ipc1-1K, and fw1-100 filter sets to evaluate the algorithm sensitivity to the configurable parameters.

Sensitivity to the Space Measure Factor In this simulation, we set the bucket size to 16, the dimension selection option to 3. The results are shown in Figure 2.7. A larger space measure factor means larger storage and better performance. acl1-10K and ipc1-5K show similar performance while fw1-100 is much

worse in terms of the storage, even though there are only about 100 filters in the filter set.

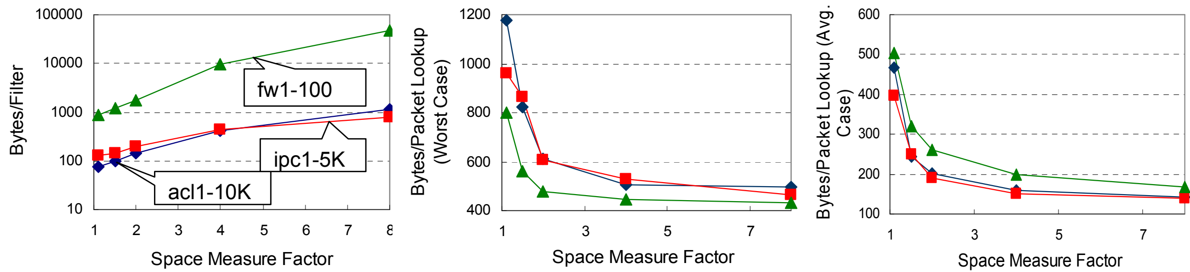

Figure 2.7: Sensitivity to Space Measure Factor

Sensitivity to the Bucket Size In the simulation, we set the space measure factor to 2, the dimension selection option to 3. The results are shown in Figure 2.8. The storage decreases monotonically when the bucket size increases. Generally a larger bucket size means a worse lookup throughput but this is not always true.

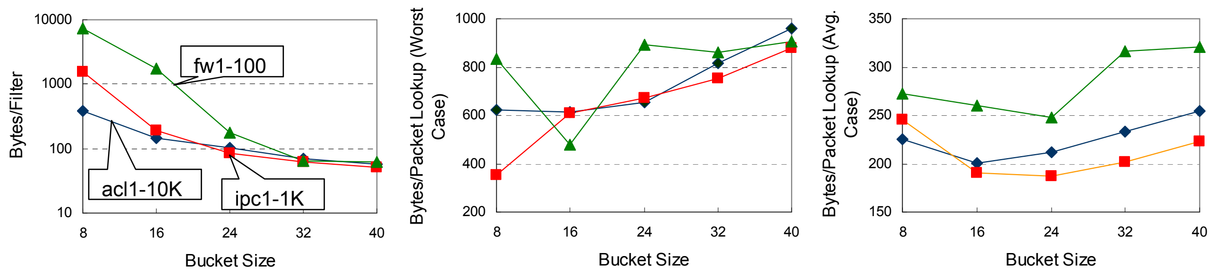

Figure 2.8: Sensitivity to Bucket Size

2.3.4 Observations

The overall evaluation results are consistent with our expectations. However, some interesting behaviors do stand out, which are not stressed in the published papers.

First, the performance of packet classification on firewall filter sets is generally poor, even for moderate sized filter sets. This is especially true for decision tree-based algorithms. For example, given a firewall filter set with only 100 filters, the HiCuts

algorithm [32] uses more than 1,000 bytes to store a filter on the average and needs to access about 500 bytes to classify a packet in the worst case. The main reason is that each field of the filter tends to cover a wide range of values, and in consequence the filters are heavily overlapped and less distinguishable. Fortunately, the number of unique ranges or prefixes on each field is not necessarily large so the decomposition-based algorithms such as RFC [33] and BV [39] work relatively better.

Second, although the decomposition-based algorithms can be very fast, their memory efficiency is poor. We observe an excessive memory consumption for moderate sized filter sets. For example, given a IPC filter set with about 1,000 filters, the RFC algorithm [33] needs 40,000 bytes to store a filter on the average. The algorithm can even exhaust the system memory when working on the filter sets with a few thousands of filters.

Third, although the decision tree-based algorithms allow a nice tradeoff between storage and throughput, their overall performance is rather disappointing. Not only can neither the storage nor the throughput be predetermined due to the limited control mechanisms, but also either end of the performance is barely satisfactory for use in high performance environments. This point is hard to see in the original papers.

We will address these problems further and propose a better algorithm in Chapter 7.

Chapter 3

Shape Shifting Tries

3.1 Introduction

The growth of Internet traffic and the growing complexity of packet processing are placing extreme demands on the design of high performance routers. More flexible and efficient methods are needed to perform high performance packet classification and route lookup.

Longest Prefix Matching (LPM) is now a well-understood problem, for which there is a variety of effective high performance algorithmic solutions [64, 42, 26, 74]. However, the expected deployment of IPv6, and the use of LPM as a component within more general packet classification mechanisms [39, 10, 46, 68] creates new challenges, justifying continuing efforts to improve the performance of LPM algorithms.

Some of the most successful methods for LPM are essentially high performance variants of the basic binary trie. The simplest variant of the binary trie is a multibit trie, in which binary nodes are replaced with d-ary nodes for values of $d > 2$. This can dramatically reduce the number of memory accesses required at the cost of less efficient use of memory. The *tree bitmap* algorithm (TBM) [26] can be viewed as a clever encoding of a multibit trie that dramatically reduces the memory penalty associated with a naive implementation. For each node in a multibit trie, the tree bitmap algorithm uses a pair of bit vectors to represent the subset of the "potential children" that are actually present and the prefixes associated with the given node. Children of a node are stored in consecutive memory locations, allowing each node to use just a single child pointer. Similarly, the next hop information associated with a node is

stored in a group of consecutive memory locations, allowing use of a single pointer to reference the next hop information. This representation allows every node in the multibit trie to be represented with a small, constant-size record. A tree bitmap algorithm implementation with an initial on-chip table of $8K$ entries (covering the first 13 bits of the IP address) and a stride of five needs just four off-chip memory accesses to traverse an IPv4 trie, with one or two additional accesses needed to retrieve the next-hop information.

Unfortunately, the time needed for trie-based lookup mechanisms grows linearly in the address length, making them less attractive for IPv6. Reference [74] describes an algorithm whose complexity grows logarithmically in the prefix length, making it much more attractive for IPv6. However, the algorithm is relatively complex to implement and its use of pre-computed markers to guide the search makes it difficult to support incremental update. An alternative approach is to extend trie-based algorithms to make them more efficient for longer address fields. The key observation needed to enable this is that as address lengths grow, the structure of the underlying binary trie intrinsically becomes much more sparse. This provides an opportunity to use alternate encodings that better match the structure of the binary trie. The *Shape-Shifting Trie* (SST) developed in this chapter is constructed from nodes that correspond to arbitrarily shaped subtrees of the underlying binary trie. This allows the SST to conform to the structure of the underlying binary trie, significantly reducing the number of SST nodes that must be traversed to perform a lookup.

General packet classification can also benefit from the SST. The decomposition-based packet classification algorithms decompose the problem into a series of single field lookups which are often conducted using LPM. Even when the filter set is very large, the number of unique prefixes on each header field is typically very small, which leads to very sparse binary tries.

In this chapter, we introduce the concept of shape shifting tries and the corresponding algorithms for high performance LPM. Sections 3.2 and Section 3.3 discuss the SST coding scheme and lookup algorithm, respectively. Section 3.4 describes the SST construction algorithms. An improved hybrid algorithm is introduced in Section 3.5. We describe a reference implementation of the algorithms in Section 3.6. The algorithm performance is evaluated for both IPv4 and IPv6 table lookups in Section 3.7.

Incremental update of SST is discussed in Section 3.8. In Section 3.9, we present several algorithm optimizations to further improve the performance. Section 3.10 summarizes the related work and Section 3.11 concludes the chapter.

3.2 SST Representation

All SST nodes have the same size (i.e. use the same amount of storage). The size determines the capacity of the SST node. The nodes of an SST correspond to subtrees of the underlying binary trie, with up to K nodes, where K is a parameter of the data structure. Since these subtrees can have an arbitrary shape, each SST node includes a *shape bitmap* (*SBM*) that represents the subtree's shape. The encoding we use was described by Jacobson [37]. To encode a tree, we first augment the tree with additional *dummy nodes*. Each original node with no children, gets two dummy children. Each original node with one child gets one dummy child. We then associate a bit with each node in the augmented tree. The value of this bit is '1' for each of the original nodes and '0' for each of the dummy nodes. The shape bitmap consists of this set of bits, listed in breadth-first order. We omit the bit corresponding to the root, since this bit is always '1'. The shape bitmap for a tree with K original nodes has $2K$ bits and any tree with up to K nodes can be represented by a shape bitmap with $2K$ bits. We can also view the shape bitmap as associating two bits with each original node. These bits indicate which of the node's potential children are present in the tree. In our illustrations, we typically adopt this viewpoint to avoid showing dummy nodes explicitly.

In addition to the shape bitmap, an SST node includes an *internal bitmap* with K bits. This identifies which of the binary trie nodes has an associated prefix. An SST node also includes an *external bitmap* with $K + 1$ bits that identifies which of the potential "exit points" from the subtree corresponds to an actual node in the underlying binary trie. The bits of the internal and external bitmaps are listed in breadth-first order of the corresponding nodes.

Each SST node also includes two pointers. The *child pointer* points to the first SST node that is a child of the given SST node. The *next hop pointer* points to the next

hop information for the first binary trie node in the SST node for which there is a prefix. The children of a given SST node are stored in sequential memory locations, allowing us to access any of the children using the *child pointer*. Similarly, the next hop information for all the nodes is stored in sequential locations, allowing us to access the next hop information for any binary node using the *next hop pointer*.

Figure 3.1 shows a binary trie that has been divided into subtrees of size less than or equal to three (K=3), along with the corresponding shape-shifting trie. In the figure, the darker binary trie nodes indicate valid prefixes.

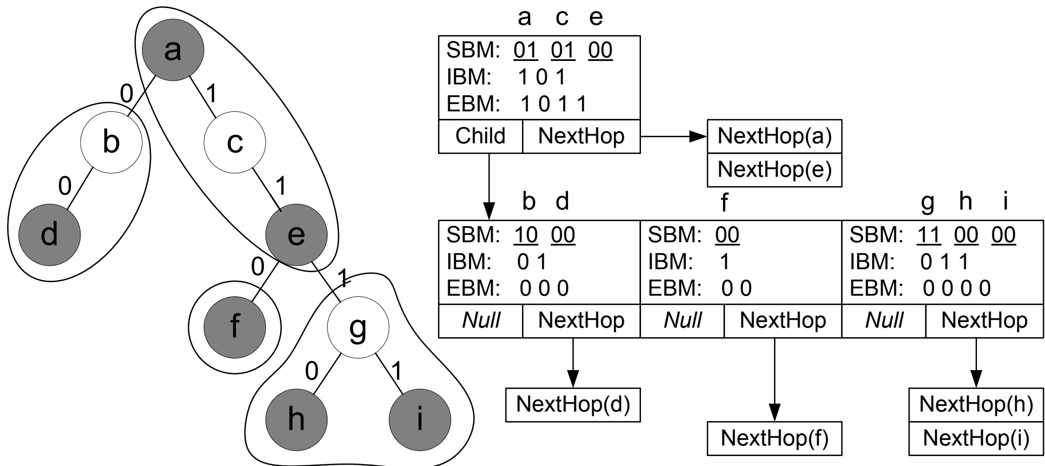

Figure 3.1: An SST and the Corresponding Data Structure

3.3 Lookup in an SST

The lookup process in an SST is similar to the lookup process for the tree bitmap algorithm [26]. The search proceeds recursively, starting from the root. At each step, we use bits from the address prefix to move through the subtree of the binary trie represented by the current SST node. We use the shape bitmap and the external bitmap to determine if the search terminates at this node or continues to one of its children. If it does continue to a child, we find the bit in the external bitmap that corresponds to the child and count the number of '1's in the bitmap that precede this bit. We then use this number as an offset to the child node of interest, from the array of children starting at the location specified by the child pointer. An

example illustrating this process is shown in Figure 3.2. Assuming the IP address being looked up is “1100”, the first SST node lookup returns the child pointer and the best matching prefix so far; the second SST node lookup returns the best matching prefix.

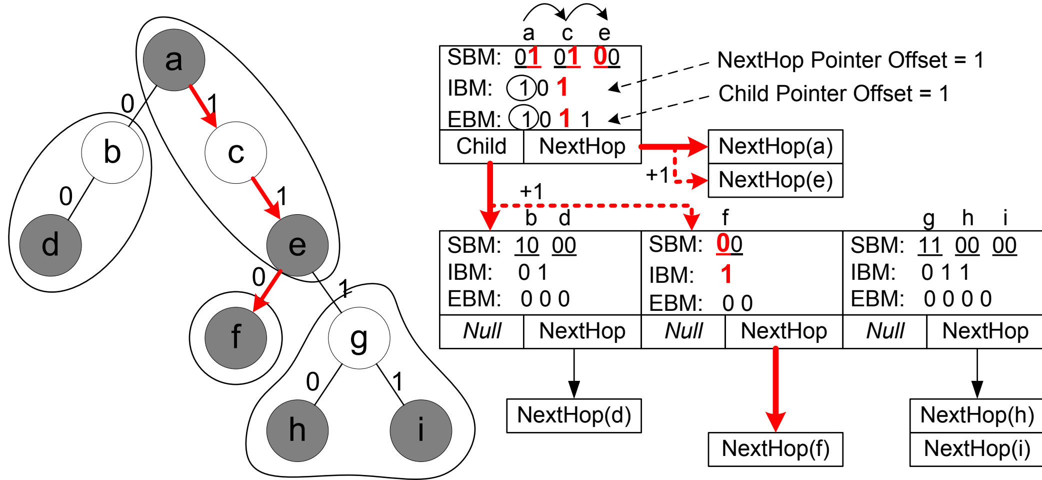

Figure 3.2: A Lookup Example Using SST

The basic step in the search algorithm requires decoding the shape bitmap. The key step is to find the bits in the shape bitmap that correspond to nodes in the path traversed by a search using bits from the IP address prefix. We start by defining n_i to be the number of nodes at distance i from the root of the augmented version of the subtree represented by the SST node (including dummy nodes). We let f_i denote the position of the bit in the shape bitmap that corresponds to the first node at distance i from the root. Note that $n_1 = 2$, $f_1 = 0$ (since we omit from the shape bitmap the bit corresponding to the root) and $f_i = f_{i-1} + n_{i-1}$. We define $ones(i, j)$ to be the number of ones in the shape bitmap in the range of bits from i through j, and note that $n_i = 2 \times ones(f_{i-1}, f_i - 1)$.

Next, we let a_i be the i-th bit of the IP address that is relevant to the node currently being decoded (so a_1 selects a child of the root of the subtree represented by the current node). We also let p_i be the index in the shape bitmap corresponding to the node on the path specified by the IP address that is at distance i from the root of the subtree. With these definitions, $p_1 = a_1$ and for $i > 1$, $p_i = f_i + 2 \times ones(f_{i-1}, p_{i-1} - 1) + a_i$.

Now, if i is the smallest integer for which the shape bitmap at position p_i is zero, then p_i corresponds to the point where the search on the IP address leaves the subtree represented by the current SST node. To determine if the search continues to another SST node, we need to consult the external bitmap. The position in the external bitmap that must be checked is the one with index equal to $zeros(0, p_i - 1)$ where $zeros(i, j)$ is defined to be the number of zeros in the shape bitmap in the range of bits from i through j. If x is the index of the proper bit in the external bitmap, and if bit x of the external bitmap is equal to '1', then the search continues at a child of the current SST node. To find the next SST node, we add an offset to the child pointer. This offset is equal to the number of ones in the external bitmap preceding bit x.

Consider the example shown in Figure 3.2. In the root SST node, we find $n_1 = n_2 = n_3 = 2$, $f_1 = 0$, $f_2 = 2$, $f_3 = 4$, $p_1 = 1$, $p_2 = 3$, $p_3 = 4$. Since bit p_3 of the shape bitmap is the first of the p_i bits that equals zero, we count the number of zeros in the shape bitmap preceding position 4. Since there are two zeros, we consult position 2 in the external bitmap to determine if the search continue to another SST node. Since bit 2 of the external bitmap is 1, there is an extending path. Also, since there is a single 1 in the external bitmap before bit 2, we add 1 to the child pointer to find the next SST node.

There are several ways to implement the lookup process for a single SST node. One conceptually simple approach is to use the equations derived above to define a combinational circuit that computes the values of p_i for $1 \leq i \leq K$. This is fast, but it does require a relatively large amount of circuitry. A simpler alternative is to use a sequential circuit that for $i \geq 1$, computes values of n_i, f_i and p_i iteratively on successive clock ticks, terminating as soon as the shape bitmap at position p_i is equal to zero. This takes up to K clock ticks in the worst case, plus another clock or two to decode the external bitmap and add the offset to the child pointer for the next memory access.

While the time needed to decode an SST node sequentially can be fairly long, note that the overall time to perform a lookup is essentially one clock tick per address bit, plus one memory access time per SST node searched. Since the lookup process does not change the SST, we can have multiple lookup engines operating in parallel on different

packets, with their memory accesses interleaved. Thus, the time to do a lookup at a single node only affects the number of engines required, not the throughput. The throughput is only a function of the memory bandwidth and the number of memory accesses needed per lookup. See [67] for a description of how this technique is used with the TBM algorithm of [26].

3.4 Constructing Optimal SSTs

A given binary trie can be represented by many different SSTs, depending on how the binary trie is partitioned. Since our primary concern is to minimize the search time, we focus on SSTs that have minimum height, where the height of a tree is defined as the length of the longest path from the root of the tree to a leaf.

However, we start by considering how to find an SST with a minimum number of nodes, ignoring the question of height. This can be done using a post-order traversal of the binary trie, pruning off subtrees to form SST nodes. Let $s(x)$ be the number of nodes in the subtree of the binary trie with root x. When we visit node x in a post-order traversal, we perform the following step.

1. if $s(x) = K$ prune the subtree at x and assign all of its nodes to a new SST node.
2. otherwise, if $s(x) > K$ and x has children a and b with $s(a) \geq s(b)$, prune the subtree at a and assign its nodes to a new SST node.

We call this the *Post-Order Pruning* (POP) algorithm. Figure 3.3(a) shows an example of the partitioning produced by the POP algorithm for $K = 3$. Figure 3.3(b) also shows a minimum height partitioning. Notice that the minimum height partition has a height of one and yields five SST nodes, while the minimum size partition has a height of three and yields four SST nodes. The example makes it clear that a single SST cannot be optimal with respect to both criteria.

Theorem 1 *The SST constructed by the POP algorithm for a given binary trie has the minimum number of nodes.*

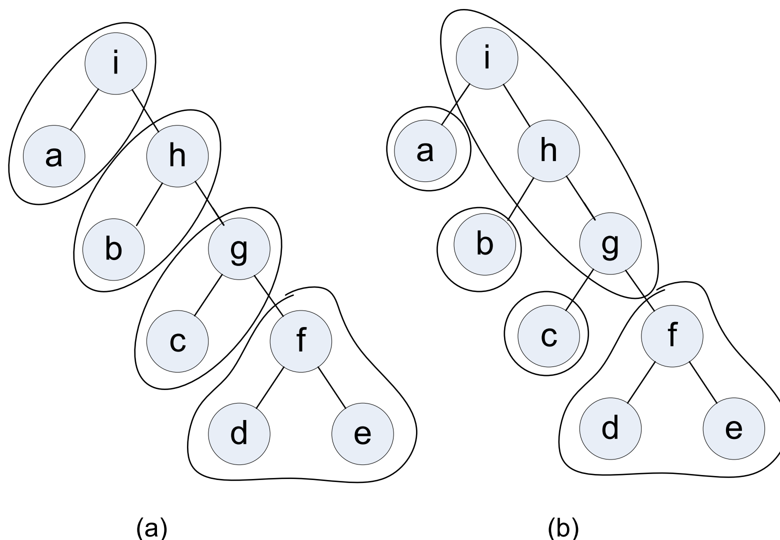

Figure 3.3: Minimum Size and Height Partitions of a Binary Tree

We sketch the proof of the optimality of the POP algorithm. We claim that the algorithm maintains the following invariant.

- *Invariant:*after every step, there is some minimum sized SST that includes all the nodes formed so far.

This is clearly true when the algorithm starts and if it is held to be true when the algorithm completes, then the constructed SST must be optimal. So, it suffices to show that the pruning rules maintain the invariant. Consider an application of the first pruning rule and let T be a minimum size SST that includes the nodes formed so far. If T does not include a node for the entire subtree at x, then at least one descendant of x must be in a different SST node than x is. This SST node cannot contain any nodes that are not descendants of x. Consequently, we can modify T so that it does form a single node from the subtree at x. The partition that T imposes on the rest of the binary trie remains unchanged. This SST cannot have any more nodes than T has.

Now, consider the second pruning rule. Again, let T be a minimum size SST that includes the nodes formed so far. Note that due to the post-order traversal, $K > s(a) \geq s(b)$. Because $s(x) > K$, T cannot form a single node from the subtree at

x. Any subtree that is pruned from the subtree at x leaves behind at least $1 + s(b)$ nodes. Consequently, we can modify T so that it includes a node for the subtree at a, but is otherwise unchanged. This modified SST cannot have any more nodes than T. So far the optimality of the POP algorithm is proved.

We now turn our attention to constructing minimum height SSTs. This requires a somewhat more complicated method that we call the *Breadth-First Pruning* (BFP) algorithm. BFP operates in multiple passes, successively pruning off subtrees with at most K nodes. It starts by computing $s(x)$, the number of descendants of node x in the binary trie, for each binary trie node x. It then repeats the following step until there is nothing left of the binary trie.

- Scan the current pruned binary trie in breadth-first order. Whenever a binary trie node y with $s(y) \leq K$ is found, prune y and its descendants from the trie and assign them to a new SST node. For all ancestors x of y, subtract $s(y)$ from $s(x)$.

The BFP algorithm can be implemented to run in $O(n^2)$ time, where n is the number of nodes in the underlying binary trie. We now show that it does produce minimum height SSTs.

3.4.1 Optimality of BFP

Consider any minimum height SST for a given binary trie. We say that a binary trie node u "belongs" to an SST node x, if u is in the subtree corresponding to x. We assign each binary trie node u a label $h(u)$ equal to the height of the SST node it belongs to. To establish the optimality of the BFP algorithm we first prove a few properties concerning these labels.

Lemma 1 *For any node u, the number of descendants v of u (including u itself) with $h(v) = h(u)$ is at most K.*

Proof: Let S be the set of descendants v of u with $h(v) = h(u)$. Assume that S contains more than K nodes and note that, they cannot all belong to the same SST

node. If U is the SST node that u belongs to, there must be some node v in S that belongs to a child V of U. But the height of U cannot equal the height of V, contradicting the assumption that S contains more than K nodes. □

We call each of the steps performed by the BFP algorithm a pass.

Lemma 2 *After i passes of the BFP algorithm, the binary trie contains no nodes u with $h(u) \leq i - 1$.*

Proof: Proof by induction. The basis ($i = 0$) is trivially satisfied, since $h(u) \geq 0$ for all u.

For the inductive step, assume that at the beginning of pass i, the trie contains no nodes u with $h(u) \leq i - 2$. Suppose that at the end of pass i, there is some node u with $h(u) = i - 1$. Since u was not removed from the trie, it must have been considered in the breadth-first scan performed by the BFP algorithm. Since it was not removed from the trie, it must have had more than K descendants at the time it was considered. But since all of its descendants v have $h(v) = i - 1$, this contradicts Lemma 1. □

Lemma 3 *Let x and y are two SST nodes formed by the BFP algorithm in the same pass, then neither is an ancestor of the other.*

Proof: The BFP algorithm scans the underlying binary trie in breadth-first order. In one pass, if a node is pruned, all of its ancestors have already been scanned and will not be touched again in the same pass. □

With these lemmas, we are now prepared to show that the BFP algorithm produces minimum height SSTs.

Theorem 2 *The SST constructed by the BFP algorithm for a given binary trie has the minimum height. The height is one less than the number of passes performed by BFP.*

Proof: Let r be the root of the binary trie and let T be the SST constructed by BFP. By Lemma 2, the SST node containing r is formed by the end of pass $h(r)+1$. By Lemma 3, no path from the root of T to one of its descendants passes through more than one node formed in the same pass. Hence, the height of T is at most $h(r)$. Since $h(r)$ was defined relative to a minimum height SST, it follows that T has minimum height also. □

3.4.2 Effectiveness of Shape Shifting Trie

The shape shifting trie method for longest prefix matching is a generalization of the Tree Bitmap algorithm (TBM) [26]. In both algorithms, the data structure node includes an internal bitmap, an external bitmap, a single child pointer and a single next hop pointer. However, SST also requires a shape bitmap that must be taken into account when comparing the two.

If we let $K = 2^S$ be the SST node size, then an SST node needs $4K+1$ bits for its three bitmaps. A TBM node can use these bits to implement a multibit trie node with a stride of $S+1$, corresponding to a subtree of the binary trie with $2K-1$ nodes. So, if the underlying binary trie is dense, the TBM data structure can be more space-efficient than the SST. But if the binary trie is sparse (fewer than half the "potential" nodes are actually present), the SST is more space-efficient. Because such sparse subtrees are very common in the tries that represent large routing tables, SST is typically more space-efficient than TBM.

The most important advantage of SST is its potential to reduce the trie height. In the extreme case of a trie that consists of one long path with m nodes, a TBM data structure has a height of approximately $m/(S+1)$, while a comparable SST has a height of $m/2^S$. For $S=4$, this is more than a three-to-one improvement. In practice we don't expect such dramatic gains, but we do find improvements as high as two-to-one for IPv6 in Section 3.7.

3.5 A Hybrid Algorithm

The discussion of the last subsection suggests that it may be worthwhile to use a hybrid approach in which TBM nodes are used to represent dense parts of the trie, while SST nodes are used to represent sparse parts. We use a bit in the node data structure to identify the format used for the current node. If the bit specifies a TBM node, we use $2K$ bits for the external bitmap, and $2K-1$ bits for the internal bitmap. If the bit specifies an SST node, we use $2K$ bits for the shape bitmap, $K+1$ bits for the external bitmap and K bits for the internal bitmap.

When building a hybrid trie, we must decide which node type to use. We modify the BFP algorithm to take this into account. During each breadth-first scan, when we encounter a node u, we first check to see if the number of nodes in the subtree is small enough to fit into a single SST node. If so, we prune the subtree and form an SST-type node. Otherwise, we check to see (1) if the height of the subtree with root u is small enough to fit into a TBM-type node, and (2) if there is no extending path from the subtree nodes other than those with the largest stride. When both are satisfied, we prune the subtree and form a TBM-type node. Note that whenever we encounter a node in a breadth-first scan, we know that the height of its parent is too large for a TBM node and the size of its parent's subtree is too large for an SST node. Also, note that the height of the hybrid data structure cannot be any larger than the height of an optimal SST. On the contrary, the hybrid data structure can potentially reduce the trie height further.

3.6 Reference Implementations

The performance evaluation to follow, is based on reference implementations of the TBM, SST and hybrid algorithms. We assume that in all three cases, the lookup data structure is stored in a 200MHz QDRII SRAM with a 36-bit wide data interface. These devices have a minimum burst size of two words, which can be read and written in a single clock cycle. In our reference implementations, the nodes for each data structure are stored in three words. For the TBM data structure (and for TBM nodes in the hybrid data structure), this allows us to implement a stride of 5 (32 bits

for the external bit map and 31 bits for the internal bitmap). For SST nodes, there is enough space for $K = 16$.

All three algorithms use a variation of the *prefix bit optimization* described in references [26, 67]. This optimization reduces the number of off-chip memory accesses substantially. It's based on the observation that we don't really need to look at the next hop pointer and the internal bitmap for most nodes visited during a search. We only need to examine these fields for the node corresponding to the longest prefix. The prefix bit optimization allows us to identify this node without looking at the next hop pointer and internal bitmap fields of all but two nodes visited. The optimization is implemented using an extra bit in each data structure node. This bit is set to '1' if the portion of the underlying binary trie corresponding to the parent node has a prefix that is relevant to the child's subtree. During the search, we remember the parent of the most recently visited node whose prefix bit was set. At the end of the search, we examine the next hop pointer and internal bitmap of this parent node. We also examine the next hop pointer and internal bitmap of the node where the search terminates. If all but the next hop pointer and internal bitmap are placed in the first two words of the three words used to store a data structure node, we only need to do one two-word access per data structure node visited, plus one or two more to retrieve the best matching next hop. Thus, if the data structure has a height of H, the worst-case number of memory accesses is $H + 3$.

These considerations lead to the node formats shown in Figure 3.4. In all cases, the third word contains the internal bitmap. For the SST node format it also contains the next hop pointer. Because the parameter K for an SST node does not have to be a power of two, one can increase the SST node size at the expense of reducing the number of bits in the child pointer. The child pointer size is then set to 20. This allows us to have up to a million SST nodes. We allocate 20 bits to the next hop pointer, allowing for up to a million prefixes. Since the largest IPv4 prefix tables currently contain fewer than 200,000 prefixes, this seems more than adequate.

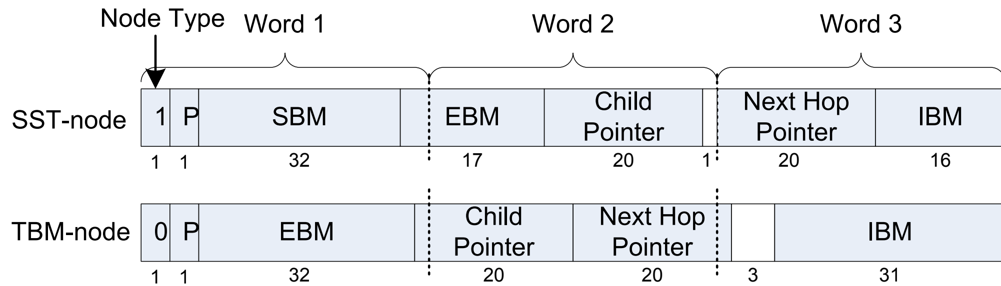

Figure 3.4: Data Structure Node Formats

3.7 Performance Evaluation

Using our reference implementations, we performed simulations on real and synthetic IP route lookup tables to examine the performance of our algorithms in terms of tree height and tree size, which determines the worst-case lookup throughput and memory consumption. Specifically, we compared three different algorithms: the tree bitmap algorithm, the original BFP SST algorithm and the BFP hybrid algorithm. We also provide the statistics of the underlying binary trie for reference. The parameter settings are illustrated in Figure 3.4.

3.7.1 Performance on IPv4 Route Lookup

To start, we simulated the algorithms for IPv4 lookup tables. We expect the largest performance improvement on small tables since we can expect the prefix tree to be sparse and contain a lot of long and skinny paths. However, we are particularly interested in the algorithm performance on very large IP lookup tables. We used a recent snapshot of the $AS1221$ BGP table from [1] for analysis. This table contains about $184K$ prefixes and has the prefix length distribution shown in Figure 3.5.

The prefixes lengths are distributed from 8 to 32. Almost half of the prefixes have length 24. Table 3.1 shows the test results.

In summary, the BFP Hybrid algorithm improves the trie height by 17% and improves the trie size by 46% over the tree bitmap algorithm. The BFP SST algorithm reaches the optimal trie depth while the multibit trie is one layer deeper. On the other

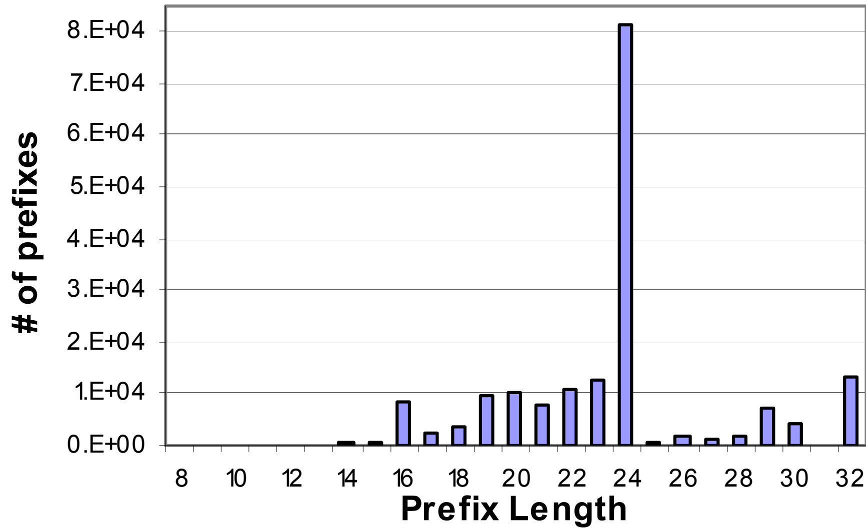

Figure 3.5: Prefix Length Distribution of the IPv4 BGP Table

Table 3.1: Performance on IPv4 BGP Table

	Trie Depth	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Binary Trie	32	487,696	-	-
Tree Bitmap	6	64,245	22.2M pkts/s	845.0K
BFP SST	5	49,177	25.0M pkts/s	648.3K
BFP Hybrid	5	34,515	25.0M pkts/s	455.0K
SST Optimal Bound	5	37,760	25.0M pkts/s	497.8K

hand, the BFP algorithms decrease the total number of nodes significantly compared with the tree bitmap algorithm. Surprisingly, the size of the trie generated by the BFP Hybrid algorithm is even lower than the pure SST optimal bound. In all cases, the data structures are small enough to fit in a single SRAM chip (4 MB chips are currently available).

Assuming we fully utilize the memory bandwidth by deploying multiple lookup engines and interleaving the memory accesses, the BFP hybrid algorithm needs only 8 memory accesses in the worst case, per route lookup. Since the QDRII SRAM can perform 200 million two-word accesses per second, it can sustain a throughput of 25 million packets per second. Assuming a worst-case packet size of 40 bytes, the system can support 8 Gbps throughput, which is close to the OC-192 link rate.

3.7.2 Performance on IPv6 Route Lookup

Evaluation is somewhat more difficult for IPv6, as there are no large real-world IPv6 routing tables available for analysis. We start with an available IPv6 BGP table from [1]. This table has fewer than 900 prefixes, with the prefix length distribution shown in Figure 3.6.

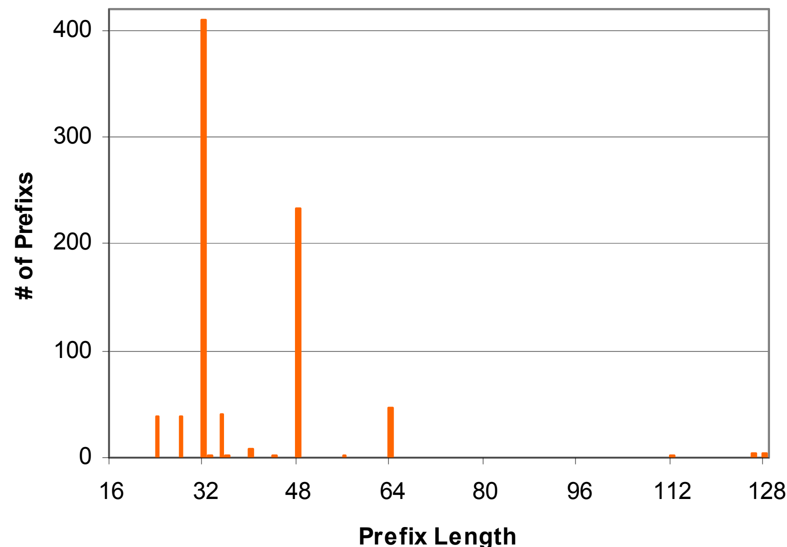

Figure 3.6: Prefix Length Distribution of the IPv6 BGP Table

In this table, prefixes with length 32, 48 and 64 dominate and only a few prefixes have length of 112, 126 and 128 bits. Table 3.2 shows the test results. Actually, if we only support this size of table, 10 bits are enough for the Child Pointer since the test result shows that there are at most 1,013 nodes in the multibit trie and even fewer in the SST. Thus using our node format layout, we can support $K = 19$ and $S = 5$. Clearly, this will make the performance of our algorithms even better while the performance of the tree bitmap algorithm stays the same. Even though we still use the same parameters, the BFP SST and the BFP hybrid algorithms yield a trie height less than one third that required by the tree bitmap algorithm. This allows them to sustain a throughput that is almost three times higher.

We see a dramatic 68% reduction on the trie depth in our algorithms as well as a 52% reduction on the number of trie nodes, compared with the tree bitmap algorithm.

Table 3.2: Performance on IPv6 BGP Table

	Trie Depth	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Binary Trie	128	5,415	-	-
Tree Bitmap	25	1,013	7.14M pkts/s	13.4K
BFP SST	8	530	18.2M pkts/s	7.0K
BFP Hybrid	8	493	18.2M pkts/s	6.5K
SST Optimal Bound	8	413	18.2M pkts/s	5.4K

Less than 7K bytes are needed to store the data structure except for the next hop information.

One can argue that this comparison is unrealistic, since current IPv6 address allocation schemes [4] use the lower half of the 128-bit IPv6 address for an interface ID. This makes it unnecessary to store more than the first 64 bits of the IP address prefix in the trie. To correct for this, we do a second comparison in which all prefixes with length longer than 64 have been removed. Table 3.3 summarizes the results. In this case, the BFP SST and BFP hybrid algorithms still provide more than a 2:1 reduction in the trie height and nearly a 2:1 reduction in the trie size, when compared to the tree bitmap algorithm.

Table 3.3: Performance on Trimmed IPv6 BGP Table

	Trie Depth	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Binary Trie	64	5,015	-	-
Tree Bitmap	12	934	13.3M pkts/s	12.3K
BFP SST	5	498	25.0M pkts/s	6.6K
BFP Hybrid	5	459	25.0M pkts/s	6.1K
SST Optimal Bound	5	386	25.0M pkts/s	5.1K

To more fully evaluate our algorithms for the IPv6 case, we resort to synthetic IPv6 prefix sets, since there are no large real-world IPv6 tables available yet. We adopt the methodology developed in [75]. The authors observe that while it is difficult to predict the structure of future large scale IPv6 route lookup tables, it's possible to use the IPv6 address allocation schemes and the characteristics of current IPv4 tables to infer information that can be used to generate realistic IPv6 tables. For evaluation,

we generate an IPv6 table with about 200K prefixes using the method proposed in [75]. The prefix length distribution of this table is shown in Figure 3.7.

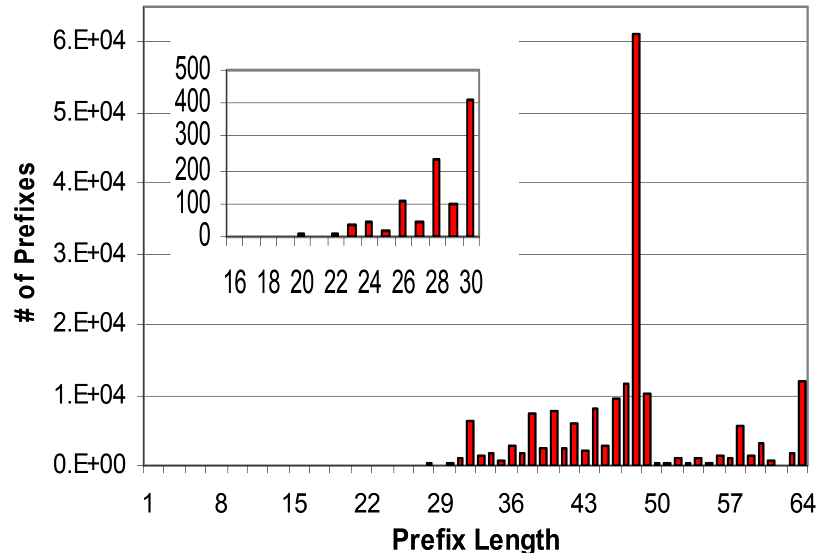

Figure 3.7: Prefix Length Distribution of the Synthetic IPv6 Table

All the prefixes in this table are for global unicast addresses, which start with the first three bits of “001”. The prefix length also retains some statistical characteristics of the IPv4 BGP table used in our earlier experiments, but scaled to IPv6. For example, the ratio of the number of even length prefixes to the number of odd length prefixes is 3 : 1. A large portion of the prefixes have length of 32, 48 and 64. This characteristic is also consistent with the IPv6 address allocation schemes and seems likely to hold true in the future IPv6 route lookup tables. Each address prefix is generated by starting with the three bit prefix 001, appending a 13 bit random number, then appending an IPv4 prefix, and finally appending some additional random bits whose length is selected to produce the desired prefix length distribution. The IPv4 prefixes were selected from the BGP table used in our earlier experiment. Figure 3.8 illustrates the prefix value distribution at each bit position.

The simulation results on this synthetic route lookup table are summarized in Table 3.4. The trie height for the BFP hybrid algorithm is about half that of the tree bitmap algorithm, and the memory required is about 40% of that required by the tree bitmap algorithm. The pure BFP SST algorithm is only slightly less efficient than the BFP hybrid algorithm.

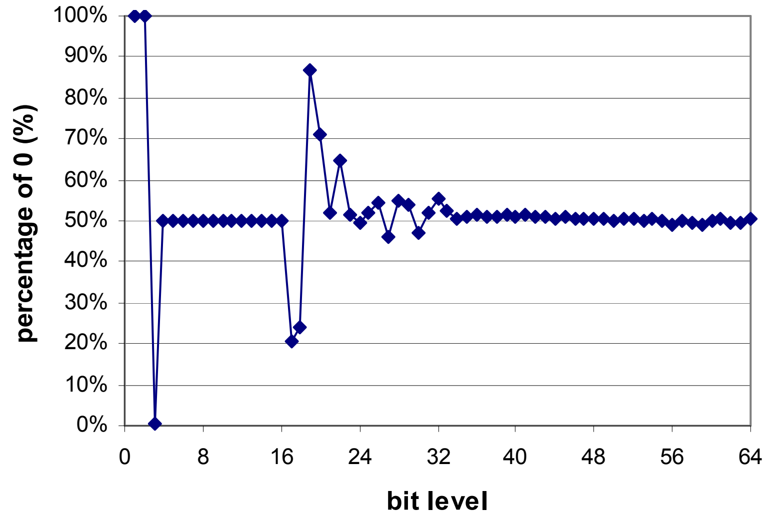

Figure 3.8: Bit Value Distribution of the Synthetic IPv6 Table

Table 3.4: Performance on the Synthetic IPv6 Table

	Trie Depth	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Binary Trie	64	4,565,260	-	-
Tree Bitmap	12	892111	13.3M pkts/s	11.8M
BFP SST	7	345,222	20.0M pkts/s	4.55M
BFP Hybrid	6	345,166	22.2M pkts/s	4.55M
SST Optimal Bound	7	312,132	20.0M pkts/s	4.12M

With similar numbers of prefixes, the binary tries for IPv6 route lookup tables are sparser than those for IPv4. Comparing the simulation results for the IPv4 and IPv6 route lookup tables, we note that the BFP hybrid algorithm makes a bigger difference in the space efficiency for the IPv4 case, apparently due to the greater density in the underlying trie.

For this scale of route lookup tables, we can do 22.2 million route lookups per second. Assuming the worst-case IPv6 packet size to be 60 bytes, a single SRAM chip can sustain 10.7 Gbps link speed.

While the height of trie-based data structures with a fixed stride length grows in proportion to the underlying binary trie height, we find that the SST height increases

only by two as we go from IPv4 to IPv6. The number of memory accesses needed is actually comparable to the number of hash table probes needed for the method described in scheme [74]. The method of [74] requires $\log_2 n$ hash probes, where n is the address length, which is 64 for IPv6.

3.7.3 Scaling Characteristics of SST

We performed some additional experiments to show how the performance of SST improves as more bits are available for the per node bit maps. We used the synthetic IPv6 BGP table used in our earlier experiment and varied the *total* number of bits available for the bitmaps from 16 to 128. The results are summarized in Figure 3.9. At most of the data points, the SST algorithm shows substantial advantages over the tree bitmap algorithm.

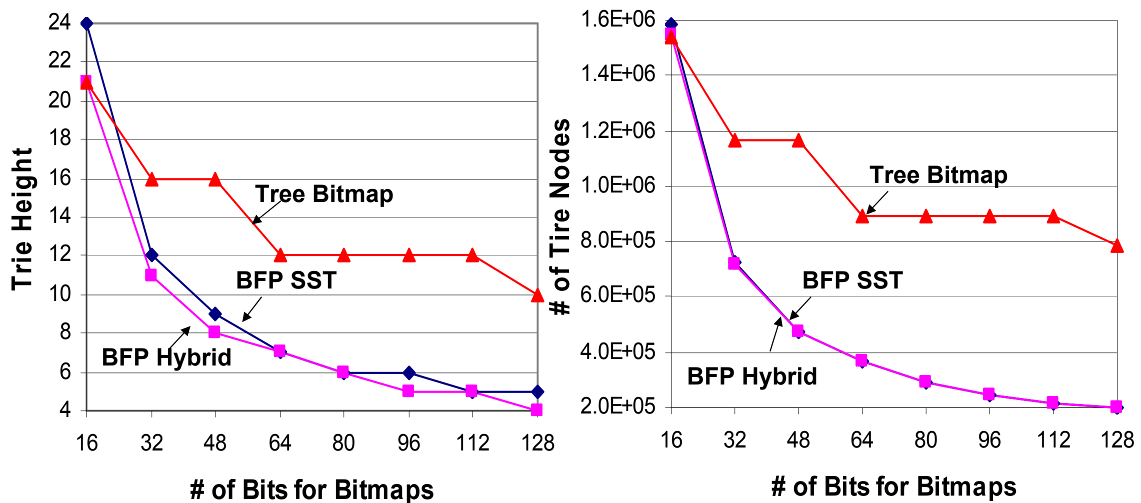

Figure 3.9: Effects of the Bit Assignment

3.8 Updating an SST

Because an SST is essentially an encoding of a binary trie, it is relatively easy to add and remove prefixes. Prefixes that convert a non-prefix node in the underlying binary trie to a prefix node are trivial to handle. It's also easy to add binary trie nodes to

SST nodes that are not yet "full". In some cases, this may require restructuring an SST node, but so long as this restructuring does not change the set of child nodes of the SST node being restructured, it affects only the one SST node. Adding new SST nodes is also straightforward, as is removing SST nodes that are no longer needed.

However, incremental modifications to an SST can result in poor performance. In particular, one can construct a sequence of insertions and deletions that results in an SST with nodes that all have depth $\log_2 K$. This can lead to worst-case performance that is worse than that of the tree bitmap algorithm. One can avoid this by restructuring the SST occasionally, should the height exceed some target bound. Determining the frequency with which such restructuring should be done is left as a subject for future study.

3.9 SST Optimizations

Given a fixed SST node size, we have three approaches to further optimize the SST algorithm. First, compress the underlying binary trie by removing the redundancies. Second, increase the SST node capacity by removing the nonessential information. Third, come up with a better SST node encoding technique by exploiting the underlying trie structure. We explore the first two approaches in this section and leave the last one as future work.

3.9.1 Compressing the Underlying Binary Trie

Since the size of the underlying binary trie correlates directly with the size of the SST, a smaller underlying trie is preferred to reduce memory usage. The real route lookup tables contain some kinds of redundancies that can be exploited to compress the underlying binary trie. We present two simple techniques called *child promotion* and *nearest ancestor collapse* to remove the redundancies.

Figure 3.10 illustrates an example to perform the *child promotion*. The darker nodes represent the valid prefixes. If two child nodes of a binary trie node both represent the

valid prefixes, we can use a one-bit shorter prefix to replace one of these two longer prefixes without changing the LPM lookup results. If this promoted child node is a leaf node, we can safely delete this child node after the promotion. In this example, we promote the prefix A in the first step and C in second step.

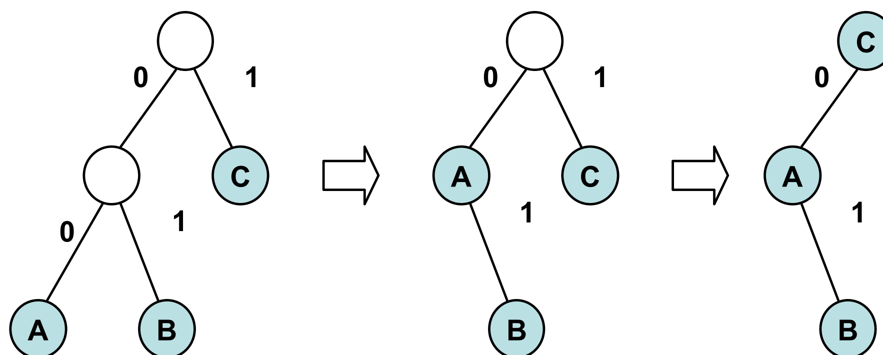

Figure 3.10: Child Promotion Optimization

To perform this optimization, after building the binary prefix trie, we traverse the trie in postorder. If the two child nodes of a binary trie node are valid prefixes, we promote one of them to be the parent node. If either node is also a leaf node, we promote it in order to delete the redundant node after the promotion.

In practice, this optimization works very well. It deletes 5.23% (5,426) of the tree nodes and promotes 7,179 prefixes for the Mae-West route lookup table. For the much larger IPv4 BGP table, It deletes 10.78% (52,585) tree nodes and promotes 74,804 prefixes.

Our second optimization is based on the fact that for route lookups, it does not matter which prefixes are matched so long as the next hop information is correct. The number of next hops or forwarding ports is limited and typically small in a router. So if the next hop of the longest matching prefixes is same as the next hop of the second longest matching prefixes for an IP address, then the longer prefix is redundant and the search for it is fruitless. We can safely collapse this prefix to the shorter one. Figure 3.11 illustrates the nearest ancestor collapse optimization. The darker nodes represent the valid prefixes and the number in the nodes indicates the next hop.

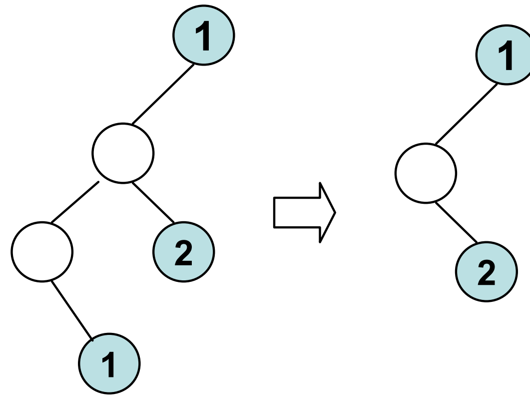

Figure 3.11: Nearest Ancestor Collapse Optimization

To perform this optimization, we traverse the binary trie in postorder. For each valid prefix, we examine its nearest ancestor that is also a valid prefix and compare their next hops. If they are same, we delete the next hop information in the longer prefix and invalidate this prefix. If this prefix node happens to be a leaf node, we recursively delete the nodes upwards until we meet its nearest ancestor or a tree branch.

This optimization decreases the number of binary trie nodes as well as the valid prefixes. Liu uses similar technique to compress the route lookup tables and shows that the number of valid prefixes can be reduced up to 26.6% [45].

When both of the optimizations help to produce a better SST, we note that they do not help to improve the multi-bit trie. Actually, one multi-bit trie implementation [64] has to apply opposing mechanisms such as the prefix expansion in order to work.

3.9.2 Increasing SST Node Capacity

We consider alternate node representations that allocate a larger share of the node space to the SBM. Given a fixed SST node size, every two more bits assigned to the SBM can increase the node capacity by one. First, we can eliminate the EBM and assign the saved bits to the SBM. Although this scheme requires allocating space for all potential children of an SST node even some of which may not be present at all, it can increase the maximum node capacity by 50%. We gain some extra throughput through this arrangement. Moreover, the reduction of the total number of SST nodes

can compensate for the extra memory consumption for the empty nodes. To further compress the SST node, we can remove the IBM from the SST nodes and store it with the next hop array instead. This costs an extra memory access to retrieve the next hop, but does not affect the worst-case number of accesses.

Assume we can read three 36-bit words per clock cycle and the basic BFP SST algorithm uses the node format shown in Figure 3.4. We then remove the EBM, IBM, and both from the SST node, respectively. Table 3.5 and Table 3.6 shows the performance comparison for the IPv4 BGP table and the synthetic IPv6 table. The throughput estimation is based on a clock frequency of 200MHz.

Table 3.5: Performance Optimization on IPv4 BGP Table

	K	Trie Depth	# of Nodes	Worst Case Throughput	Memory (Bytes)
BFP SST	16	5	49,177	33.3M pkts/s	648.3K
- EBM	22	4	42,519	40.0M pkts/s	560.6K
- IBM	22	4	37,413	33.3M pkts/s	493.2K
- EBM - IBM	34	4	26,878	33.3M pkts/s	354.3K

Table 3.6: Performance Optimization on Synthetic IPv6 Table

	K	Trie Depth	# of Nodes	Worst Case Throughput	Memory (Bytes)
BFP SST	16	7	345,222	25.0M pkts/s	4.55M
- EBM	22	6	1,209,595	28.6M pkts/s	15.57M
- IBM	22	6	252,845	25.0M pkts/s	3.26M
- EBM - IBM	34	4	445,631	33.33M pkts/s	5.74K

For sparser underlying trie as in the IPv6 case, more empty nodes are included into the data structure, so the storage may become less efficient. However, the throughput gain becomes more significant. On the other hand, when the underlying trie is relatively dense as in the IPv4 case, the storage saving is dominant.

3.10 Related Work

IP route lookup is a well-studied problem. The algorithmic approaches organize the prefixes using some sophisticated data structure and store them in memories. The lookup is conducted by a series of memory accesses. The multibit trie is the best representative of this type of technique. Another type of technique directly uses the brute-force parallel search in hardware, like TCAMs. Since our algorithm can easily support an OC-192 throughput in the worst case with extremely efficient memory usage (several bytes per prefix), there is no point using expensive and power-hungry TCAM devices for the job. Simulations in [48] also show that the multibit tries scale better than TCAMs with increasing routing table sizes in terms of the number of transistors. The disparity increases with the increased use of multi-homing and load-balancing.

For trie-based IP lookup, some techniques have been developed to improve the lookup efficiency by exploiting the structure characteristics of the prefix tree. The original binary trie can have long sequence of one-child nodes. The path compression technique [57] collapses the one-way branch nodes. Additional information must be kept in remaining nodes so that a search operation can be performed correctly. Specifically, a compressed tree node contains a variable-length bit string which records the address prefix at this point, the next hop information if a valid prefix is present, a bit position field which tells the address bit to be checked, as well as two child pointers. The lookup procedure can be described like this: Descend in the trie to the node indicated by the bit position field. If the node is marked as a valid prefix, compare the current IP fragment with the bit string which decompress as the path. If the comparison returns a match, the next hop information is kept as a best match so far. The procedure ends when finding a mismatch or reaching a leaf node. The best match kept is returned as the longest prefix match. The algorithm can decrease the storage requirement and potentially increase the lookup speed, but it is hard to implement in hardware and the effect is not so significant if the prefix tree is dense. The SST data structure can be viewed as a generalization of this approach.

The multibit trie is the more common technique to accelerate the IP route lookup speed. In this scheme, multiple bits are inspected simultaneously, so that the throughput is improved in proportion to the stride. One way to implement it is through prefix expansion: arbitrary prefix lengths are transformed into an equivalent set with the prefix length allowed by the new structure. Specifically, if the stride is S, the prefix lengths that are not a multiple of S need to be expanded to make the lengths equal to the nearest multiple of S. The prefix expansion increases the memory consumption if a fixed stride is used.

The multibit trie algorithm is generalized to enable a different stride at each trie level. Given the IP route lookup table and a desired number of memory accesses in the worst case, the selection of the stride size is implemented by controlled prefix expansion [64]. A dynamic programming algorithm is used to compute the optimal sequence of strides that minimizes the storage requirements. The algorithm is further improved by providing alternative dynamic programming formulations for both fixed and variable-stride tries [52]. The disadvantages of the controlled prefix expansion are two-fold: the update is slow and leads to sub-optimality; the hardware implementation is difficult due to the variable trie node size. Moreover, the storage optimality is only under the worst-case throughput constraints. The prefix expansion tends to increase the memory consumption anyway. For example, the memory usage of our data structure on the BGP table is roughly equal to the memory usage of the multibit trie with the controlled prefix expansion on the MaeEast table, when their worst-case tree depths are both five [64]. However, the BGP table is about five times larger than the MaeEast table. Clearly, our algorithm scales with the route table size much better.

The breakthrough to enable fast hardware implementation and eliminate prefix expansion was the tree bitmap algorithm [26]. The major idea of tree bitmap also forms the foundation of our work. A coding scheme is used to effectively compress the node size and enable fast lookup. Another similar node coding scheme can be found in [72]: A depth-1 trie numbering scheme is actually a combination of the SBM and the IBM, while the EBM is implied by the trie scanning order. However, the major concern in that paper is to compress the trie representation. Though the data structure also supports multiple bit search in one memory access, it only uses naive trie partition and does not provide an optimal trie in terms of either trie depth or trie size. Besides,

the algorithm does not consider the case when the compression actually turns out to be more inefficient than the simple *tree bitmap* representation as addressed by our hybrid algorithm.

In [22], the underlying binary trie is also partitioned to build the FSMs using the hardware logics for IP route lookups, where the partitioning is only aimed to reduce the overall number of FSMs. We also find the similarity of the SST construction problem with the technology mapping problem in the reconfigurable logic technology. Many algorithms have been proposed to optimize the depth and size of the partitions over the underlying binary tree, separately or simultaneously [31, 19]. Actually, the SST construction can be considered as a special and the simplest case of this problem, hence these algorithms can be applied to our problem with corresponding modifications. Likewise, the BFP SST algorithm, which is simple and optimal in terms of trie depth, can also be used in technology mapping scenarios.

3.11 Conclusion

We present a novel data structure, the *shape shifting trie* and a corresponding LPM algorithm in this chapter. The algorithm outperforms the classical tree bitmap algorithm and can be used in high performance routers to perform IP route lookup at even higher line speed. The algorithm also scales well to fast growing route lookup tables and is especially attractive for IPv6 route lookups, by taking advantage of the sparsity of the prefix tree. The efficient SST node coding and the SST construction algorithm both contribute to a faster LPM solution. We prove that our algorithm achieves the optimal bound on the SST depth and is close to the optimal bound of the SST size through analysis and simulations. A hybrid algorithm which leverages the benefits of both the tree bitmap algorithm and the SST algorithm can even beat the bounds and achieves better performance.

We show that using a single QDRII SRAM chip, in the worst case, the hardware implementation of the algorithm can perform large scale IPv6 route lookups at the wire-speed of OC-192. By deploying more QDRII SRAM chips, we can scale the throughput to OC-768. Based on the fact that the underlying binary tree appears to

be much denser when close to the root, we may also borrow the idea in [42] to build a jump table using some number of prefix bits of the IP address, then build an SST for each table entry. Each SST will then be considerably shorter so that the worst-case number of memory accesses per lookup is further reduced.

We have mentioned that some underlying trie structure can be used to improve the algorithm performance. For a sparse portion of the underlying binary tree which contains only a few branches, another shape encoding scheme may be more efficient. For example, using an "up/down counts" scheme as illustrated in Figure 3.12, we can directly describe the paths and their relationships in a subtree so that the subtree shape can be easily recovered with fewer bits. In the example, the up and down counters count the lengths of path segments. Paths are listed in depth-first order. This scheme needs 19 bits to encode the subtree shape while the breadth-first encoding needs 24 bits. Though this scheme makes the decoding process more complex, it may generate a more efficient SST. We consider applying this in the hybrid algorithm in future work to evaluate its impact to the SST construction algorithm, the implementation cost, and the performance gain.

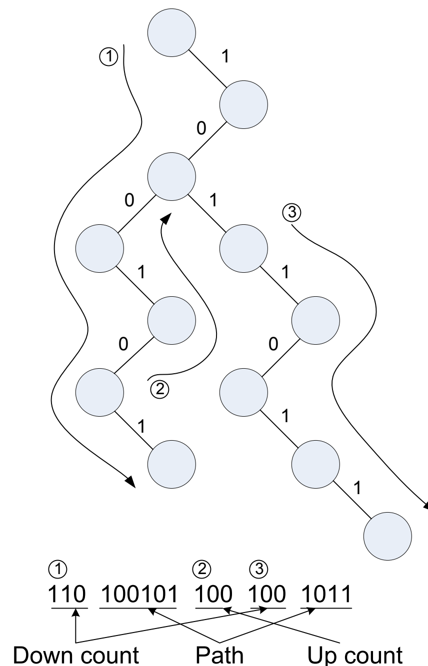

Figure 3.12: Up Down Counts Shape Encoding

Chapter 4

Fast Hash Table

4.1 Introduction

A hash table is a versatile data structure for performing fast associative lookups, which requires $O(1)$ average memory accesses per lookup. Due to its wide application in network packet processing, including IP lookup, packet classification, and per-flow state management, some modern network processors even provide built-in hash units [36]. The applications using hash tables typically appear in the data-path of high-speed network devices. Hence, they must be able to process packets at line speed, which makes it imperative for the underlying hash tables to deliver good lookup performance.

4.1.1 Hash Tables for Packet Processing

Following is a short discussion of how various network processing applications use hash tables and why their lookup performance is important.

IP Route Lookup

Efficient hash tables are crucial for some IP routing lookup algorithms. In particular, the Binary Search on Prefix Lengths [74] algorithm, which has the best theoretical performance of any sequential algorithm for the best-prefix matching problem, uses

hash tables. The algorithm described in [23] uses parallel lookups of on-chip Bloom filters to identify which off-chip hash tables must be searched to find the best matching prefix for a given packet.

In [74], prefixes are grouped according to their lengths and stored in a set of hash tables. A binary search on these tables is performed to find the matching prefixes of the destination IP address. Each search step probes a corresponding hash table to find a match. By storing extra information along with the member prefixes in hash tables, a match in a given table implies that the longest matching prefix is at least as long as the length of prefixes in the table, whereas a failure to match implies the longest matching prefix is shorter. If there are W different possible prefix lengths, the search requires at most $\log W$ probes of the hash tables. For IPv4 lookup, this means we need to perform lookups in five hash tables in the worst case. Even with the use of controlled prefix expansion [64] we need multiple hash table lookups depending on the resulting number of unique prefix lengths. This algorithm critically demands better hash table lookup performance to preserve the performance gained by binary search.

Reference [23] presents a hardware based LPM algorithm for IP lookups. The technique improves the performance of a regular hash table using Bloom filters. When unsuccessful searches in a hash table are dominant, most of them can be avoided by first filtering them through a Bloom filter. In this algorithm, prefixes are also grouped by length. The prefixes in each group are programmed in a Bloom filter and stored in a separate hash table as well. The Bloom filters are maintained in high-bandwidth and small on-chip memory while the hash table resides in the slow and high-volume off-chip memory. Before a search is initiated in the off-chip hash table, the on-chip Bloom filter is probed to check if the item has been programmed. This typically allows one to just probe a single off-chip hash table. However, if the probe of the off-chip hash table requires multiple memory accesses, the performance of the algorithm suffers.

The BART scheme [47] also uses hash tables for routing table lookup. It constructs simple hash functions by picking a few bits in the IP address. To bound the collisions in a hash bucket, it selects the bits for use in the hash function based on an exhaustive

search of the space of possible bit sets. This makes the configuration of the lookup engine for a particular set of address prefixes less flexible and time-consuming.

Packet Classification

Hash tables are also used for some packet classification algorithms. Fundamentally, many packet classification algorithms first perform lookups on each individual header field and then leverage the results to narrow down the search to a smaller subset of filters [39, 10, 46, 30]. Since the lookups on an individual header field can also be performed using one of the hash table-based algorithms, improving the hash table performance also benefits these packet classification algorithms.

Some packet classification algorithms directly apply hash tables. The tuple space search algorithm [63] groups the rules into a set of “tuples” according to their prefix lengths specified for different header fields. Each group is then stored in a hash table. The packet classification performs exact match operations in all the hash tables. While the algorithm analysis in [63] centers on the number of distinct tuples, the hash table lookup performance also directly affects the classification throughput.

Exact flow matching is an important subproblem of the general packet classification, where the lookup performs an exact match on the packet 5-tuple header fields. In [66], exact filters are used for flows with reserved bandwidth and multicast in high performance routers as an auxiliary component to general packet classification. The search technique employs a hash table with chaining to resolve collisions. A hash key based on the low-order bits of the source and destination addresses is used to probe an on-chip hash table containing “valid” bits. For a packet under lookup, if the corresponding “valid” bit is set, the hash key is then used to index another hash table in off-chip SRAM. This architecture resembles our schemes with only a single hash function. It has the limited ability to filter out some of the unnecessary off-chip hash table queries, but in essence, this is still a naive hash table implementation. The hash collisions directly impact the system throughput.

General packet classification is a difficult problem and tends to be more time consuming. Fortunately, the network flow temporal locality can be exploited to improve the

system throughput through caching. Since the header fields used to identify a flow contain more than a hundred bits, caching the whole string uses excessive resources. Reference [16] presents a scheme using a multi-predictive Bloom filter to save memory usage while maintaining high hit rate. However, this scheme can cause misclassification and only supports a few actions. We believe that by using our fast hashing scheme, we can achieve precise matching and impose no limitation on the number of actions.

Maintaining Per-flow Context

One of the most important applications of hash tables in network processing is in the context of maintaining connection records or per-flow state. Per-flow state is useful in providing QoS, flow measurements and monitoring, and payload analysis for Network Intrusion Detection Systems (NIDS).

For example, the network intrusion detection systems such as Bro [50] and Snort [5] maintain a hash table of connection records for active TCP connections. A record is created and accessed by computing a hash over the 5-tuple of the TCP/IP header. This record contains certain information describing the connection state and is updated upon the arrival of each packet of that connection. Efforts have been made to implement intrusion detection systems in hardware for line speed packet processing [54, 25]. In these implementations, connection records are maintained in DRAM due to their vast size. Similarly, hardware-based network monitoring systems such as NetFlow [2] or Adaptive NetFlow [28] maintain a hash table of connection records in DRAM.

In these applications, it is crucial to be able to keep up with the pace of accessing the records for the back-to-back minimum sized packets in order to maintain wire-speed throughput. Unfortunately, in a naive hash table there are always collisions which force multiple connection records to be in the same hash bucket. In the worst case, back-to-back packets can access the same connection record which is at the end of the record list in a bucket. This is undesirable for a system with a little or no buffer at all.

The above discussion illustrates the role of hash tables in a variety of network packet processing applications and clearly states their direct impact on performance.

4.1.2 Related Work

A hash table lookup involves hash computation followed by memory accesses. While memory accesses due to collisions can be moderately reduced by using sophisticated cryptographic hash functions such as MD5 or SHA-1, they are difficult to compute quickly. In the context of high-speed packet processing devices, even with specialized hardware, such hash functions can take quite a few clock cycles to produce an output. For instance, some of the existing hardware implementations of the hash cores consume more than 64 clock cycles [35], which exceeds the budget of minimum packet time. Moreover, the performance of such hash functions is no better than the theoretical performance with the assumption of uniform random hashing. Hence, we are forced to use simple and practical hash functions which, unfortunately, may suffer from high collision rates.

Another avenue to improve the hash table performance would be to devise a perfect hash function based on the items to be hashed. While this would deliver the best performance, searching for a suitable hash function can be a slow process and needs to be repeated whenever the set of items undergoes changes. Moreover, when a new hash function is computed, all the existing entries in the table need to be re-hashed for correct search. This impedes the normal operations on the hash table making it impractical in high-speed packet processing. Some applications instead settle on using a “semi-perfect” hash function which can tolerate a predetermined collision bound. However, even searching for such a hash function requires time in the order of minutes [64, 47].

Multiple hash functions are known to perform better than a single hash function [12]. With multiple hash tables, each having a different hash function, the items colliding in one table are hashed into the other tables. Each table has smaller size and all hash functions can be computed in parallel. Another multi-hashing algorithm, the d-random scheme, uses only one hash table but d hash functions [8]. Each item is hashed by d independent hash functions, and the item is stored into the least loaded

bucket. A search needs to examine d buckets but the bucket's average load is greatly reduced. A simple variation of d-random, the d-left scheme, is proposed to improve IP lookups [13]; this approach generalizes the 2-left scheme in [73]. In this scheme, the buckets are partitioned into d sections, each time a new item needs to be inserted, it is inserted into the least loaded bucket (left-most in case of a tie). Simulation and analysis show the performance is better than d-random. While these ideas are similar to our fast hash table algorithm, our approach uses an on-chip Bloom filter to eliminate the need to search multiple buckets in the off-chip memory.

A Bloom filter [11] can be considered a form of multi-hashing. The Counting Bloom Filter [29] extends the simple binary Bloom filter by replacing each bit in the filter with a counter, which counts the number of items that is hashed to each bucket. This makes it possible to implement a deletion operation on the set represented by the Bloom filter. Some lookup schemes also use Bloom filters to avoid unnecessary searches of an off-chip hash table [23, 24]. However, they do nothing to reduce the time needed to search the off-chip table, so the lookup performance can still be unpredictable. In contrast, our fast hash table lookup algorithm fully uses the information gained from the front-end Bloom filter to optimize the following exact match lookup in the off-chip memory.

4.1.3 Scope for Improvement

From a theoretical perspective, hash tables are among the most extensively studied data structures. From an engineering perspective, designing a good hash table can still be a challenging task with potential for several improvements. The main engineering aspect that differentiates our hash table design from the rest is the innovative use of the advanced embedded memory technology in hardware. Today it is possible to integrate a few megabits of SRAM into a fairly small amount of chip area. For instance, some modern FPGA devices contain hundreds of embedded SRAM blocks with two read/write ports, totaling over 10 Mbits [79]. We exploit the high lookup capacity offered by such memory blocks to implement more efficient hash tables.

At the same time it is important to note that embedded memory on its own is not sufficient to build a good hash table when we need to maintain a large number of

items. For instance, we cannot squeeze 100,000 TCP connection records each of 32 bytes into a hash table built with only 5 Mbits of on-chip memory. Thus, we must resort to using the commodity memory such as SDRAM to store the items in the hash table. Since SDRAM is inherently slow, we have to reduce the off-chip memory accesses resulting either from hash collisions or unsuccessful searches for efficient processing. This leads us to the question: Can we make use of the small but high bandwidth on-chip memory to improve the lookup performance of an off-chip hash table? The answer to this question forms the basis of our algorithm. We show that a small amount of on-chip memory can be used effectively to avoid unsuccessful searches in the off-chip hash table as well as to reduce the hash collisions by orders of magnitudes.

We start from the well-known Bloom filter data structure [11] and extend it to support hash table lookups with reduced lookup time. We use a small amount of on-chip multi-port SRAM to realize a counting-Bloom-filter-like data structure such that it not only answers the membership query on the search items but also helps us reduce the search time in the off-chip hash table.

The rest of this chapter is organized as follows. Section 4.2 introduces our fast hash table data structure and lookup algorithm. Section 4.3 provides a detailed mathematical analysis of the proposed hash table algorithm. Sections 4.3 and 4.4 provide comparisons on the average search time and the expected collision list length of the naive hash table and our fast hash table, theoretically and experimentally. Section 4.5 briefly discusses some implementation considerations and Section 4.6 concludes the chapter.

4.2 Fast Hash Table and Lookup Algorithm

For the purpose of clarity, we develop our algorithm and hash table architecture incrementally starting from a naive hash table (NHT). We consider the hash table algorithm in which collisions are resolved by chaining since it has better performance than open addressing schemes and is one of the most popular methods [20].

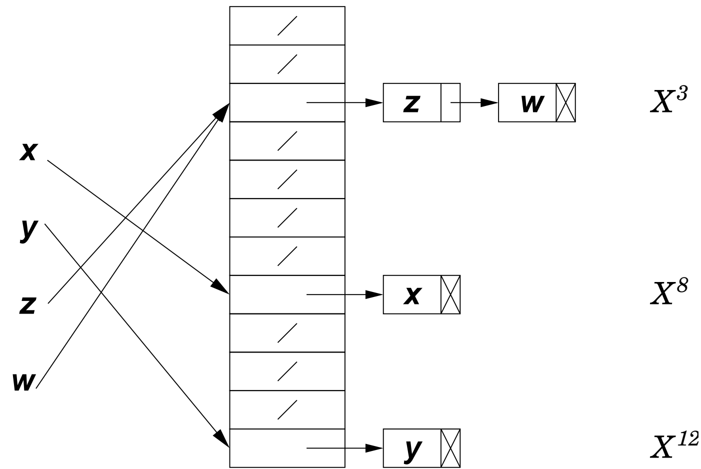

Figure 4.1: A Naive Hash Table

An NHT consists of an array of m buckets with each bucket pointing to the list of items hashed to it. We denote by X the set of items to be inserted into the table. Further, let X^i be the list of items hashed to bucket i and X^i_j the jth item in this list. Thus,

$$X^i = \{X^i_1, X^i_2, X^i_3, ..., X^i_{a_i}\}$$
$$X = \bigcup_{i=1}^{L} X^i$$

where a_i is the total number of items in the bucket i and L is the total number of lists present in the table. In the Figure 4.1, $X^3_1 = z$, $X^3_2 = w$, $a_3 = 2$ and $L = 3$.

The insertion, search and deletion algorithms are straight-forward:

InsertItem$_{NHT}(x)$
1. $X^{h(x)} = X^{h(x)} \cup x$

SearchItem$_{NHT}(x)$
1. if $(x \in X^{h(x)})$ return true

2. else return false

DeleteItem$_{NHT}$**(**x**)**

1. $X^{h(x)} = X^{h(x)} - x$

where $h()$ is the hash function based on uniform random hashing.

4.2.1 Basic Fast Hash Table

We now present our Fast Hash Table (FHT) algorithm. First we present the basic form of our algorithm which we call Basic Fast Hash Table (BFHT) and then we improve upon it.

We begin with the description of the Bloom filter which is at the core of our algorithms. A Bloom filter is a hash-based data structure to store a set of items compactly. To insert an item in a Bloom filter, we compute k hash functions on each item, each of which returns an address of a bit in a bitmap of length m. All the k bits chosen by the hash values in the bitmap are set to '1'. By doing this, we essentially program the filter with a signature of the item. By repeating the same procedure for all the input items, the Bloom filter can be programmed to contain a summary of all the items. This filter can be queried to check if a given item is programmed in it. The query procedure is similar—the same k hash functions are calculated over the input and the corresponding k bits in the bitmap are probed. If all the bits are set then the item is said to be present, otherwise it is absent. However, since the bit-patterns of multiple items can overlap within the bitmap, the Bloom filter can give false-positive results.

For the ensuing discussion, we use a variant of Bloom filter called Counting Bloom Filter [29] in which each bit of the filter is replaced by a counter. Upon the insertion of an item, each counter indexed by the corresponding hash value is incremented. Therefore, a counter in this filter essentially gives us the number of items hashed to

that position. We will show how this information can be used to minimize the search time in an associated hash table.

We maintain an array C of m counters where each counter C_i is associated with a bucket i of the hash table. We compute k hash functions $h_1(), ..., h_k()$ over an input item and increment the corresponding k counters indexed by these hash values. Then, we store the item in the lists associated with the k buckets. Thus, a single item is stored k times in the off-chip memory. The following algorithm describes the insertion of an item in the table.

InsertItem$_{BFHT}(x)$

1. for $(i = 1 \text{ to } k)$
2. if $(h_i(x) \neq h_j(x)\ \forall j < i)$
3. $C_{h_i(x)}$++
4. $X^{h_i(x)} = X^{h_i(x)} \cup x$

Note that if more than one hash function maps to the same address then we increment the counter only once and store just one copy of the item in that bucket. To check if the hash values conflict, we keep all the previously computed hash values for that item in registers and compare the new hash value against all of them (line 2).

The insertion procedure is illustrated in the Figure 4.2. In this figure, four different items, x, y, z, and w are shown to have been sequentially inserted into the data structure. Each of the items is replicated in $k = 3$ different buckets and the counter value associated with the bucket reflects the number of items hashed in it.

The search procedure is similar to the insertion procedure: given an item x to be searched, we compute k hash values and read the corresponding counters. When all the counters are non-zero, the filter indicates the presence of the input item in the table. We then proceed to verify it in the off-chip table by comparing it with each item in the linked list associated with one of the buckets. If the counters are kept in the fast on-chip memory such that all of the k counters associated with the item can be checked in parallel, then in almost all cases we avoid an off-chip access if any bucket counter checked is zero. Given the recent advances in the embedded memory

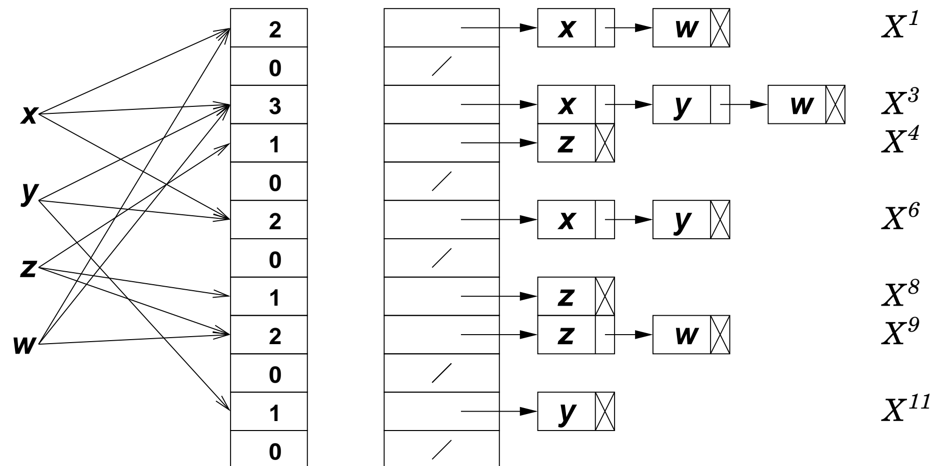

Figure 4.2: Basic Fast Hash Table (BFHT)

technologies, it is feasible to implement these counters in a high speed multi-port on-chip memory.

Secondly, the choice of the linked list to be inspected is critical since the list traversal time depends on the length of the linked list. Hence, we choose the list associated with the counter with the smallest value to reduce the off-chip memory accesses. The speedup of our algorithm comes from the fact that it can choose the smallest list to search while an NHT does not have any choice but to trace only one linked list which can potentially have several items in it.

As will be shown later, in most cases, for a carefully chosen value of the number of buckets, the minimum valued counter has a value of one requiring just a single memory access to the off-chip memory. In our example shown in Figure 4.2, if item y is queried, we need to access only the list X^{11}, rather than X^3 or X^6 which are longer than X^{11}, according to the bucket counters.

When multiple counters indexed by the input item have the same minimum value then somehow the tie must be broken. We break the tie by simply picking the minimum valued counter with the smallest index. For example, in Figure 4.2, item x has two bucket counters set to 2, which is also the smallest value. In this case, we always access the bucket X^1. This step turns out to be critical to enable some further optimizations.

Finally, if the input item is not present in the item list, then clearly it is a false positive match indicated by the CBF.

The following pseudo-code summarizes the search algorithm of BFHT.

SearchItem$_{BFHT}(x)$

1. $C_{min} = min\{C_{h_1(x)}, ..., C_{h_k(x)}\}$
2. if ($C_{min} == 0$)
3.   return false
4. else
5. i = SmallestIndexOf(C_{min})
6.   if ($x \in X^i$) return true
7.   else return false

With the data structure above, deletion of an existing item is easy. We simply decrement the counters associated with the item and delete all the copies from the corresponding lists. The following pseudo-code summarizes the deletion algorithm of BFHT.

DeleteItem$_{BFHT}(x)$

1. for ($i = 1$ to k)
2.   if ($h_i(x) \neq h_j(x)\ \forall j < i$)
3. $C_{h_i(x)} - -$
4. $X^{h_i(x)} = X^{h_i(x)} - x$

4.2.2 Pruned Fast Hash Table (PFHT)

In BFHT, we need to maintain up to k copies of each item which requires k times more external memory compared to NHT. However, it can be observed that in a BFHT only one copy of each item, i.e., the copy associated with the first minimum valued counter, is accessed when the table is probed. The remaining $(k-1)$ copies of the item are therefore redundant. This observation offers us the first opportunity

to optimize the memory usage: all the other copies of an item except the one that is accessed during the search can now be deleted. After this pruning procedure, we have exactly one copy of the item which makes the memory consumption the same as that of NHT. We name the resulting hash table a Pruned Fast Hash Table (PFHT).

The following pseudo-code summarizes the pruning algorithm.

PruneSet(X)

1. for (each $x \in X$)
2. $C_{min} = min\{C_{h_1(x)}, ..., C_{h_k(x)}\}$
3. $i =$ SmallestIndexOf(C_{min})
4. for $(l = 1$ to $k)$
5. if $(h_l(x) \neq i)$ $X^{h_l(x)} = X^{h_l(x)} - x$

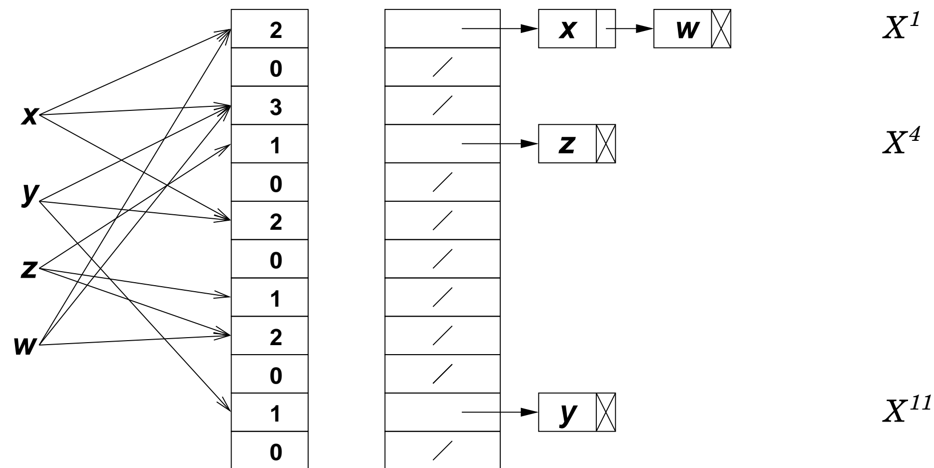

Figure 4.3: Pruned Fast Hash Table (PFHT)

The pruning procedure is illustrated in Figure 4.3. It is important to note that during the pruning procedure, the counter values are not changed. Hence, after the pruning is completed, the counter value no longer reflects the number of items actually present in the associated linked list and is usually greater than it. However, for a given item, the bucket with the smallest counter value always contains it. This property ensures the correctness of the search results. Another property of pruning is that it is independent of the order in which the items are pruned since it depends just on the

counter values, which are not altered. Hence, pruning in sequence x-y-z-w will yield the same result as pruning it in z-y-x-w.

A limitation of the pruning procedure is that now the incremental updates to the hash table are difficult to perform. Since counter values no longer reflect the number of items in the associated list, if counters are incremented or decremented for any new insertion or deletion, then it can disturb the counter values corresponding to the existing items in the bucket which in turn will result in incorrect lookups. For example, in Figure 4.3, the item y maps to the lists $\{X^3, X^6, X^{11}\}$ with counter values $\{3, 2, 1\}$ respectively. If a new item, say v, is inserted which also happens to share the bucket 11 then the counter will be incremented to 2. Hence, the minimum counter valued bucket with the smallest index associated with y is no longer bucket 11 but bucket 6 which does not contain y at all. Therefore, a search on y will result in an incorrect result. With this limitation, new insertions and deletions may require us to reconstruct the data structure from scratch which makes this algorithm impractical for dynamic item sets.

We now describe a version of *InsertItem* and *DeleteItem* algorithms which can be performed incrementally. The basic idea used in these functions is to maintain the invariant that out of the k buckets indexed by an item, the item should always be placed in a bucket with the smallest counter value. In case of a tie, it should be placed in the one with the smallest index. If this invariant is maintained at every point then the resulting hash table configuration will always be the same irrespective of the order in which the items are inserted.

In order to insert an item, we first increment the corresponding k counters. If there are any items already present in those buckets then their corresponding smallest counter might be altered. However, the counter increments do not affect all the other items. Hence, each of these items must be relocated to another bucket that satisfies the invariant. In other words, for inserting one item, we need to reconsider all and only the items existing in the chosen k buckets.

The following pseudo-code describes the insertion algorithm.

InsertItem$_{PFHT}(x)$

1. $Y = x$

2. for ($i = 1$ to k)
3. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
4. $Y = Y \bigcup X^{h_i(x)}$
5. $X^{h_i(x)} = \phi$
6. $C_{h_i(x)}$++
7. for (each $y \in Y$)
8. $i = argmin\{C_{h_1(y)}, ..., C_{h_k(y)}\}$
9. $X^i = X^i \cup y$

In the pseudo-code above, Y denotes the list of items to be considered for insertion. It is first initialized to x since that is definitely the item we want to insert (line 1). Then for each bucket x mapped to, if the bucket was not already considered (line 3), we increment the counter (line 6), collect the list of items associated with it (line 4) since now all of them must be considered for relocation. We also delete the items from the bucket (line 5). Finally, all the collected items are re-inserted (lines 8-9). Note that we do not need to increment the counters while re-inserting them since the items were already inserted earlier.

The data structure has n items stored in m buckets, so the average number of items per bucket is n/m. Hence the total number of items read from buckets is nk/m requiring as many memory accesses. Finally $1 + nk/m$ items are inserted in the table which again requires as many memory accesses. Hence the insertion procedure has an expected complexity of the order $O(1 + 2nk/m)$ operations totally. Since for an optimal Bloom filter configuration, $k = m \ln 2/n$, the overall memory accesses required for insertion are only $1 + 2\ln 2 \approx 2.44$.

Unfortunately, incremental deletion is not as straight-forward as incremental insertion. When we delete an item, we also need to decrement the corresponding counters. This might cause these counters to be eligible as the smallest counter for some items which are hashed to them but stored in other buckets. However, now that we keep just one copy of each item, we cannot tell which items hash to a given bucket if they are not in that bucket. This information can be acquired from the pre-pruning data structure i.e. BFHT in which an item is inserted in all the k buckets. Therefore, in order to perform an incremental deletion, we must maintain an off-line BFHT like

the one shown in Figure 4.2. This shadow copy need not contain any actual data but must allow each item to be unambiguously identified. Such a data structure can be maintained in the system management software which is responsible for updating the hash table.

In order to differentiate between the off-line BFHT and the on-line PFHT we denote the off-line lists by χ and the corresponding counter by ζ. Thus, χ^i denotes the list of items associated with bucket i, χ^i_j the jth item in χ^i, and ζ_i the corresponding counter. The following pseudo-code describes the deletion algorithm.

DeleteItem$_{PFHT}(x)$

1. $Y = \phi$
2. for ($i = 1$ to k)
3. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
4. $\zeta_{h_i(x)} --$
5. $\chi^{h_i(x)} = \chi^{h_i(x)} - x$
6. $Y = Y \bigcup \chi^{h_i(x)}$
7. $C_{h_i(x)} --$
8. $X^{h_i(x)} = \phi$
9. for (each $y \in Y$)
10. $C_{min} = min\{C_{h_1(y)}, ..., C_{h_k(y)}\}$
11. i= SmallestIndexOf(C_{min})
12. $X^i = X^i \cup y$

When deleting an item is desired, we first perform the deletion operation on the off-line data structure using the DeleteItem$_{BFHT}$ algorithm (lines 2-5). Then we collect all the items in all the affected buckets (buckets whose counters are decremented) of BFHT for relocation. At the same time, we delete the list of items associated with each bucket from the PFHT since each of them now must be reinserted (lines 7-8). Finally, for each item in the list of collected items, we re-insert it (lines 9-12) just as we did in InsertItem$_{PFHT}$. Notice the resemblance between the lines 6-12 of DeleteItem$_{PFHT}$ with lines 4-10 of InsertItem$_{PFHT}$. The only difference is that in DeleteItem$_{PFHT}$, we collect the items to be re-inserted from the BFHT and we decrement the counter rather than incrementing it.

Before we derive the expression for the complexity of DeleteItem algorithm, we notice that we have two types of operations involved: on the BFHT and on the PFHT. We derive the complexity for only the PFHT operations since the BFHT operations can be performed in the background without impeding the normal operations of PFHT. With this consideration, we note that the number of items per non-empty bucket in BFHT is $2nk/m$ since only half of the buckets in the optimal configuration are non-empty (see Section 4.3). Because we collect the items from k buckets, we have totally $2nk^2/m$ items to be relocated in the loop of line 9. For relocation, we need to read as well as write each item. Hence the overall expected complexity of the deletion operation is $4nk^2/m$. With the optimal configuration of the fast hash table it boils down to $4k \ln 2 \approx 2.8k$.

4.2.3 PFHT List-balancing Heuristic

After the pruning procedure, more than one item can still reside in one bucket. We show a heuristic list-balancing scheme to further balance the bucket load by manipulating the counters and a few items. The reason that a bucket contains more than one items is because this bucket is the first least loaded bucket indicated by the counter values for the stored items in this bucket. Based on this observation, if we artificially increment this counter, all the involved items will be forced to reconsider their destination buckets to maintain the correctness of the algorithm. There is a chance that by relocating these items, each of them can be put into an actually empty bucket. The feasibility is based on two facts: First, analysis and simulations show that for an optimal configuration of Bloom filter, there are very few collisions and even fewer collisions involving more than two items. Each items has k possible destination buckets and in most cases the colliding bucket is the only one they share. The sparsity of the table provides a good opportunity to resolve the collisions by simply giving those items a second choice. Second, this process does not affect any other items, we need to only pay attention to the involved items in the colliding buckets.

However, incrementing the counter and relocating the items may potentially create other collisions. So we need to be careful when using this heuristic. Before we increment a counter, we first test the consequence. We perform this optimization

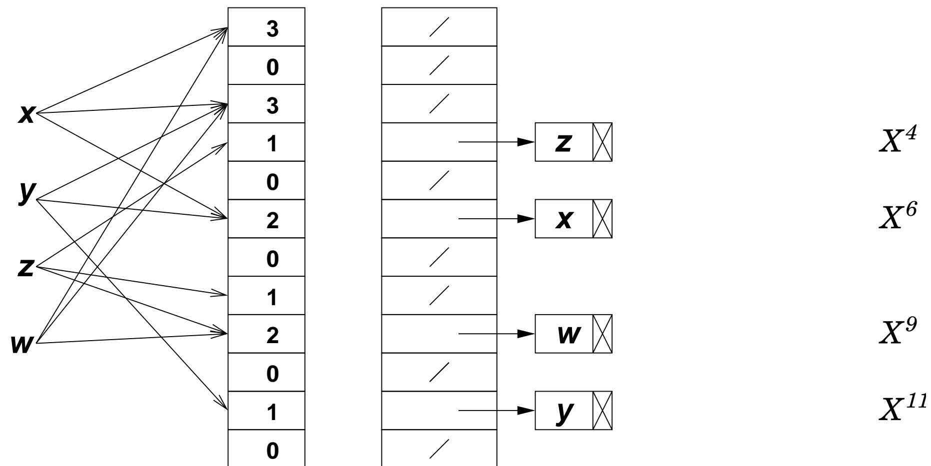

Figure 4.4: PFHT List-balancing

only if it does not result in any other collision. The algorithm scans the colliding buckets for several rounds and terminates if no more progress can be made or the involved counters are saturated. We will show that this heuristic is quite effective and in our simulations all collisions are resolved so that each non-empty bucket contains exactly one item. Figure 4.4 illustrates the list balancing optimization. By simply incrementing the counter in bucket X^1 and relocating the involved items x and w, we resolve the collision and now the lookups to this table always give the best possible performance.

4.2.4 Shared-node Fast Hash Table (SFHT)

In the previous section, we see that in order to perform incremental updates, we need an off-line BFHT. With the assumption that the updates are relatively infrequent compared to the query procedure, we can afford to maintain such a data structure in the control software which performs updates on the internal data structure (which is slow) and later update the pruned data structure accordingly. However, some applications involve time critical updates which must be performed as quickly as possible. An example is the TCP/IP connection context table where connections are set up and broken frequently and the time for hash table query is comparable to the time for addition/deletion operations of connection records [25].

We present an alternative scheme which allows easy incremental updates at the cost of a little more memory than that required for PFHT but significantly less than that of BFHT. The basic idea is to allow the multiple instances of the items to share the same item node using pointers. We name the resulting fast hash table Shared-node Fast Hash Table (SFHT). The lookup performance of SFHT is the same as that of BFHT but slightly worse than PFHT. Moreover, with the reduced memory requirement, this data structure can be kept on-line. The new algorithm is illustrated in Figure 4.5.

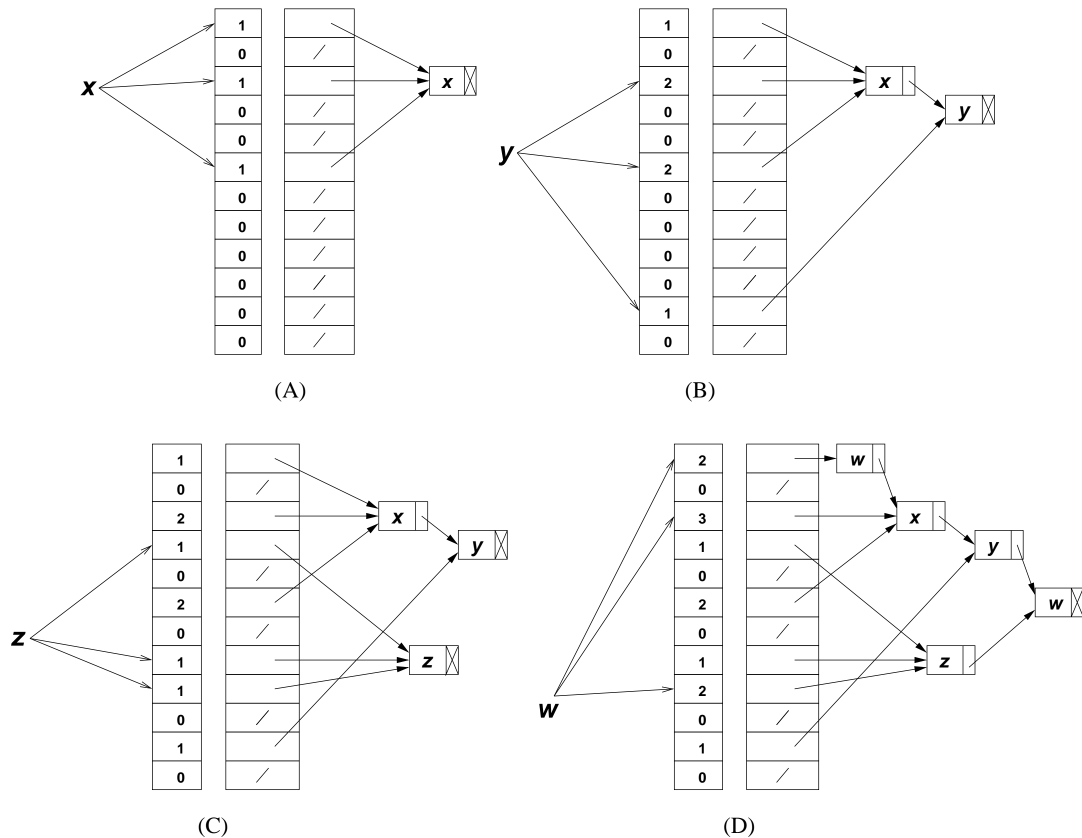

Figure 4.5: Shared-node Fast Hash Table (SFHT)

We start with an empty table and insert items one by one. When the first item, x, is inserted we just create one node for the item. Instead of inserting a copy in each of the lists corresponding to the k hashed buckets, we simply make the buckets point to the item. This clearly results in a great deal of memory savings. When we insert the next item y, we create the node and make the empty buckets point to the item directly. However, two of the three buckets already have a pointer pointing to the earlier item, x. Hence we make the item x point to the item y using the next pointer.

Note that the counters are incremented at each insertion. More importantly, the counter values may not reflect the physical length of the linked list associated with a bucket. For example, the first bucket has a value of one but there are two items, x and y in the linked list associated with this bucket. Nevertheless, it is guaranteed that we will find a given item in a bucket associated with that item by inspecting the number of items equal to the associated counter value. For example, when we wish to locate x in the first bucket, it is guaranteed that we need to inspect only one item in the list although there are two items. The reason that we have more items in the linked list than indicated by the counter value is because multiple linked lists can get merged as in the case of x and y.

The insertion of the item z is straightforward. However, an interesting situation occurs when we insert the item w. Notice that w is inserted in bucket 1, 3, and 9. We create a node for w, append it to the linked lists corresponding to the buckets and increment the counters. For the 3rd and 9th bucket, w can be located exactly within the number of items indicated by the corresponding counter value. However, for the first bucket this is not true: while the counter indicates two items, we need to inspect three in order to locate w. This inconsistency will go away if instead of appending the item, we *prepend* it to the list. Therefore, if we want to insert w in the first bucket and we find that the number of items in the list is 2 but the counter value is 1, we prepend w to the list. This will require replication of the node. Once prepended, the consistency is maintained. Both items logically stored in the first bucket list can be located by inspecting at most two items as indicated by the counter value.

The following pseudo-code describes the insertion algorithm for SFHT.

InsertItem$_{SFHT}(x)$

1. for $(i = 1 \text{ to } k)$
2. if $(h_i(x) \neq h_j(x)\ \forall j < i)$
3. if $(C_{h_i(x)} == 0)$
4. Append$(x, X^{h_i(x)})$
5. else
6. $l \leftarrow 0$
7. while $(l \neq C_{h_i(x)})$
8. $l{+}{+}$

9. read $X_l^{h_i(x)}$
10. if $(X_{l+1}^{h_i(x)} \neq NULL)$
11. Prepend$(x, X^{h_i(x)})$
12. else
13. Append$(x, X^{h_i(x)})$
14. $C_{h_i(x)}$++

In the pseudo-code, l is used as a counter to track the number of items searched in the list. We search up to $C_{h_i(x)}$ items in the list. If the list does not end after $C_{h_i(x)}$ items (line 10) then we prepend the new item to the list otherwise we append it. Note that prepending and appending simply involves scanning the list for at the most $C_{h_i(x)}$ items. Hence the cost of insertion depends on the counter value and not on the actual linked list length. In SFHT, we have nk items stored in m buckets giving us an average counter value nk/m. We walk through nk/m items of each of the k lists and finally append or prepend the new item. Hence the complexity of the insertion is of the order $O(nk^2/m + k)$. For an optimal counting Bloom filter configuration where $k = mln2/n$ (see Section 4.3), the memory accesses for deletion are proportional to k.

The item node replication causes the memory requirement to be slightly more than what we need in NHT or PFHT where each item is stored just once. The simulation results presented in Section 4.3.4 show that the memory consumption is typically just one to three times that of NHT (or PFHT). This is significantly smaller than that of BFHT.

The pseudo-code for deletion on SFHT is as shown below. We delete an item from all the lists by tracing each list. However, since the same item node is shared among multiple lists, after deleting a copy we might not find that item again by tracing another list which was sharing it. In any case we do not need to trace a list to the end but only need to consider the number of items equal to the counter value. If the list is actually shorter than the counter value, we simply start with the next list (line 4). The deletion operation has similar cost as the insertion operation.

DeleteItem$_{SFHT}(x)$

1. for $(i = 1 \text{ to } k)$
2. if $(h_i(x) \neq h_j(x)\ \forall j < i)$
3. $l \leftarrow 1$
4. while $(l \neq C_{h_i(x)}$ AND $X_l^{h_i(x)} \neq NULL)$
5. if $(X_l^{h_i(x)} == x)$
6. $X^{h_i(x)} = X^{h_i(x)} - x$
7. break
8. $l{+}{+}$
9. $C_{h_i(x)} - -$

4.2.5 Memory Compression

Our last optimization is aimed at reducing the memory usage for counters and initial bucket pointers. After all member items are programmed, if the minimum linked list length for each member item x is $C_{min}(x)$, a value B is maintained along with the fast hash table, where

$$B = max\{C_{min}(x), \forall\ x \in X\}$$

Based on B's value, each Bloom filter bucket needs only b bits for a saturated counter where

$$b = \lfloor log(B + 1) \rfloor + 1$$

As our analysis will show, using the optimal configuration of a Bloom filter, the counter value rarely exceeds 7. In practice, B is almost always less than 2. So three bits are enough to represent the counter and differentiate useful scenarios: "000" means no item is hashed to the bucket; "001" to "110" means one to six items are hashed to the bucket; and at last, "111" means seven or more items are hashed to the bucket. Thus, the counter value does not necessarily reflect the actual number of items hashed to a bucket.

The value B also provides us another subtle improvement to the lookup performance: whenever the Bloom filter reports a match but the minimum counter value is greater

than B, we know immediately that this is a false positive so no extra memory access is needed. This subtle arrangement allows us to avoid a pathological memory access pattern in which the Bloom filter shows a false positive but the smallest counter value is greater than the threshold B. In the absence of this arrangement, we need to traverse the entire list with length greater than B just to figure out that the item is not in the table. This is actually a great improvement for the worst-case scenario, since we avoid some of the unnecessary memory accesses which are also the most expensive.

It can be seen from Figure 4.3 that out of the 12 buckets only four are occupied. In reality, it can be even sparser. We now describe a technique to exploit this sparsity and reduce the memory requirement for maintaining buckets. Assume we have m buckets of which only L are occupied. In order to compress this array, we first maintain a bitmap l in which each bit l_i set to '1' indicates that there is a list of items associated with the corresponding bucket. Then we divide this bitmap into multiple segments of g bits. Furthermore, we store the starting items of all the lists within the same segment in consecutive slots in the off-chip memory. For each segment, we now keep just one pointer pointing to the first list in that slot. This is illustrated in Figure 4.6(A). As the figure shows, the bitmap is divided into three segments each containing four bits. The pointer associated with the first segment points to the first list in this segment which is the only list and contains only z. The third segment has two lists. Their starting items, w and y, are kept in the consecutive slots in the off-chip memory. Now we number all the lists within each segment, starting from zero. The list $\{z\}$ is numbered 0 in the first segment, $\{x\}$ is numbered 0 in the second segment, and $\{w\}$ is numbered 0 and $\{y\}$ is numbered 1 in the third segment. The offset or the number of the list can be calculated by counting the number of '1's up to but not including the bit corresponding to the list in the segment bitmap. Now it is easy to see that within a segment, if we want to access a particular list, we can first read the base pointer to the set of lists of that segment and then add the number of the list of interest. The memory layout of the scheme is shown in Figure 4.6(B).

With the optimization described above, the memory used by the initial pointers is for the bitmap and the segment base pointers. For m buckets, we need $\lceil m/g \rceil$ pointers each with $\lceil \log n \rceil$ bits. The segment size depends on how quickly we can compute the number of '1's within a segment. It is conceivable to construct a circuit that can

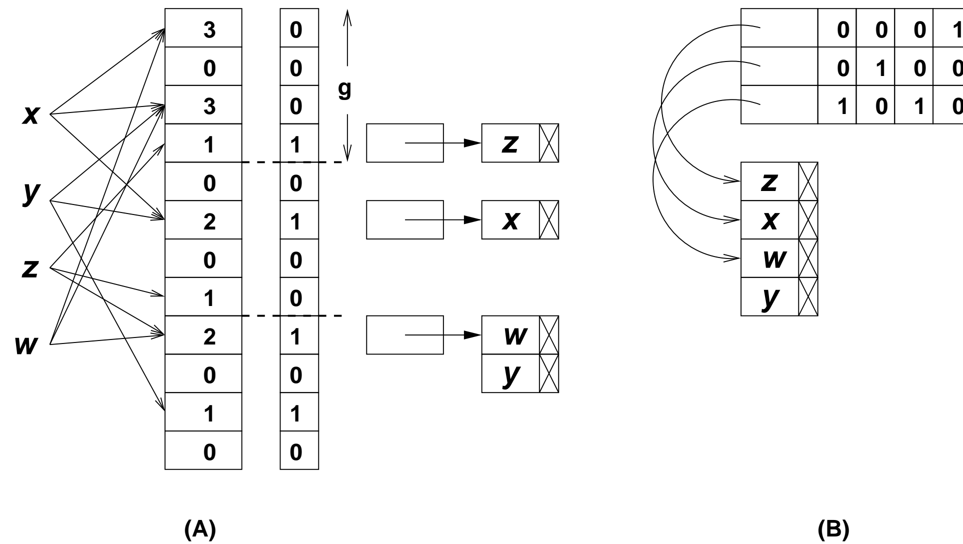

Figure 4.6: Pointer Array Compression

compute the number of bits set to '1' up to a given index, in a single clock cycle, provided the segment size is small (say 64 bits). However, for a large segment size the delay becomes large. On the other hand, pipelining this circuit will increase the latency of the overall operation. More importantly, the computation requires reading the bitmap of the entire segment. Hence, a larger segment requires more memory bandwidth which will ultimately impose a limit on the usable segment size. It should be noted that the use of such array compression scheme is not new. Similar techniques have been used previously in space efficient IP Lookups like [21, 26, 58] and string matching for network intrusion detection [71].

4.3 Analysis

We analyze and compare the FHT algorithm with NHT in terms of the expected lookup time and the lookup time tail probability to demonstrate the merit of our algorithm. We assume that NHT and all versions of FHT have the same number of buckets, m. As we have seen, given same number of items, PFHT should consume exactly the same amount of off-chip memory as NHT. SFHT consumes slightly more memory due to the item replication. Therefore, the only extra cost of our algorithm is the use of on-chip memory to store the bucket counters.

The lookup performance of PFHT is difficult to analyze. Hence we analyze only the performance of BFHT. It is important to note that PFHT will always have fewer items in each bucket than BFHT. Hence the lookup time on PFHT will always be shorter than that of BFHT or equal. We also note that SFHT has the same lookup performance as BFHT.

4.3.1 Expected Linked List Length

Consider the lookups in an NHT when the linked list is not empty. Let Y be the length of the searched bucket. We have:

$$\Pr\{Y = j | Y > 0\} = \frac{\Pr\{Y = j, Y > 0\}}{\Pr\{Y > 0\}} = \frac{\binom{n}{j}(1/m)^j(1-1/m)^{n-j}}{1-(1-1/m)^n} \tag{4.1}$$

Now we analyze the distribution of the linked list lengths of FHT. Recall that in order to store n items in the table, the number of actual insertions being performed are nk (or slightly less than that if same item could be hashed into same bucket by different hash functions), each of which is independent of each other. Under the assumption of simple uniform hashing, we can derive the average length of the list in any bucket.

With nk insertions in total, the probability that a bucket received exactly i insertions can be expressed as:

$$f_i = \binom{nk}{i}\left(\frac{1}{m}\right)^i\left(1-\frac{1}{m}\right)^{(nk-i)} \tag{4.2}$$

The question we try to answer is: when the Bloom filter reports a match for a given query (i.e. all the k' counters > 0, where k' is the number of unique buckets for an item calculated by k hash functions. We know that $1 \leq k' \leq k$), what is the probability that the smallest value of the counter is j?

Let X denote the value of the smallest counter value among k' counter values corresponding to a query item *when all the counters are non-zero.* Hence,

$$\Pr\{X = s\} = \sum_{j=1}^{k} \Pr\{k' = j\} \times \Pr\{X = s | k' = j\} \tag{4.3}$$

Let $d(j, r)$ be the probability that the first r hashes of an item produce exactly j distinct values. To derive $d(j, r)$, we know that if the first $r - 1$ hashes of an item have already produced j distinct values, the rth hash has to produce one of the j values with probability j/m. Otherwise, the first $r - 1$ hashes of an item must have already produced $j - 1$ distinct values and the rth hash should produce a value which is different from the $j - 1$ values. The probability of this is $(m - (j - 1))/m$. Hence,

$$d(j, r) = \frac{j}{m} d(j, r - 1) + \frac{m - j + 1}{m} d(j - 1, r - 1) \tag{4.4}$$

with the boundary conditions $d(j > r, r) = 0$, $d(0, 0) = 1$, $d(0, r > 0) = 0$. So, based on the fact that $Pr(k' = j) = d(j, k)$, now we can write

$$\Pr\{X = s\} = \sum_{j=1}^{k} d(j, k) \times \Pr\{X = s | k' = j\} \tag{4.5}$$

Now, let

$q(r, s, j) = \Pr\{$smallest counter value in any r of the j buckets is $s\}$

$p(i, j) = \Pr\{$a counter value in a set of j non-empty buckets is $i\}$

Since there is at least one item in any non-empty bucket, j non-empty buckets contain at least j items. We consider the probability to allocate the $i-1$ out of the remaining $nk - j$ items in one of the j buckets to make the bucket has exactly i items. Thus,

$$p(i, j) = \binom{nk - j}{i - 1} (1/m)^{(i-1)} (1 - 1/m)^{((nk-j)-(i-1))} \tag{4.6}$$

With these definitions, we can write

$$q(r, s, j) = \sum_{i=1}^{r} \begin{pmatrix} r \\ i \end{pmatrix} p(s, j)^i \times \left(1 - \sum_{h=1}^{s} p(h, j)\right)^{r-i} \tag{4.7}$$

This is because in r buckets we can have i buckets ($1 \leq i \leq r$) with counter value s while all other $r - i$ buckets have counter values greater than s. $Q(r, s, j)$ is simply the sum of the probabilities for each choice. The boundary conditions are $q(1, s, j) = p(s, j)$.

Putting things together we get:

$$\Pr\{X = s\} = \sum_{j=1}^{k} d(j, k) \times q(j, s, j) \tag{4.8}$$

Based on Eq. 4.8, Figure 4.7 shows the linked list length comparisons of FHT and NHT when $n = 10,000$, $m = 128K$, and $k = 10$. The figure tells us once we do need to search a non-empty linked list, what is the length distribution of these linked lists. In the next section we use simulations to show how the pruning and balancing optimizations improve the performance.

It is shown that given a probability of the inspected linked list length being within a bound, the bound on NHT is always larger than the bound on FHT. For instance, with a probability of 10^{-3}, NHT have about 3 items in a list where as FHT have only 2, thus improving the performance of NHT by a factor of 1.5. The improvement of the bound keeps getting better for smaller probabilities.

For NHT, the expected number of buckets that the attached linked list exceeds a given length j, $E_{>j}$, is expressed as:

$$E_{>j} = m \times B(n, 1/m, > j) \tag{4.9}$$

Where $B(n, 1/m, > j)$ is the the probability that a binomial random variable (or the load of a bucket) is greater than j:

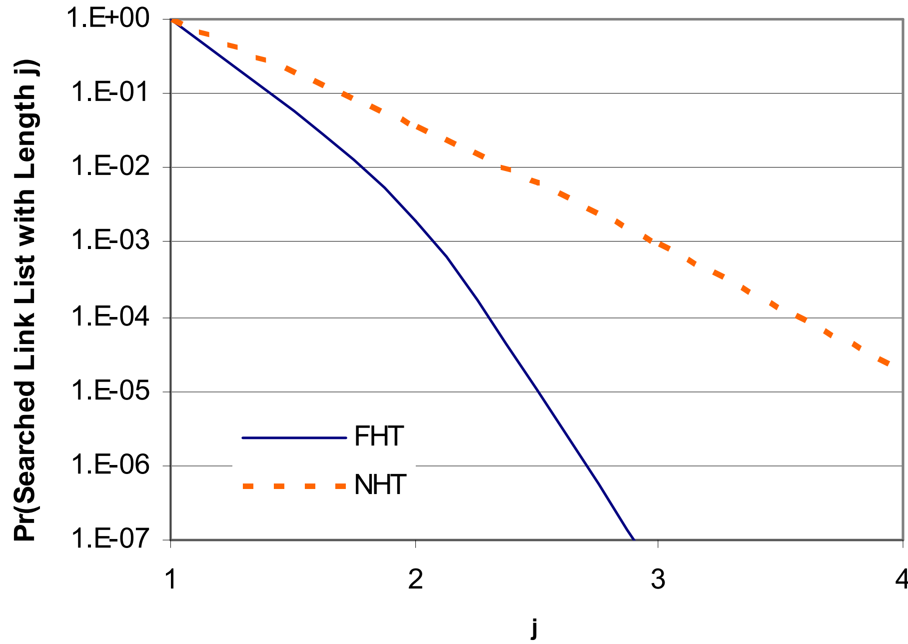

Figure 4.7: Probability Distribution of Searched Linked-list Length

$$B(n, 1/m, > j) = 1 - \sum_{i=0}^{j} \binom{n}{i} (1/m)^i (1 - 1/m)^{n-i} \tag{4.10}$$

For NHT, if the expected number of buckets with linked list length j is i, we can equivalently say that the expected number of items for which the bucket linked list lengths are j is $i \times j$. So the expected number of items for which the bucket linked list length $> j$ for an NHT, $E'_{>j}$, can be expressed as:

$$E'_{>j} = \sum_{i=j}^{n} (i+1)(E_{>i} - E_{>i+1}) = m \sum_{i=j}^{n} (i+1)(B(n, 1/m, > i) - B(n, 1/m, > i+1)) \tag{4.11}$$

Now we derive $E''_{>j}$, the expected number of items in an FHT for which all buckets have more than j items (before pruning and balancing), . We use an approximate expression for this:

$$E''_{>j} = n \times B((n-1)k, 1/m, > (j-1))^k \tag{4.12}$$

The idea behind this is that, if we consider a single item that hashes to k distinct buckets and a particular bucket for that item, the number of additional items that map to the same bucket is given by the binomial distribution with $(n-1)k$ trials. We can approximate the probability that all buckets for that item have $> j$ items by raising this probability to the k-th power. This is not quite precise, since the probabilities for the sizes of the different buckets are not strictly independent. However, the true probability is slightly smaller than what we get by multiplying probabilities, so this gives us a conservative estimate. On the other hand, the expression is only approximate, since it assumes that all n items are mapped by the k hash functions to k distinct buckets. It's likely that for a small number of items, this will not be true, but we show through simulations that this does not have a significant impact on the results.

Figure 4.8 shows the expected number comparisons of FHT and NHT when $n = 10,000$, $m = 128K$, and $k = 10$. This expected number tells us the number of items that are in linked lists with at least j entries.

The results show a definite advantage for FHT even before the pruning and balancing optimizations. We can interpret that there are only two items in a billion for which the smallest bucket has more than 3 entries. For NHT, there are about two items in ten thousands for which the bucket has more than five entries. Also in this configuration, only a few tens of items need more than one node access in FHT, but near 1000 items need more than one node access in NHT.

4.3.2 Effect of the Number of Hash Functions

We know that for an ordinary Bloom filter, the optimal number of hash functions k is related to the number of buckets m and the number of items n by the following relation [29]

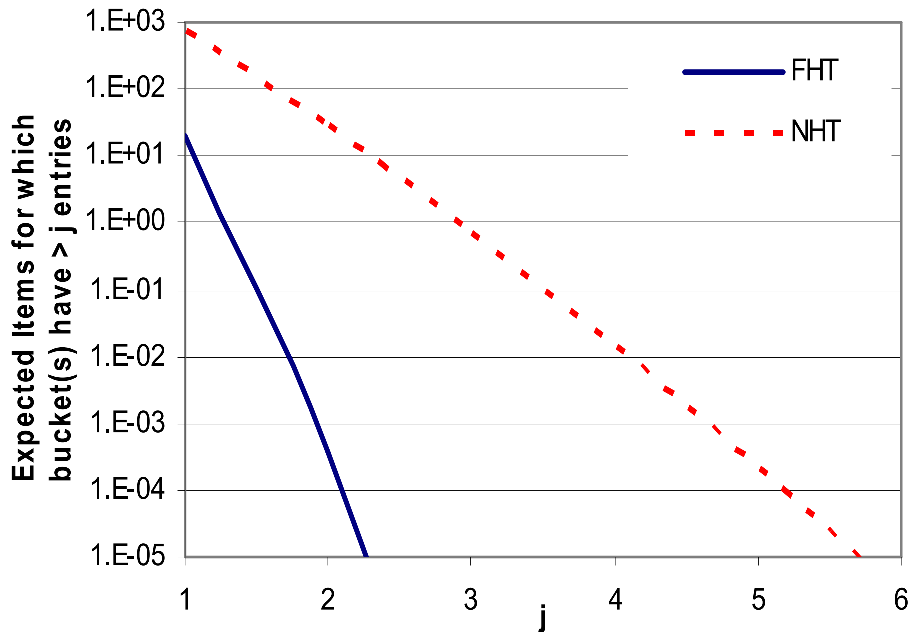

Figure 4.8: Expected Items for Which the Searched Bucket Contains $> j$ Items

$$k = \frac{m}{n} \ln 2 \tag{4.13}$$

Now we justify analytically why the same number of hash functions is also optimal for the FHT's lookup performance. From Equation 4.12, we know the expected number of items for which each bucket has more than j items. It is desirable to have at least one bucket with just one item in it. Hence we wish to minimize the probability of all the buckets corresponding to an item having more than one item. This translates into minimizing the following with respect to k.

$$B((n-1)k, 1/m, > 0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{(n-1)k}\right)^k \tag{4.14}$$

This expression is the same as the expression for the false positive probability of the ordinary Bloom filter containing $n - 1$ items in m buckets [29]. Hence, the optimal number of hash functions for the counting Bloom filters is given by

$$k = \frac{m}{n-1} \ln 2 \approx \frac{m}{n} \ln 2 \qquad (4.15)$$

for a large number of items. Therefore, the optimal configuration of the ordinary Bloom filter for minimizing the false positive probability is the same as the optimal configuration of FHT for reducing the item access time. Figure 4.9 shows the performance of FHT for different optimal configurations. In the figure, $H(i, j)$ indicates $i = k$ and $j = m/n$. When $i = 1$, it implies an NHT. For a fixed number of items n, we vary k and always ensure that m is optimally allocated for FHT. For each configuration we use the same number of resulting buckets for NHT. The performance is compared for FHT and NHT. We can make two observations from the figure. First, the performance is always better if we have more buckets per item (i.e. larger m/n). Secondly, the performance of FHT is always significantly better than that of NHT. This can be observed by comparing the curves $H(1, 3)$ and $H(2, 3)$, $H(1, 6)$ and $H(4, 6)$ and so on.

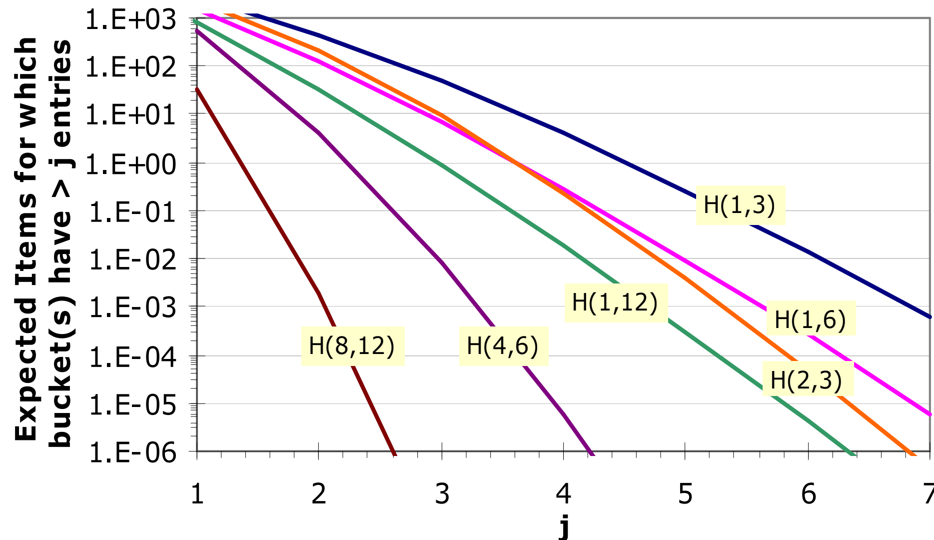

Figure 4.9: The Effect of Optimal Configuration of FHT

We also plot the performance when we use fewer hash functions than the optimal, and fix m and n. This is shown in Figure 4.10. The optimal number of hash functions for the configuration used is 10. Although the performance degrades as we use less than 10 hash functions, it is still significantly better than NHT ($k = 1$ curve). An

advantage of having a smaller number of hash functions is that the incremental update cost is reduced. Moreover, the associated hardware cost is also reduced.

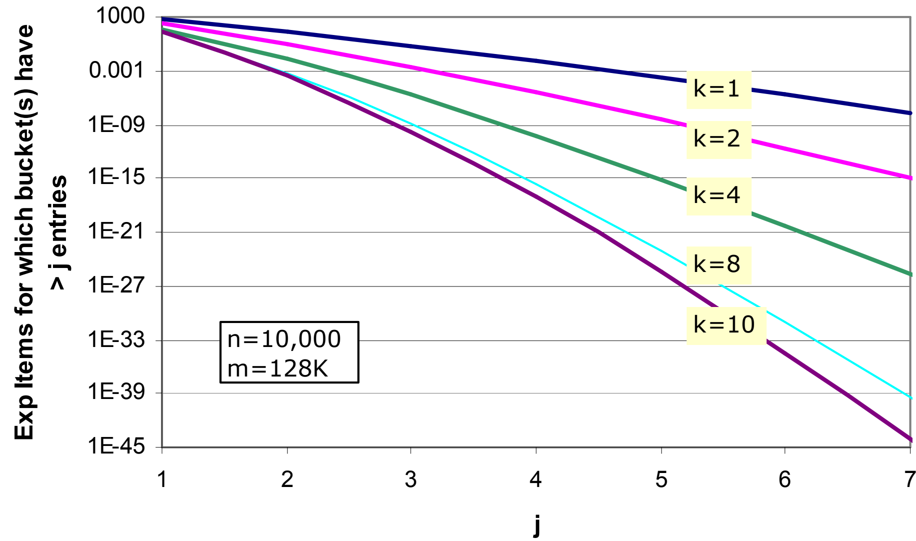

Figure 4.10: The Effect of Non-optimal Configuration of FHT

4.3.3 Average Access Time

The load factor of a hash table is defined as the average length of lists in the table [20]. For an NHT, the load factor α can be given as:

$$\alpha = n/m \tag{4.16}$$

Let T_1, T_1^s and T_1^u denote the time for an average, successful and unsuccessful search respectively (ignoring the hash computation time). For an NHT, the following can be shown [20]:

$$T_1^s = 1 + \alpha/2 - 1/2m \tag{4.17}$$

$$T_1^u = \alpha \tag{4.18}$$

In order to evaluate the average search time, we need to introduce another parameter p_s which denotes the probability of a *true positive*, i.e., the frequency of searches which are successful. Similarly, $p_u = 1 - p_s$ denotes the frequency of issuing unsuccessful searches.

With these notations, the average search time can be expressed as:

$$T_1 = p_s T_1^s + p_u T_1^u = p_s \left(1 + \frac{n-1}{2m}\right) + (1 - p_s)\frac{n}{m} \tag{4.19}$$

For FHT, let E_p be the expected length of linked list in FHT for a member item and E_f be the expected length of linked list in FHT for a false positive match. E_p can be derived form Equation (4.11) and E_f can be derived from Equation (4.8). So the average search time T_2 is:

$$T_2 = p_s E_p + p_u f E_f = p_s E_p + (1 - p_s)\left(\frac{1}{2}\right)^{(m/n)\ln 2} E_f \tag{4.20}$$

We compare our algorithm with NHT by using the same set of configurations. Figure 4.11 shows the expected search time in terms of the number of off-chip memory accesses for the two schemes under different successful search rates when $m = 128K$, $n = 10,000$, and $k = 10$.

We see that the lower the successful search rate, the better the performance of our algorithm is. Note that this estimation is conservative for our algorithm. We do not take into account the potential benefit of some optimizations such as pruning and list-balancing.

4.3.4 Memory Usage

There are three distinct blocks in the FHT architecture which consume memory. The first is the on-chip counting Bloom filter. Second is the hash table buckets and the third being the actual item memory. In the analysis so far, we have always considered the same number of buckets for both FHT and NHT. NHT does not require on-chip

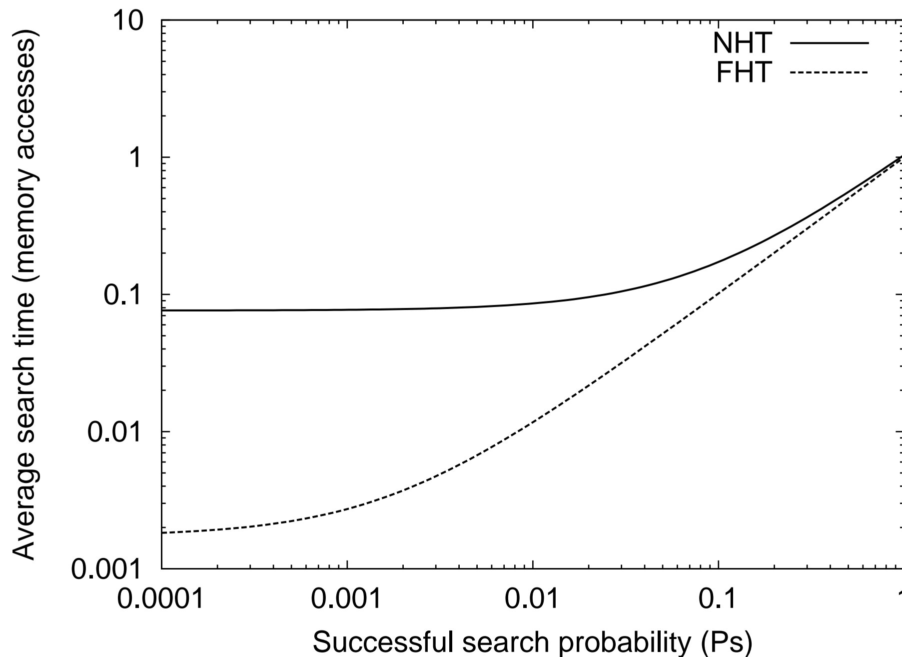

Figure 4.11: Expected Search Time

memory though FHT needs a small amount of it. Finally, while NHT needs memory for exactly n items, the different versions of FHT need different amounts of memory depending on how many times an item is replicated. BFHT needs to store each item k times and therefore needs a space of nk. PFHT keeps exactly one node for each item and therefore the storage is same as NHT. SFHT trades off memory for better incremental update support. We computed the memory requirement for SFHT using simulations with $m = 128K$, $n = 10,000$ and $k = 10$. Figure 4.12 shows the memory consumption of all the three schemes. The results show that for the chosen configuration, SFHT uses 1 to 3 times more memory than NHT or PFHT, which is much less than BFHT memory requirement.

We now elaborate on the memory usage for on-chip counters. The memory consumption for the counter array depends on the number of counters and the width of each counter. While the number of counters is decided by Equation 4.13, the counter width depends on how many items can get hashed to a counter. The worst case when all the nk items land up in the same bucket is highly improbable. We calculate how many items can get hashed in a bucket on an average and choose a counter width to support it. For any counter that overflows by chance, we make a special arrangement

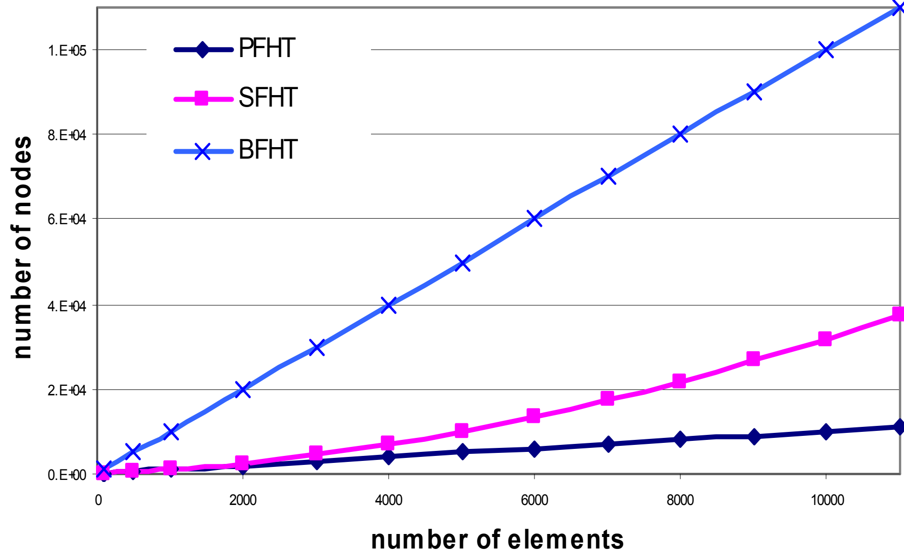

Figure 4.12: Item Memory Usage of Different Schemes

for it. We simply keep the counter on the chip and attach the index of the counter in a small Content Addressable Memory (CAM) with just a few entries. When we want to address a counter, we check to see if it is one of the "oversize" counters and access it from the special hardware. Otherwise, the normal operations proceed. Given the optimal configuration of counting Bloom filters (i.e. $m \ln 2/n = k$) and $m = 128K$, we can show that the probability of a counter being > 8 is 1.4×10^{-6}, which is negligible for our purpose. In other words, one in a million counters can overflow when we have only 128K counters. Hence, we can comfortably choose the counter width of three bits and this consumes less than 400K bits of on-chip memory.

4.4 Simulations

We simulate the FHT lookup algorithm using different configurations and compare the performance with NHT under the condition that each scheme has the same number of buckets. First, we need to choose a set of "good" hash functions. Even with a set of simple hash functions, we show that our algorithm demonstrates a compelling

lookup performance that is much better than NHT. In the optimal case, our algorithm's successful lookup time is exactly 1, and the average unsuccessful lookup time is determined by the false positive rate of the Bloom filter.

A class of universal hash functions described in [15] are suitable for hardware implementation [51]. For any member item X with b-bit representation

$$X = \langle x_1, x_2, x_3, \ldots, x_b \rangle$$

the ith hash function over X, $h_i(x)$, is calculated as:

$$h_i(X) = (d_{i1} \times x_1) \oplus (d_{i2} \times x_2) \oplus (d_{i3} \times x_3) \oplus \ldots \oplus (d_{ib} \times x_b)$$

where '$\times$' is a multiplication operator and '$\oplus$' is a bitwise *XOR* operator. d_{ij} is a predetermined random number in the range $[0 \ldots m-1]$. For the NHT simulation, one of such hash functions is used.

We simulate the tail distribution of the expected number of items in a non-empty bucket which needs to be searched. The simulation was run 1,000,000 times with different seeds. In Table 4.1, we list both the analysis results and the simulation results.

Table 4.1: Expected # of Items for Which All Buckets Have $> j$ Entries

j	*Fast Hash Table*				*Naive Hash Table*	
	Analysis	*Simulation*			*Analysis*	*Simulation*
		basic	*pruning*	*balancing*		
1	19.8	18.8	5.60×10^{-2}	0	740.32	734.45
2	3.60×10^{-4}	4.30×10^{-4}	0	0	28.10	27.66
3	2.21×10^{-10}	0	0	0	0.72	0.70
4	1.00×10^{-17}	0	0	0	1.37×10^{-2}	1.31×10^{-2}
5	5.64×10^{-26}	0	0	0	2.10×10^{-4}	1.63×10^{-4}
6	5.55×10^{-35}	0	0	0	2.29×10^{-6}	7×10^{-6}

From the table, we can see that our analysis of FHT and NHT is quite precise. The simulation results are very close to the analytical results and confirm the accuracy of our approximate analysis. More importantly, while BFHT has already demonstrated its advantages over NHT, after the pruning and list-balancing, the results are even

better: each non-empty bucket contains exactly one item. This means, in the worst case, only one off-chip memory access is needed for a lookup.

4.5 Implementation

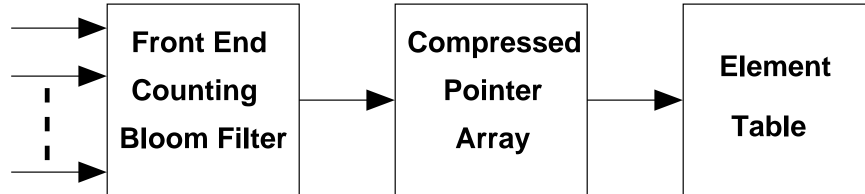

Figure 4.13: Hierarchical Structure of Fast Hash Table

We briefly discuss the implementation of our fast hash table. The memory and associate logic can be partitioned into a 3-level hierarchical structure as shown in Figure 4.13. The front end counting bloom filter supports k parallel lookup and counter comparisons. The index for the bucket of the smallest counter is fed into the compressed pointer array and the element's absolute address is calculated in this block. The address is used to retrieve the items in the associated item table.

The implementation of the counting Bloom filter takes advantage of the fast multi-port on-chip memory in FPGAs or ASICs. A modern FPGA chip can contain more than 10 Mbits SRAM in a single chip and can be configured in many different sizes and widths [79]. In the case there are only two-port SRAM available, each SRAM block can support two hash functions and the hash keys are restricted only to this memory block. Several SRAM blocks can work in parallel, each with different set of hash functions. Analysis shows this architecture can achieve equivalent performance as the strict Bloom Filter implementation.

Note that except the front-end counting Bloom filter, other memory blocks are all single port memory and are not necessarily restricted to be on-chip or off-chip. The decision is totally an engineering consideration and depends on the design criteria and memory availability. Using this memory hierarchy, lookups can be pipelined to further improve the lookup performance. This regular and reconfigurable architecture

with tunable parameter settings can be implemented as a hardware core and used in any occasion where fast hash table lookups are crucial to the system performance.

4.6 Conclusion

Hash tables are extensively used in several packet processing applications such as IP route lookup, packet classification, per-flow state management, and network traffic monitoring. Since these applications are often used as components in the data-path of a high-speed router, they can potentially create a performance bottleneck if the underlying hash table is poorly designed. In the worst case, back-to-back packets can access an item in the most loaded bucket of the hash table leading to several sequential memory accesses, which in turn will deplete the system buffer and cause packet drops.

Among the conventional avenues to improve hash table performance, using sophisticated cryptographic hash functions such as MD5 does not help, because they are too computationally intensive to be computed in a minimum packet-time budget and the performance is still not good enough; Devising a perfect hash function by preprocessing keys does not work for dynamic data sets and real-time processing; Multiple-hashing techniques to reduce collisions demand multiple parallel memory banks (requiring more pins, memory bandwidth, and power). Hence, engineering a resource efficient and high-performance hash table is indeed a challenging task.

In this chapter, we present a novel hash table data structure and algorithm which outperforms conventional hash table algorithms by providing better bounds on the hash collisions and the memory accesses per lookup. Our hash table algorithm extends the multi-hashing technique, the Bloom filter, to support the exact match. Unlike the conventional multi-hashing schemes, it requires only one external memory for lookups. Combined with the current advances in embedded fast memory technology, FHT offers a promising approach to hardware-based hash table design to meet network throughput demands by providing faster and more predictable lookup performance.

Chapter 5

LPM using Hash Tables and Tries

5.1 Introduction

We show in Chapter 4 that we can take advantage of on-chip memory to improve hash table lookup performance. Now we consider applying this technique to further improve the LPM performance.

Using hash tables for LPM is not new. Dharmapurikar et. al. have presented a scheme to assign each unique prefix length a Bloom filter [23]. The queries to the Bloom filters are performed in parallel, and then the search for the longest matching prefix in an off-chip hash table starts from the longest length for which the corresponding Bloom filter reports a positive match. In case no false positive is present, only one hash table query is needed to retrieve the best matching prefix.

However, LPM using Bloom filters has some disadvantages for IP lookups. Each distinct prefix length requires a Bloom filter, so the total number of Bloom filters might be too large. Although the total number of items programmed in these Bloom filters is simply the number of prefixes in a table, the item distribution among these Bloom filters is highly skewed. This makes engineering the system to best use the on-chip memory resource a challenging problem. More important, a large number of Bloom filters result in poor worst-case performance. In the worst case, if all the Bloom filters show a false positive, we need as many hash table queries as the number of Bloom filters for a packet lookup. Therefore, reducing the number of Bloom filters not only lowers the system complexity and but also improves the worst-case performance. To reduce the number of Bloom filters, the algorithm in [23] selects a few thresholds

based on the prefix length distribution and expands the prefixes to their nearest thresholds. Now we need only one Bloom filter for each threshold. However, the prefix expansion increases the total number of prefixes in a route table a great deal. The expanded prefix table increase the number of items that must be stored in both the on-chip memory and the off-chip hash table.

In addition to increasing the memory required, prefix expansion also significantly increases the incremental update cost. One single update might need a large number of memory operations on both the Bloom filter and the associated hash table. In an environment where the route table changes frequently, the update cost can become prohibitively large.

On the other hand, we have mentioned in Chapter 3 that most successful IP lookup algorithms are essentially variations of the basic binary trie that allow for examining multiple bits per memory access. Smart encoding techniques such as Tree Bitmap [27] and Shape Shifting Tries [58] avoid the prefix expansion, improving storage efficiency and providing faster lookup throughput. However, searching in a trie always starts from the root, so the worst-case performance of these algorithms is proportional to the maximum trie depth [27] and is sensitive to the underlying trie structure [58].

Combining the hash table and trie data structures leads to a new LPM algorithm which is presented in this chapter. It retains the memory efficiency of the trie-based algorithm and meanwhile allows the search to bypass intermediate trie nodes with the assistance of hash tables. The algorithm can be used in a high-performance IP lookup engine, especially for IPv6. It is suitable for hardware implementation and can sustain OC-192 and above line-speed processing by using only one commodity memory chip. The algorithm exhibits a nice tradeoff between throughput and storage, which allows system designers to decide the configurations based on the available on-chip and off-chip memory resource and the desired lookup throughput. The algorithm can also be used as a building block of packet classification algorithms.

The remainder of this chapter is organized as follows. The related work is discussed in Section 5.2, the algorithm and its implementation are described in Sections 5.3 and 5.4. We evaluate the algorithm in Section 5.5 and conclude the chapter in Section 5.6.

5.2 Related Work

Some LPM algorithms take advantage of the trie data structure to support a pipelined architecture [34]. Ideally, pipelined lookups allow the completion of one packet lookup per clock cycle. Unfortunately, this technique has serious problems. It consumes too much memory bandwidth and the skewed storage requirement of the pipeline stages makes engineering the system difficult and inefficient. Another technique interleaves memory accesses from multiple parallel IP lookup engines [27, 58]. When these lookup engines share the same memory interface, they try to fully utilize the available memory bandwidth to gain a high aggregated lookup throughput. The bandwidth of a single SRAM chip today can be higher than 14 Gbps. Current VLSI technology makes it easy and low-cost to deploy multiple engines and synchronize their behavior. So the core problem here is to lower the bandwidth share of each engine. In other words, we should focus on reducing the number of off-chip memory accesses needed for a single packet lookup in order to achieve a higher overall lookup throughput.

The central piece of our LPM algorithm is a set of on-chip Bloom filters. As discussed in Chapter 4, Bloom filters have drawn significant attention in the networking research community recently due to their efficient use of memory. Reference [23] discusses using Bloom filters for IP lookups. Our work is built upon this algorithm and significantly improves it.

Sangireddy et. al. present an Elevator-Stair algorithm that combines hash tables and PATRICIA trees [53]. Hash tables are built on selected levels to indicate if there are longer prefixes starting from these levels. However, as the name of algorithm implies, the LPM starts from the tree root, searching the hash tables level by level to determine where to find the potential longest matching prefix. While this is akin to our algorithm, our algorithm supports directly jumping to the destination hash table, resulting in a faster search speed. Their algorithm uses the PARTRICIA tree for the second layer search. However, the PARTRICIA tree can only compress tree paths without any branch. Hence it is not as effective as other encoded multibit trie algorithms. Moreover, the algorithm does not use Bloom Filters to summarize the items in hash tables, so the algorithm has to physically access many of the off-chip hash tables.

5.3 Algorithm

The easiest way to organize data for IP lookup is to group the prefixes based on their lengths and store each group in a hash table. When lookups are performed in software, the binary search on these hash tables based on the prefix lengths is the best choice [74], resulting in the $O(\log W)$ lookup time performance, where W is the number of unique prefix lengths. When lookups are performed in hardware, however, we can take advantage of the embedded fast memory and the parallel processing capability of hardware to use the brute force method. As proposed in [23], an on-chip Bloom filter is used to summarize the items in each hash table. The lookup process probes all the Bloom filters simultaneously and uses the output of the Bloom filters to determine which hash table to query. In practice, the lookups can be very fast. Unfortunately, due to the possibility of false positive in Bloom filters, the worst-case lookup time performance is as poor as $O(W)$. When using the prefix expansion technique [64] to reduce W, i.e. the number of Bloom filters and hash tables, significantly more storage is required.

The high level idea of our algorithm is simple: with the reduced number of Bloom filters, instead of performing the prefix expansion, we encode the subtree between two length thresholds using the TBM or SST encoding technique. In a sense our new algorithm can be seen as a multi-bit trie algorithm with multilevel jump tables.

For example, assume we have a prefix table shown in Table 5.1. If we assign each unique length a Bloom filter, we need at least five Bloom filters. Future updates can drastically change the situation so more Bloom filters are expected. Here we get a sense of the difficulty of engineering such a system. Now we assume the table is just as it is. Six items are programmed in the Bloom filters and in the worst case we need five hash table queries to find the best matching prefix when all the Bloom filters show a false positive.

Now we want to use the prefix expansion technique to reduce the number of Bloom filters to two. By carefully analyzing the prefix length distribution, we decide to set the two length thresholds to 4 and 7. The expanded table is shown in Table 5.2. The table size is doubled and 15 items need to be programmed in the two Bloom filters. Although the worst-case number of hash table queries is reduced to only 2,

Table 5.1: Prefix Table

ID	Prefix
p0	*
p1	1*
p2	000*
p3	101*
p4	1000*
p5	10010*
p6	1001101*

the storage required is significantly increased. Moreover, if now we need to remove the prefix $p1$, we need to remove five items from the on-chip Bloom filters and the off-chip hash tables. This is a high update cost.

Table 5.2: Expanded Prefix Table

ID	Prefix
p0	*
p2	0000*
p2	0001*
p4	1000*
p1	1001*
p3	1010*
p3	1011*
p1	1100*
p1	1101*
p1	1110*
p1	1111*
p5	1001000*
p5	1001001*
p5	1001010*
p5	1001011*
p6	1001101*

Rather than expanding the prefix table, our algorithm seeks to encode the prefixes between the length thresholds using the trie data structure. As shown in Figure 5.1, the binary trie nodes are grouped into subtrees and the subtrees are encoded using either TBM or SST (Note that if the subtrees have only one layer of encoded nodes, EBM and the child pointer are not required in the node data structure). Now in the

first Bloom filter, we need to program only two items “10010” and “10011”, and in the second Bloom filter, we need to program only one item “1001101”. The root nodes of subtrees, associated with the items in the Bloom filters, are stored in the off-chip hash tables. In addition, the best matching prefix so far for each item is also stored along with the item in hash tables. For example, in the hash table entry associated with the item “10010”, the best matching prefix so far is itself, “10010*”. However, in the hash table entry associated with the item “10011*”, the best matching prefix so far is $p1$ or “1*”. Now there are only three items in the Bloom filters. Compared with the previous scheme, it is a huge saving.

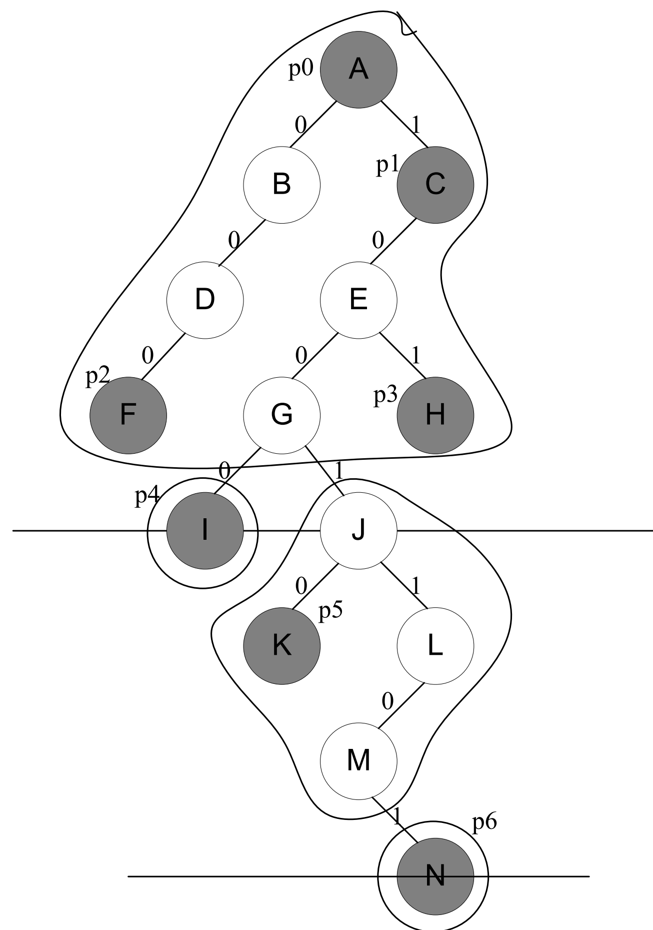

Figure 5.1: LPM Data Structure for the Example Table

There are some subtle points about this data structure. First, the items programmed in the Bloom filters may not be the prefixes in the Table. Rather, they are the prefixes of the paths that cross the length thresholds. See the path “1001101” in Figure 5.1

for an example. Second, if a long path cross multiple thresholds, then the prefixes of this path with different threshold lengths are programmed in multiple Bloom filters. We created a sort of dependency among the Bloom filters. This feature can help filter out certain false positive pattern. A match in a Bloom filter for a longer threshold can be a true match only if all the Bloom filters for the shorter thresholds show matches too. In other words, if any Bloom filter shows a mismatch, we know that the matches in Bloom filters for longer thresholds are definitely false positive, so we do not need to query their associated off-chip hash tables.

With this data structure, the LPM lookups are quite simple. Give an IP address, we extract the prefixes according to the length thresholds of the Bloom filters and then use these prefixes to query the Bloom filters in parallel. We then determine which hash table to query based on the Bloom filter outputs. If a Bloom filter and all the Bloom filters for shorter length thresholds report a match, we then query the hash table for this Bloom filter to verify the match. If it turns out to be a true match, then the best matching prefix is either the one stored in the hash table entry or a longer prefix in the subtree. So we traverse the subtree to search for a longer prefix. The best matching prefix is returned according to the search result. If the query to a hash table shows that a match in a Bloom filter is a false positive, then we go ahead to query the hash table for the Bloom filter with a shorter length threshold.

This architecture suggests that the worst-case lookup performance is determined by the number of Bloom filters and the cost to traverse a subtree. In the example shown in Figure 5.1, we can read a subtree in just one memory access, so in the worst case, a packet lookup needs two hash table queries to retrieve a valid multibit trie node and one extra memory access to retrieve the next hop per lookup. For this example, the worst-case performance is identical to that of the original method with prefix expansion. However, our algorithm uses much less memory and provides better support of incremental updates.

Now we describe the algorithm formally. The data structure construction algorithm starts from the binary prefix tree. The tree is partitioned into k segments at depth $d_0, d_1, d_2, ..., d_k$, where $0 = d_0 < d_1 < d_2 < ... < d_k \leq w$. We then assign an on-chip Bloom filter B_i and an off-chip hash table H_i for each depth d_i when $i > 0$. For any path starting from the root with its length $\geq j$, there is a record in each Bloom filters

B_r if $d_r \leq j$. In the Bloom filter B_i, the d_i-bit prefix of the path is programmed. A unique path prefix is only programmed in a Bloom filter once. Since the prefixes of a path with different lengths are present in a sequence of Bloom filters, we call it *Chained Path Bloom Filters* (CPBF). All the paths ended in a segment (d_i, d_{i+1}) forms a set of subtrees for which the roots are the tree nodes at the depth d_i. We use either Tree Bitmap or Shape Shifting Tries to encode these subtrees. Each such encoded subtree is called a *Segment Multibit Trie* (SMT). The stride or the node capacity is determined by the word size of the off-chip memory. All the path prefixes programmed in a Bloom filter B_i are also stored in the hash table H_i. Along with the path prefixes, the hash table stores the corresponding SMT root node and the length of the longest matching prefix of the root node.

The IP lookup process includes two steps. First, construct the Bloom Filter keys, query the Bloom Filters, and use the outputs to determine which Hash Table to search. Second, retrieve the best match so far and the SMT root from the Hash Table, traverse the SMT, and determine the best match.

In the first step, we use the prefixes of the IP address with length $d_1, d_2, ...d_k$ as keys to query the corresponding Bloom filters in parallel. We examine the match status from B_1 to B_k. If the first negative match is reported by B_j, then the length of the longest matching prefix must be shorter than d_j, even if some Bloom filters with index greater than j report a positive match. The dependency of the CPBF is able to filter out this kind of false positive without requiring any off-chip memory access. If $j = 1$, we know the best match exists in the SMT between depth 0 and depth d_1; otherwise, we query the hash table H_{j-1} to verify the match. If it turns out the match in B_{j-1} is a false positive, we then back to query the hash table with smaller index and so on. Finally, we can find exactly the segment which contains the best match. Once we find a true match in a hash table, in the second step, we retrieve the associated SMT root and traverse the SMT to find a longer matching prefix. The longest matching prefix in this SMT is returned as the best match. If the search fails, the stored best prefix is returned. The best matching prefix can then be used as a key to retrieve the associated information, such as the next hop for IP lookups.

CPBF provides a mechanism to fast jump the search to the target segment when doing LPM. The encoded SMT efficiently uses the memory and supports fast lookups. The combination of these two forms a more scalable and faster LPM algorithm.

5.4 Implementation

When designing the LPM system, we need to determine the number of Bloom filters as well as the set of length thresholds. Our algorithm is very flexible: each segment can have a different number of bits. We can optimize the segment partition to fit different applications. When the algorithm is used for IP lookups, we can engineer the design to gain the best throughput. For example, if we use TBM to encode the SMTs and one memory word can encode a node with a stride of s, we can partition the binary tree into $\lceil W/s \rceil$ segments, where W is the longest length of the prefixes in a table. So $\lfloor W/s \rfloor s$ Bloom filters are needed to cover path lengths s, $2s$, ... , $\lfloor W/s \rfloor s$. Since an SMT is encoded using a single memory word, one to $\lfloor W/s \rfloor$ off-chip memory accesses are needed to find the best matching prefix. Here we assume each hash table query requires just one off-chip memory access, which is reasonable with the use of FHT for implementing the hash tables.

If the on-chip memory resource becomes a concern, we can reduce the number of Bloom filters by letting each segment cover αs bits. Now only $\lfloor W/\alpha s \rfloor$ Bloom filters are needed and an SMT could have a depth of α. This means one to $\lfloor W/\alpha s \rfloor + \alpha$ memory accesses are required to find the best matching prefix. Our algorithm allows a tradeoff between throughput and storage. This is especially attractive for IPv6, where W is a large number. Assume the longest prefix length is 64 and the memory word size supports TBM nodes with a stride of 5, Figure 5.2 shows the number of Bloom filters required versus the worst-case number of memory accesses required for a packet lookup when we vary the value of α. The upper curve shows the absolute worst-case performance when Bloom filters can show false positives. If we assume there is no false positive from the Bloom filters, the worst-case performance is determined by the depth of SMTs, which is shown in the lower curve in Figure 5.2.

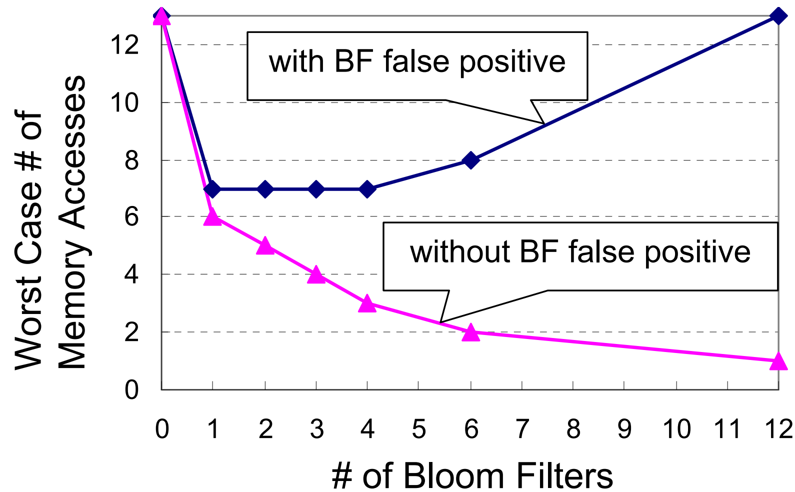

Figure 5.2: The Worst-Case Performance vs the Number of Bloom Filters

When no Bloom filter is used, the performance is simply that of the multibit trie algorithm. When just one Bloom filter is used, the performance improves almost two times. While more Bloom filters tend to worsen the absolute worst-case performance, the average-case performance and the usual performance are both improved substantially.

Recall that in Chapter 3 we take advantage of the tree sparsity to encode the subtrees using SST which can generally cover a larger stride per node. Similarly, we can expect a better performance if SST can also be used to encode SMTs. We use the following algorithm to partition the binary tree and construct the data structure dynamically.

Step 1 Traverse the binary tree and find the largest depth d_i such that every subtree rooted at depth d_i can be encoded as an SMT of depth k, for some specified k. The SMTs combine the TBM-type and SST-type nodes in order to minimize d_i. If $d_i > 0$, go to the step 2. Otherwise, the binary tree root is reached, so encode the subtree rooted at the binary tree root and halt.

Step 2 Build a Bloom filter for depth d_i. All the tree nodes at depth d_i have a record in the Bloom filter. Encode the subtrees and store them in the associated hash table. Store the best matching prefix so far for each item in the associated hash table.

Step 3 Prune the binary tree at depth d_i and repeat the previous steps on the remainder tree.

Figure 5.3 shows an example of such a tree partition when $k = 1$. Assume an SST node can encode five binary nodes and an TBM node can support a stride of 3. At the first iteration, we get the threshold at the depth 4. At the second iteration, we get the second threshold at the depth 1. The algorithm terminates at the third iteration.

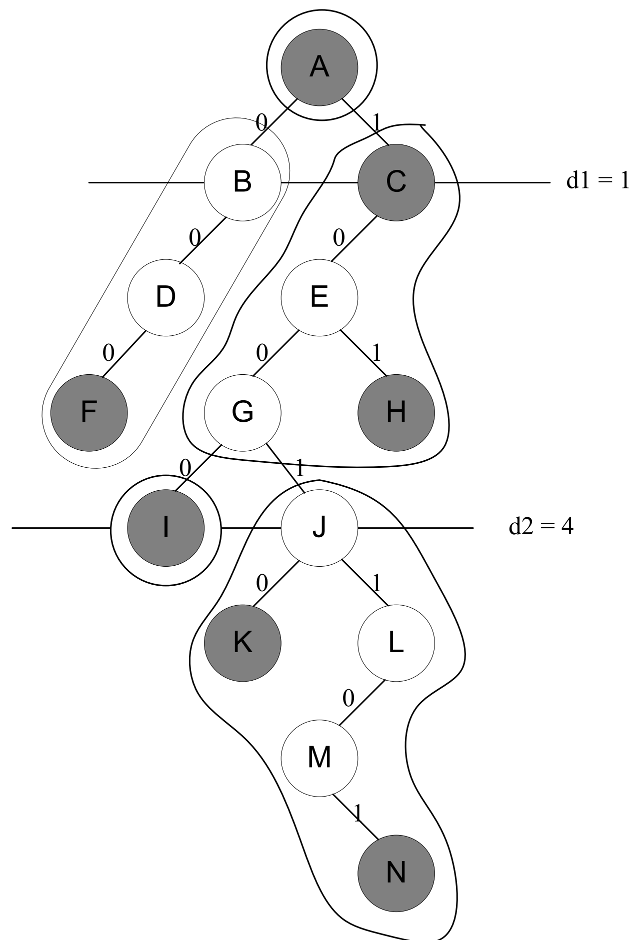

Figure 5.3: Tree Partition Example Using TBM and SST

Using this algorithm, the size of segments is not necessary to be the same, but adapts to the structure of the binary tree. Hence, we can reduce the required number of Bloom filters to the minimum. Note that we can also increase the desired SMT depth to further reduce the number of Bloom filters required at the cost of a little lower throughput.

5.5 Evaluation

Assume there is no false positive from Bloom filters. Hence, traversing one SMT is the only cost incurred to find the longest matching prefix. In the worst case when all the k Bloom filters show a false positive, the extra cost is k hash queries. In this case, the performance is the same as the worst-case performance of the multi-bit trie algorithm.

Assume the best matching prefix is stored in H_i. Clearly, B_i should report a true match. We need to access only one SMT if and only if B_{i+1} does not show a false positive, so the probability is $(1-f)$, where f is the false positive rate of a Bloom Filter. Similarly, we need to access two SMTs if and only if B_{i+1} shows a false positive and B_{i+2} does not show a false positive, so the probability is $f(1-f)$. Therefore, the average number of STMs accessed per packet lookup is:

$$(1-f) + 2f(1-f) + ... + (k-i)f^{k-i-1}(1-f) + (k-i+1)f^{k-i}$$

f is typically a very small value, so we can let $1-f=1$. Assume for the lookups, the best matching prefixes are evenly distributed in all the hash tables, then the average number of STMs accessed per packet lookup is:

$$1 + \frac{2(k-1)}{k}f + \frac{3(k-2)}{k}f^2 + ... + \frac{2(k-1)}{k}f^{k-2} + f^{k-1}$$

A QDRII SRAM has an equivalent word size of 72 bits, which is sufficient to encode a TBM node with a stride of 5 or an SST covering a 16-node binary subtree (see Chapter 3)[1]. We use the largest available BGP route table to demonstrate our algorithm's performance. There are about 200K prefixes in this table, and the lengths are distributed between 8 and 32. We evaluate our algorithm using only five Bloom

[1] When each SMT can be encoded using a single node, the node data structure does not need the EBM and the child pointer fields. Therefore, the node can support a larger stride or can cover more binary nodes. However, here we use the conservative numbers for evaluation

filters for path lengths of 8, 13, 18, 23 and 28. This partition ensures that each SMT contains a single TBM node.

Table 5.3: BGP Table Results I

Depth	# SMTs	# expanded prefixes
8	128	22
13	2,648	405
18	23,689	47,155
23	71,913	286,769
28	10,333	1,319,789
32	—	89,640
Total	108,711	1,743,780

As shown in Table 5.3, the number of SMTs defines the number of items that must be stored in both the on-chip Bloom filters and the off-chip hash tables. If the original method was used with Bloom filters for lengths 8, 13, 18, 23, 28, and 32, the number of items stored corresponds to the number of expanded prefixes, shown in the right column. Thus, our method reduces the storage required by more than an order of magnitude.

We can reduce the number of Bloom filters further at the cost of one more memory access per lookup. The results are shown in Table 5.4. By reducing the number of Bloom filters, our algorithm needs fewer and fewer items in Bloom filters. However, the prefix expansion scheme needs more and more items. By eliminating two Bloom filters, now the prefix expansion scheme generates more than 50 expanded prefixes per original prefix on average.

Table 5.4: BGP Table Results II

Depth	# SMTs	# SMT nodes	# expanded prefixes
8	128	2,776	22
18	23,689	95,602	58,240
28	10,333	10,333	10,206,672
32	—	—	89,640
Total	34,150	108,711	10,354,574

Figure 5.4 shows the worst-case number of memory accesses per packet lookup versus the number of items stored per original prefix, for our algorithm and the prefix

expansion scheme. Our algorithm provides a good worst-case performance with very low storage overhead. For the prefix expansion scheme to reach the similar worst-case performance, significantly more storage is required.

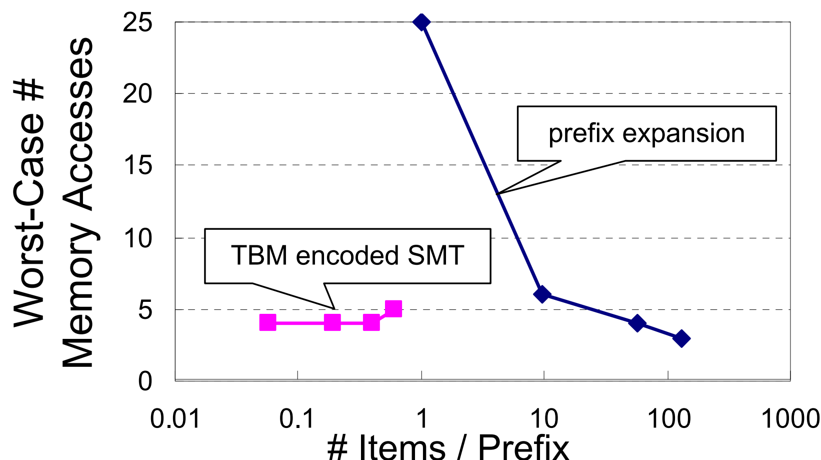

Figure 5.4: The Worst-Case Performance vs the Number of Items per Prefix

Finally, we investigate the effect of dynamically determining the segment sizes using SST and TBM together on the synthetic IPv6 table we used in Chapter 3. There are 47 unique prefix lengths distributed from 18 to 64. If the naive Bloom filter scheme is applied, up to 47 Bloom filters are required and the worst-case performance is 47 hash table queries per packet lookup. If TBM is used to encode the SMTs and the memory word supports a stride of 5, then we can reduce the number of Bloom filters to just 10 and the worst-case performance now is 10 hash table queries per packet lookup. Table 5.5 summarizes the results. Because the binary trie is very sparse, the number of SMTs exceeds the number of expanded prefixes, so we can see the storage of our scheme is actually worse than that of the prefix expansion scheme. However, if we reduce the number of Bloom filters further, the number of SMTs will decrease and the number of expanded prefixes will increase.

By using SST and TBM together, an encoded SMT node can cover either 16 binary tree nodes or support a stride of 5. Letting each SMT contain a single node, we run the dynamic algorithm to determine the segment size and reduce the number of Bloom filters to 7. Now in the worst case, we need just seven hash queries per packet lookup to find the best matching prefix. Table 5.6 summarizes the results. The table also shows when the prefix expansion is used to support these thresholds,

Table 5.5: IPv6 BGP Table Results (Using TBM only)

Depth	# SMTs	# expanded prefixes
18	31,667	—
23	86,617	142
28	126,489	1,285
33	130,347	18,819
38	128,264	35,464
43	111,077	86,807
48	85,447	204,972
53	28,730	133,787
58	24,141	34,729
63	13,460	51,464
64	—	11,840
Total	766,239	579,309

the table size is expanded to almost 100 times larger, while in our scheme, the items stored in Bloom filters are only 3.5 times more than the number of original prefixes. When a single QDRII SRAM chip is used, our algorithm can perform lookups for 200 million packets per second in the usual case and 25 million packets per second in the worst case. With a false positive rate of 0.008, our algorithm requires 12 Mb on-chip memory when FHT is used to implement the Bloom filters and the associated hash tables. On the other hand, the prefix expansion scheme requires 339 Mb on-chip memory.

Table 5.6: IPv6 BGP Table Results (Using SST and TBM)

Depth	# SMTs	# expanded prefixes
18	31,667	—
21	56,935	24
26	116,436	632
31	132,874	3,667
36	130,320	99,013
41	117,051	73,021
49	38,465	1,083,665
64	—	16,515,728
Total	623,748	17,775,750

We can see our new LPM algorithm has significant advantages over the previous algorithms for IP lookups. In the next Chapter, we will examine its use in a 2D packet classification algorithm.

5.6 Conclusion

High speed backbone routers running at OC-768 line speed need to find the routing information for up to 125 million packets per second from a database with more than 200K IPv4 or IPv6 prefixes. This challenging task demands new research efforts to provide better algorithms. System designers feel comfortable with a design only if it can work in the worst-case scenarios. Although the LPM using Bloom filters provides a compelling average-case lookup performance, its worst-case performance is poor due to the possibility of false positive in Bloom filters.

In this chapter, we present a new LPM algorithm which can be used for fast IPv4 and IPv6 lookups. Leveraging the state-of-art IP lookup algorithms and the availability of fast on-chip memory, our new algorithm not only scales well to the prefix length and the database size but also exhibits a desirable tradeoff between storage and throughput.

Chapter 6

2D Coarse-grained Tuple Space Search

6.1 Introduction

Packet classification becomes difficult when more header fields are involved and more general filters are specified. Although TCAMs can do the job easily, considering the unparalleled advantages of commodity memory such as high density, low cost, and low power consumption, we believe faster and scalable algorithms can still win the battle. Coupled with efficient data structures, hardware-based algorithms can achieve high throughput and low storage by taking advantage of parallel processing and pipelining techniques. Moreover, today's ASICs and FPGAs embed fast, multi-port on-chip memory which can be used to buffer and store critical data. In computer systems, a small amount of fast on-chip cache plays an important role in accelerating system performance. Similarly, the smart use of embedded memory can make a significant difference. The use of on-chip memory to build a small lookup cache while keeping the major data structure off-chip and taking advantage of the temporal locality of packet classification is reported in [17, 43]. However, systems relying only on the temporal locality suffer from unpredictable performance degradation due to inevitable cache misses. Alternatively, noticing that the off-chip memory access is still the system performance bottleneck, one can implement a part of the algorithm using on-chip memory, making it an integral part of the algorithm data structure so that the on-chip memory can help reduce the number of off-chip memory accesses.

The relatively small size of the on-chip memory requires us to use it efficiently. Memory-efficient data structures, such as Bloom filters, are well-suited to on-chip implementation. We have built the FHT data structure based on Bloom filters in Chapter 4 and used it for a fast LPM algorithm in Chapter 5. In this chapter, we present a new packet classification algorithm which uses hash tables as the underlying data structure. Again, we can use the Bloom filter-based FHT to optimize the on-chip memory usage and improve the system performance.

A special case of general packet classification, called 2D packet classification, takes only the source and destination IP addresses into account. In 2D filter sets, each filter specifies a pair of prefixes. 2D packet classification is widely used in basic Access Control Lists (ACL) [18]. Moreover, EGT-PC [9] shows that it can also be used in general packet classification with some extensions.

Some previous work addresses this problem. The AQT algorithm [14] applies the 2D cutting technique. Its performance highly depends on the filter set structure. The GOT [65] and EGT [9] algorithms are both trie-based, and have a worst-case lookup performance of $\alpha \times w$, where w is the longest prefix length and α is some constant factor. They traverse the prefix trees and jump between them to examine all possible matches. Another type of algorithm uses the tuple space search technique. A tuple is defined as a pair of unique prefix lengths (u, v). Filters belonging to the same tuple are stored in one hash table. The lookups are conducted by querying the hash tables. The storage and the lookup time depend on the number of tuples. The 2D tuple space search requires w^2 hash queries per lookup in the worst case. An enhanced version, called rectangle search, reduces the number to $2w - 1$, at the cost of preprocessing and more storage [63]. However, this still requires up to 63 hash queries per lookup for IPv4 in the worst case. Another algorithm [77] that operates on conflict-free filter sets uses binary search to reduce the number of hash queries to $\log^2 w$, which is 25 for IPv4. Unfortunately, most 2D filter sets are not conflict-free. Despite the differences, the performance of the above algorithms depends on the prefix length w, which makes these algorithms less attractive for the IPv6 case.

Our new 2D packet classification algorithm combines the tuple space search algorithm [63] and the crossproducting algorithm [65] and overcomes their drawbacks.

To classify a packet, it performs LPM on each field first and then combines the results to perform several hash table queries. The algorithm can be categorized as a decomposition-based algorithm using the filter set grouping technique.

The remainder of this chapter is organized as follows. Since our algorithm is directly derived from the tuple space search algorithm [63] and the crossproducting algorithm [65], we discuss them briefly and suggest optimizations in Section 6.2. We describe the algorithm in Section 6.3 and discuss the tuple partition issues in Section 6.4. We evaluate the algorithm performance using different filter sets in Section 6.5. We conclude the chapter in Section 6.6

6.2 Related Work

6.2.1 Tuple Space Search

Each filter in a 2D filter set is specified as a pair of prefixes. We define the lengths of these prefixes as a tuple, denoted as (i, j), where i is the length of the source IP address prefix and j is the length of the destination IP address prefix. Hence, filters can be grouped into different tuples. Filters in a tuple can be easily stored in a hash table with the i-bit prefix of source IP address and the j-bit prefix of destination IP address as the key. Figure 6.1 illustrates the idea, in which each grid represents a tuple. Although there are $33 \times 33 = 1089$ tuples in total, it is possible that some tuples contain no filter at all so we do not assign hash tables for these tuples. The number of nonempty tuples in some ACL filter sets are reported in Table 6.1.

Table 6.1: Number of Nonempty Tuples in ACL Filter Sets

Filter Set	# filters	# tuples
ACL1	426	31
ACL2	527	50
ACL3	1,588	89
ACL-syn	6,826	31

When each nonempty tuple is assigned a hash table, the lookup can simply query all the hash tables to find the best matching filter. However, we cannot afford to

perform so many hash queries per lookup for high performance packet classification. A simple optimization, called tuple pruning, can help reduce the number of hash tables queried per lookup. It performs single field lookups first to determine a subset of tuples for which there are matching filters. For example, assume there is a tuple (i, j) and we perform the LPM on the source IP address to return the lengths of all the matching prefixes. If none of these lengths equals i, we do not need to search the tuple (i, j). If the cost of performing LPMs can be kept low, this optimization can help improve the average-case performance. However, the worst-case performance remains the same. Another optimization, called rectangle search, aims to improve the worst-case performance. It uses the property that more specific filters have higher priority than less specific filters. For example, in Figure 6.1, if we find a matching filter in the tuple $(20, 12)$, shown as the dark grid, then we do not need to search the hash tables in the region A because they represent less specific tuples. With this optimization, the worst-case number of hash table queries is just $2w - 1$ as opposed to w^2 in the original scheme.

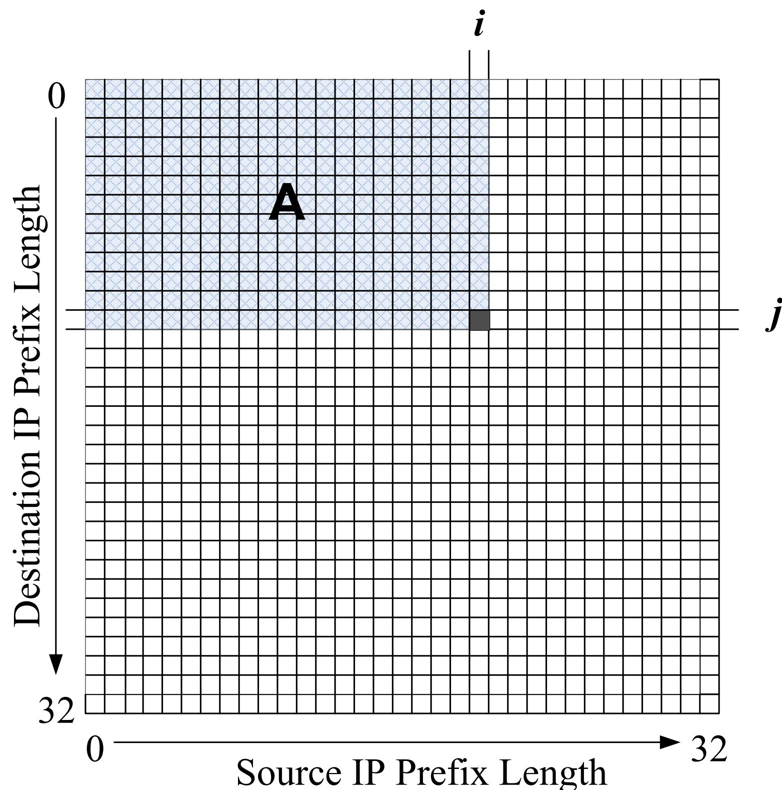

Figure 6.1: 2D Tuple Space Search

While we also use the similar optimizations, we attack the problem from a different angle. The problem with the tuple space search algorithm is that the number of tuples is too large. We reduce this number by identifying groups of tuples that can be searched efficiently using the crossproducting technique, which we discuss next. This allows us to dramatically reduce the number of separate searches that must be performed.

6.2.2 Crossproducting

The crossproducting algorithm is the most straightforward way to do packet classification. We assign each prefix on each field a unique ID. To classify a packet, the crossproducting algorithm performs LPMs on both fields first. The resulting IDs are combined to form an index, and then the index is used to retrieve the matching filter from a direct lookup table. If the source IP address field has m unique prefixes and the destination IP address field has n unique prefixes, the number of entries in the direct lookup table is $m \times n$. Although the lookup is very fast, the storage can be excessively large.

In the direct lookup table, not all the entries contain original filters. There are many "pseudo filters" that come from the cross-products of original filters. These pseudo filters must be stored and significantly expand the space used to represent the filter set. In Figure 6.2, A and C are nested prefixes in the source IP address field, as are B and D in the destination IP address field. Assume the prefix pair (A, B) is an original filter, $R1$. If the prefix pairs (A, D), (C, B), or (C, D) are not original filters, we need to add pseudo filters for each to guarantee correct lookups. For example, if the single field lookups return the matching prefixes C and D, the results imply a match to the filter $R1$. Without the pseudo filter (C, D), we will miss this match.

The expanded filter set can be produced easily. Let s be any source prefix and d be any destination prefix. If there is an original filter (s_i, d_i) such that s_i is a prefix of s, and d_i is a prefix of d, then let (s_j, d_j) be the highest priority such prefix and include an pseudo-filter (s, d) that maps to (s_j, d_j). We evaluate the filter set expansion effect for some ACL filter sets. The expansion factors are shown in Table 6.2. The synthetic ACL filter set is expanded by more than 500 times.

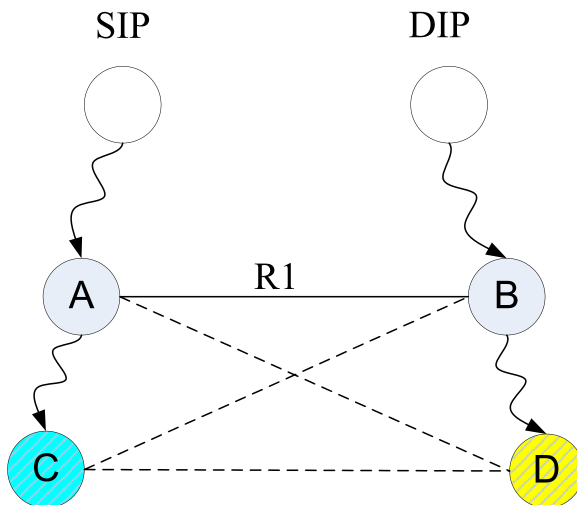

Figure 6.2: 2D Filter Expansion

Table 6.2: ACL Filter Set Expansion

Filter Set	Original filters	After Expansion	Table Entries
ACL1	426	19,885	29,294
ACL2	527	37,624	70,798
ACL3	1,588	222,396	408,157
ACL-syn	6,826	3,511,456	26,404,362

We can also see form the last column of Table 6.2 that 32% to 87% of entries in the direct lookup table actually do not contain any matching filter. Having the filters (p_{s1}, p_{d1}) and (p_{s2}, p_{d2}) does not necessarily mean we need to have two pseudo filters (p_{s1}, p_{d2}) and (p_{s2}, p_{d1}), because a match on (p_{s1}, p_{d2}) or (p_{s2}, p_{d1}) may not incur a match on any original filters. The empty table entries waste memory resources. This fact suggests that we use a hash table instead of a direct lookup table for better storage efficiency. By using FHT to implement the hash tables, we can achieve the similar lookup throughput as that of direct lookup tables.

6.3 Combining Tuple Space Search and Cross Products

Using a hash table rather than a direct lookup table for the crossproducting algorithm can significantly reduce the storage requirement. However, if we intend to keep all the filters in a single hash table, the expanded filter set due to the pseudo filters may still be too large. To mitigate the filter expansion effect, we split the filters into several subsets and build a hash table for each of them. Now the pseudo filters are only from the cross-products of the filters in each subset. Because the number of nested prefixes on each field in each subset can be much smaller than that when all the filters are put together, the overall number of expanded filters can be significantly reduced. As a tradeoff, now we need to query multiple hash tables to find a matching filter.

Combining the above ideas with the coarse-grained tuple specification, our new algorithm allows a nice throughput-storage tradeoff and therefore is fast and scalable.

We defer the discussion of the methods used to group tuples to Section 6.4. Now assume we have partitioned the tuples into k groups. For each group, we store the filters in a hash table, adding pseudo filters as needed to ensure that we can correctly identify the matching filter. Figure 6.3 illustrates a tuple partition. There are nine tuple sets from A to I in the figure. The tuple set I's specification is $([8, 16], [9, 21])$, for instance.

The lookup process is described as follows. We perform single field LPMs for both fields first. Since our coarse-grained tuples divide each IP address field into several segments, the LPMs need to return the longest matching prefix in each segment[1]. We use the results to determine the set of hash tables to query. Then we query these hash tables from more specific tuples to less specific tuples and terminate the search once the best matching filter is found. We use an example as shown in Figure 6.3 to illustrate the lookup algorithm. Suppose that for a given packet, the single field lookups show its source IP address has a matching prefix in segments $[0, 7]$ and $[17, 32]$, and its destination IP address has a matching prefix in segments $[9, 21]$ and $[22, 32]$.

[1]We can easily adapt our LPM algorithm discussed in Chapter 5 to support this. Through preprocessing, we embed such information in each SMT root so that the LPM performance is not affected.

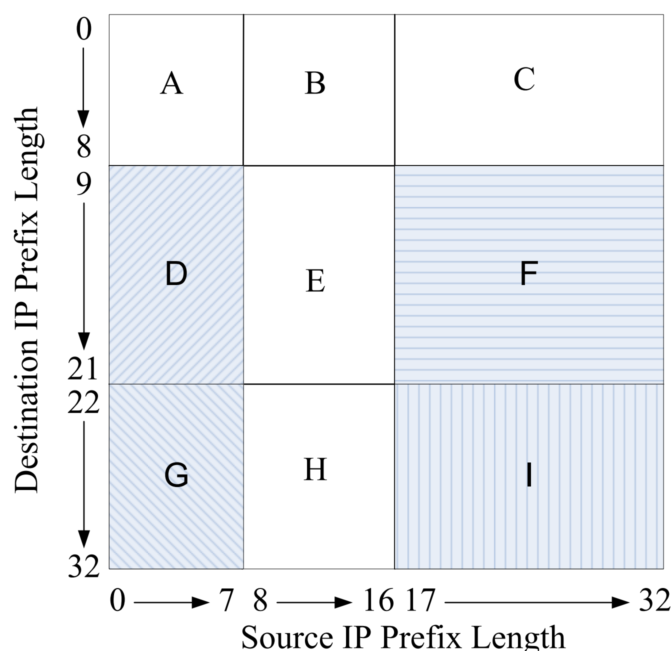

Figure 6.3: 2D Coarse-grained Tuple Space Partition

So the best matching filter can exist only in the hash tables for the tuple sets D, F, G, and I. Now the logical choice is to start the search from the tuple sets I, G, and F, because if we find matching filters belonging to these tuple sets, we do not need to search the tuple set D. If no match is found in these tables, we proceed to search the tuple set D.

With the above mentioned tuple partition, each tuple set is mapped to a grid in the 2D plane as shown in Figure 6.3. Choosing the number of segments for each field is a tradeoff of throughput and storage. At one extreme, when both fields have 33 segments, the algorithm regresses to a naive tuple space search algorithm. At another extreme, when both fields have only one segment, the algorithm regresses to a naive crossproducting algorithm.

We use the LPM algorithm discussed in Chapter 5 to perform LPM on each field and the FHT data structure discussed in Chapter 4 to improve the hash query performance. The architecture of the hardware implementation is similar to that discussed in Chapter 5.

6.4 Tuple Partition

If we visualize the tuple space in a 2D plane, the key for good performance of our algorithm is to come up with effective tuple partitions. There are two dimensions to approach this problem. First, if we are given the acceptable worst-case throughput, we seek to minimize the storage. Second, if we are given the storage we can use, we seek to maximize the throughput.

Now we consider the first approach. With the simple grid tuple definition, once the worst-case number of hash queries is chosen, we need to determine how to assign the number of segments and the boundary of each segment on each field. The goal is to minimize the size of the expanded filter set. This optimization problem can be achieved through dynamic programming. In practice, we need also to consider the performance of LPM so it is better to have regular sized segments. Fortunately, we find that the regular sized segments can result in good performance.

Besides the grid-based partition, it turns out that we can have arbitrary tuple partitions. Figure 6.4 shows a simple example. There are only three tuple sets. The tuple set B can be represented as $([12, 23], [1, 25]) \cup ([1, 11], [9, 25])$, for instance. Note that we have removed the tuple sets $(0, 0)$, $(0, [1, 32])$, and $([1, 32], 0)$ from the tuple space, because the filters in them can be handled by the LPMs on both fields so that these filters do not need to be stored in any hash table. The lookup process for such partitions is almost the same with a minor difference. It is possible we find in a tuple we may have multiple prefix length combinations that we need to check. However, we only need to check the most specific combinations, thanks to the "pseudo filters". Hence, the number of hash table queries is still bounded by the number of tuples.

We can preset the tuple partition in favor of the LPM implementation. In this case, we have little control of the filter expansion but we can guarantee the worst-case performance. On the other hand, we can dynamically determine the tuple partition by constraining the filter set expansion so we can control the storage but not the worst-case performance. For example, we may want the overall filter set expansion factor to be no more than α. On way to achieve this is to partition the space incrementally. If the expansion ratio for a particular tuple set is too high, we divide it into smaller subsets.

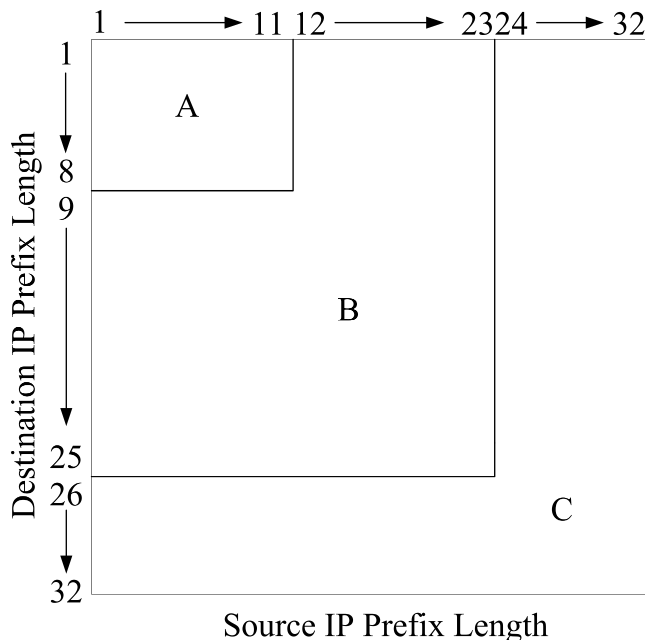

Figure 6.4: Another Tuple Partition Scheme

While simple tuple partitions seem to work well on current filter sets, it makes sense to study better tuple partition algorithms for larger filter sets in the future. We believe there are many opportunities for future work in this direction.

6.5 Evaluation

We first evaluate the grid tuple partition. We perform experiments on several ACL filter sets and summarize the results in Table 6.3 and Figure 6.5. In the first row of the table, $\alpha \times \beta$ means that the source IP address field has α equal sized segments and the destination IP address field has β equal sized segments. Therefore, the values of $\alpha \times \beta$ give the worst-case number of hash queries for a packet. In the figure, the dotted lines indicate the original size of the filter sets. We can see that at the cost of a very small number of hash queries, the size of the expanded filter set decreases quickly to approach its original size.

We use the LPM algorithm discussed in Chapter 5 to perform single field lookups. We assign Bloom filters at the same segment boundaries. Table 6.4 shows the total

Table 6.3: Filter Set Expansion for Different Configurations

Filter Set	# filters	1×1	1×2	2×1	2×2	2×4	4×2	3×3
ACL1	426	19,885	2,851	997	472	445	472	445
ACL2	527	37,674	9,317	8,978	1,225	922	899	596
ACL3	1,588	222,396	55,046	28,537	3,212	3,160	1,779	1,737
ACL-syn	6,826	3,511,456	384,353	34,579	9,666	7,992	9,666	7,992

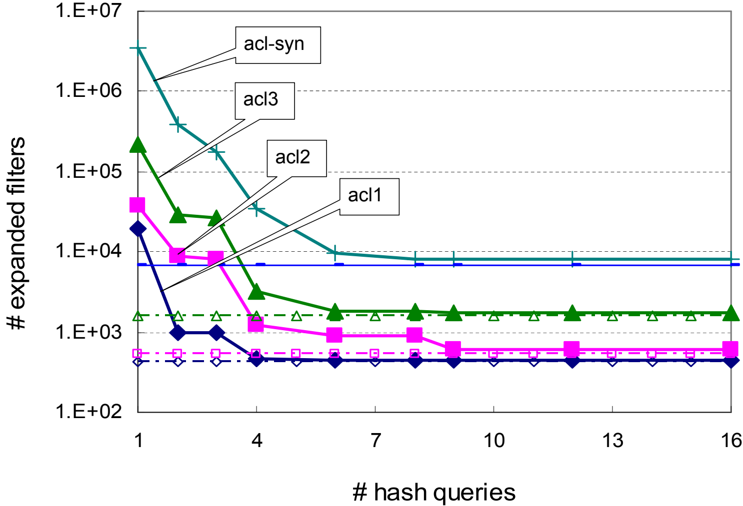

Figure 6.5: Filter Set Expansion vs. Number of Hash Queries

number of items that need to be programmed into the Bloom filters for LPM. The numbers are typically very small, which implies a small amount of on-chip memory usage.

Table 6.4: Number of Items in Bloom Filters for LPM

Filter Set	1×1	1×2	2×1	2×2	2×4	4×2	3×3
ACL1	0	61	3	64	184	72	120
ACL2	0	49	43	92	163	160	207
ACL3	0	59	57	116	368	310	385
ACL-syn	0	387	191	578	1,168	943	1,103

We evaluate the algorithm performance when using irregular tuple partitions as shown in Figure 6.6. There are two to four tuple sets for different configurations. The tuple sets are equalled spaced on each axis. The simulation results are shown in Table 6.5. The size of the expanded filter sets is significantly reduced when a finer tuple partition is used. Sometimes when a finer tuple partition is used, the overall number of filters

in the expanded filter set does not change. This is because some tuple sets do not contain any filter at all. Actually, for these evaluated filters sets, at most two tuple sets need to be searched even with the configuration (c) where there are four tuple sets, so in the worst case, a packet lookup requires only two hash table queries.

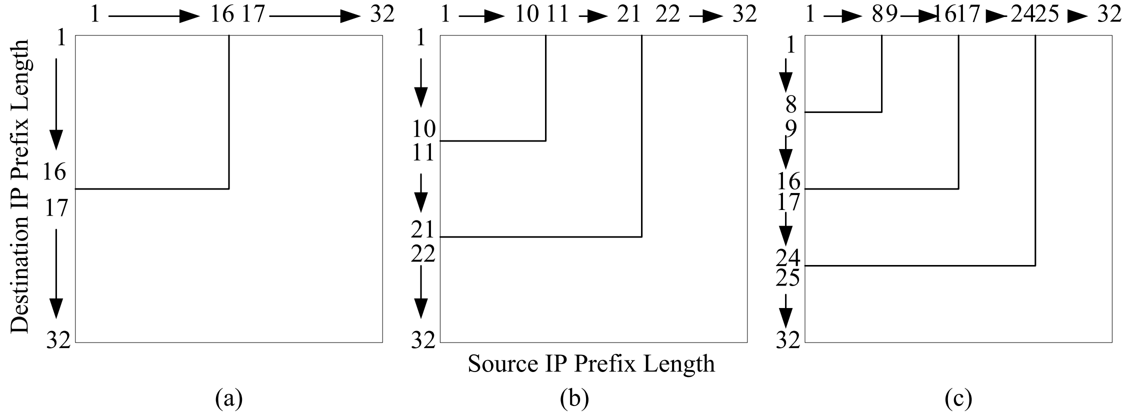

Figure 6.6: Experiments on Irregular Tuples

Table 6.5: # Filters for Different Tuple Configurations

Filter Set	# filters	Single Tuple	Config. (a)	Config. (b)	Config. (c)
ACL1	426	1,004	1,004	1,004	542
ACL2	527	3,027	3,027	2,604	1,914
ACL3	1,588	174,561	6,730	6,730	6,692
ACL-syn	6,826	185,799	185,799	185,799	12,306

Finally, we dynamically determine the tuple partition by setting the filter set expansion factor to 2. For ACL2 and ACL3, we cannot achieve this goal with the approach shown in Figure 6.4. This means we need better tuple partition algorithms. For ACL1 and ACL-syn, we get the tuple partition shown in Figure 6.7. The figure also shows the number of filters before and after filter set expansion. The results show that two tuple sets are enough to satisfy our storage constraint.

6.6 Conclusion

In this chapter we deal with a special case of the packet classification problem where each filter is specified as two prefixes. We present a novel 2D packet classification

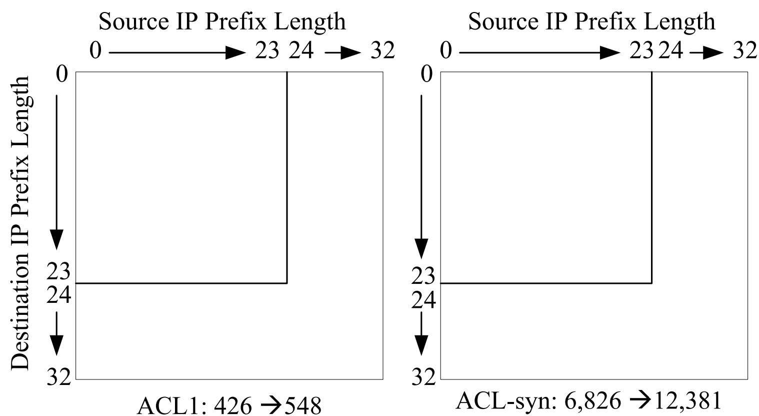

Figure 6.7: Dynamic Tuple Partition

algorithm derived from the tuple space search algorithm and the crossproducting algorithm. When implemented with our LPM and FHT techniques, the algorithm performs much better than previous algorithms for 2D IPv4 packet classification. It becomes even more attractive in IPv6 scenario, where the previous algorithms suffer from the much longer prefixes. With a flexible tuple partition scheme, our algorithm exhibits an attractive tradeoff between storage and throughput, which allows the designer to control the system performance based on the available resource and the desired classification throughput. Our algorithm is fast yet yields small on-chip and off-chip memory consumption.

Chapter 7

Adaptive Binary Cutting

7.1 Background

Designing a general packet classification system involves a lot of tradeoffs. It requires significant engineering considerations. If an algorithmic solution can satisfy the worst-case throughput performance with a reasonable amount of memory, it is not worth investing in a TCAM chip to do the job. According to the size of the algorithm data structure, different types of commodity memory, such as DRAMs, SRAMs, or on-chip SRAMs, can be used. They all have different impacts on the system complexity and the achievable performance. Generally, smaller storage allow us to use faster memory devices. If we can squeeze everything on-chip and avoid any off-chip memory, the system performance is maximized accordingly.

However, the storage used by the algorithms is often given little attention, as if the memory components are free. To compete with TCAMs, the algorithms need to ensure that their storage is more scalable than TCAMs. Although this is hard to measure, we can still take some hints from the cell density and the manufacturing cost. Typically, a TCAM component requires 14 to 16 transistors to store a bit, an SRAM component requires six transistors, and an SDRAM component requires only one transistor and a capacitor [48]. Taking the manufacturing cost into account, we can reasonably assume that a bit in a TCAM component is worth about ten bits in an SRAM component [3]. In other words, since a TCAM component usually consumes 18 bytes (144 bits) to store a filter, an SRAM-based algorithm should consume no more than about 180 bytes per filter in order to compete with TCAM. Unfortunately, many

well-known algorithms fail to satisfy this criterion. For example, the Recursive Flow Classification (RFC) algorithm consumes more than 1,600 bytes per filter for a filter set with only about 600 filters [33]. Similar inefficiencies arise with other algorithms, such as the Cross-producting algorithm [65], the Bit Vector (BV) algorithm [39], and the Aggregated Bit Vector (ABV) algorithm [10]. They all suffer a significant storage penalty, even though their throughputs are comparable to TCAM. In such cases, we need a better justification for applying these algorithms rather than directly using TCAMs.

Decision tree-based algorithms [32, 56, 78] offer more flexible control over the storage. A decision tree is built by splitting the filter set recursively using partial filter information. We stop splitting a subset when it contains fewer filters than a predefined *bucket size*. The filters stored in a decision tree node are organized in a list based on their priorities. Note that it is computationally infeasible to accomplish a globally optimal decision tree, so all these algorithms are based on some heuristics and try to achieve local optimality. The packet classification is performed by traversing the tree and linearly searching the stored filters. The search in a list stops once the first matching filter is found. The decision tree-based algorithms are very easy to implement but their performance also suffers from uncertainty. After making the fundamental observations on these algorithms, we change the decision tree building philosophy to better comply with the high level design goal and introduce extra degrees of freedom to allow more intelligent decisions. The new algorithm comes with three variations that are able to scale to large filter sets and provide sufficient throughput for OC-48 and faster networks.

The chapter is organized as follows. We first review the related work in Section 7.2. We then analyze decision tree-based algorithms and make some observations in Section 7.3. We provide a detailed algorithm description and implementation in Section 7.4. We present several novel algorithm optimizations in Section 7.5. We evaluate our algorithm and compare it with HiCuts, HyperCuts, and Woo's algorithm in Section 7.6. Finally, we conclude the chapter in Section 7.7.

7.2 Related Work

The decision tree building process is all about splitting the filter set recursively using partial header information. The keys are to limit the storage that is used to implement the data structure as well as the effort required to traverse the tree in order to find the best matching filter. When we build a decision tree (DT), a decision is made at each step to split the filter set, S, into the subsets $s_1, s_2, ...s_n$. Each subset represents a child DT node. We know that

$$s_1 \cup s_2 \cup ... \cup s_n = S$$

If $|s_i| >$ *bucket size*, we keep splitting s_i. Otherwise, the corresponding DT node becomes a leaf node. It is common that $s_i \cap s_j \neq \emptyset$ and $|s_i| \neq |s_j|$ for some i and j, which make the decision tree less efficient.

Woo proposed a generic approach to split the filter set [78]. At each node in the tree we consider some header bit that was not examined in any of the ancestors of the current node. The bit to be examined is chosen according to heuristic criteria that seek to minimize tree depth and size. Filters are stored in one or both subsets of a give node according to whether the selected bit for that node is '1', '0' or "don't care". At each decision step, all the remaining filters are evaluated first to get the statistic on the 1/0 distribution and the number of "don't care" specifications for every bit position. Then we choose the bit at the position where there are the fewest "don't care" specifications and the most uniform distribution of '1's and '0's to split the set. One disadvantage of this algorithm is that it splits a filter set using only one header bit per step. It is desirable to use more bits to make more subsets in order to accelerate the search. However, evaluating the "entropy" of multiple bits among 100+ filter bits is time consuming. In addition, the algorithm does not work directly on most real filter sets because of the prerequisite that the filters are represented as ternary bit strings. Real filter sets typically specify the port fields as arbitrary ranges that usually cannot be directly represented in the required format.

HiCuts [32] and HyperCuts [56] can generate multiple subsets per step and are practical for handling general filter sets. They both take the geometric viewpoint. A space region is cut into some equal-sized subregions at each recursive cutting step.

The corresponding subset contains all the filters that overlap the subregion. Locally optimal cutting decisions are made to best reduce the size of subsets and avoid storage expansion. The major difference between HiCuts and HyperCuts is that the latter allows cutting along multiple dimensions simultaneously in one step.

These algorithms provide better controls on the throughput-storage tradeoff. They can significantly affect the storage and the throughput by varying the tree branch fan out and the bucket size of leaf nodes. However, under reasonable storage consumption constraints, the throughput of these algorithms is often too poor to be useful; under an acceptable throughput constraint, the storage becomes too large to be satisfactory. There are some other drawbacks that we will discuss in the following section. Despite these problems, the decision tree-based algorithms naturally support a pipelined design; hence, they can provide a very high throughput if the number of pipeline stages is limited. Moreover, we can come up with very simple and efficient implementations for them based on binary encoding techniques. These advantages make this kind of algorithm attractive and motivate us to examine closely what causes the inefficiency of the decision-tree construction process and how to overcome it.

7.3 Observations

A good decision tree should have the following properties: the tree consists of as few nodes as possible, the path from the root to any leaf node is short, and the tree shape is well balanced. For the previous algorithms, the filter distribution can affect the resulting decision tree significantly.

The first problem, *filter duplication* (i.e. $s_i \cap s_j \neq \emptyset$), is caused by the fact that many filters are weakly specified on some dimensions (i.e. They have wildcards or large ranges in those dimensions). To reduce tree depth, we would like to make many cuts at each tree node, but this exacerbates the duplication problem. As a tradeoff, the previous algorithms typically set a *space expansion factor* to bound the number of duplicated filters. Without exceeding the threshold, we make as many cuts as possible at one step.

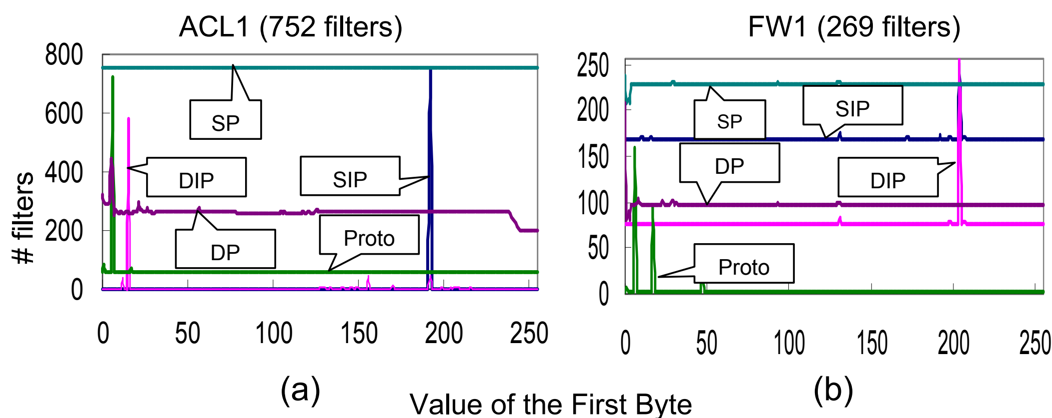

Figure 7.1: Filter Distribution on the Value of First Header Field Byte

Our simulation profiles show that during the DT building process, bits from the source or the destination IP address fields are chosen to split the filter set at about 80% of the DT nodes on the average. The filter distribution shown in Figure 7.1 implies this too. In these charts, the value on the y-axis is the number of filters that match packets with a given value for a particular field. In the figure, the header fields are the source IP address (SIP), the destination IP address (DIP), the source port (SP), the destination port (DP), and the protocol (Proto). Real world firewall filter sets typically contain many heavy wildcard filters; hence they suffer most from the filter duplication effect. Figure 7.1(b) shows the filter distribution based on the first byte of the IP header fields in a firewall filter set with only 269 filters. Any single cut on an IP address field duplicates more than 50 filters. We want to separate the filters concentrated in the spikes quickly. However, HiCuts and HyperCuts can only approach the spikes through equal sized cuttings; thus a large number of duplications are unavoidable. Woo's algorithm tends to build a tall tree in this case because the bit selected cannot separate the majority of the filters.

The second problem is *skewed filter distribution* (i.e. $|s_i| \gg |s_j|$ for some i and j). Filter distribution in real filter sets is often very skewed. From a geometric view, most filters are concentrated in a small region while a small number of filters are distributed across larger regions. Figure 7.1(a) shows the filter distribution on the first byte of the IP header fields in a real Access Control List (ACL) filter set with 752 filters. The filters are fairly specific on the IP address fields but the distribution is highly skewed. HiCuts and HyperCuts can only make even-sized cuts per step so

the cuts containing more filters need more steps to split. For example, a firewall filter set is used to protect a network in which most hosts own a class D IP address, for which the first four bits are "1110". If we use these four bits to split the filter set, most filters drop into the 14th child DT node, so this step has little effect but to duplicate some heavy wildcard filters.

We come to the conclusion that the filter distribution directly affects the DT efficiency. Unfortunately, the cutting strategy of the previous algorithms fails to react to this property. Actually, we can conceive a simple procedure like this: first we find the set of optimal cutting points that maximize the uniform distribution of filters among the cuts and minimize the filter duplication effect; then we sort and register these cutting points. When a DT node is retrieved during the lookups, we can simply perform a binary search on the point values. The search returns the pointer to the corresponding child DT node. This method leads to a smaller and shorter decision tree intuitively. One drawback is that the binary searching for the pointer can be slower than the direct indexing in HiCuts and HyperCuts, which take constant time to get the child pointer. Moreover, storing the cutting points consumes too much storage, which, in turn, would consume too much memory bandwidth for lookups. Ideally, we need to optimize the depth and size of the decision tree as well as the size of the DT node. These insights lead us to propose the *Adaptive Binary Cuttings* (ABC) algorithm, which actually includes three variations.

Exploiting the skewed distribution of filters, the key new idea of this algorithm is *to split the filter set based on the evenness of the filter distribution, rather than the evenness of the cut volumes.* Technically, we have developed three filter set splitting strategies so that the algorithm is able to adapt to the filter distribution geometrically or virtually. This additional degree of freedom leads to a more balanced tree and reduces the filter duplication effect. We use an efficient binary encoding scheme similar to the one used in the SST algorithm (see Chapter 3), which mimics the space cutting and can directly map to the bit string of packet header fields. The encoding scheme makes these strategies practical and easy to implement in both software and hardware. A simple example is depicted in Figure 7.2 to show the flavor of our algorithm, compared with the HiCuts and the Hypercuts algorithms. There are five filters distributed in a 2-D plane. At one cutting step, our algorithm best splits the

filters and avoids any filter duplication. Note that we cannot use the geometric view to illustrate the third variation of the ABC algorithm or Woo's algorithm.

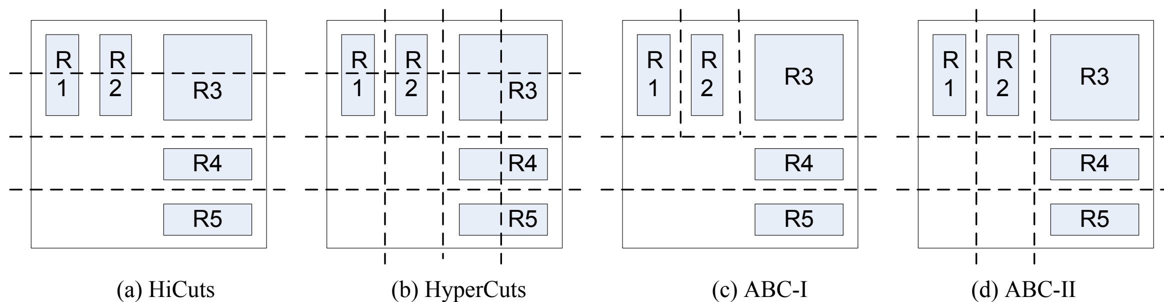

Figure 7.2: Cuttings on a DT Node for Different Algorithms

Another important observation concerning the previous algorithms is that their optimization criteria are weak. First, the HiCuts and HyperCuts algorithm both have to set an expansion factor to control the number of child nodes produced per cutting step, because although more equal-sized cuts can split the filter set better, they incur a storage penalty. Hence each DT node may have different numbers of child nodes. In a real design, it is convenient to make every DT node the same size. This causes some nodes to be under-utilized. It is also difficult to determine the proper value for the expansion factor, and a single expansion factor may not be suitable for all nodes. Since our set splitting strategies adapt to the filter distribution, each cut counts and will not negatively impact the storage efficiency. Therefore, our algorithm does not need such a parameter. Given the DT node size, we can fully utilize the capacity by making as many cuts as possible.

Second, all the previous algorithms stop splitting a set only if the number of filters is smaller than the predefined bucket size. Such an arrangement cannot guarantee either throughput or storage. Again, given a filter set it is difficult to determine the proper value and to compare the performance of different algorithms on a fixed basis, multiple runs of trial and error are required. Moreover, sticking with such a parameter blindly can lead to worse performance in some cases. In our algorithm design, we would like to answer the following questions: Given a fixed amount of storage, what is the algorithms' achievable throughput? In order to achieve a throughput, what is the minimum amount of storage required? While these two questions are actually interchangeable, the second question is a little harder to answer because without

any prior knowledge, we might set a throughput that we can never achieve. On the other hand, the first question is relatively easy to answer. We know that the minimum storage requirement is simply the storage needed to list the filter set and if we represent filters this way, the throughput is determined by the number of filters. When more storage is given, we can build a decision tree to intelligently split the filter set in order to achieve higher throughput. Clearly, each decision tree splitting step will increase the storage monotonically. Our goal is to make the decisions that best improve the throughput until the given storage is used up. Hence, we do not need the bucket size parameter either. Since the actual performance can be guaranteed only by the worst-case bound, we always try to improve the worst-case throughput performance at each step. We achieve this by evaluating all the current DT branches and the number of filters remaining in each current leaf node and then choosing the branch that causes the current worst-case throughput to continue working on, if the storage budget still allows us to do so.

Taking this into account, our algorithm is not only easy to understand but also easy to evaluate. In addition to all the above improvements, we also introduce several new algorithm refinements to further improve performance.

7.4 Algorithm

In essence, the decision-making processes use different degrees of freedom for HiCut, HyperCuts and Woo's algorithm. HiCuts chooses some number of prefix bits from only one dimension to split the filter set at each step; HyperCuts chooses some number of prefix bits from multiple dimensions at each step; and Woo's algorithm chooses any single bit from any dimension per step. These have different implications on the storage requirements of DT nodes. For example, if a filter involves D dimensions and L bits, HiCuts needs $\log_2 D$ bits to encode the cutting dimension at a DT node, HyperCuts needs a D-bit bitmap to encode the cutting dimensions, and Woo's algorithm needs $\log_2 L$ bits to encode the bit position. Moreover, with every additional bit chosen, the number of child DT nodes is doubled. We will show that this effect significantly affects the overall storage efficiency.

Our new algorithm introduces more degrees of freedom. Not only can we choose bits from any dimension or at any position to split the filter set, but also each resulting subset may require the examination of a different number of filter bits. Basically, the set splitting decision at each DT node can be represented as one or more full binary *Cutting Shape Trees* (CST). We encode each CST with a *Cutting Shape Bitmap* (CSB) which is essentially identical to the SBM in SST. The following pseudo code describes the basic decision tree construction algorithm.

Build_DT

1. while (current storage does not exceed the predefined storage limit &&
   some current leaf DT node has >3 filters)
2. let S_3 = set of leaf nodes with >3 filters;
3. select $v \in S_3$ with largest required time to search;
4. split node v to produce the CSTs and the new child DT nodes;

A DT node is not worth splitting further if it contains ≤ 3 filters, because in the best case, the resulting child DT nodes contain one or two filters each but the path is now one layer deeper. The cost of decoding one more DT node is greater than simply performing a linear search on the filters.

The chosen leaf node v identifies the current worst-case searching path. The current worst-case searching path is determined by the maximum cost (i.e. the largest number of bytes) for accessing a filter that is the last one in the list at a leaf DT node. Clearly, it is a function of the leaf DT node depth, the DT node size, the number of filters in the list, and the filter size. It is easy to find the current worst-case path if we maintain a dynamic sorting data structure such as a heap that uses the cost as the key. Each time we remove the highest cost path to process and then insert the new paths generated into the data structure.

The critical part of the algorithm is how to split a DT node and produce the CSTs. Here we derive three different approaches. We discuss them individually and then compare them. Before that, we define an important parameter that is used as a metric for the quality of a given cut. If a proposed cut divides the current filter set into subsets of size $r_1, r_2, ..., r_k$, we let:

$$pref = \sqrt{\sum_{i=0}^{k-1} {r_i}^2} \tag{7.1}$$

For example, in Figure 7.2(a), the preference value is $\sqrt{3^2+3^2+1^2+1^2} = 2\sqrt{5}$ and in Figure 7.2(b-d), the preference values are $2\sqrt{3}$, $\sqrt{5}$, and $\sqrt{5}$, respectively. The best cut decision should minimize the preference value. This choice of metric can be justified heuristically. Note that the preference value is smallest when the sets are equal in size. It is also made smaller when the number of duplications is minimized. Hence, it simultaneously seeks to optimize both of our high level criteria.

7.4.1 ABC Variation I

Producing and Encoding the CSTs

This variation produces a single CST at each DT node. Mapping each header field to a space dimension, we perform multiple cuts per DT node. The maximum number of cuts is determined by the DT node size. Each cutting step we choose one of the cuts produced so far and split it into two equal sized cuts along a certain dimension until we run out of space in the DT node. Actually, each of these cuttings resembles a binary tree node splitting and each cut corresponds to a CST leaf node. Therefore, we map the cutting sequence at a DT node to a full binary tree with k leaf nodes. Figure 7.3 shows an example on a 2D plane. Assume the DT node size allows us to split the region into eight cuts and we end up with the cuts as shown in the figure. The binary tree in the figure uniquely describes the cutting process: we cut the region on the x-axis first; then we cut the left sub-region on the x-axis, and cut the right sub-region on the y-axis, and so on. All information for reconstructing the cutting process is embedded in this tree: the number of cuts (i.e. k), the cutting sequence, and the cutting shape. Of course, we also need to associate the cutting dimension for each CST node, which needs $\lceil \log_2 D \rceil$ bits if the filter involves D fields.

We encode the CST with a CSB. The encoding scheme is identical to the SBM used in the SST algorithm. We associate a bit with each CST node. The bit value '0' is

Figure 7.3: ABC-I: Cuts and Lookup on a DT Node

assigned to the leaf nodes and '1' is assigned to all the other nodes. The CSB consists of this set of bits, which are listed in the breadth-first order. Since the CST root must have two child nodes and the bit associated with it is always '1', we omit this bit and effectively use only $2 \times (K-1)$ bits to encode the CST if there are K leaf nodes. In the example shown in Figure 7.3, we encode the CST with the CSB "11 10 11 10 00 00 00". The potential child DT nodes (the leaf CST nodes) are indexed in breadth first order.

Note that each internal CST node must store the information for the splitting dimension, which is encoded as the Cut Dim vector as shown in Figure 7.3. The Cut Dim vector is collected from the CST in breadth first order too. For a D-dimensional filter set, it is often the case that some dimensions are never used in a particular CST. We can include a D-bit vector in each DT node with a bit per dimension. A bit will be set if there is some CST node splitting on the corresponding dimension. Then, if only

two bits in the bit vector are set, we only need one extra bit per internal CST node. This optimization can save bits to allow more cuts per DT node.

After performing the cuttings at a DT node, each cut (i.e. each CST leaf node) represents a potential child DT node, depending on whether any filter remains in its set. Typically, not all of the possible child DT nodes are present, so it is inefficient to keep a pointer for each of them. Instead, as in the Tree Bitmap algorithm [27], an *Extending Path Bitmap* (*EPB*) of K bits is used to indicate the presence of child DT nodes. Bit i of the EPB equals '1' if a child DT node corresponding to the cut i is present. The pointer to the first child DT node and the EPB are sufficient to address any child DT node, as long as the child DT nodes are stored in consecutive memory locations and have the same size. Therefore, our algorithm avoids the necessity of recording the boundaries of the cuts and provides a very compact node representation.

We have shown how to encode a DT node with arbitrary K cuts using the compact data structure. Now we explain how to come up with the optimal CST at a DT node.

Starting from a single CST root, which represents the whole region covered by the current DT node, we need to figure out the CST leaf node to cut and the dimension to cut on at each cutting step. There are different ways to do this. For example, we can decide the CST leaf node first and then decide the dimension to cut on the node. In our implementation, we determine the two factors jointly. A new cut on each CST leaf node and on each dimension is evaluated. The CST leaf node and the cutting dimension that can minimize the preference value are chosen to grow the CST. Although this method is a little slow, it results in the best performance.

Formally, the current CST divides filters into sets of size $r_1, r_2, ...r_k$. If we split the node i on the dimension d, r_i is replaced with $r_{i,d,l}$ and $r_{i,d,r}$. We want to find the node i and the dimension d that can minimize the preference value $pref_{i,d}$. From Equation 7.1, we have

$$pref_{i,d} = \sqrt{r_1^2 + ... + r_{i,d,l}^2 + r_{i,d,r}^2 + ... + r_k^2} = \sqrt{pref^{\,2} + r_{i,d,l}^2 + r_{i,d,r}^2 - r_i^2} \qquad (7.2)$$

Hence, to find the minimum $pref_{i,d}$, we only need to evaluate each of the current CST leaf nodes to find the i and d that minimize $r_{i,d,l}^2 + r_{i,d,r}^2 - r_i^2$.

Decoding the CST

The lookup process traverses the decision tree and compares the packet header with the filters stored in the leaf DT node (or internal DT nodes in some cases). For each DT node, a CSB needs to be decoded to determine the child DT node to follow.

The CSB decoding algorithm is similar to the SBM decoding algorithm for SST. In the CSB, each '0' corresponds to a cut; each '1' corresponds to a splitting decision except the first one. The goal is to locate the index of the child DT node, for which the corresponding cut covers the packet header. To achieve this, we label the bits in the CSB with $0, 1, ..., 2(K-1)-1$, from left to right, and then perform the following step recursively starting from bit 0 until we find a bit position bearing the value '0', which means a cut on this DT node has been reached.

- Let the current bit position be i and the value of the next prefix bit from the dimension d under examination be x (x could be either '1' or '0'). Let $Ones(r, s)$ be the number of '1's in the CSB between the bit position r and s. In particular, $Ones(0, 0) = 0$. Then the next bit position j needs to be examined is $2 \times Ones(0, i) + x$.

We let $Zeros(r, s)$ be the number of '0's in the CSB between the bit position r and $(s-1)$. Once we quit the loop at the bit position t, the index of the child DT node is simply $Zeros(0, t)$. A node decoding example is illustrated in Figure 7.3, where a packet drops in the subregion with the child DT node ID 3.

During the DT traversal, the number of prefix bits for each dimension that has already been examined is implied by the DT node traversal and decoding history, so we do not need to maintain them explicitly.

7.4.2 ABC Variation II

Producing and Encoding the CSTs

In this variation, a DT node is also cut on multiple dimensions. The difference is that the cutting can end up with 1 to D separate CSTs, each for a chosen dimension. Assume our cutting strategy on a 2D plane ends up with the cuts in Figure 7.4. The x-dimension is split into six cuts and a 10-bit CSB is used to describe the cutting shape: "11 00 10 01 00". Likewise, the y-dimension is split into four cuts and the corresponding CSB is "01 01 00". Overall $6 \times 4 = 24$ cuts are produced by the two-dimensional cuttings.

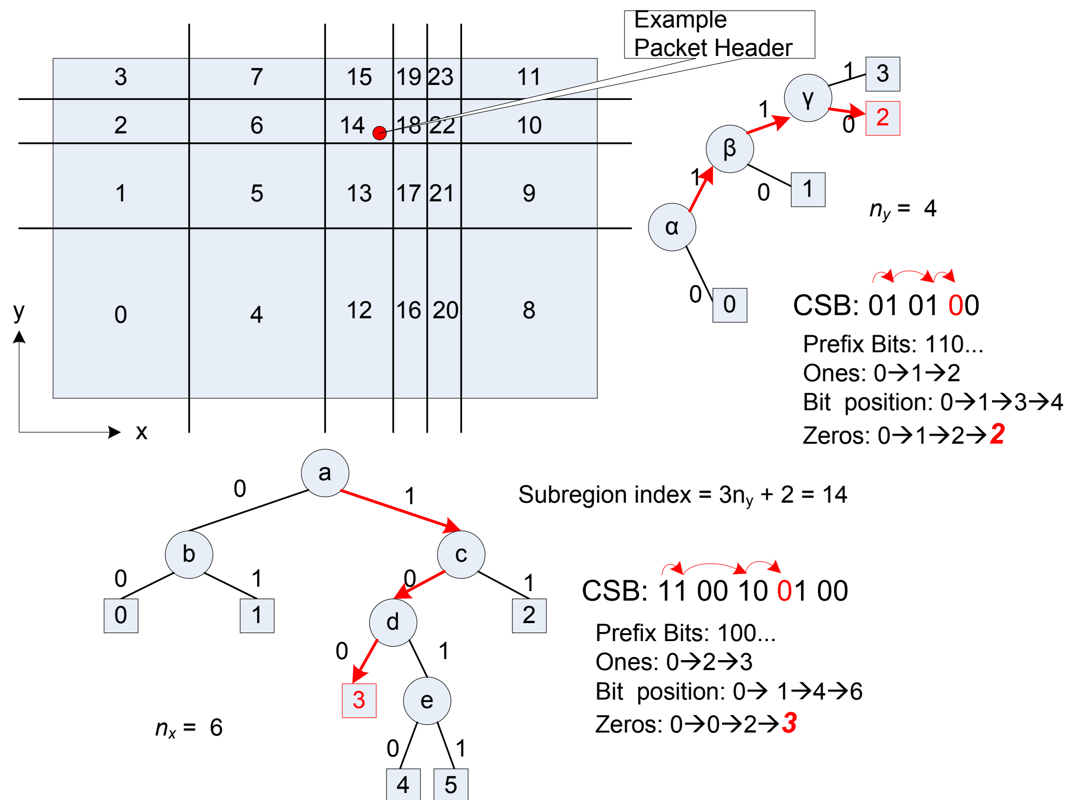

Figure 7.4: ABC-II: a DT Node and Decoding Example

To index these cuts, we incrementally label the leaf nodes of each CST starting from zero in breadth first order. This label, named as *Cutting Label*, uniquely identifies a cut on that dimension. Let the number of cuts on each dimension be $K_1, K_2, ..., K_D$.

Let the Cutting Labels on each dimension be $l_1, l_2, ..., l_D$. The cut index CI can be calculated as:

$$CI = \sum_{i=1}^{D}(l_i \prod_{j=i+1}^{D} K_j) \tag{7.3}$$

Figure 7.4 shows the index of the cuts. Each cut represents a potential child DT node, depending on whether any filter remains in its set. Again, we use an EPB of $K = \prod_{i=1}^{D} K_i$ bits to indicate the presence of child DT nodes.

In this variation, there is no need to maintain the cutting dimension for each CST node; hence more CST nodes are allowed, given the same storage space. Moreover, the total number of potential child DT nodes are now determined by the product of the number of leaf nodes of all the CSTs. Assume we have n CSTs and each CST has K_i leaf nodes. If we use a CSB to encode a CST, the overall storage consumption for the CSTs is

$$2\sum_{i=1}^{n}(K_i - 1) + \prod_{i=1}^{n} K_i$$

and the cuttings result in $K = \prod_{i=1}^{n} K_i$ cuts.

To produce the CSTs at a DT node, we start with D CSTs each for a dimension and with a single root node that represents the current range on that dimension. At each following step, a leaf node on a CST that minimizes the preference value is chosen to split, which resembles a range to be cut into two equal subranges. The process terminates when the assigned N bits are used up for encoding the CSTs. When we finish the process for a DT node, some dimensions may never be chosen for splitting. In order to avoid representing the CSTs only with a root node in the lookup data structure, we include a D-bit vector with one bit per dimension to indicate which dimensions are chosen to split at each DT node.

The naive computation of the preference value requires evaluation of all the current leaf CST nodes in each step, which is relatively expensive, although for a DT node, the total number of evaluations T is bounded by:

$$T = D + (D+1) + ... + (\sum_{i=1}^{D} K_i - 1) = \sum_{i=1}^{D} K_i(\sum_{i=1}^{D} K_i - 1)/2 - D(D-1)/2 \quad (7.4)$$

We can perform the similar transformation as in Equation 7.2 to avoid the redundant computations. For example, in Figure 7.4, we want to evaluate the new preference value if we cut the left most range on the dimension x. Since only the cut 0, 1, 2, and 3 will be split, we can only evaluate their effect to the resulting preference value.

Decoding the CSTs

The lookup process is similar as in the ABC-I, except now we need to decode multiple CSBs and the EPB to figure out the child DT node index. Equivalently, in each DT node decoding step some variable number of prefix bits of the selected packet header fields are examined and used to traverse the decision tree, narrowing down the searching scope. A DT node decoding example is shown in Figure 7.4, where a packet drops in the subregion with the index 14.

7.4.3 ABC Variation III

Producing and Encoding the CST

This variation produces only a single CST at each DT node. Unlike the first variation where each filter field is seen as a dimension, here the filter is treated as a ternary bit string and any bit can be chosen to split the filter set. For this reason, we cannot map it geometrically. It also implies we have to first convert the port ranges to prefixes. This step expands the filter set. This method is similar to Woo's algorithm but there are some significant differences. First, as we have discussed, the high level decision tree building approaches are different. Second, our algorithm encodes multiple filter bits per DT node. Third, the algorithms use different preferences to choose the bit to split the filter set. Based on simulations, our preference leads to better performance.

Table 7.1: Bit Consumption of CSBs and EPB

Variation	# Bits
ABC-I	$(2+\lceil\log_2 D\rceil)(K-1)+K$
ABC-II	$D+\sum_{i=1}^{D} 2(K_i-1)+\prod_{i=1}^{D} K_i$
ABC-III	$(2+\lceil\log_2 L\rceil)(K-1)+K$

To produce the CST at a DT node, we start from a root node and keep splitting some leaf node using a bit from the filter string until we run out of space. At each step, we examine the new preference value for all the leaf nodes if any of the filter bits was chosen to split it. The leaf node and the filter bit that minimize the preference value are actually used to grow the CST. The final CST is encoded with a CSB. Each internal CST node also needs to record the bit used to split the node, which takes $\lceil\log_2 L\rceil$ bits if the filter is L bits long. We use an EPB of K bits to indicate the presence of child DT nodes, where K is the number of leaf nodes in the final CST.

Decoding the CST

The CST decoding algorithm is similar to that in the first variation.

7.4.4 Comparison

DT Node Capacity

The DT node size is fixed in real implementations. In addition to all the other common information held in a DT node, we have fixed N-bits to realize the CSBs and the EPB. Table 7.1 summarize the bits used by the CSB and the EPB for the three algorithm variations. Note that for the algorithm variation I, we assume no encoding optimization is used.

The performance is better if more cuts can be produced at each DT node. Assume 128 bits are assigned for encoding the CSBs and the EPB and each filter consists of five fields ($D=5$) and 104 bits ($L=104$), from Table 7.1, ABC-I supports at most 22 cuts per DT node and ABC-III supports at most 13 cuts per DT node. For ABC-II,

the maximum number of cuts per DT node is variable, but generally it can produce more cuts per DT node than the other two variations.

Implementation Complexity

The second difference affects the implementation. Since ABC-I and ABC-III generate a single CST per DT node and the CST can be very tall, the DT node processing latency is typically larger than for ABC-II, in which all the CSTs can be decoded in parallel. In case a pipeline or multiple lookup engines are needed to fill the memory bandwidth, ABC-II has smaller system complexity. However, the preprocessing time of ABC-II is the largest because it requires more evaluations per DT node.

7.4.5 Implementation

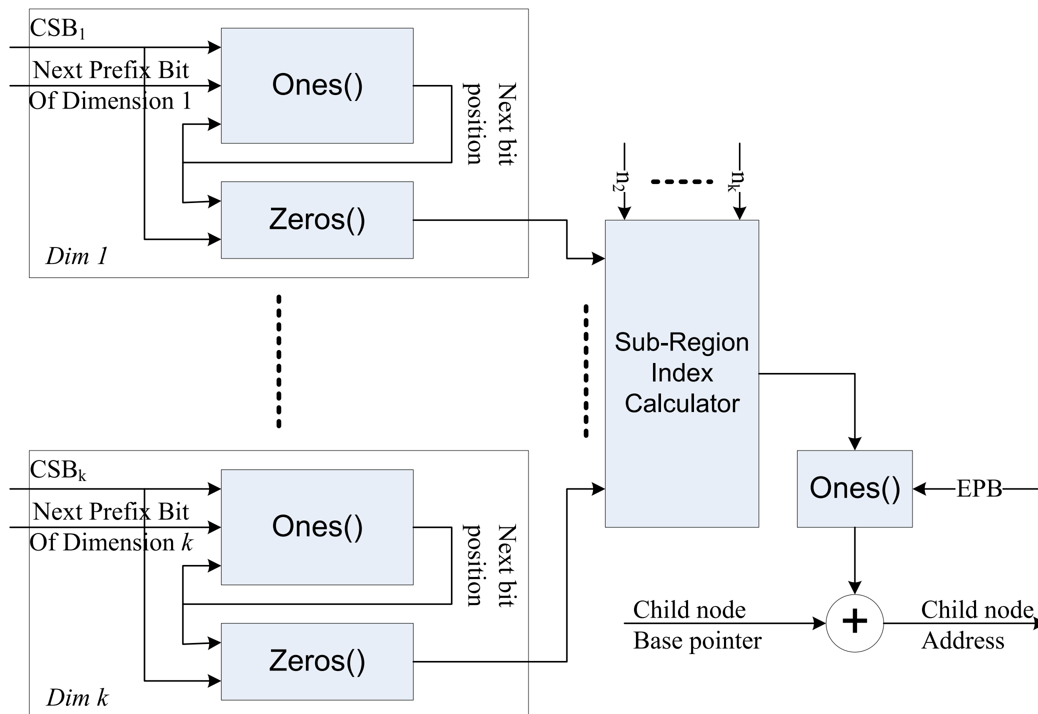

Figure 7.5: Decoding the DT Node to Find the Child DT Node Address

The hardware or network processor-based multi-threaded software implementations are more promising for meeting the throughput demands than conventional software-based implementations. Multiple lookup engines can work on different packets in parallel to fill the available memory bandwidth. The core component of the lookup engine is the CSB decoding logic. A simple hardware implementation of the CSB decoding uses a sequential circuit that computes values of $Ones(0, i)$, $Zeros(0, i)$, and the new bit position iteratively on successive clock cycles, terminating as soon as the position j of the CSB is equal to zero. For ABC-I and ABC-III, this takes up to $K - 1$ clock cycles. For ABC-II, multiple instantiations of the circuit can work in parallel, each for a CSB of a selected dimension. This takes up to $max_i(K_i)$ clock cycles. We need another one or two clock cycles to calculate the child DT node index and add the offset to the base pointer for the next memory access. The time to do a lookup at a single DT node only affects the number of lookup engines required, but not the throughput. The throughput is merely a function of the memory bandwidth and the number of memory accesses needed per packet lookup. A diagram of the DT node decoding circuit is shown in Figure 7.5. Note that only one CSB decoding block is required for ABC-I and ABC-III.

The data structure for the algorithm implementation is illustrated in Figure 7.6. Note that the internal DT node may also hold some filters due an optimization we adopt. Each filter is only stored once. When a filter must be duplicated, we only duplicate pointers to the filter. Since the size of a pointer is much smaller than that of a filter, this arrangement saves on overall storage. We also attach the priority value of each filter to each pointer so that in some cases a lookup can determine if a filter needs to be compared without actually reading the filter. For example, if we have had a matching filter with the priority value i and the current filter in the list has the priority value $j > i$, we know the new filter and all the following filters in the list cannot lead to a better match, so we can avoid reading the filter and stop searching the list.

Table 7.2 shows the DT node encoding scheme of a reference design in which each DT node consumes 16 bytes, each filter pointer consumes 2 bytes, and each 5-tuple filter consumes 18 bytes.

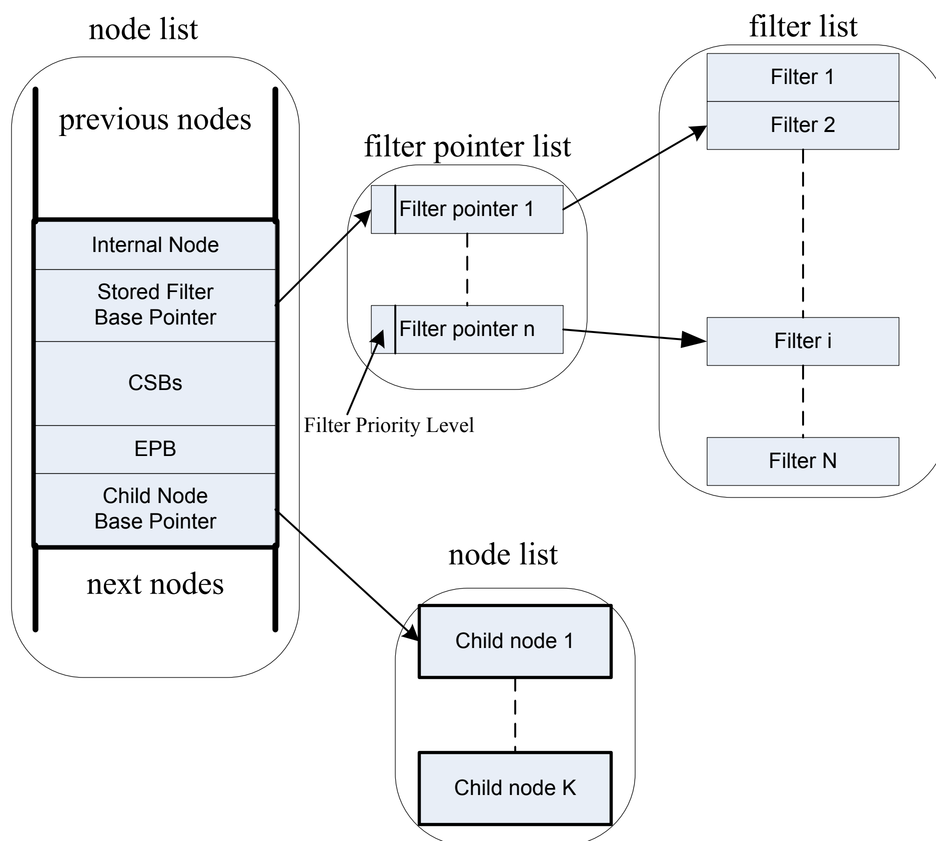

Figure 7.6: Data Structure of the Algorithm Implementations.

Note that for the ABC-II, 86 bits are left for the CSBs and the EPB. The assignment of these bits is dynamically determined by the number of CSTs and the size of each CST, according to Table 7.1.

7.5 Optimizations

The redundant filter removal optimization introduced in HiCuts [32] and HyperCuts [56] is embedded in the ABC algorithm by default. See Chapter 2 for a description of this optimization. Other optimizations are either inapplicable or improved. We will discuss them in later sections. This section discusses several new optimizations that improve the basic ABC algorithm.

Table 7.2: ABC DT Node Encoding Scheme(# Bits)

	ABC-I	*ABC-II*	*ABC-III*
isLeaf	1	1	1
Cut Dimension Bitmap	N/A	5	N/A
CSB(s)	75	*variable*	81
EPB (i.e. K)	16	*variable*	10
Child Base Pointer	18	18	18
Filter Base Pointer	18	18	18

7.5.1 Reduce Filters Using Hash Table

More filters in a filter set often mean larger storage and slower lookup throughput. Since hash tables allow fast lookups, we can use a hash table to handle a portion of the filters so that only the remaining filters participate in the DT construction. To build the hash table, we consider only the source IP and destination IP addresses for two reasons. First, when more fields are considered, the preprocessing time is significant. Second, the other fields in a filter are not specified as prefixes, so they are not easy to incorporate in a hash table. For the two IP address fields, we evaluate all the tuples (i, j), where $0 \le i \le 32$ and $0 \le j \le 32$. A filter belongs to a tuple (i, j) if the length of its source IP prefix specification is $\ge i$ and the length of its destination IP prefix specification is $\ge j$. We can build a hash table for each tuple by hashing on the first i bits of the source IP prefixes and the first j bits of the destination IP prefixes. Since many filters can share the same value on these selected bits, they will collide within the hash table. In order to bound the lookup time, we allow a hash table bucket to hold at most T filters. Here we assume for a tuple (i, j), any two filters with different i-bit source IP prefix and j-bit destination IP address prefix do not drop in the same bucket. We can achieve this using the FHT (see Chapter 4). We select the top tuple (i, j) that maximize the number of filters in the hash table without exceeding the bucket threshold T. These filters are removed from the filter set and inserted into a hash table. The hash table guarantees that in the worst case, at most T memory lookups are required to find a match. We evaluate some real filter sets as shown in Figure 7.7 and find that 18% to 44% of filters can be removed by setting the threshold to only one.

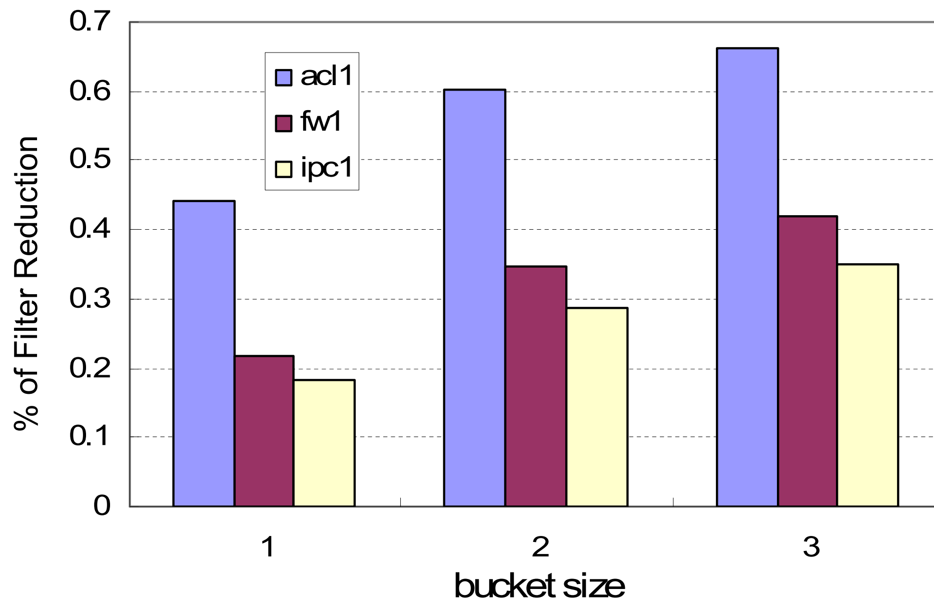

Figure 7.7: Effect of Filter Reduction by Using a Hash Table

The lookup process needs to search the hash table first. If a matching filter is found, the filter's priority value can be used to guide the search in the decision tree. For this reason, if more than T candidate filters are for the same hash bucket, we choose the T filters with higher priority (i.e. smaller priority values).

7.5.2 Filter Partition on the Protocol Field

The cutting or bit choosing method does not work well on the protocol field because the protocol values are not optimized for the purpose of decision tree building. This 8-bit field is used to encode only a few protocol values. In 10 real filter sets, only three to six protocol values are in use and all filter sets together use only eight unique protocol values. On average, 13% of the filters have a wildcard protocol specification. We need to examine five bits of the protocol field, accounting for 32 cuts, before we can separate the TCP (0x06) and the ICMP (0x01) protocol. This causes too many duplications of the filters with the wildcard protocol specification. A solution is to re-encode the protocol field or simply use the entire protocol field as the decision tree root.

Because re-encoding requires an extra decoding step, our implementation takes the latter method. We build a 256-entry table. Each entry stores a pointer to a decision tree. The table index is the protocol value. Each specified protocol value points to a unique tree; all unspecified protocol values point to a common tree. This step partitions the filters into a minimum number of subsets. Filters with the wildcard protocol specification are duplicated in each subset. The lookups first use the protocol field value to retrieve a tree's root, and then traverse the tree as usual. The data structure is shown in Figure 7.8. The example filter set only specifies the TCP, UDP, and ICMP protocols.

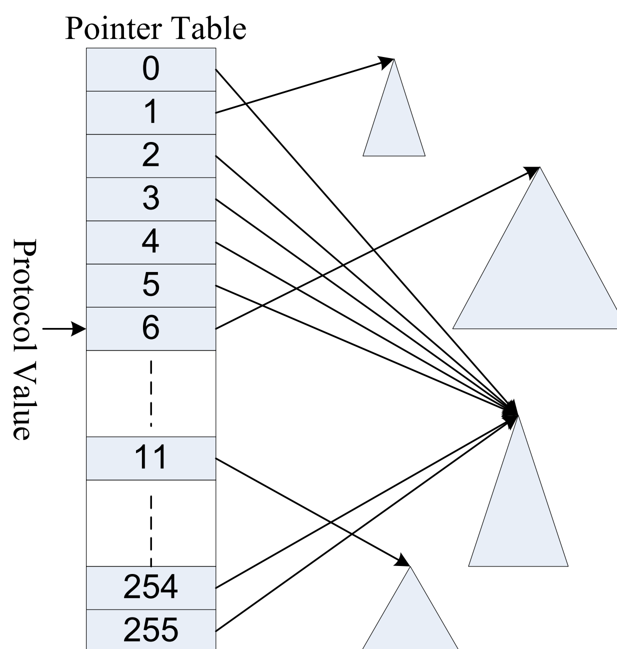

Figure 7.8: Protocol Pointer Table Structure

This simple optimization reduces the number of dimensions that need to be considered in the following steps. Therefore, some node bits can be saved to allow more cuts to be encoded. For example, in ABC-I, each CST node now needs only two bits to encode the cutting dimension information. Therefore, in the reference design shown in Table 7.2, a DT node can support a K of 19 rather than 16. Even if we do not change the DT node format, this optimization can still reduce the memory storage and increase the lookup throughput to some extent.

Since the third algorithm variation takes a unified view of the filter bits, we do not need to apply this optimization to this variation.

7.5.3 Partition Filters Based on Duplication Factor

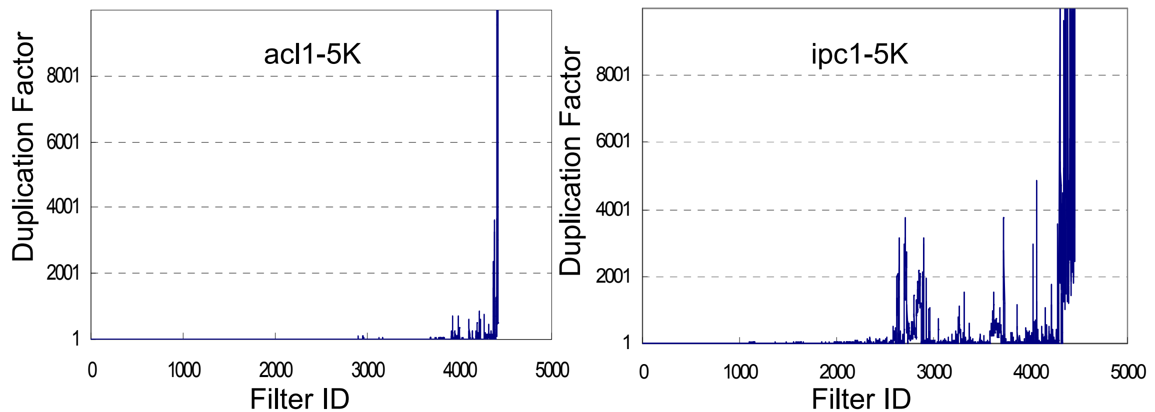

Figure 7.9: Filter Duplication Factor Distribution I

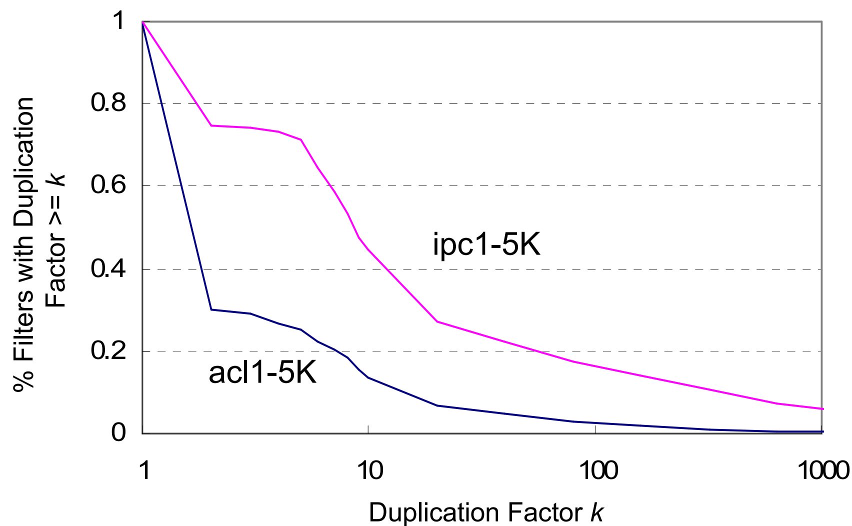

Figure 7.10: Filter Duplication Factor Distribution II

The cutting dimensions and the cutting shape are chosen in favor of the majority filters at a DT node. This causes some filters to suffer more duplications. We profile the duplication numbers of each individual filter for two filter sets when running ABC-II and depict the results in Figure 7.9 and 7.10. First, most of the filters receive

no or very few duplications while a relatively small fraction of filters account for a very large number of duplications. Second, higher-priority filters tend to receive fewer duplications than lower-priority filters. The results imply that some “spoiler” filters contribute significantly to the storage expansion and they should be handled in other ways.

We remove a few “spoiler” filters that cause excessive duplications from the filter set and then evaluate the algorithm performance on the remaining filters. The “spoiler” filters can be handled by a small on-chip TCAM. Our simulation shows that this optimization significantly improve the throughput performance, given the same storage budget.

7.5.4 Hold Filters Internally and Reverse Search Order

The HyperCuts algorithm [56] introduces an optimization called filter pushing up, which can reduce the filter duplications but cannot change the tree size and the throughput. We perform this optimization in a forward manner. At a DT node, if a filter would otherwise be duplicated into all the child DT nodes, we keep it in the current DT node to avoid duplications. Although storage efficient, this method actually can worsen the throughput performance, since the lookups also need to search the filters stored in internal DT nodes. We consider improving the throughput performance while retaining the gain on the storage efficiency.

Figure 7.9 tells us that the lower-priority filters tend to receive a larger number of duplications. The large number of duplications is directly due to the fact that these filters are less specific; hence, if we enable the rule pushing optimization, these filters are more likely to be held internally. The less specific the filters are, the more likely they are held closer to the tree root. However, since our search order is from the root to a leaf, even if we find a match at an internal DT node, we cannot avoid the searches on other filter lists in the deeper tree nodes. Indeed, we have a good chance to find a better match. This is the major reason for the performance deterioration.

This observation suggests that we should search the filter lists using the bottom-up order, and yet still traverse the tree from the root. When we find a stored filter list,

we do not start searching it right away. Instead, we put the pointer to the list into a stack. We begin to pop the pointers in the stack and search the filter lists only when we reach a leaf node. Using this order, we are more likely to search the filters in their natural priority order, resulting in early search termination.

7.6 Evaluation

We are concerned with the two most important performance characteristics of the ABC algorithm: storage efficiency and throughput. The storage is made up of two parts: the decision tree and the filters. The storage of the decision tree is determined by the number of DT nodes and the size of a DT node. The storage of the filters is determined by the number of original filters and the total number of duplicated filters (recall that each duplicated filter only consumes a pointer). The storage efficiency and scalability are evaluated by the number of bytes consumed per filter. As for the throughput performance, the depth of a DT branch and the number of filters stored along this branch determine the worst-case performance on the branch. We also need to take into account the cost of retrieving a DT node or a filter. We use the total number of memory bytes required per packet lookup to evaluate the throughput performance. Both the overall worst-case throughput performance and the average-case throughput performance are provided.

We use a suite of synthetic filter sets generated by *ClassBench* [6]. For each filter set, we also generate a packet header trace in which the number of packets is then times the number of filters. We run the lookup algorithm on these traces to collect the average number of bytes retrieved per packet lookup. This number roughly reflects the average-case throughput performance.

7.6.1 Comparison of ABC Variations

Scalability to Filter Set Size

First, we assume all the above mentioned algorithm optimizations are used when applicable, with the exception of the "spoiler" filter removal optimization. The hash table bucket size is set to one.

The size of the filter sets ranges from about 100 to about 10,000 filters. They are synthesized from an Access Control List (ACL) seed filter set, an IP Chain (IPC) seed filter set, and a firewall (FW) seed filter set. Because the ClassBench tool removes a few redundant filters after synthesizing the filter set, the actual filter size is slightly smaller than the target size. The simulation allows the storage of 100 bytes per filter on the average. Figure 7.11 shows the results.

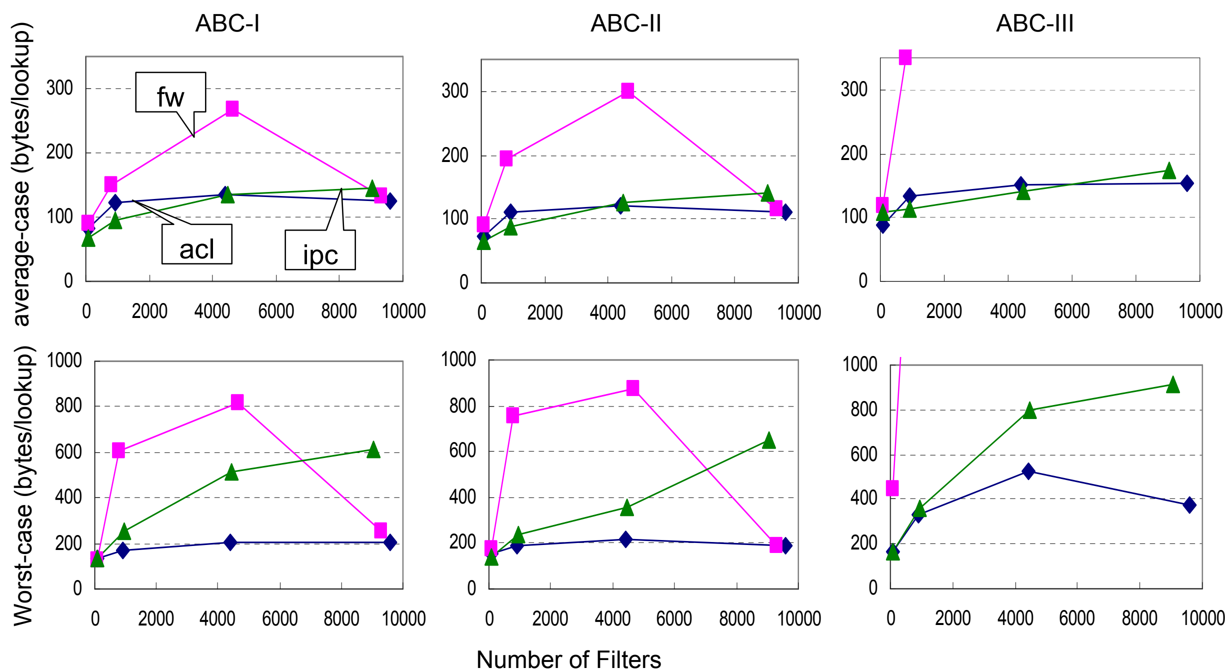

Figure 7.11: Algorithm Scalability on Filter Set Size

It is interesting to note that although we assign a storage budget of 100 bytes per filter, for the ACL filter sets, the algorithm ends up using much less storage. This implies the algorithm works best on the ACL filter sets. This also implies we have already

reached the limitation of the algorithm. More storage does not help to improve the algorithm performance further.

The worst-case performance is two to four times worse than the average-case performance. This is mainly because the imbalance of the decision tree, although our algorithm has tried the best to adapt the decision tree shape to the skewness of the filter distribution.

The performance is worst for the FW filter sets because they contain too many filters with lots of wildcards specified. The results confirm our previous analysis.

ABC-I and ABC-II show comparable performance. If we optimize the DT node encoding scheme for ABC-I to allow more cuts, ABC-I will outperform ABC-II. The performance of ABC-III is only acceptable for the ACL and IPC filter sets. This is because ABC-III gives the lowest DT node capacity and requires filter set expansion. For example, a FW filter set with 9,311 filters is expanded to having 32,136 filters.

Another interesting point is that the algorithm performance for the FW filter set with about 10K filters is better than that for the FW filter sets with 1K to 5K filters. This is due to the ClassBench tool. For a filter set with a larger size, it tends to generate it with more structured and specific filters.

To interpret the throughput performance, we consider a single 200 MHz QDR-II SRAM chip. It provides a memory bandwidth of 200 MHz $\times$ 9 Bytes = 1.8 GB/s. For the ACL filter set with about 10,000 filters, a packet lookup needs to retrieve 125 bytes on the average. So we can classify 1.8 GB / 125 Bytes = 14.4 million packets per second. In the worst case, when all the packets are just 40 bytes in size, a fully-loaded OC-48 link requires processing 7.8 million packets per second and a fully-loaded OC-192 link requires processing 30 million packets per second. The performance of our algorithm is sufficient for two OC-48 links but is not enough for an OC-192 link when a single memory device is used.

Throughput and Storage Tradeoff

In this simulation, we vary the storage to examine its effect on the achievable lookup throughput. The simulation runs on the synthetic IPC filter set with 10,000 filters. We disable the filter reduction optimization. Figure 7.12 shows the results.

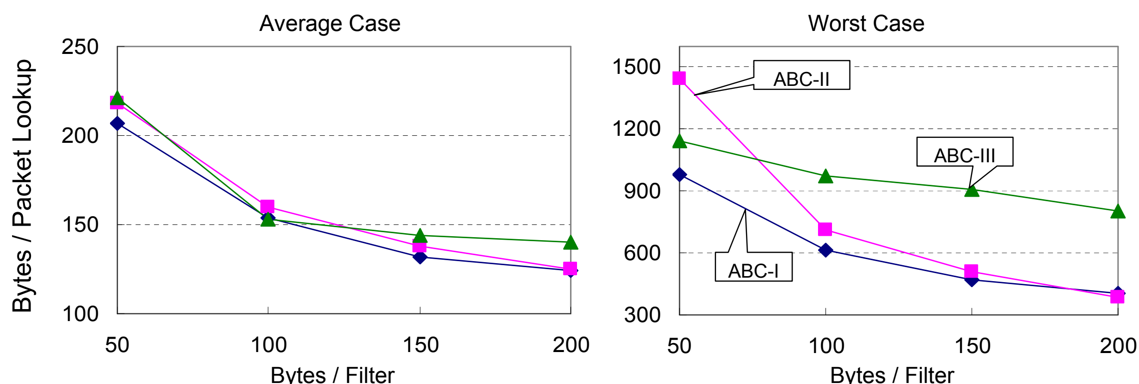

Figure 7.12: The Tradeoff of the Storage and the Throughput

When more storage is granted, the lookup performance steadily becomes better. All three variations have the similar average-case lookup performance, but there are significant differences in the worst-case lookup performance. ABC-I gives the overall best performance.

Sensitivity to Optimizations

Now we examine the algorithm sensitivity to different optimizations. In the simulations we use the synthetic ACL filter set with 10,000 filters and allow 50 bytes used per filter on the average. To isolate the effect of different optimizations, we turn them on one by one to compare with the bare algorithm without any optimization.

Figure 7.13 shows the effect of the filter reduction using a Hash Table. This optimization can significantly improve the worst-case performance (almost 2x for ABC-II) but only moderately for the average-case performance.

Figure 7.14 shows the effect of performing the protocol field lookup first. Note that this optimization is only applied to the first two algorithm variations.

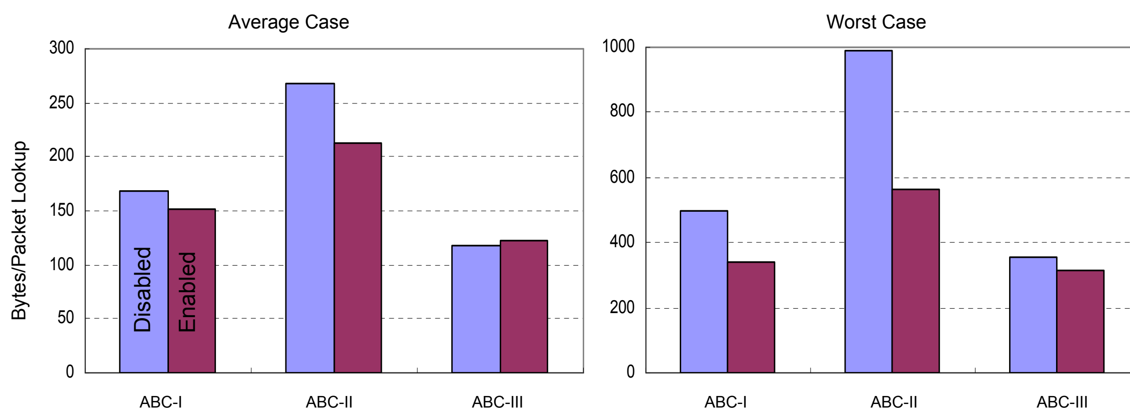

Figure 7.13: The Effect of Filter Reduction Using a Hash Table

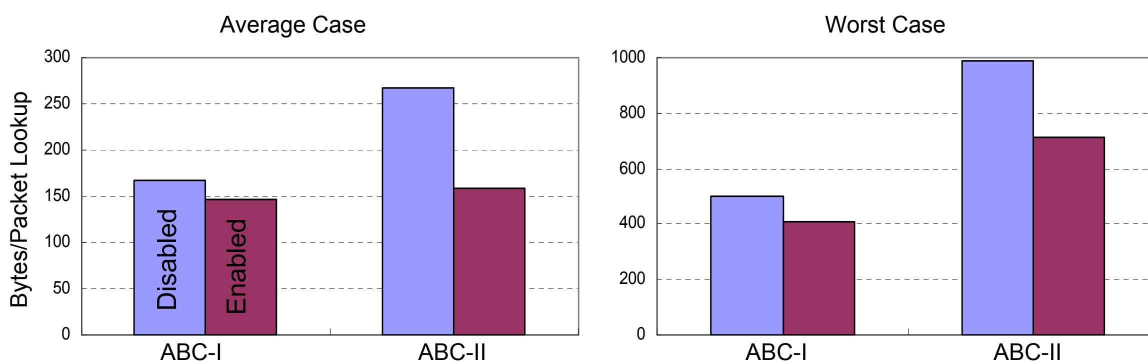

Figure 7.14: The Effect of Looking Up on Protocol Field First

Figure 7.15 shows the effect of holding filters internally and reversing the search order. We see this algorithm refinement actually only helps improve the first two variations of the ABC algorithm and it also has the best effect compared with the other optimizations. The performance of ABC-III gets worse. This is because its performance is near optimal. Holding filters internally increases the overhead of filter lookups.

Finally, we examine the effect of removing some highly duplicated filters from the filter sets. The duplication statistics are collected from an implementation of the HyperCuts algorithm. Only three to 14 filters (0.1% to 0.3%) are removed from the three filter sets. However, from Figure 7.16 we can see a significant performance improvement. This fact suggests that we should differentiate the filters that produce a large amount of duplications and use another method, such as a small on-chip TCAM, to deal with them.

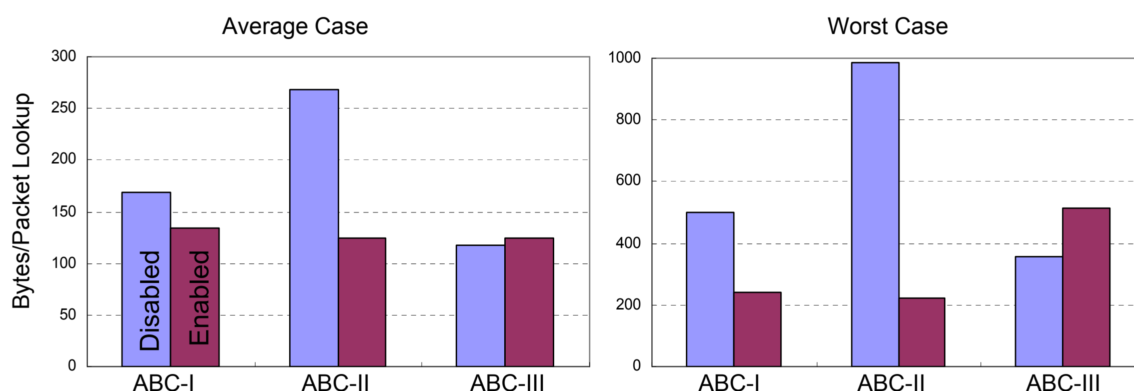

Figure 7.15: The Effect of Holding Filters Internally and Reversing Search Order

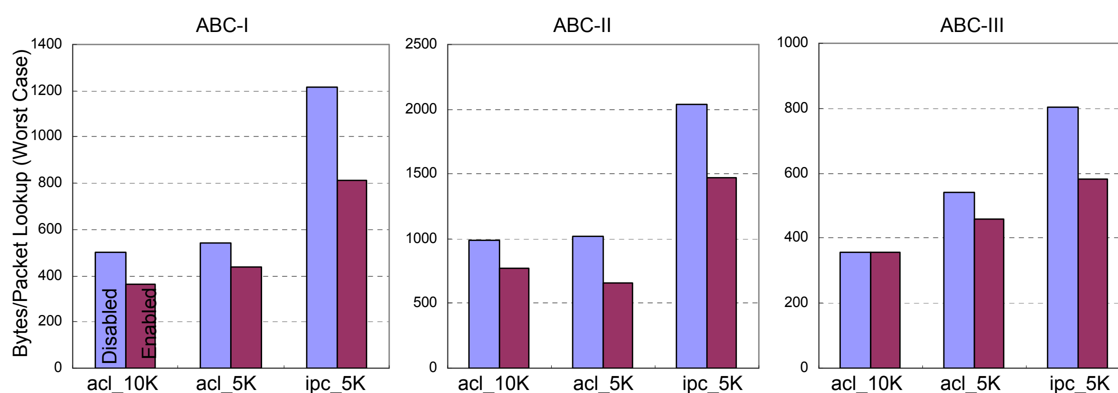

Figure 7.16: The Effect of Removing Highly Duplicated Filters

Effect of DT Node Capacity

The above evaluations are based on our reference design. Now we examine the algorithm performance when different DT node sizes are used. We evaluate five cases with DT node sizes 8, 12, 16, 20, and 24 bytes, respectively. We assume the size only affects the DT node capacity (i.e. the CSBs and the EPB). We turn off all the optimizations and allow 50 bytes per filter. The ACL filter set with 10,000 filters is used for the simulation.

As Figure 7.17 shows, in most of the cases, increasing the DT node size actually decreases the throughput performance. This is because under the same storage restriction, larger node size implies fewer DT nodes. Larger DT node size can gives

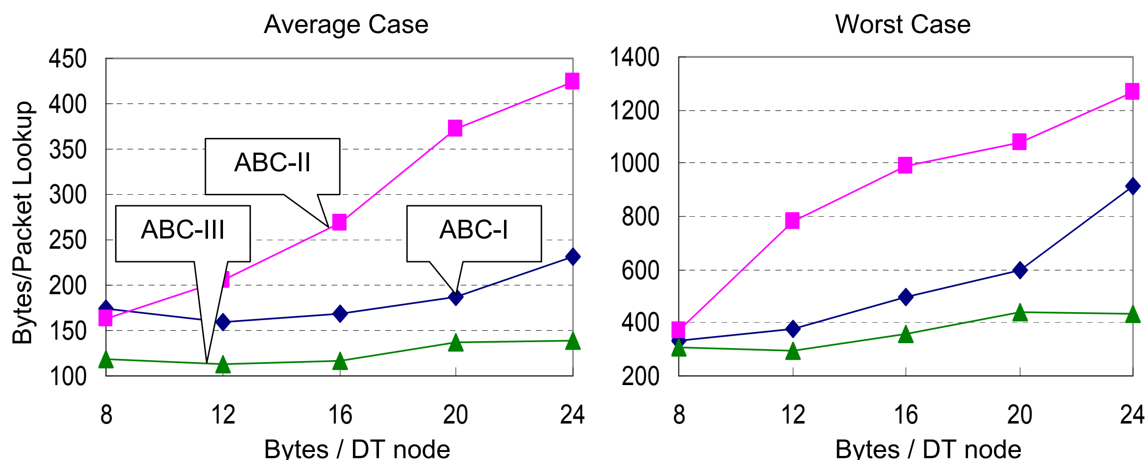

Figure 7.17: The Effect of Changing DT Node Size

better performance only when we increase the storage budget accordingly. This is another tradeoff that needs to be considered in the actual implementation.

7.6.2 Comparison with Other DT-based Algorithms

Implementation

We have shown how we can use CSB and EPB to efficiently encode a DT node. The HiCuts and the HyperCuts algorithms also support the similar binary-encoded implementation if the number of cuts on a dimension is limited to a power of two. In such a case, the geometric cutting process is actually identical to the process of examining several prefix bits on some fields in sequence. Since the cuts are regular, no CSB is needed and we only need to record which dimensions to choose and how many prefix bits on these dimensions to be examined at each DT node (For Woo's algorithm, things are even simpler. We only need to record which filter bit to choose per DT node). If r bits are examined, we concatenate these r bits and use the value of the string as the index of the corresponding child DT node. As long as the 2^r child DT nodes are stored in the order of their index values, they can be directly addressed.

Unfortunately, certain optimizations, like *region compaction* [56] and *node merging* [32], can no longer be applied because they require the DT nodes to explicitly

store the end points of each cut. The region compaction optimization can be applied when the actual region covered by the filters is smaller than the current cut. The border of the cut is trimmed off to make the filters fill the new compact cut. Now the child DT node's index can only be calculated using the number of cuts and the cut boundary. Storing the boundary of one dimension uses up to eight bytes of memory. The node merging optimization requires a 2^r array per DT node to store the pointers to child DT nodes. Explicit pointers increase the DT node size significantly when the number of cuts is large. Basically, such optimizations require huge DT nodes which impact both the storage and the throughput; hence, their benefit on the reduction of the DT size and the DT depth is compromised. In addition, these optimizations need significant preprocessing time and the resulting data structure eliminates the possibility for incremental updates at all. Therefore, we propose a simplified implementation. Note that the *overlapped filter redundancy removal* [32] and the *filter pushing* [56] optimizations can still be applied.

In order to compare with the ABC algorithm, we layout the DT node format that also consumes four 32-bit memory words for the HiCuts and the HyperCuts algorithms. A DT node in Woo's algorithm requires only two 32-bit memory words. Just as in the ABC algorithm, after cutting a node in HiCuts and HyperCuts, some cuts contain no filter at all. Creating empty DT nodes wastes a lot of memory. Again, we store an 18-bit base child node pointer and an EPB at each DT node for these algorithms. Non-empty child nodes of a DT node are stored in consecutive memory locations so that they can be addressed by the base pointer and the EPB. Note that when the number of cuts per DT node in HiCuts and HyperCuts is increased at a factor of two, so does the number of bits of the EPB. Hence, a DT node allows at most 64 cuts for HiCuts and HyperCuts, which means at most six bits can be examined at each DT node.

For HiCuts and HyperCuts, a 5-bit bitmap indicates which dimensions are chosen to cut and each selected dimension is assigned three bits to indicate how many prefix bits are examined (our assignment allows up to six prefix bits to be examined per step). The DT node encoding scheme is summarized in Table 7.3.

Table 7.3: DT Node Encoding Scheme for Other Algorithms(# Bits)

	HiCuts	*HyperCuts*	*Woo's*
isLeaf	1	1	1
Cut Dimension(s) (Bitmap)	5	5	N/A
# of prefix bits (Binary Encoded)	3	5*3	N/A
Bit Selection (Binary Encoded)	N/A	N/A	7
Child Base Pointer	18	18	18
Filter Base Pointer	18	18	18
EPB	64	64	2
Total (bits)	109	121	46

Comparison

Recall that the previous decision tree-based algorithms terminate the DT construction algorithm only if all the leaf nodes contain fewer filters than a predefined threshold, which results that neither the storage nor the throughput can be determined before the simulation. To set a basis for comparison with our algorithm, we run the simulation with different parameters. To make the comparison fair, we also apply the same set of algorithm optimizations to all the implementations. Figure 7.18 illustrates the results on three different filter sets. The x-axis stands for the storage and the y-axis stands for the average lookup throughput performance. Since ABC-I has the best overall performance, we only show the curve for ABC-I. The closer the data point is to the left-bottom corner, the better the overall performance. Clearly, the ABC algorithm significantly outperforms all the other algorithms.

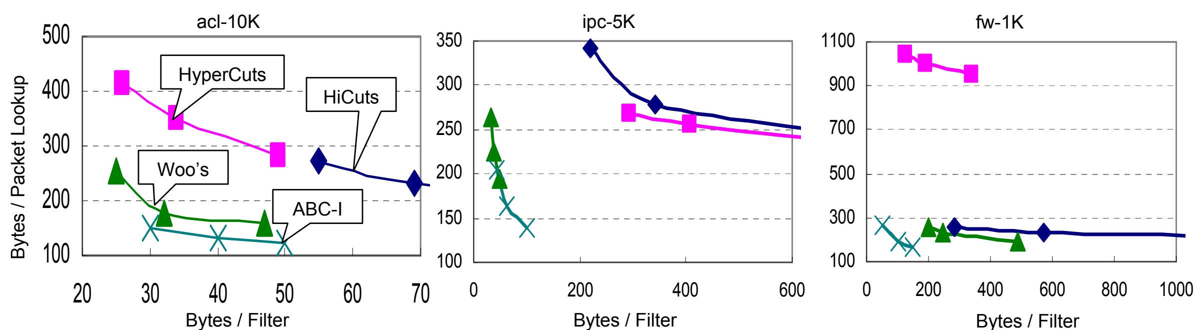

Figure 7.18: Compare ABC with the Other DT-based Algorithms

7.6.3 Incremental Updates

Generally, the decision tree data structure does not support incremental updates very well. The major reason is filter duplication. To insert a new filter, we may need to push a filter to many leaf nodes. Deletion requires a similar amount of work. More important, insertion and deletion may lead to suboptimal performance of the data structure, so we have to rebuild the decision tree from scratch at some point.

When using certain optimizations like region compaction and node merging proposed in [32, 56], the HiCuts and HyperCuts algorithms actually disable this kind of incremental updates. Our implementation helps regain the capability. Moreover, since the filter can also be stored in the internal tree nodes, we can use this feature to reduce the number of duplications made when inserting a new filter without degrading the throughput performance too much: if pushing a filter down to the leaf nodes makes too many duplications, we simply store it in some internal nodes with a restricted number of duplications.

7.7 Conclusion

A branch of packet classification algorithms is based on the decision tree data structure. Some of these algorithms build the decision tree through geometric cuts, but the cuttings are performed in favor of the evenness of the cut size rather than the evenness of the filter distribution. Due to the skewness of the filter distribution found in real filter sets, this approach exaggerates the effect of filter duplications, and in turn, results in a poor decision tree. Woo's algorithm aims to split the filter set more evenly and keep the filter duplication to a minimum. However, it only produces a binary decision tree. The algorithm cannot fully utilize the available memory bandwidth so the throughput suffers. Moreover, all these algorithms use some unnatural criteria to control the decision-tree building process, which make the algorithm evaluation and implementation very difficult. The merits of the heuristics are hard to justify and both the throughput and the storage are unknown before the experiments.

We introduce a new degree of freedom to enable variable sized cuts per decision step in order to make the filter distribution more even, resulting in a much better decision tree. A simple and compact encoding scheme makes this feasible. Since our algorithm guarantees that each micro cut counts, it ensures that each DT node can have the same size and be fully utilized. Furthermore, a more natural and implementation-oriented decision making process is applied. We can predetermine the storage budget to get the best achievable throughput, which allows better observability and controllability over the algorithm. Based on the similar idea, three variations are derived.

We also propose hardware-oriented implementations for the other decision tree-based algorithms: HiCuts, HyperCuts and Woo's. We compare the ABC algorithm with them through simulations. The ABC algorithm significantly improves the storage and throughput performance of the previous algorithms and is scalable to large filter sets. Although it is difficult to push the algorithm's performance to work on OC-192 networks with a single SRAM chip, it is still quite attractive for OC-48 networks. The simple implementation and its efficient use of memories make it a better candidate than TCAMs and other algorithms in such environments.

Chapter 8

Fast TCAM Filter Updates

8.1 Introduction

TCAMs are widely deployed in high performance network routers for packet classification because of their unmatched lookup throughput, and their generality. In TCAMs, packet filters are represented as ternary bit strings and stored in decreasing priority order. Given a packet header, the search for the best matching filter with the highest priority is performed on all the entries in parallel. The index of the first matching filter is then used to access an associated data memory to retrieve the data associated with the matching filter. This elegant architecture allows classification of packets in just a single clock cycle, allowing a state-of-art TCAM chip to support a sustained search rate of 250 million packets per second [3]. Even for backbone network routers supporting OC-192 (10Gbps) links, the peak packet rate in the worst case is no more than 30M packets per second, far less than the search capability that a TCAM provides.

In this chapter, we focus on TCAM filter set management, a problem that has received relatively little attention in the research literature. In an operating router, filter sets must change over time, in response to changes in network management policies and link availability. New filters may be inserted and existing ones deleted or modified. Because TCAMs return only the first matching filter, based on storage position within the TCAM, insertion of a new filter can require many other filters to be moved in order to place the new filter at the appropriate position in the filter set. In the worst case, a large fraction of the filters in a filter set may need to be moved for

each insertion. While many filter deletion and modification operations can be done without moving filters (using "lazy deletion" and in-place modification), in the worst case these operations can also require large numbers of filters to be moved.

While the rate at which filters are updated is much smaller than the rate at which lookups are processed, filter updates can have a significant impact on lookup rate, since updates must be suspended while a control processor makes the changes needed to complete a TCAM update. Wang et. al. show that the movement of just 16 TCAM entries in an OC-192 router can trigger the dropping of 18 packets [76]. As applications requiring more frequent updates emerge, the impact of updates on lookup performance may becomes much worse.

As we have mentioned, the lookup throughput of TCAMs actually exceed the requirements in typical applications. This suggests the possibility of trading off lookup throughput for more efficient filter updates. We show that this trade-off can be exploited to good effect, by encoding the priority as a field in the TCAM and using multiple lookups to identify the matching filter with the highest priority. The resulting system can sustain worst-case lookup rates of more than 65 million packets per second, and average rates of more than 80 million packets per second.

The remainder of this chapter is organized as follows. Section 8.2 discusses the related work. Section 8.3 presents our new algorithm and Section 8.4 evaluates it. Section 8.5 concludes the chapter.

8.2 Related Work

When TCAM is used for longest prefix matching (LPM), updates can be performed efficiently, using at most W moves to insert a new prefix, where W is the number of unique prefix lengths [55]. Because for any packets, there is at most one matching prefix among prefixes of the same length and we prefer the longest matching prefix, prefixes can be placed in the TCAM in decreasing order of their lengths. This ensures the correct IP lookup result and makes it relatively easy to update an entry. The update algorithm uses the property that changing the relative order of prefixes of the same length does not affect the lookup result, so one can insert a new prefix

by moving at most one prefix for every distinct prefix length. Since there are only $W \leq 32$ distinct prefix lengths, the update time is bounded and reasonably small. One can do even better by storing prefixes in chain-ancestor order [55]. In this case an update requires at most D moves for an insertion, where D is the longest prefix chain comprising the updated prefix. Carefully refining the memory layout can further reduce the total number of entry moves. Unfortunately, as will be explained in the next section, this approach cannot be directly used for general packet classification.

Rererence [76] is one of the few prior studies of the TCAM update problem for packet classification. The authors of [76] focus on how to maintain consistent filter table lookup throughput during the update process. They show that TCAM locking can be avoided by carefully managing the update process so that correct filter matches are ensured, even while the filter update is in progress. However, their method significantly increases the number of moves required, and while they do not lock up lookups during the update process, the filter moves do still consume TCAM bandwidth. In addition, the filter set management process is relatively complex and introduces a significant latency, which delays the time for an update to take effect.

A typical TCAM component provides 144 bits for matching a packet header. In IPv4 applications, some of these bits are not needed because the standard 5-tuple packet header contains only 104 bits. These otherwise unused bits can be used for other purposes. The MUD algorithm uses these bits to attach a filter index to each filter in order to support multi-match classification [40]. In this algorithm, the filters are stored in incremental index order. If the first lookup returns a matching filter with index j, then in the subsequent lookups, we only need to search the filters with index greater than j. This is achieved by converting the range "$> j$" into a set of subranges (e.g. prefixes) that can be represented by ternary bit strings. These subranges are then used to configure the TCAM's Global Mask Registers. Each subsequent lookup uses the key plus one of the Global Mask Registers to search for a matching filter among the filters whose index is in a given range. Our algorithm is similar to the MUD algorithm in the sense that it also encodes additional information in the TCAM entries. However, the information that we add is different and, we use it to improve the efficiency of filter set updates rather than to enable multi-match classification.

8.3 Algorithm

If enough empty entries are allocated between any two filters in a TCAM, then to insert a new filter, we can simply insert it in an appropriate empty entry, without moving any other filters. Although this is a tempting solution, there are two problems with it. First, in order to reduce TCAM power consumption, we prefer to store the filter set in as few TCAM segments as possible. Allocating empty entries between filters, makes it necessary to search more segments for a given filter set, increasing the power consumption. Second, because we cannot predict future updates, we cannot guarantee that there will always be an empty position in the TCAM where we need one. When there is no empty entry available, filter moves become necessary.

From this discussion, we identify two important objectives. First, we would like to store the filter set in a TCAM compactly without allocating empty entries between filters. This allows a linear growth of occupied entries and segments as the size of the filter set grows, reducing the power required for lookups. Second, we would like to minimize the movement of entries, so as to reduce the amount of work that must be done for each update and to minimize the impact of updates on lookup throughput.

8.3.1 Real Filter Priority

A filter's order in a filter set naturally reflects its priority, so the filter index can be used as its priority value. In fact, only overlapping filters need to be ordered relative to one another, in order to ensure the correctness of lookup results. Therefore, filters can be divided into groups in such a way that filters in the same group can exchange their order at will, without affecting the lookup results. The order of the groups, however, cannot be exchanged. To be specific, each group is assigned a priority value. The group of filters with a higher priority (i.e. a smaller priority value) must be stored in a lower address region of a TCAM than the group of filters with a lower priority (i.e. a larger priority value).

The algorithm for grouping the filters and assigning the priority values can be described as follows. We start from a graph in which each vertex denotes a filter.

For each filter, r_i, we examine all the other filters, r_j, which overlap with r_i (i.e. $r_i \cap r_j \neq \emptyset$). If $i > j$, we create a directed edge from r_j to r_i; otherwise, we create a directed edge from r_i to r_j. This step generates a directed acyclic graph. The topological order of the vertices in this graph reflects the relative priorities of filters. In the second step, we assign priority values to filters. Each vertex with no predecessors is assigned a priority value of zero. Other vertices are assigned a priority value only after all their predecessors have been assigned a priority value. The priority value assigned to a vertex is one plus the largest priority value assigned to any of its predecessors. An example of this process is shown in Figure 8.1.

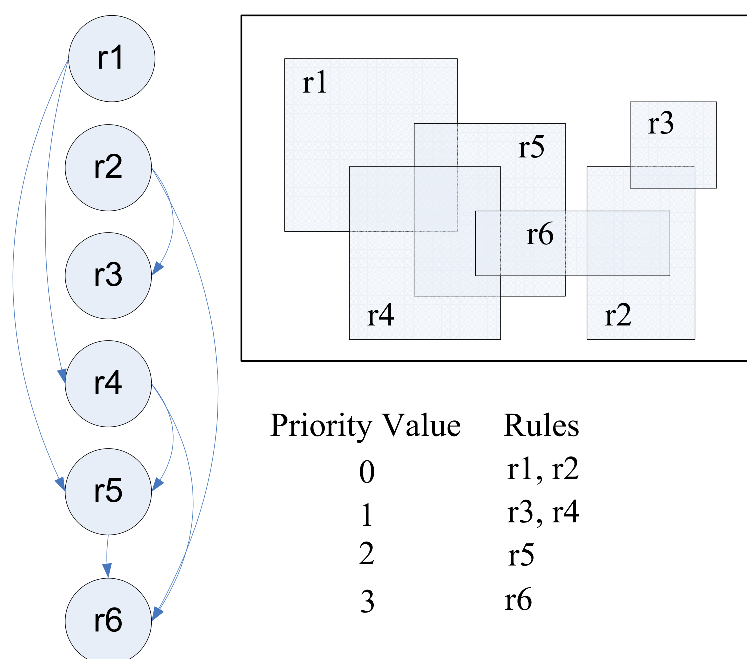

Figure 8.1: Grouping and Priority Value Assignment

We have evaluated real world filter sets and found that the number of distinct priority values needed is typically much smaller than the number of filters, as shown in Table 8.1. We have also evaluated large synthetic filter sets generated using the ClassBench [70] tool and found that the number of priority levels is insensitive to the number of filters. Even for filter sets with 10 thousand filters, the number of priority levels is less than 64. This property implies that if filters are updated based on their priority values, significantly less work needs to be done than if they are updated using their absolute position in the TCAM.

Table 8.1: Real Priority Levels in Real Filter Sets

filter set	# filters	# priorities
acl1	814	40
acl2	623	35
acl3	2,400	22
acl4	3,061	22
acl5	4,557	13
fw1	283	53
fw2	184	55
fw3	160	49
ipc1	1,702	42
ipc2	192	42

The filter grouping is analogous to the prefix grouping by the prefix lengths or the chain-ancestor ordering for LPM [55], but updating filters for general packet classification is much more complex than updating prefixes for LPM. First, the number of priority levels in filter sets for general packet classification is much more than the number of unique prefix lengths in prefix sets for IP lookup. Second, updating a prefix in a prefix length group does not affect any other prefixes. Therefore, the number of entry moves required is bounded by the number of unique prefix lengths. However, for general packet classification, updating a filter may change multiple filters' priority values. Figure 8.2 illustrates the grouping result after a new filter R, which has an index between $r1$ and $r2$, is inserted into the set. Notice that $r4$, $r5$, and $r6$ all have to change their priority values. This implies that after a new filter is inserted, the priority values of several others need to be adjusted to maintain a correct topological priority order. Similar actions need to be taken after a filter is removed or modified in the filter set.

If we apply the similar update algorithm used for LPM for packet classification, it can cause too many TCAM entry moves and the associated data memory updates. In the worst case, all filters need to change their priority values. Fortunately, in reality this is unlikely to happen. We will show this point through analysis and simulation.

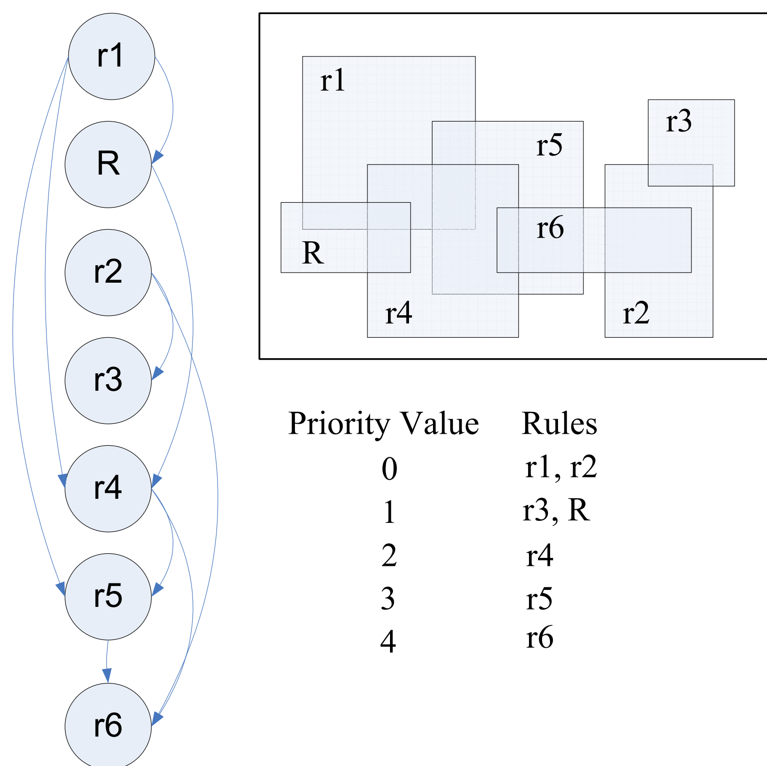

Figure 8.2: Effect of Inserting a New Filter R

8.3.2 Using Extended Filter

We have shown that a typical TCAM entry configuration results in some unused bits. Using a few of these unused bits, we attach the real priority value to each filter. Now if the packet classification always looks up the extended filters, the filters need not be stored in their priority order in a TCAM. Actually, a new filter can be written in any empty entry in a TCAM and no other filter needs to be moved. Clearly, now the search key has to also include the priority value. The lookup process is no longer looking for the matching filter with the minimum TCAM index but the matching filter with the minimum attached priority value. Without the prior knowledge of the priority, multiple lookup attempts are needed to figure out the best matching filter with the minimum priority value.

A linear search on the priority values does not scale to large filter sets with many priority levels. Fortunately, TCAMs have a set of reconfigurable Global Mask Registers (GMR) which can selectively mask out any bits in all entries as “don’t care”.

Each filter has a priority value with all bits enabled. By configuring GMR bits, we can determine which bits of the priority value should be considered as if each GMR enables a range of priority values. Each TCAM lookup therefore designates one of the preset GMRs to search only a range of priority values. The result narrows down the search range for the next lookup, and eventually identifies the best match. Typically, a TCAM has up to 64 GMRs. They are more than enough for our purpose.

Figure 8.3 illustrates an example where there are at most 32 priority values in the filter set. We only show the priority level, a part of the key, in the figure for simplicity. The binary decision tree is traversed based on the search result of the previous lookup. If the TCAM reports a match, we follow the upper branch of the tree; otherwise, we follow the lower branch of the tree. For example, given a packet header, we first search any matching filter with the priority value between zero and 15. If the result is positive, we then search any matching filter with the priority value between zero and seven; otherwise, we search the range eight to 11, and so forth. Since each lookup step halves the searched priority range, this scheme needs only $\log N$ lookups per packet to find the best matching filter, where N is the number of unique priority values.

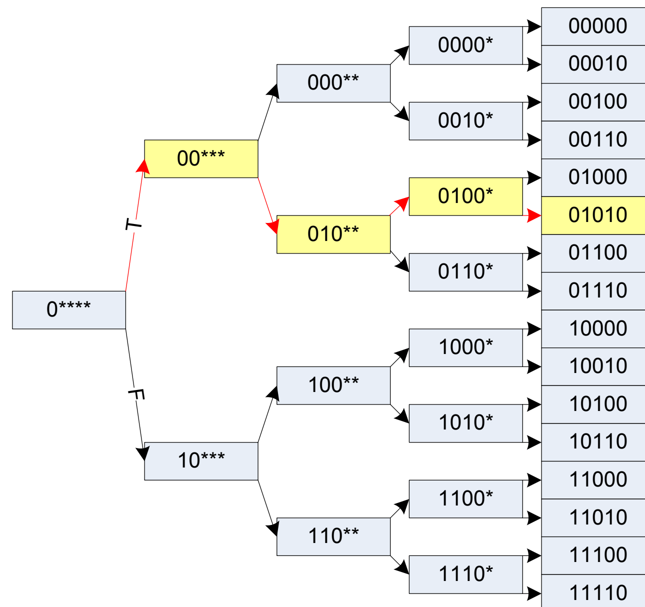

Figure 8.3: Searching for the Filter with the Minimum Priority Value

Note that in this scheme, the only information used is whether or not the TCAM reports a match. If we also know the matching filter's priority value, we can accelerate

the search. For example, if in the first attempt, we find a matching filter with a priority value of zero, then there is no further search needed. Even when the priority value is not zero, we can use the value to advance the search range quickly in the decision tree. For example, if the first search in the range of zero to 15 returns a match with the priority value of two, then in the next step we can directly search the range zero to one rather than zero to seven.

To achieve this, we store the filter's priority value in the associated data memory as a part of the filter's associated data. Each lookup step reads this value if there is a match in the TCAM. The control logic then uses this information to choose another GMR for the next lookup or terminates the lookup. Accessing the associated data memory is pipelined with the TCAM lookups, so the TCAM throughput is unchanged.

We also use another TCAM feature to help improve the lookup performance. Along with the matching filter index, the TCAM also has a multi-match output signal indicating if there is more than one matching filter for the given key. Since our search order is in favor of the higher priority filters, during the search, if the multi-match signal shows only one single match for the given key, the filter is guaranteed to be the best matching one. In such a case, no further search is needed.

8.3.3 Lookup

The lookup of a filter (*SearchTCAM*[*key*]) involves a sequence of recursive calls to the sub-procedure (*SearchPriorityRange*[*key, low, high*]) that searches a range of priority values using a GMR. In the following pseudo code, *IsMultiMatch* is asserted by the TCAM if more than one filters are matched. *Priority(i)* is filter i's priority value acquired from the associated data memory. *Low* and *high* define the priority value range which can be represented with a prefix string.

SearchTCAM [*key*]

1. $low = 0$
2. $high = 2^{\lceil \log_2 MaxPriority \rceil} - 1$
3. $SearchPriorityRange\ [key, low, high]$

SearchPriorityRange $[key, low, high]$

1. get filter index i
2. if $(i \neq NULL)$
3. if $(priority(i) = low$ OR ! $IsMultiMatch)$
4. return i as the best match
5. else if $(priority(i) \neq low$ AND $IsMultiMatch)$
6. $high = low + 2^{\lfloor \log_2 (priority(i) - low) \rfloor} - 1$
7. $regi = i$
8. $SearchPriorityRange\ [key, low, high]$
9. else
10. if (is the first TCAM lookup)
11. return $NULL$
12. else if $(low = high$ OR $priority(regi) = high + 1)$
13. return $regi$ as the best match
14. else
15. $low = high + 1$
16. $high = low + 2^{\lfloor \log_2 (priority(regi) - low) \rfloor} - 1$
17. $SearchPriorityRange\ [key, low, high]$

8.3.4 Update

The update process includes inserting, deleting, and modifying filters. All of these may result in multiple filters changing their priority values. Inserting a filter implies some filters need to increase their priority value, deleting a filter implies some filters need to decrease their priority value, and modifying a filter can do both. The analysis can be done through the DAG we built in Section 8.3.1.

An update involves a sequence of accesses in the TCAM and the associated data memory. By performing accesses in the proper order, we can do an update using the spare TCAM cycles without blocking the normal lookups. For example, to insert a new filter, we first get the set of filters that need to increase their priority value. We sort these filters in decreasing priority value order and then increase their priority value in turn. At last, we insert the new filter in any empty entry. Of course, for a better lookup performance, we should choose the best available entry for the new

filter. Ideally, among all the filters that overlap the new filter, those with smaller priority values should be located in the small indexed entries and those with larger priority values should be located in the large indexed entries.

8.4 Evaluation

8.4.1 Filter Distribution

The efficiency of our algorithm highly depends on the filter distributions. We have shown that even for very large filter sets, the number of unique priority values, which determines the worst-case performance bound, is small. We also examine the filter distributions in different priority value groups. An example is shown in Figure 8.4. We found that the majority of filters are concentrated in groups with small priority values. This fact has two favorable implications. First, it benefits the lookup process since a packet has a higher possibility to match a filter with small priority value and our search starts from the filters with small priority values. Second, it implies the long dependent chains comprise only a few filters; hence our update process will not affect too many filters.

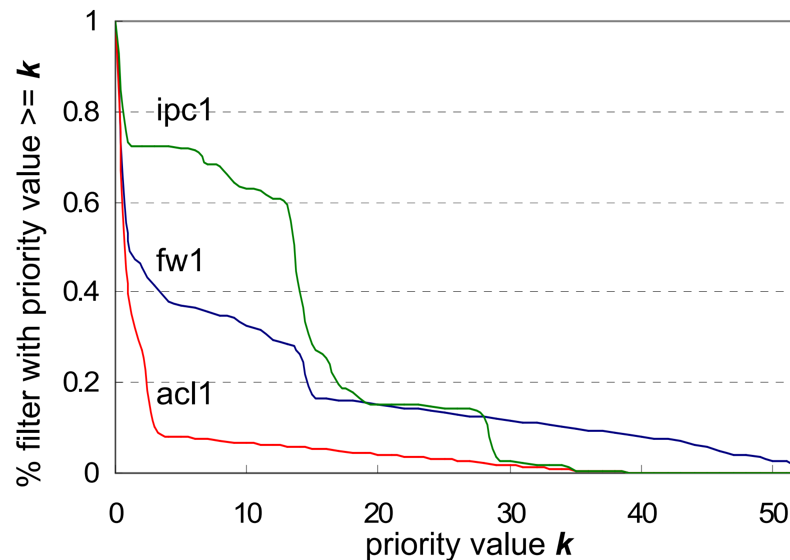

Figure 8.4: Priority Value Distribution for Real Filter Sets

Indeed, the priority dependency is a result of filter overlaps. If the maximum number of overlapped filters that a packet can match is small, our lookup and update algorithms both work better. In [40], 112 real filter sets are analyzed. In only one filter set does a packet match as many as eight filters. In the majority of filter sets, no packet matches more than five filters.

8.4.2 Lookup Throughput Performance

For each filter set, we generate a packet header trace using the ClassBench tools [70] to evaluate the lookup performance. We evaluate the worst-case lookup performance by storing all the filters in a TCAM in decreasing priority value order. The best case lookup performance happens when the filters are stored in increasing priority value order and the average case performance happens when the filters are randomly permuted in TCAM entries.

The simulation results for some filter sets are shown in Table 8.2. In the simulation, we assume that the TCAM runs at 250MHz clock rate. Note that for any case, a packet needs at most six TCAM accesses to find the best matching filter, so in the absolute worst case, the TCAM can still classify 42 million packets per second, which is sufficient for the OC-192 link speed.

Table 8.2: Lookup Throughput Performance

	# accesses			throughput (Mpkt/s)		
filter set	*best*	*average*	*worst*	*best*	*average*	*worst*
acl1 (814 filters)	1.38	2.65	3.20	181	94	78
fw1 (283 filters)	2.18	2.68	2.90	115	93	86
ipc1 (1,702 filters)	3.23	4.14	5.37	77	60	47
acl1_syn (4,415 filters)	2.08	2.97	4.66	120	84	54

8.4.3 Update Performance

The update performance is determined by the number of TCAM entry writes needed when inserting, deleting or modifying a filter. The worst-case update performance

happens when we insert the filters in the reversed priority order or delete the filters in priority order. To evaluate the worst-case update performance, we first reverse the filters' order as they appear in the original filter set, and then we insert the filters into the TCAM one by one. After each filter is inserted, we reevaluate all the filters' current priority value and count the number of filters that need to update their priority values. This number plus one more TCAM write that actually inserts the new filter is the overall number of TCAM writes needed for an update.

In Table 8.3, we show the average number of TCAM writes and the maximum number of TCAM writes needed after all the filters are inserted into a TCAM. We can see the average number of TCAM writes is small but the maximum number of TCAM writes can be very large. Figure 8.5 shows the cumulative distribution of the TCAM write numbers.

Table 8.3: The Worst-Case Update Performance

filter set	average # TCAM writes	maximum # TCAM writes
acl1 (814 filters)	3.2	110
fw1 (283 filters)	10.8	239
ipc1 (1,702 filters)	12.3	1,099

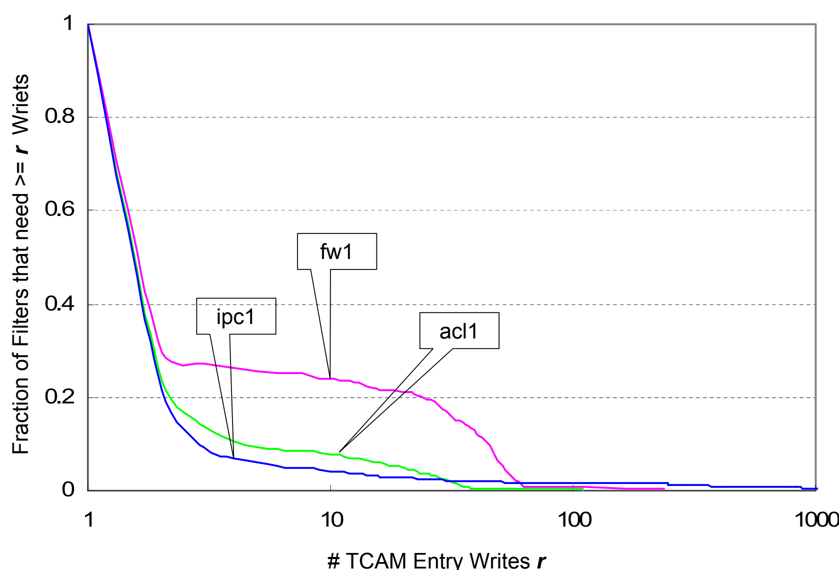

Figure 8.5: The Worst-Case Distribution of TCAM Accesses

These results further affirm us that the similar update algorithm used for LPM is not applicable for the general packet classification since too large number of memory

moves can be involved. On the other hand, our algorithm needs only to rewrite the portion of the extended filters that holds the priority value, which can be done very fast in general and does not need to block the normal lookup process.

8.5 Conclusion

In this chapter we present an algorithm that trades off the surplus search capability of TCAMs for efficient filter set updates for the general packet classification problem. The real priority values of filters are derived and attached to the filters. Using the binary search on the priority values and some other common features of TCAMs, the algorithm maintains a lookup throughput that is sufficient for backbone routers running at OC-192+ speeds. At the same time, that algorithm greatly reduces the work required for filter set management thus it is quite suitable for the dynamic environment where filter updates occur frequently.

Chapter 9

Summary

The work described in this dissertation focuses on the design and evaluation of high performance packet classification systems, which are needed to allow tomorrow's routers and switching systems to meet QoS and security challenges in a high speed environment.

9.1 Contributions

Since many surveys on existing packet classification algorithms are available today, in this dissertation we do not repeat the descriptions of the previous work. Instead, in Chapter 1 we summarize the previous work from a high level perspective and try to categorize the algorithms according to their basic approaches. This method has a clear advantage to help the researchers and designers avoid digging into the algorithm details while not losing sight of the big picture. Understanding the problem from a high level also provides insights that can lead to further improvements in the state of the art.

It is also important to understand the technical merit of each algorithm. We are often in a situation to ask which algorithm is indeed better or if we can use one for a particular application with reasonable confidence. Unfortunately, such questions are difficult to answer just based on published results. In Chapter 2, we describe an open-source project to address this problem. Most of the representative algorithms are actually implemented under uniform conditions and assumptions. The free available implementations allow others to easily adapt them for different scenarios. We

also enforce more consistent criteria for the algorithm evaluation so that their performance and potential are directly comparable. This project relieves researchers and designers from duplicating the previous work and helps them quickly evaluate algorithms for any application. We also encourage external contributions of new algorithm implementations and evaluations that can be incrementally added to the library in the same framework. We believe this will benefit the research and design community as a whole.

Based on the insights gained from the previous work, we design several new algorithms to tackle different variations of the packet classification algorithms.

The *Shape Shifting Tries* (SST) presented in Chapter 3 is a trie-based *Longest Prefix Matching* (LPM) algorithm. The algorithm takes full advantage of the sparsity of the underlying binary trie and uses an efficient encoding technique. It outperforms the well-known Tree Bitmap algorithm and is particularly attractive for IPv6 route lookup and for use as a building block in general packet classification algorithms.

Chapter 4 presents a special hash data structure, *Fast Hash Table* (FHT), and the associated maintenance and search algorithms. The FHT extends the classical Bloom Filter data structure to support exact match. In addition to playing its original role as a match filter, the front-end Bloom Filter also acts as as a multi-hash load-balancing mechanism, which is proved to be able to lower the hash collision probability by orders of magnitude. The FHT is ideal for use in exact match packet classification where high throughput and predicability are crucial.

Combining the trie-based and the hash-based LPM algorithms and applying the techniques developed in the previous two chapters, we derive a flexible and high performance LPM algorithm in Chapter 5. The algorithm allows a tradeoff between storage and throughput. It can support very fast lookups on average and meanwhile the worst-case performance is no worse than the trie-based algorithms.

2D packet classification is a special case of general packet classification where each filter is defined as a prefix pair. Chapter 6 presents a new way to solve this problem. It can be seen as a tradeoff between the Tuple Space Searching algorithm, which is slow but memory efficient, and the Cross-producting algorithm, which is fast but

memory inefficient. The essence of the idea can be used to extend the algorithm to general packet classification.

Chapter 7 presents a decision-tree based algorithm, the *Adaptive Binary Cuttings* (ABC), for general packet classification. The ABC algorithm takes the filter distribution into account when constructing the decision tree and realizes the decision tree using a technique similar to the one used in the SST algorithm. Following the same central idea, the algorithm comes with three different styles, each with its own advantages and disadvantages. The ABC algorithm significantly outperforms the previous decision-tree based algorithms, such as HiCuts, HyperCuts, and Woo's algorithm.

TCAMs are widely used in packet classification systems. However, TCAM also faces a lot of challenges when used for general packet classification. Chapter 8 contributes to this field by tackling the little studied researched problem of dynamic filter set management. We trade off the surplus search capability of the TCAM for a fast and simple filter set update process, which is suitable for a highly dynamic environment.

9.2 Future Directions

It is generally believed that packet classification is a well understood and researched problem. The vast body of previous work is a dauntingly high hurdle for anyone who wants to address it. However, as the Internet keeps evolving, one thing is known for sure: we are still far from a point when we can label the packet classification problem as "solved".

Fortunately, through the course of our study and beyond the attainment we have accomplished, we can still spot plenty of opportunities to push the research forward. For example, our algorithm evaluation is incomplete in some sense. It is not an exhaustive collection of all known algorithms. So far the implementations provided are only behavior models for the purpose of simulation. We prefer "real" implementations that can be directly implanted into ASICs, FPGAs, and network processors for real performance evaluation and direct application. Moreover, we expect to acquire more real filter sets from industry and refine the ClassBench tool to better predict the future evolution of large scale filter sets for all kinds of applications.

For the development of new algorithms, in addition to making continuing efforts to understand the existing algorithms, we should start to closely examine the filter sets and try to answer the question: What factors contribute to the deterioration of algorithm performance and how can we avoid their effects? For example, in the decision tree-based algorithms, we find that a very small fraction of the filters contribute significantly to the memory consumption, so it is reasonable to use a hybrid algorithm to handle these filters separately. A related question from another perspective is: Can we construct the filter set in a more structured way so we can make the algorithms perform better? This is a question that needs to be answered by network operators and system designers together.

Based on our observation of the existing algorithms, we find that a single algorithm can never handle all the scenarios equally well. In addition to making more tradeoffs and introducing more degrees of freedom as we have presented in this dissertation, it makes sense to develop some hybrid algorithms that leverage the strength of different approaches.

It is our intent to provide a solid foundation to encourage further investigations on design and evaluation of packet classification systems. All our efforts lead to this direction and we expect the next breakthrough is on the horizon.

References

[1] BGP Reports. In *http://bgp.potaroo.net/*.

[2] Cisco netflow. *http://www.cisco.com/warp/public/732/Tech/netflow*.

[3] IDT Network Search Engines. In *http://www.idt.com/products*.

[4] IPv6 Address Allocation and Assignment Policy (APNIC). In *http://www.apnic.net/docs/policy/ipv6-address-policy.html*.

[5] Snort - The Open Source Network Intrusion Detection System. *http://www.snort.org*.

[6] ClassBench Web Site. In *http: //www.arl.wustl.edu /d̃et3 /ClassBench*, 2005.

[7] Packet Classificaton Evaluation Web Site. In *http: //www.arl.wustl.edu /h̃s1 /PClassEval.html*, 2006.

[8] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of 26th ACM Symposium on the Theory of Computing*, 1994.

[9] Florin Baboescu, Sumeet Singh, and George Varghese. Packet classification for core routers: Is there an alternative to CAMs? In *IEEE INFOCOM*, 2003.

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

[11] Burton Bloom. Space/Time Trade-offs in Hash Coding With Allowable Errors. *Communications of the ACM*, 13, July 1970.

[12] A. Broder and A. Karlin. Multilevel adaptive hashing. In *Proceedings of 1st ACM-SIAM Symposium on Discrete Algorithm*, 1990.

[13] Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. In *Proceedings of IEEE INFOCOM*, 2001.

[14] M.M. Buddhikot, S. Suri, and M. Waldvogel. Space Decomposition Techniques for Fast Layer-4 Switching. In *Conference on Protocols for High Speed Networks*, 1999.

[15] L. Carter and M. Wegman. Universal classes of hashing functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[16] Francis Chang, Wu chang Feng, and Kang Li. Approximate caches for packet classification. In *Proceedings of IEEE Infocom*, 2004.

[17] M. M. I. Chvets. Multi-zone caches for accelerating ip routing table lookups. In *Proceedings of High-Performance Switching and Routing*, 2002.

[18] Edith Cohen and Carsten Lund. Packet Classification in Large ISPs: Design and Evaluation of Decision Tree Classifiers. In *ACM SIGMETRICS/Performance*, 2005.

[19] Jason Cong and Yuzheng Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimaization in Lookup-Table Based FPGA Designs. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1992.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd Edition*. The MIT Press, 2001.

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

[22] Madhav Desai, Ritu Gupta, Abhay Karandikar, Kshitiz Saxena, and Vinayak Samant. Reconfigurable Finite-State Machine Based IP Lookup Engine for High-Speed Router. *IEEE Journal on Selected Areas in Communications*, 21, May 2003.

[23] Sarang Dharmapurikar, P. Krishnamurthy, and Dave Taylor. Longest Prefix Matching using Bloom Filters. In *ACM SIGCOMM*, August 2003.

[24] 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.

[25] Sarang Dharmapurikar and Vern Paxson. Robust TCP stream reassembly in the presence of adversaries. In *USENIX Security Symposium*, August 2005.

[26] Will Eatherton. Fast IP Lookup Using Tree Bitmap. *Washington University Master Thesis*, 1999.

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

[28] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a Better NetFlow. In *ACM SIGCOMM*, August 2004.

[29] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8, March 2000.

[30] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *IEEE INFOCOM*, 2000.

[31] Robert J. Francis, Jonathan Rose, and Kevin Chung. Chortle: A Technology Mapping Program for Lookup Table-based Field Programmable Gate Arrays. In *27th Annual ACM/IEEE Conference on Design Automation*, 1991.

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

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

[34] Jahangir Hasan and T. N. Vijaykumar. Dynamic Pipelining: Making IP Lookup Truly Scalable. In *ACM SIGCOMM*, 2005.

[35] HDL Design House. HCR_MD5: MD5 crypto core family, December, 2002.

[36] Intel Corporation. Intel IXP2800 Network Processor. Datasheet, 2002.

[37] G. Jacobson. Succinct Static Data Structure. *Carnegie Mellon University Ph.D Thesis*, 1988.

[38] Abhishek Kumar, Jun Xu, Li Li, and Jia Wang. Space-code bloom filter for efficient traffic flow measurement. In *Internet Measurement Conference*, 2003.

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

[40] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for Advanced Packet Classification with Ternary CAMs. In *ACM SIGCOMM*, 2005.

[41] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups Using Multiway and Multicolumn Search. *IEEE/ACM Transactions on Networking*, 7, 1999.

[42] Butler Lampson, Venkatachary Srinivasan, and George Varghese. IP Lookups Using Multiway and Multicolumn Search. *IEEE/ACM Transactions on Networking*, 7, June 1999.

[43] K. Li, F. Chang, D. Berger, and W. Chang Fang. Architectures for packet classification caching. In *Proceedings of IEEE ICON*, 2003.

[44] Huan Liu. Efficient Mapping of Range Classifier into Ternary-CAM. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, August 2002.

[45] Huan Liu. Routing Table Compaction in Ternary CAM. *IEEE Micro*, 22, January 2002.

[46] J. Lunteren and T. Engbersen. Fast and Scalable Packet Classification. *IEEE Journal on Selected Areas in Communications*, 21, May 2003.

[47] Jan Van Lunteren. Searching Very Large Routing Tables in Wide Embedded Memory. In *IEEE Globecom*, 2001.

[48] Harsha Narayan, Ramesh Govindan, and George Varghese. The Impact of Address Allocation and Routing on the Structure and Implementation of Routing Table. In *ACM SIGCOMM*, 2003.

[49] M.H. Overmars and A.F. van der Stappen. Range Searching and Point Location Among Fat Objects. *Journal of Algorithms*, 21, 1996.

[50] Vern Paxson. Bro: A System for Detecting Network Intruders in Real Time. *Computer Networks*, December 1999.

[51] 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.

[52] Sartaj Sahni and Kun Suk Kim. Efficient Construction of Multibit Tries for IP Lookup. *IEEE/ACM Transactions on Networking*, 11, August 2003.

[53] Rama Sangireddy, Natsuhiko Futamura, Srinivas Aluru, and Arun K. Somani. Scalable, Memory Efficient, High-Speed IP Lookup Algorithms. *IEEE/ACM Transactions on Networking*, 13, August 2005.

[54] David V. Schuehler, James Moscola, and John W. Lockwood. Architecture for a hardware-based TCP/IP content scanning system. In *IEEE Symposium on High Performance Interconnects (HotI)*, Stanford, CA, August 2003.

[55] Devavrat Shah and Pankaj Gupta. Fast Incremental Updates on Ternary-CAMs for Routing Lookups and Packet Classification. In *Hot Interconnects 8*, 2000.

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

[57] K. Sklower. A Tree-based Routing Table for Berkeley Unix. In *Winter Usenix Conference*, Dallas, TX, 1991.

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

[59] 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.

[60] Haoyu Song and John Lockwood. Efficient Packet Classification for Network Intrusion Detection using FPGA. In *International Syposium on Field-Programmable Gate Arrays (FPGA)*, 2005.

[61] R. Souza, P. Krishnakumar, C. Ozveren, R. Simcoe, B. Spinney, R. Thomas, and R. Walsh. GIGASwitch: A High-Peformance Packet Switching Platform. *Digital Tehnical Journal*, 1994.

[62] Ed Spitznagel, David Taylor, and Jonathan Turner. Packet Classification Using Extended TCAMs. In *IEEE International Conference on Network Protocols (ICNP)*, 2003.

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

[64] V. Srinivasan and George Varghese. Fast Address Lookups using Controlled Prefix Expansion. *ACM Transaction on Computer Systems*, 17, February 1999.

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

[66] David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar, John Lockwood, Wenjing Tang, and Jonathan Turner. System-on-Chip Packet Processor for an Experimental Network Services Platform. In *Proceedings of IEEE Globecom*, 2003.

[67] David Taylor, John Lockwood, Todd Sproull, Jon Turner, and David Parlour. Scalable IP Lookup for Programmable Routers. In *IEEE INFOCOM*, June 2002.

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

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

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

[71] 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.

[72] Henry Hong-Yi Tzeng. Longest Prefix Search Using Compressed Trees. In *Proceedings of IEEE Global Communication Conference*, July 1998.

[73] B. Vocking. How asymmetry helps load balancing. In *Proceedings of 40th IEEE Symposium on Foundations of Computer Science*, 1999.

[74] M. Waldvogel, George Varghese, Jon Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM*, 1997.

[75] Mei Wang, Stephen Deering, Tony Hain, and Larry Dunn. Non-Random Generator for IPv6 Tables. In *12th Annual IEEE Symposium on High Performance Interconnects*, August 2004.

[76] Zhijun Wang, Hao Che, Mohan Kumar, and Sajal Das. CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking. *IEEE Transactions on Computer*, 53, December 2004.

[77] Priyank Warkhede, Subhash Suri, and George Varghese. Fast Packet Classification for Two-Dimensional Conflict-Free Filters. In *IEEE INFOCOM*, 2001.

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

[79] Xilinx Inc. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, November, 2004.

[80] Fang Yu and Randy H. Katz. Efficient Multi-Match Packet Classification with TCAM. In *Hot Interconnects*, 2004.

[81] 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 *Symposium on Architectures for Networking and Communications Systems*, 2005.

[82] Francis Zane, Girija Narlikar, and Anindya Basu. CoolCAMs: Power-efficient TCAMs for Forwarding Engines. In *IEEE INFOCOM*, 2003.

[83] Kai Zheng, Hao Che, Zhijun Wang, and Bin Liu. TCAM-Based Distributed Parallel Packet Classification Algorithm with Range-Matching Solutin. In *IEEE INFOCOM*, 2005.

Vita

Haoyu Song

Date of Birth	July 26, 1973
Place of Birth	Da Tong, P.R.China
Degrees	B.E. Electronics Engineering, July 1997 M.S. Computer Engineering, August 2003 D.Sc. Computer Engineering, September 2006
Professional Societies	Association for Computing Machines (ACM) Institute of Electrical and Electronics Engineers (IEEE)

Publications

Haoyu Song, Jonathan Turner, Fast Filter Updates for Packet Classification using TCAM, In Proceedings of IEEE Globecom, San Francisco, CA, November 28-December 2, 2006.

Haoyu Song, John Lockwood, Multi-pattern Signature Matching for Hardware Network Intrusion Detection Systems, In Proceedings of IEEE Globecom, St. Louis, MO, November 28 - December 2, 2005.

Haoyu Song, Jonathan Turner, John Lockwood, Shape Shifting Tries for Faster IP Route Lookup, In Proceedings of 13th IEEE International Conference on Network Protocols (ICNP), Boston, MA, November 6-9, 2005.

Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, John Lockwood, Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing, In Proceedings of ACM SIGCOMM, Philadelphia, PA, August 22-26, 2005.

Haoyu Song, Todd Sproull, Michael Attig, John Lockwood, Snort Offloader: A Reconfigurable Hardware NIDS Filter, In

Proceedings of 15th International Conference on Field Programmable Logic and Applications (FPL), Tampere, Finland, August 24-26, 2005.

Haoyu Song, John Lockwood, Efficient Packet Classification for Network Intrusion Detection Using FPGA, In Proceedings of 13th IEEE International Symposium on Field-programmable Gate Array (FPGA), Monterey, CA, F February 20-22, 2005.

Haoyu Song, Jing Lu, John Lockwood, James Moscola, Secure Remote Control of Field-programmable Machines (FCCM), Napa, CA, April 20-23, 2004.

Haoyu Song, Secure Remote Control and Configuration of FPX Platform in Gigabit Ethernet Environment, Technical Report of the Department of Computer Science and Engineering, Washington University in St. Louis, August 2003.

September 2006

Short Title: Packet Classification Systems Song, D.Sc. 2006