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

ABSTRACT

HIGH PERFORMANCE PACKET CLASSIFICATION

by Edward W. Spitznagel

ADVISOR: Professor Jonathan S. Turner

December 2005
Saint Louis, Missouri

High-performance packet classification is crucial for a multitude of emerging network services. Diffserv edge routers, firewalls, intrusion-detection devices and many QoS-enabled routers need to classify packets to determine what to do with them. With link rates of 10 Gb/s and higher this becomes a difficult task at best, and packet classification becomes a bottleneck. This dissertation addresses that problem by providing packet classification techniques with increased scalability. The contributions of this work include a new TCAM architecture called Extended TCAMs, a refinement of that approach (inspired in part by Parallel Packet Classification) and a method for reducing the size of data structures used in Recursive Flow Classification. Extended TCAMs are studied in detail and are shown to allow packet classification using over 100,000 rules at the same rate as a standard TCAM while reducing power requirements by a factor of ten to twenty.

copyright by

Edward W. Spitznagel

2005

to my family

Contents

List of Tables . . . . **vii**

List of Figures . . . . **ix**

Acknowledgments . . . . **xiii**

1 Introduction **1**
- 1.1 Internetworking Background 1
- 1.2 Current Trends And The Need For Packet Classification 4
- 1.3 The Packet Classification Problem 5
- 1.4 Metrics For Evaluating Classification Methods 7
- 1.5 Outline of Dissertation 8

2 The State of the Art **9**
- 2.1 Ternary Content-Addressable Memories (TCAMs) 9
- 2.2 Recursive Flow Classification (RFC) 11
  - 2.2.1 Equivalence Classes in RFC 12
  - 2.2.2 Use of Crossproducting 14
  - 2.2.3 Extending to k Dimensions 18
- 2.3 Parallel Packet Classification 22
- 2.4 Summary of Additional Related Work 26
  - 2.4.1 Classification in One Dimension 26
  - 2.4.2 Classification in Two Dimensions 27
  - 2.4.3 General Multi-dimensional Classification 27
  - 2.4.4 Packet Classification Repository 29
- 2.5 The ClassBench Benchmark 29

3 Compressed Data Structures for RFC **30**
3.1 Compression of Data Structures 30
3.1.1 Simple Compression Algorithm 31
3.1.2 Results 32
3.2 Improving Compression 35
3.2.1 TSP Heuristic 37
3.2.2 Results 37
3.3 Performance on Larger Databases 41
3.4 Choice of Reduction Trees 42
3.4.1 Impact of Reduction Tree Selection 43
3.4.2 Finding an Optimal Reduction Tree 43
3.4.3 Results 46
3.5 Summary 46

4 Extended TCAMs **48**
4.1 Overview 48
4.2 Partitioning of the TCAM 49
4.2.1 Region-Splitting Algorithm 51
4.2.2 Region-Splitting Results 58
4.2.3 Trie-Carving Algorithm 62
4.2.4 Trie-Carving Results 69
4.3 Range Check Hardware 72
4.3.1 Implementation with four inter-stage signals 73
4.3.2 Implementation with three inter-stage signals 75
4.3.3 Revised implementation with three inter-stage signals 77
4.3.4 Subfield Chaining 79
4.3.5 Summary of Range Check Hardware 83
4.4 Combined Results 83
4.5 Handling Filter Updates 86
4.5.1 Trie-Carving Update Algorithm 87
4.5.2 Results 90
4.6 Multilevel Indexing 97
4.6.1 Using Region-Splitting Algorithm 98
4.6.2 Using Trie-Carving Algorithm 101
4.7 Summary 103

5 Partitioned Encoded Search of TCAMs **104**
5.1 PEST Architecture 104
5.1.1 Encoding of Field Values 106
5.1.2 Partitioning of TCAM 108
5.2 TCAM Update Technique 116
5.3 Evaluation and Results 118
5.4 Lower Bounds on Encoding Width 125
5.4.1 Initial Lower Bounds on $\mathrm{P^2C}$ Encoding Widths 125
5.4.2 Generalized Lower Bound 128
5.4.3 Evaluation of $\mathrm{P^2C}$ Encoding Results 132
5.5 Summary 134

6 Conclusion **136**
6.1 Contributions 136
6.2 Future Directions 138

References **139**

Vita **144**

List of Tables

1.1 Example: simplified firewall filter database 6

2.1 Example 1-dimensional filter database 12
2.2 Equivalence sets for the 1-D example 13
2.3 Lookup table for the 1-D example 13
2.4 Best matching filter for each equivalence class (1-D example) 14
2.5 Example 2-dimensional filter database 15
2.6 Equivalence sets for destination address (2-D example) 15
2.7 Equivalence sets for source address (2-D example) 15
2.8 Lookup table for destination address (2-D example) 17
2.9 Lookup table for source address (2-D example) 17
2.10 Equivalence classes for crossproducting table (2-D example) 18
2.11 2-dimensional crossproducting table (2-D example) 18
2.12 An example set of chunks for a typical 5-tuple classifier. 20
2.13 Example encodings for match conditions (filter fields) 25
2.14 Encoding widths for some real-world filter databases, using P^2C encoding style I 26

3.1 Chunks used in evaluating RFC compression scheme 33
3.2 Experimental compression results 34
3.3 Two-dimensional crossproducting table with poor compression 36
3.4 Two-dimensional crossproducting table with better compression . . . 36
3.5 Experimental TSP-heuristic compression results 38
3.6 Results from optimal reduction tree selection for real-world filter sets 47

4.1 Example set of filters 51
4.2 Example set of filters 62

5.1 Example set of filters 106
5.2 Encoding for source addresses in example rules 107
5.3 Encoding for destination addresses in example rules 107
5.4 Encoded representation of example filters 108
5.5 Filters for partitioning example 109
5.6 TCAM width required, in bits 119
5.7 Lower bound on TCAM width required 119
5.8 Layers required for encoding fields 120
5.9 Results of expanding ranges for standard TCAM representation . . . 121
5.10 Block sizes used for partitioned TCAM 122
5.11 Encoding width results for Source Address field 133
5.12 Encoding width results for Destination Address field 134

List of Figures

1.1 Internet Protocol packet structure 2
1.2 Internetworking example 3

2.1 Conventional 6-transistor SRAM storage cell 10
2.2 TCAM cell (details of SRAM storage cells omitted) 10
2.3 Filters projected onto an axis (1-D example) 12
2.4 Filters projected onto axes (2-D example) 16
2.5 Reduction of three equivalence class identifiers to one 20
2.6 An example reduction tree for classifying on three fields 21
2.7 Simplified representation of example reduction tree 21
2.8 An example reduction tree for a typical 5-tuple classifier 22
2.9 P^2C lookup process 23
2.10 Example primitive range hierarchies 24

3.1 Example structure of RFC classifier. 31
3.2 Example representations of a two-dimensional array. 32
3.3 Overall compression ratios from experimental results. 35
3.4 Compression ratios for crossproducting tables only, in experimental results. 35
3.5 Overall TSP compression ratios from experimental results. 39
3.6 TSP compression ratios for crossproducting tables only, in experimental results. 39
3.7 TSP compression ratio relative to simple compression, in terms of overall data structure size. 40
3.8 TSP compression ratio, relative to simple compression, in terms of size of crossproducting tables. 41
3.9 Compression Efficiency vs. Number of Filters. 42

4.1 An example of filters and index filters in an Extended TCAM 50
4.2 Example filters plotted in two dimensions 52
4.3 First split performed in region-splitting example 53
4.4 Second split performed in region-splitting example 54
4.5 Second phase of region-splitting example 55
4.6 Contents of Extended TCAM for region-splitting example 56
4.7 Power fraction results for different filter storage block sizes 60
4.8 Power fraction results for region-splitting algorithm 60
4.9 Storage complexity results for region-splitting algorithm 61
4.10 Source address trie for example filter set 63
4.11 Destination address trie for example filter set 63
4.12 First carving step for source address trie 65
4.13 Second carving step for source address trie 66
4.14 Third carving step for source address trie 66
4.15 Fourth carving step for source address trie 67
4.16 First carving step for destination address trie 67
4.17 Second carving step for destination address trie 68
4.18 Extended TCAM contents for trie-carving example 69
4.19 Power fraction results for different filter storage block sizes 70
4.20 Power fraction results for trie-carving algorithm 71
4.21 Storage complexity results for trie-carving algorithm 71
4.22 Iterative structure of range check circuit 73
4.23 Range-check sub-circuit for upper bound comparison 74
4.24 Range-check sub-circuit for lower bound comparison 74
4.25 Iterative structure of range check circuit 75
4.26 Range-check sub-circuit using three inter-stage signals 77
4.27 Iterative structure of revised range check circuit 77
4.28 Sub-circuit for *out-of-range* signal 79
4.29 Sub-circuit for *equals-hi-if-in-range* signal 80
4.30 Sub-circuit for *equals-lo-if-in-range* signal 80
4.31 Sub-circuit for *out-of-range* signal for input to stage $si - 1$ 81
4.32 Sub-circuit for *equals-hi-if-in-range* signal for input to stage $si - 1$ . . 82
4.33 Sub-circuit for *equals-lo-if-in-range* signal for input to stage $si - 1$ . . 82
4.34 Power fraction results, including effects of range support 84
4.35 Storage complexity results, including effects of range support 85

4.36 Power fraction after switching to the final filter set 91
4.37 Storage complexity after switching to the final filter set 92
4.38 Average rate of update processing in software on 700 MHz Pentium III 93
4.39 Average number of TCAM writes needed per update 94
4.40 Maximum number of TCAM writes needed for an update 95
4.41 Time history of performance for updating from 2,000 filter acl1 to fw1 96
4.42 Example of a two-level indexing structure 97
4.43 Effects of varying storage block size, for region-splitting with a two-level index 99
4.44 Power fraction for region-splitting with a two-level index 99
4.45 Storage complexity for region-splitting with a two-level index 99
4.46 Power fraction for region-splitting with a two-level index, including effects of range-check circuit 100
4.47 Storage complexity for region-splitting with a two-level index, including effects of range-check circuit 100
4.48 Effects of varying storage block size, for trie-carving with a two-level index 101
4.49 Power fraction for trie-carving with a two-level index 102
4.50 Storage complexity for trie-carving with a two-level index 102
4.51 Power fraction for trie-carving with a two-level index, including effects of range-check circuit 102
4.52 Storage complexity for trie-carving with a two-level index, including effects of range-check circuit 102

5.1 PEST lookup architecture. 105
5.2 Destination address primitive range hierarchy from example 107
5.3 Filters for partitioning example 110
5.4 Partitioning after one step 111
5.5 Partitioning after two steps 111
5.6 Partitioning during second phase 112
5.7 Extended TCAM contents for simplified partitioning example 112
5.8 Extended TCAM contents for example 116
5.9 TCAM power fraction resulting from partitioning 122
5.10 Storage complexity factor resulting from partitioning 123
5.11 Overall TCAM power fraction 124

5.12 Overall TCAM storage complexity 124

Acknowledgments

I would like to express my sincere gratitude to the many people who have made this dissertation possible.

First and foremost I would like to thank my advisor Jon Turner for all of his guidance in my scholarly endeavors and for encouraging me to be unafraid to reach my full potential. I would also like to thank Sally Goldman, Sergey Gorinsky, Joseph O'Sullivan and George Varghese for all the guidance and constructive input they have provided as members of my dissertation committee. I would furthermore like to thank Subhash Suri for his assistance with my early research in packet classification.

I would like to thank all the professors who have been a part of my education, and I would like to thank all the staff members who have made it possible; without them, I would never have made it this far. In particular I would like to acknowledge Ken Wong, John DeHart, and W. David Richard, among the many contributors to my experiences at Washington University.

For countless insightful discussions and for the wonderful camaraderie found in the ARL back hallway, I would like to thank Anshul Kantawala, Ralph Keller, Ruibiao Qiu, Jai Ramamirtham, Sherlia Shi, Prashanth Pappu, Sumi Choi, Samphel Norden, Sarang Dharmapurikar, Dave Lim, Todd Sproull, and all the other students with whom I have interacted over the years. I would like to thank David Taylor in particular for many collaborations in the field of packet classification; I am also grateful for having Tilman Wolf as an exceptionally helpful and dependable officemate.

Many thanks also go to Chuck Cranor, for encouraging me many years ago to select computer science as a field of study, and for encouraging me to pursue graduate school after completion of my undergraduate education.

Finally, I would like to thank my parents, for everything they have done to support and encourage me, and for teaching me the joys of learning.

Edward W. Spitznagel

Washington University in Saint Louis
December 2005

Chapter 1

Introduction

In recent years, the Internet has become an incredible success story, and new applications (such as video on demand, Internet telephony, and programmable networking) promise to provide an even richer, more useful experience to Internet users. Many of these new applications, however, require classification of packets at certain points in the network. This is already a difficult task at best, at current network speeds of 10 Gb/s and 40 Gb/s, and it will become harder as link rates increase. In this dissertation we seek to design packet classification techniques which can support emerging applications at high network speeds.

1.1 Internetworking Background

In order to set the stage for the packet classification problem, let us begin with a brief explanation of internetworking with packet-switched networks.

A network consists of a set of *hosts* and the *links* connecting them. A host might be a workstation, server, or a personal computer; it could also be a laptop, PDA, webcam, network-enabled video monitor, or an Internet-enabled cellular telephone. Links may be wires, fiber optic cables, or radio signals; the links carry signals (e.g. in the case of wires, the signal may be the presence of a certain voltage level on a wire, relative to a reference voltage on another wire) representing data.

The Internet is a packet-switched network, which means its functionality is based on the sending and receiving of *packets* of data. When a host needs to send data to another host, it splits the data into pieces of reasonable size and puts each piece into a packet. The host then sends these packets on the network.

A packet consists of a set of header fields and the data contained within the packet. The header fields are used to store information about the packet, such as the address of the host that sent the packet, the address of the host to which the packet should be delivered, and information regarding how the packet should be processed when it arrives. For example, the format of header fields currently used in the Internet is shown in Figure 1.1. These fields are fully described in the Internet Protocol version 4 (IPv4) [34] specification.

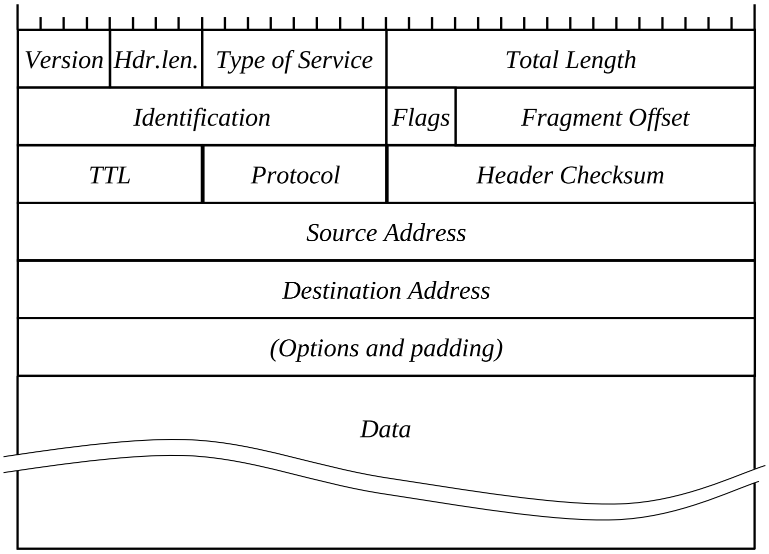


Figure 1.1: Internet Protocol packet structure

The Internet is composed of a large number of networks connected to each other by devices called *routers*. A router is a device which is connected to two or more networks and can forward packets from one network to another. That is, a packet can be sent from one network to a different network by sending it to a router connecting the two. Additional routers can continue to send the packet to further and further networks on its journey.

Figure 1.2 shows an example where three networks A, B, and C with various hosts are connected by means of the two routers R1 and R2. Let us consider what must happen to enable the host H1 to be able to send packets to host H2. H1 sends packets out on Network A, because that is the network to which is it attached. Router R1 must receive the packets and send them out on Network B, so that Router R2 can receive the packets and send them to H2 via Network C.

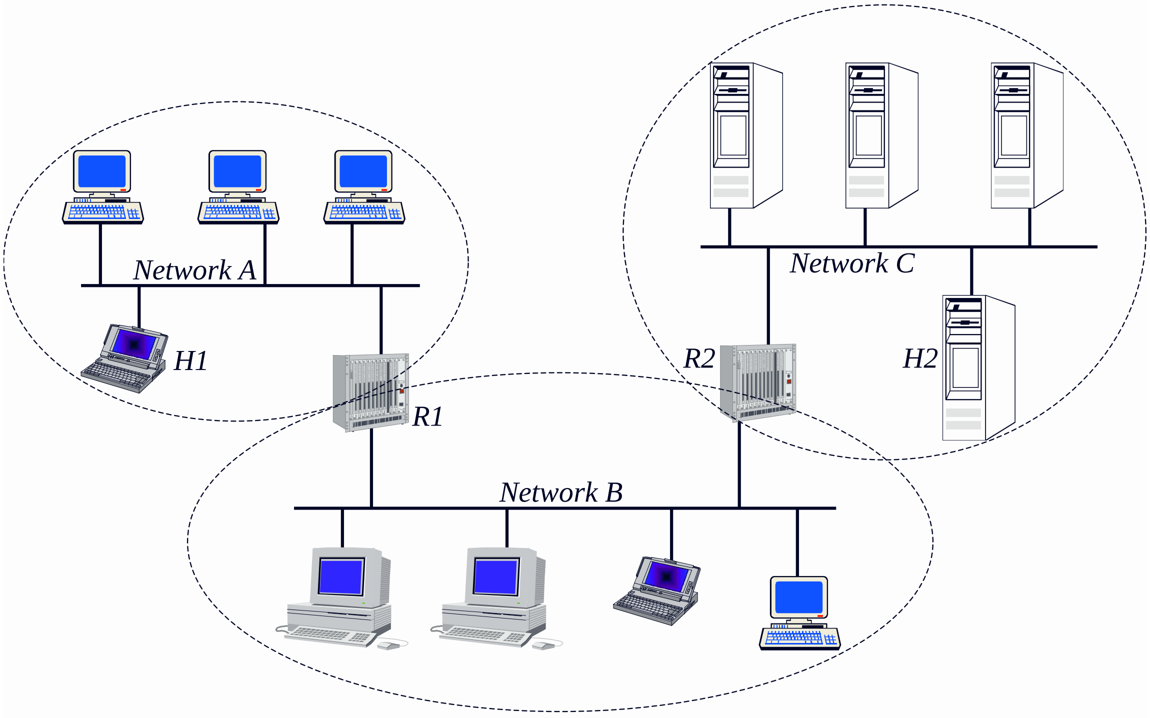


Figure 1.2: Internetworking example

In order to send packets towards the proper destinations, a router looks at the destination address contained in the packet's header. This address is a 32-bit quantity. In written text it is often expressed in dotted quad format, consisting of four decimal numbers (each representing 8 bits of the address) joined by periods. For example, 128.252.169.16 is the dotted-quad format for the 32-bit binary address 10000000111111001010100100010000.

Host addresses in the Internet are typically assigned so that all hosts on the same network have a common prefix in their address; thus routers need not keep track of the location of every host on the Internet (which obviously would not scale well). Instead, they use a set of prefixes to determine the network towards which the packet should be sent, based on the destination address in the packet. The hosts on a particular subnetwork belonging to the Washington University Computer and Communications Research Center all have addresses of the form 128.252.169.*, for example, where the asterisk (*) is used as a wildcard character.

Thus, as part of their normal operation, routers are required to match the destination addresses of incoming packets against a set of address prefixes. Each of these prefixes is associated with information about how to reach the network corresponding to that prefix. In the case where more than one prefix matches a destination address, the most specific (i.e. longest) prefix is preferred. This act of finding the longest matching prefix for a destination address is sometimes called a *routing lookup*.

The basic idea behind packet classification is the use of several of the packet header fields to identify packets belonging to particular classes or flows of interest. Most IPv4 packet classifiers use the *Source Address* field (indicating the IP address of the sending host), the *Destination Address* field (indicating the IP address of the destination host) and the *protocol* field. The protocol field indicates which higher layer protocol module should process the packet on the destination host; in the case where the higher layer protocol is the Transmission Control Protocol (TCP) [35] or the User Datagram Protocol (UDP) [32], the higher layer protocol header contains a *source port* number and a *destination port* number. Most IPv4 packet classifiers use use these port numbers, when available, in addition to the other fields mentioned.

1.2 Current Trends And The Need For Packet Classification

Increasingly, routers (and other devices in the network) are called upon to perform more advanced services. These services involve processing packets differently depending on several header fields, not simply the destination address, and thus require classification of the packets. These new applications include Quality-of-Service (QoS) based routing [5] [6], firewalls [29], Network Address Translation [15], and other emerging services.

New applications are emerging which demand more from the Internet than just a single “best effort” data forwarding service. Video-on-demand and Internet telephony, for example, require guaranteed bandwidth and low delay. Even in cases where different service levels are not explicitly required, network managers can find it advantageous to provide higher service levels for certain types of traffic (e.g. make interactive sessions higher priority than file transfers). To provide the desired quality of service to the packets belonging to such an application, a router must be able to identify the flow to which the packets belong. In the case of Integrated Services [6], this requires packet classification by all routers the packet encounters. The Differentiated Services [5] approach, on the other hand, simplifies the requirements of the core routers, but still requires packet classification to be performed by the edge routers.

Firewalls [29] are becoming a common network security measure. A firewall connects two or more networks together, but differs from a router in terms of what packets it allows to cross from one network to another. Whereas a typical router is willing to forward any valid IP datagram, a firewall uses a set of rules (filters) specifying which packets are allowed to pass through it.

Intrusion detection systems [38] are another security-related example. These devices need to process packets arriving at wire speed to detect signs of attacks on host or network security. There are many attack forms that only occur in certain types of packets; a packet classifier provides an easy way to identify exactly which packets need to be scanned for particular attack signatures, thereby increasing the efficiency of the intrusion detection system.

Network managers often seek to either measure or capture (for analysis) packets meeting certain criteria. For example, a network manager may want to measure the number of packets send from host A to network B, or the network manager may want to log information about all packets sent to host C on UDP port d. Thus there is a need for packet classification in several network management scenarios.

We expect other advanced network services to appear as well, particularly with the use of network processors [11] [22] [36] and programmable networks [9] [12] [51].

1.3 The Packet Classification Problem

The object of packet classification is to categorize packets by applying a set of rules called *filters* to the header fields of a packet. Each rule consists of a specification of header field values, and an action to perform on packets whose headers match that specification.

The information relevant for classifying a packet is contained inside the packet in K distinct header fields, denoted $H[1], H[2], ..., H[K]$. For example, the fields typically used to classify Internet Protocol (IP) packets are the destination IP address, source IP address, destination port number, source port number, protocol number and protocol flags. The number of protocol flags is limited, so they are often combined into the protocol field itself.

Using those fields for classifying IP packets, a filter F = (128.252.$*$, $*$, TCP, 23, $*$), for example, specifies a rule matching traffic addressed to subnet 128.252 using TCP destination port 23, which is used for incoming Telnet; using a filter like this, a firewall may disallow Telnet into its network.

A filter database consists of N filters $F_1, F_2, ..., F_N$. Each filter F_j is an array of K values, where $F_j[i]$ is a specification on the i-th header field. The i-th header field is sometimes referred to as the i-th dimension or the i-th axis, when considering a packet's header as specifying a point in K-dimensional space. The value $F_j[i]$ specifies what the i-th header field of a packet must contain in order for the packet to match filter j. These specifications often have (but need not be restricted to) the following forms: exact match, for example "source address must equal 128.252.169.16"; prefix match, like "destination address must match prefix 128.252.*"; or range match, e.g. "destination port must be in the range 0 to 1023."

Each filter F_j has an associated directive $disp_j$, which specifies the action to perform for a packet that matches this filter. This directive may indicate whether to block the packet, send it out a particular interface, or perform some other action. Filter databases look like the example in Table 1.1, but most real-world databases have many more filters in them.

A packet P is said to *match* a filter F if each field of P matches the corresponding field of F. For instance, let $F = (128.252.*, *, TCP, 23, *)$ be a filter with $disp = block$. Then, a packet with header (128.252.169.16, 128.111.41.101, TCP, 23, 1025) matches F, and is therefore blocked. The packet (128.252.169.16, 128.111.41.101, TCP, 79, 1025), on the other hand, doesn't match F.

Since a packet may match multiple filters in the database, we associate a *priority* for each filter to resolve ambiguous matches. The packet classification problem is to find the highest priority filter matching a given packet P. Often, the priority is implied by the ordering of filters (with highest priority first), rather than by explicitly storing a priority value for each filter.

Destination Address	*Source Address*	*Dest. Port*	*Src. Port*	*Protocol and flags*	*Comments*
host M_1	*	25	*	TCP	allow inbound mail to M_1
host M_2	*	53	*	UDP	allow DNS access to M_2
*	network N	*	*	*	allow outgoing packets
network N	*	*	*	TCP-ack	return ACKs OK
*	*	*	*	*	block everything else

Table 1.1: Example: simplified firewall filter database

To classify a packet, one could simply apply each rule, in decreasing order of priority, until a match is found. This approach is easy to use, but is clearly not fast enough when a large number of rules are used. Several more sophisticated algorithms have been developed that use data structures cleverly to improve the speed of packet classification.

1.4 Metrics For Evaluating Classification Methods

There are many techniques for packet classification. In this section we discuss criteria for measuring the performance of a packet classification scheme.

The metric of primary concern is the rate at which packets can be classified. Exactly how much speed is needed, however, depends on the application. Classification of minimum-sized IPv4 packets (20 bytes) arriving back to back on a 10 Mb/s link requires 62,500 lookups per second. With a 40 Gb/s link, the required lookup rate is 250 million classifications per second. As network speeds increase, it will be desirable to have packet classification techniques that can scale to higher speeds.

Another important metric is storage requirements of the classifier. Very often there is a tradeoff between storage requirements and lookup speed; for example, classification in one memory access (via direct table lookup) would be possible, if one has storage for a table with 2^W entries, where W is the total number of bits contained in the K header fields of interest. Another extreme case would be simply iterating through a list of all filters, requiring $N \cdot W$ bits of storage, but taking $O(N)$ time for classification.

The complexity of processing filter updates (i.e. insertion or deletion of filters) is also a metric of concern. The rate of updates depends greatly on the specific application. A small firewall with a manually-configured ruleset, for example, does not require handling thousands of filter updates per second. A high-performance QoS-enabled router, on the other hand, may need to establish hundreds or thousands of flows per second, each requiring a filter database update.

In some cases (as we shall see when TCAMs are discussed), power dissipation is important too. This is not only due to the need to supply that much power, but also due to the need to dissipate the resulting heat produced.

1.5 Outline of Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 describes related work in the field of packet classification; particular attention is directed towards TCAMs, Recursive Flow Classification and Parallel Packet Classification, since some of the work in this dissertation builds off those ideas.

Chapter 3 describes techniques for reducing the amount of memory needed by Recursive Flow Classification. These techniques include a method for compressing the data structures while still allowing high speed classification, and a method for rearranging the data structures for improving the compression efficiency. This chapter also explores the effects of reduction tree selection on the classifier size.

In Chapter 4 we discuss Extended TCAMs, a TCAM-based classification technique with greatly reduced power requirements; this is particularly noteworthy because TCAMs are currently the most practical approach to high performance packet classification, and their biggest drawback is their high power consumption. We also describe a means of avoiding inefficiencies in the representation of range match fields in TCAMs.

Chapter 5 explores the idea of packet classification via Partitioned Encoded Search of TCAMs; here we leverage techniques from Parallel Packet Classification and Extended TCAMs to produce a high performance packet classification method with even lower power requirements.

Finally, some concluding remarks are made in Chapter 6.

Chapter 2

The State of the Art

Much research has already been done in the field of high-performance packet classification. In this chapter we discuss the most prominent approaches to date, paying particular attention to those most relevant to this dissertation. These include Ternary Content Addressable Memories (Section 2.1), Recursive Flow Classification (Section 2.2), Parallel Packet Classification (Section 2.3). Other classification methods are included in Section 2.4. Finally, Section 2.5 describes ClassBench, a packet classification benchmark.

2.1 Ternary Content-Addressable Memories (TCAMs)

Ternary Content-Addressable Memories (TCAMs) [17] [21] [30] are the most popular practical approach to general multidimensional packet classification in high performance routers. Ordinary TCAMs perform matching in a bitwise fashion; a query word q matches a stored value, mask pair (v, m) if $q\&m = v\&m$, where the ampersand denotes the bitwise logical "and" operation. Another way to think about this is to consider the stored words to be sequences over not only the traditional bits 0 and 1, but also a "don't care" value (often represented as an X.) The "don't care" bits correspond to bits with mask 0 in the bitmask representation.

TCAMs are used for classifying packets by concatenating all header fields of interest, and using that concatenation as a word for TCAM lookup. This works well for prefix matching (often used on the IP address fields) but is not well-suited for range matching (which is used on the port fields.) The usual way to handle a port range is to replace each filter with several filters, each using a prefix match that covers a portion of the desired port range. For example, the range 2-10 can be expanded into the bit patterns 001*, 01*, 100* and 1010, which exactly cover that range.

In this manner, any sub-range of a k bit field can be represented as a set of prefixes, requiring up to $2(k-1)$ prefixes per sub-range. So, a 16-bit port field can require as many as 30 distinct TCAM entries. But if ranges are present in both the source and destination port fields, then we need a filter for *all combinations* of the sub-ranges for the two fields. Thus a single packet filter may require 900 TCAM entries.

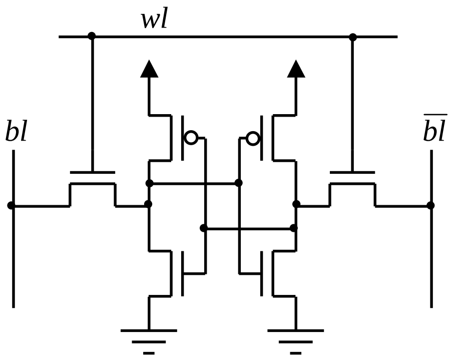
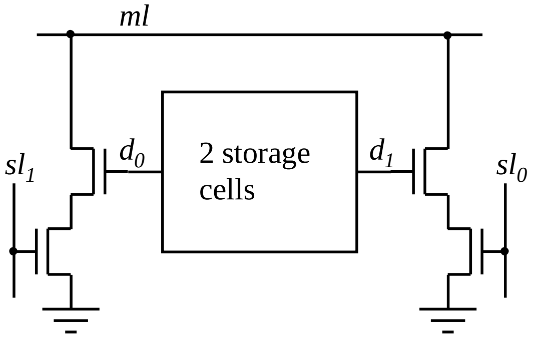


Figure 2.1: Conventional 6-transistor SRAM storage cell



Ternary bit T	Storage cell values d_0	 d_1
0	0	1
1	1	0
X	0	0

Figure 2.2: TCAM cell (details of SRAM storage cells omitted)

Figures 2.1 and 2.2 show the circuitry for a single bit in a typical TCAM design, requiring 16 transistors (proprietary designs exist requiring as few as 14 transistors per cell [21] [17]). A ternary bit T stored in the cell has three possible values: 0, 1, and X, where X is the "don't care" value that matches on both 0 and 1. To store a ternary bit, the TCAM cell uses two SRAM cells with values d_0 and d_1 as shown in Figure 2.2. The value d_0 is 0 when a zero bit in the query word should match, and the value d_1 is 0 when a one bit in the query word should match. Details of an SRAM cell are shown in Figure 2.1. In the SRAM, data is read from and written to the storage cell via the bitlines bl and $\overline{bl}$, under control of the wordline wl.

In the TCAM circuit, the query value and its complement are input via search lines sl_0 and sl_1. A mismatch in this circuit creates a path to ground from the matchline ml. Comparison for all TCAM words is done in parallel, which allows a lookup to be very fast, but it also means that power consumption is quite high [25] [31]. Indeed, TCAMs consume approximately 3 microwatts per ternary bit [28], versus SRAM's need for merely 20 to 30 nanowatts per bit [27].

A recent paper [31] describes CoolCAMs, a technique for using *partitioned TCAMs* to perform IP routing lookups with greatly reduced power dissipation. A partitioned TCAM is a TCAM divided into blocks of words, where each block can be enabled or disabled during a query; power consumption is approximately proportional to the number of blocks searched [31]. With the CoolCAMs method, IP routing lookups require searching just two TCAM blocks. This technique described relies on certain properties of the one-dimensional case, and thus does not generalize to multi-dimensional classification. But, we will take the basic idea of using partitioned TCAMs to reduce power usage, and find ways to apply it in the case of multi-dimensional classification.

2.2 Recursive Flow Classification (RFC)

RFC [19] is an algorithmic approach which has excellent performance in terms of time to classify a packet. This comes at the expense of storage requirements, which can be quite high for large filter databases.

The RFC algorithm works by processing the header fields of a packet in chunks. Effectively, the algorithm tracks which filters have been matched by the various header chunks, and combines the results for the chunks to determine the set of filters matched by the complete headers.

In order to do this efficiently, RFC uses two techniques described in detail later in this section. First, at each step it uses equivalence classes defined by the set of filters matched thus far in processing the packet; these equivalence classes are a concise way for the algorithm to keep track of which filters have been matched by the various chunks of the header fields. Secondly, in order to combine the results for different chunks together efficiently, RFC uses crossproducting tables to store precomputed results. Both of these techniques are described in the examples below, which build up from an overly simplified example to the basis of the full RFC algorithm.

2.2.1 Equivalence Classes in RFC

Consider the following example, in which only one header field is used for classification; for simplicity, let this field represent a destination address of a mere four bits in length. The set of filters in this example are shown in Table 2.1.

Filter Number	*Destination Address (in binary)*	*Cost*
1	001*	1
2	0101-0111	2
3	110*	3
4	0001-1001	4

Table 2.1: Example 1-dimensional filter database

The filters in Table 2.1 can be projected graphically along an axis representing the domain of possible values for the destination address. The axis can be divided into intervals at the endpoints of each filter, as shown in Figure 2.3. Within each interval, a particular set of filters is matched.

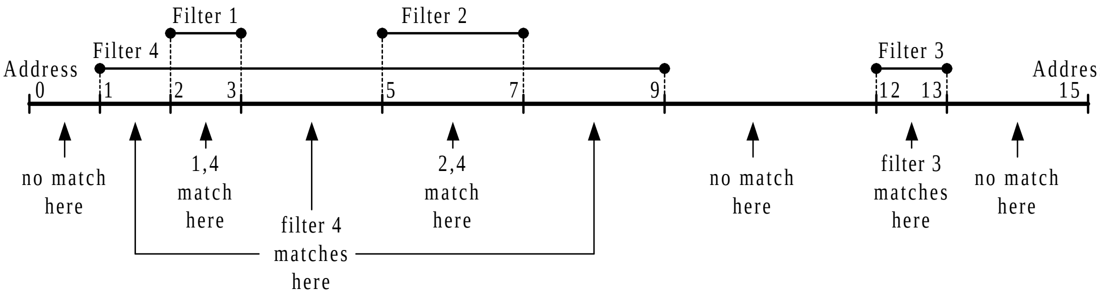


Figure 2.3: Filters projected onto an axis (1-D example)

We can use this to partition the set of possible values for this field (in this case, the space of all possible destination addresses) into equivalence sets, where all values in a set match exactly the same filters. In this example the addresses 1, 4, 8, and 9 all match exactly the same filters (i.e. only filter 4.) Therefore, those addresses belong in the same equivalence set. There are a total of five of these equivalence sets in this example, shown in Table 2.2.

Two points in the same interval always belong to the same equivalence set. Also, two intervals are in the same equivalence set if exactly the same filters project onto them.

Equivalence Class	*Filters Matched*	*Values (destination addresses) in Equivalence Set*
E_0	none	0, 10, 11, 14, 15
E_1	only 4	1, 4, 8, 9
E_2	1 and 4	2, 3
E_3	2 and 4	5..7
E_4	only 3	12, 13

Table 2.2: Equivalence sets for the 1-D example

To help us solve the best matching filter problem, we can precompute a table that maps each possible value for the address to the equivalence class to which it belongs. For the example we are using, this lookup table would look like Table 2.3.

Address	*Equivalence Class*
0	E_0
1	E_1
2	E_2
3	E_2
4	E_1
5	E_3
6	E_3
7	E_3
8	E_1
9	E_1
10	E_0
11	E_0
12	E_4
13	E_4
14	E_0
15	E_0

Table 2.3: Lookup table for the 1-D example

In actual implementation, the equivalence classes are represented by integers 0, 1, 2,... instead of symbols $E_0, E_1, E2, ...$ so they can be used to index into another table. In this case, since we know the filters matching each equivalence class, we can precompute a table that maps each equivalence class to the least cost filter matched by the values in its equivalence set. The result of this is Table 2.4.

Equivalence Class	*Least Cost Filter Matched*
E_0	none
E_1	4
E_2	1
E_3	2
E_4	3

Table 2.4: Best matching filter for each equivalence class (1-D example)

So, to perform a lookup for a packet P with header field containing x, we would do the following: Perform a table lookup of the address x (using Table 2.3) to find the equivalence class to which the address x belongs. This equivalence class indicates which filters match the packet P. Next, perform a table lookup of this equivalence class identifier (using Table 2.4) to determine the least-cost matching filter.

If, for example, we receive a packet with destination address 5, we first look up destination address 5 in Table 2.3. The fifth entry is E_3, i.e. E_3 is the equivalence class for address 5. We then look up E_3 in Table 2.4; the third entry is 2, indicating that Filter 2 is the least cost filter matched by addresses in E_3 (and thus by address 5.)

Of course, this 1-dimensional lookup can be streamlined, by storing the best matching filters instead of the equivalence class identifiers in Table 2.3. Thus, for the 1-dimensional case, the equivalence classes are not required. They are, however, a compact representation for intermediate results (i.e. which filters have been matched so far) during classification on multiple fields; this becomes more apparent in the next example, which involves a 2-dimensional lookup.

2.2.2 Use of Crossproducting

Consider now an example where two header fields are used for classification. Let the fields be source and destination addresses; for simplicity, let each address be only four bits in length. The filters in this example are listed in Table 2.5.

Filter Number	*Destination Address*	*Source Address*	*Cost*
1	*	10*	1
2	100*	010*	2
3	10*	*	3
4	010*	010*	4

Table 2.5: Example 2-dimensional filter database

As before, we can project the filters onto an axis that represents the destination address, as shown at the top of Figure 2.4. This can be used to partition the destination address space into equivalence sets; these are indicated in the same figure and Table 2.6.

Similarly, we can project the filters onto an axis that represents the source address, as shown at the left of Figure 2.4. We use this to partition the source address space into equivalence sets, where source address values in the same set match exactly the same filters. These equivalence sets are shown in the figure and in Table 2.7.

Equivalence Class	*Filters Matched*	*Destination Addresses in Equivalence Set*
ED_0	1 only	0..3, 6, 7, 12..15
ED_1	1 and 4	4..6
ED_2	1, 2, and 3	8, 9
ED_3	1 and 3	10, 11

Table 2.6: Equivalence sets for destination address (2-D example)

Equivalence Class	*Filters Matched*	*Source Addresses in Equivalence Set*
ES_0	3 only	0..3, 6, 7, 12..15
ES_1	2, 3, and 4	4, 5
ES_2	1 and 3	8..11

Table 2.7: Equivalence sets for source address (2-D example)

A table can now be constructed mapping each destination address to the equivalence class to which it belongs; this mapping is shown in Table 2.8. Another table can be constructed to map each source address to the equivalence class to which it belongs; this mapping is shown in Table 2.9.

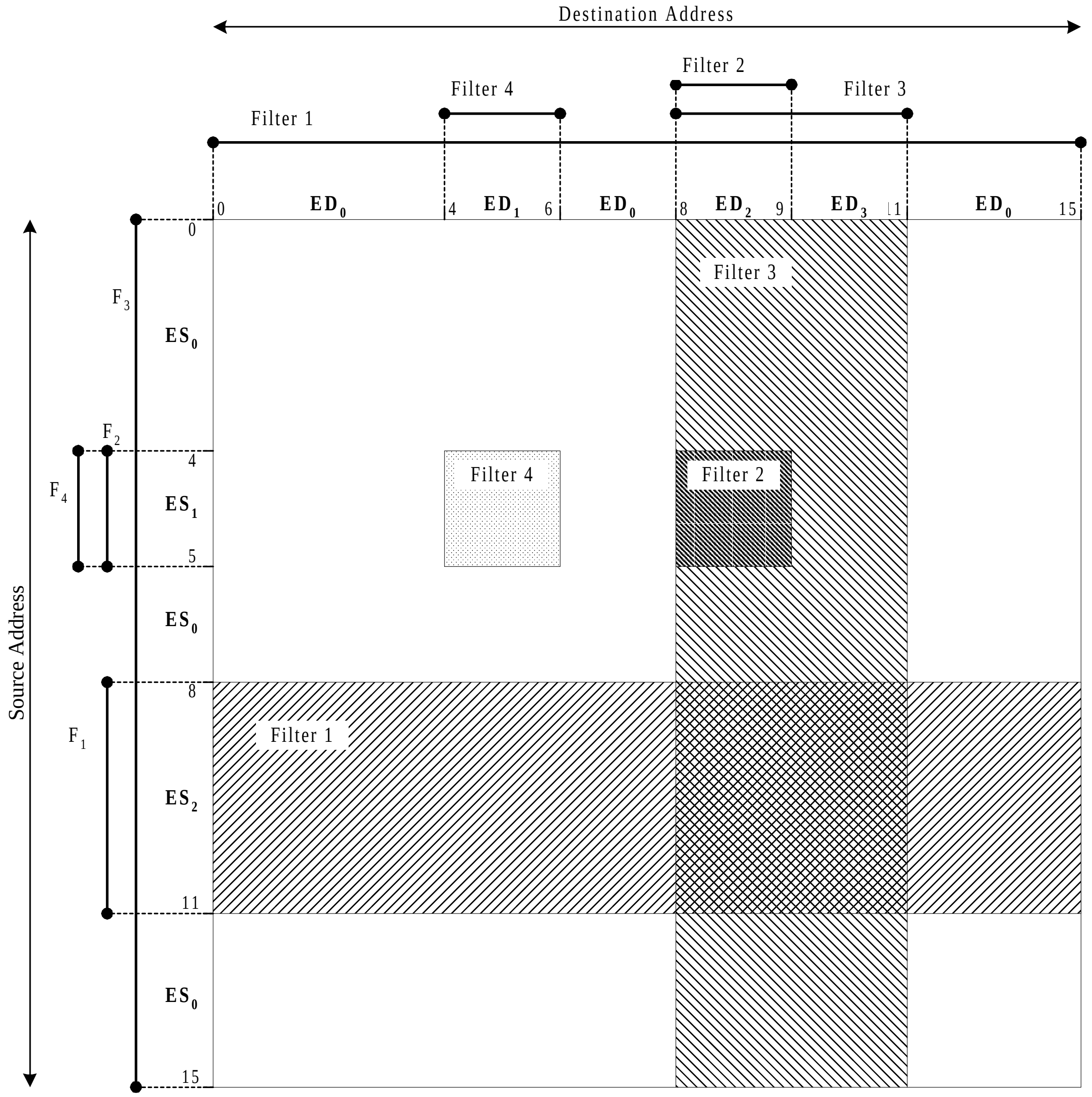


Figure 2.4: Filters projected onto axes (2-D example)

By looking up a packet's destination address in Table 2.8, we obtain an equivalence class identifier which indicates the set of filters matched by that destination address. Similarly, by looking up a packet's source address in Table 2.9, we obtain an equivalence class identifier which indicates the set of filters matched by that source address. But, what we really want is an indication of which filters are matched by both the destination and the source addresses.

Dest. Address	*Equivalence Class*
0	ED_0
1	ED_0
2	ED_0
3	ED_0
4	ED_1
5	ED_1
6	ED_0
7	ED_0
8	ED_2
9	ED_2
10	ED_3
11	ED_3
12	ED_0
13	ED_0
14	ED_0
15	ED_0

Table 2.8: Lookup table for destination address (2-D example)

Source Address	*Equivalence Class*
0	ES_0
1	ES_0
2	ES_0
3	ES_0
4	ES_1
5	ES_1
6	ES_0
7	ES_0
8	ES_2
9	ES_2
10	ES_2
11	ES_2
12	ES_0
13	ES_0
14	ES_0
15	ES_0

Table 2.9: Lookup table for source address (2-D example)

We can compute this by finding the intersection of the set of filters matched by the destination address and the set of filters matched by the source address. This, however, can be too expensive to compute at lookup time if there are many filters (if there are N filters, an N-bit wide AND operation would be needed), so we precompute the results of these intersections and store the results in a 2-dimensional table; the table is computed such that the entry $table[x][y]$ indicates the intersection of the set of filters matched in equivalence class x and the set of filters matched in equivalence class y.

Each entry in this 2-dimensional crossproducting table is used to indicate a set of matching filters. The same set of filters may occur more than one time in the table; thus it makes sense define a new set of equivalence class identifiers to represent these sets, so the table itself only contains equivalence class identifiers. So, to precompute an entry $table[x][y]$ in the crossproducting table, we must do the following:

1. Look up the set of filters matched by equivalence classes x and y,
2. Compute the intersection of that set (bitwise AND),
3. Determine the equivalence class to which that result belongs; store this as $table[x][y]$.

The new equivalence classes for this example are listed in Table 2.10. These classes are defined during the creation of the crossproducting table, shown in Table 2.11, for it is only then that we know which sets of filters will occur.

Equivalence Class	*Filters Matched*
EC_0	none
EC_1	only 2
EC_2	only 4
EC_3	2 and 3
EC_4	only 1
EC_5	1 and 2

Table 2.10: Equivalence classes for crossproducting table (2-D example)

	ED_0	ED_1	ED_2	ED_3
ES_0	EC_0	EC_0	EC_1	EC_1
ES_1	EC_0	EC_2	EC_3	EC_1
ES_2	EC_4	EC_4	EC_1	EC_5

Table 2.11: 2-dimensional crossproducting table (2-D example)

To perform classification, we need both one-dimensional lookup tables (Tables 2.8 and 2.9), the two-dimensional crossproducting table (Table 2.11), and the mapping from final equivalence class identifier to classifier output (Table 2.10.) The other tables are only needed during initialization. To see how this works, consider the following example:

Suppose a packet arrives with destination address 9 and source address 5. To classify this packet, we first look up destination address 9 in Table 2.8, which gives us the result ED_2. We also look up source address 5 in Table 2.9, giving us the result ES_1. These results, ED_2 and ES_1, indicate the filters matched by the destination address and by the source address respectively; we use these equivalence class identifiers to index into Table 2.11 to find which filters are matched by both destination and source addresses. In this example, we would use entry (2, 1) of Table 2.11, which is EC_3. Using Table 2.10 we can see that filters 2 and 3 were matched by the packet.

The last step (using Table 2.10) can be eliminated by storing the least-cost matching filter directly in the entries of Table 2.11; in our example, then, entry (2, 1) of that table would contain the number 2 (identifying the least cost filter matched.)

2.2.3 Extending to k Dimensions

Classification in two dimensions starts by finding a pair of equivalence class identifiers, and uses a precomputed 2-dimensional table to map those to a single equivalence class identifier.

In the case of three dimensions, we start by finding three equivalence class identifiers; let us call these x, y, and z. Each identifier indicates which filters are matched by the corresponding header field. To find which filters match in all three dimensions, we need to compute the intersection of these three sets of filters. Again, this intersection can be too costly to evaluate during a lookup, so we wish to precompute as much as we can.

One approach to this in RFC is to create a 3-dimensional crossproducting table, where each value $table[x][y][z]$ is precomputed by finding the intersection of the sets of filters matched in equivalence sets x, y, and z. But this approach can scale poorly in terms of memory requirements, especially when extending to more than three dimensions; thus, it is not considered in depth in this paper.

Another approach to this in RFC is to use multiple 2-dimensional crossproducting tables; this is the approach preferred in this paper. To classify packets in three dimensions, we need two such 2-dimensional crossproducting tables. The first table is computed such that $table1[x][y] = a$ where a identifies an equivalence class corresponding to the intersection of the filters matched in x and y. The second table is computed such that $table2[a][z] = b$ where b identifies an equivalence class corresponding to the intersection of the filters matched in a and z. This example is shown in Figure 2.5.

The equivalence class identified by b, then, corresponds to the set of filters matched by all three header fields. Thus, we can get the same result while generally requiring less memory than the 3-dimensional table.

This idea can be extended to handle k dimensions, by using $k - 1$ separate 2-dimensional tables. Each table combines two equivalence class identifiers into one equivalence class identifier; thus, with $k - 1$ two-dimensional crossproducting tables, we can go from k equivalence class identifiers (one for each field) to just one.

The order in which these identifiers are combined corresponds to a structure called a reduction tree, where each node in the tree represents a crossproducting table, and its children are the source of the equivalence class identifiers used to index into that table. Figure 2.6 shows an example of a reduction tree for classification using three dimensions (source address, destination address, protocol information.) A more compact representation of the same tree is shown in Figure 2.7; the 1-dimensional lookup tables are implied, but omitted from the figure for brevity.

Prefix matching on a large field can be performed by splitting it up into more than one chunk. This is useful for fields exceeding 16 bits in length (e.g. IP addresses),

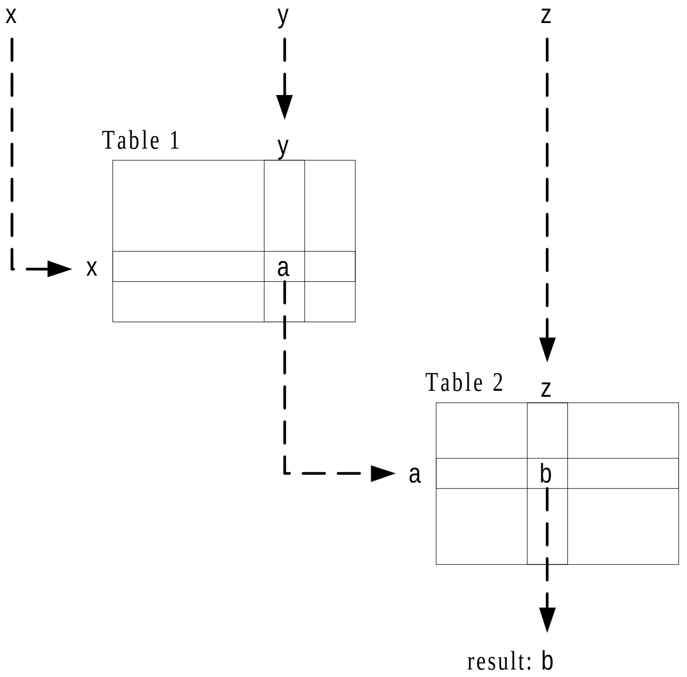


Figure 2.5: Reduction of three equivalence class identifiers to one

Chunk Number	*Chunk Contents*
0	First 16 bits of IP source address
1	Last 16 bits of IP source address
2	First 16 bits of IP destination address
3	Last 16 bits of IP destination address
4	Source port number
5	Destination port number
6	Transport protocol number and flags

Table 2.12: An example set of chunks for a typical 5-tuple classifier.

since a field W bits wide requires a table with 2^W entries to map values to equivalence classes. Fields with range matches cannot be split this way, but a method exists for transforming the range location problem into a prefix matching problem [16].

Using these techniques, we can build a classifier for the standard 5-tuple used in the Internet context. Table 2.12 shows one way to define the header chunks, and Figure 2.8 shows one of the possible reduction trees for this set of chunks.

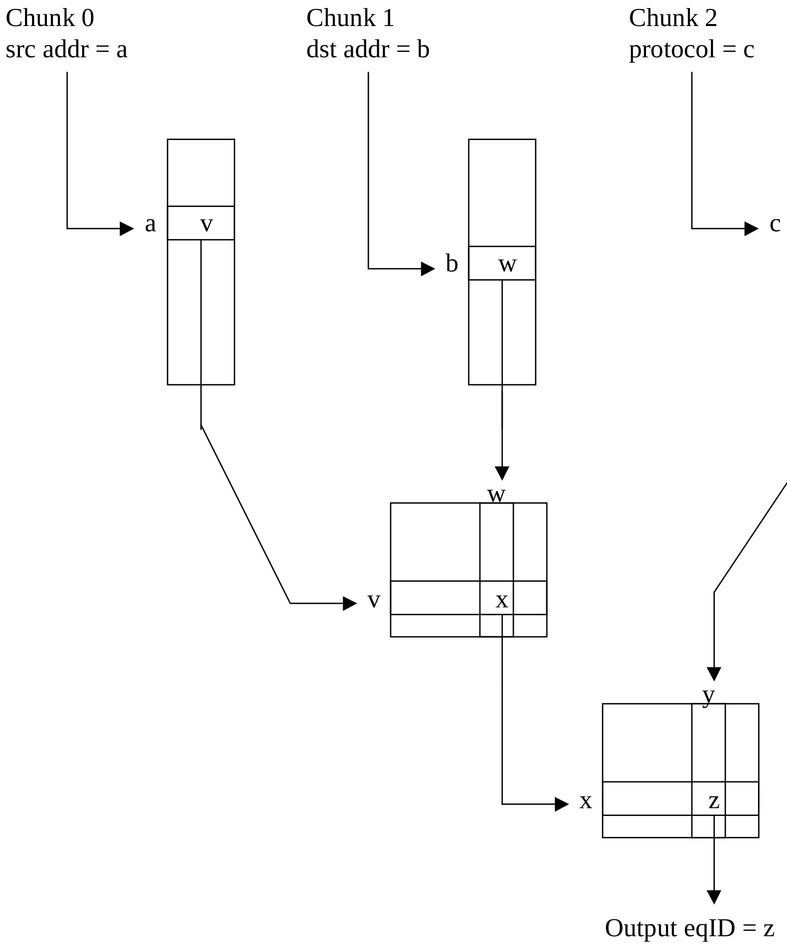


Figure 2.6: An example reduction tree for classifying on three fields

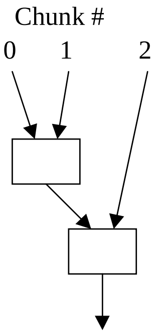


Figure 2.7: Simplified representation of example reduction tree

Thus, with these extensions, what began as a simple example has been extended to become Recursive Flow Classification. The authors of [19] also describe an optional adjacency group optimization which can be used in some applications.

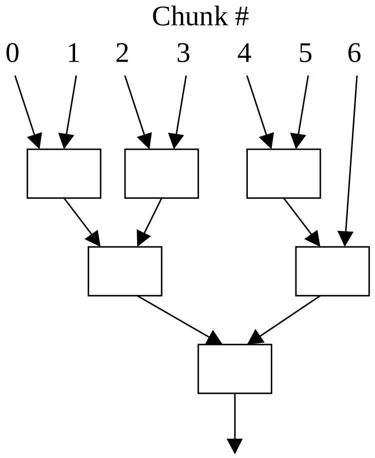


Figure 2.8: An example reduction tree for a typical 5-tuple classifier

2.3 Parallel Packet Classification

Parallel Packet Classification (P^2C) [53] is a packet classification technique which uses an encoding scheme to reduce the width required of a TCAM. Classification of a packet requires converting the values of each field of the packet header into encoded values, and then querying the TCAM with the encoded values. This is motivated by the observation that, for each field in a typical filter database, only a small number of distinct match conditions are used; thus the encoded representation of a set of packet headers will only require a small number of bits.

A lookup in P^2C proceeds as shown in Figure 2.9. First, the values from the packet's header fields are converted into their encoded representations; the concatenation of these form the encoded query, which is presented to a TCAM containing encoded versions of all the filters. The encoding is done in such a way that standard TCAM matching will find the entry corresponding to the highest priority matching filter. The authors of [53] suggest using a modified version of BART [52] as an efficient means of converting header fields to the encoded values during lookup.

The authors of [53] describe three encoding styles, each involving the construction of a primitive range hierarchy. Primitive ranges are the intervals of field values specified in the match conditions for a given dimension in a filter database. Hierarchies can be constructed from these as follows:

With the first encoding style, the set of ranges used in a given field of the database are grouped into layers, such that no two ranges in the same layer overlap;

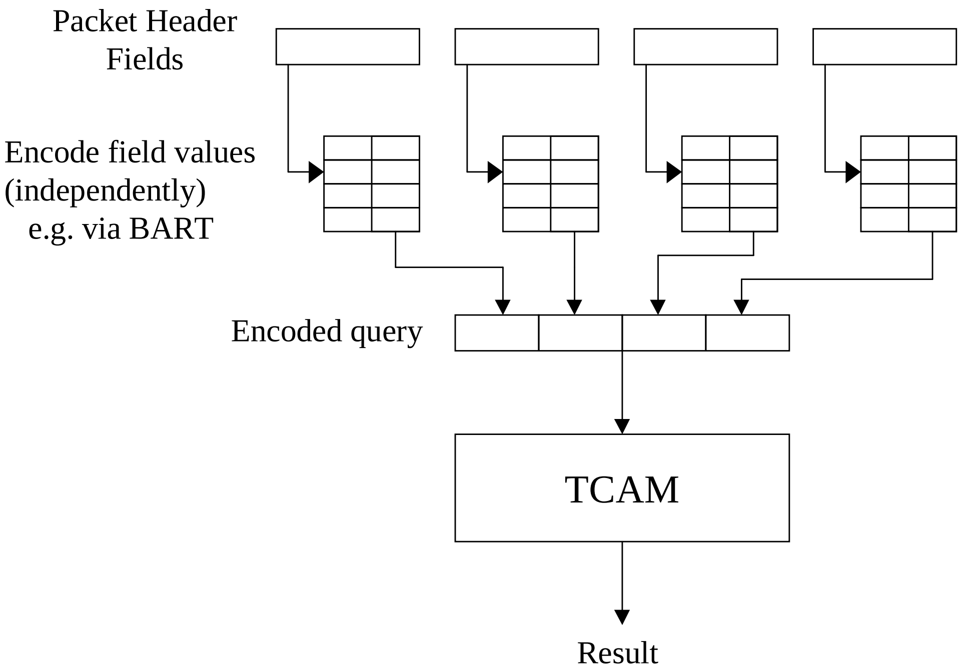


Figure 2.9: P^2C lookup process

this can be accomplished incrementally by adding each range one at a time, or all at once via interval graph coloring. Within each layer, identifiers are assigned as follows: 0 corresponds to all regions not in any range in the layer, 1 corresponds to the first range in the layer, 2 corresponds to the next range in the layer, and so on. Each layer, then, is represented by some set of bits in the encoded representation of a field value; the exact number of bits needed for each layer depends on the number of ranges within that layer. An example of this encoding style is shown at the top of Figure 2.10; this example has four primitive ranges, and this style of encoding requires two layers for the primitive range hierarchy.

When encoding a field in a rule using this technique, the range used in that field is added to a layer in the primitive range hierarchy, if not already present. The bits corresponding to that layer are set to the identifier for that range; bits corresponding to other layers in that field are set to the "don't care" value. Table 2.13 continues the example from Figure 2.10, indicating how to encode match conditions that occur in filters, using the three different encoding styles.

When encoding a field of a search query, the encoded value must include identifiers for every layer in the primitive range hierarchy; for example, using the encoding style I entries in Table 2.13, a packet header field value of 0x24 would be encoded as 0101 (since it falls within range 01 from layer 1 and range 01 of layer 2), and a packet header field value of 0x13 would be encoded as 0100 (since it falls within range 01 from layer 1, and no range in layer 2).

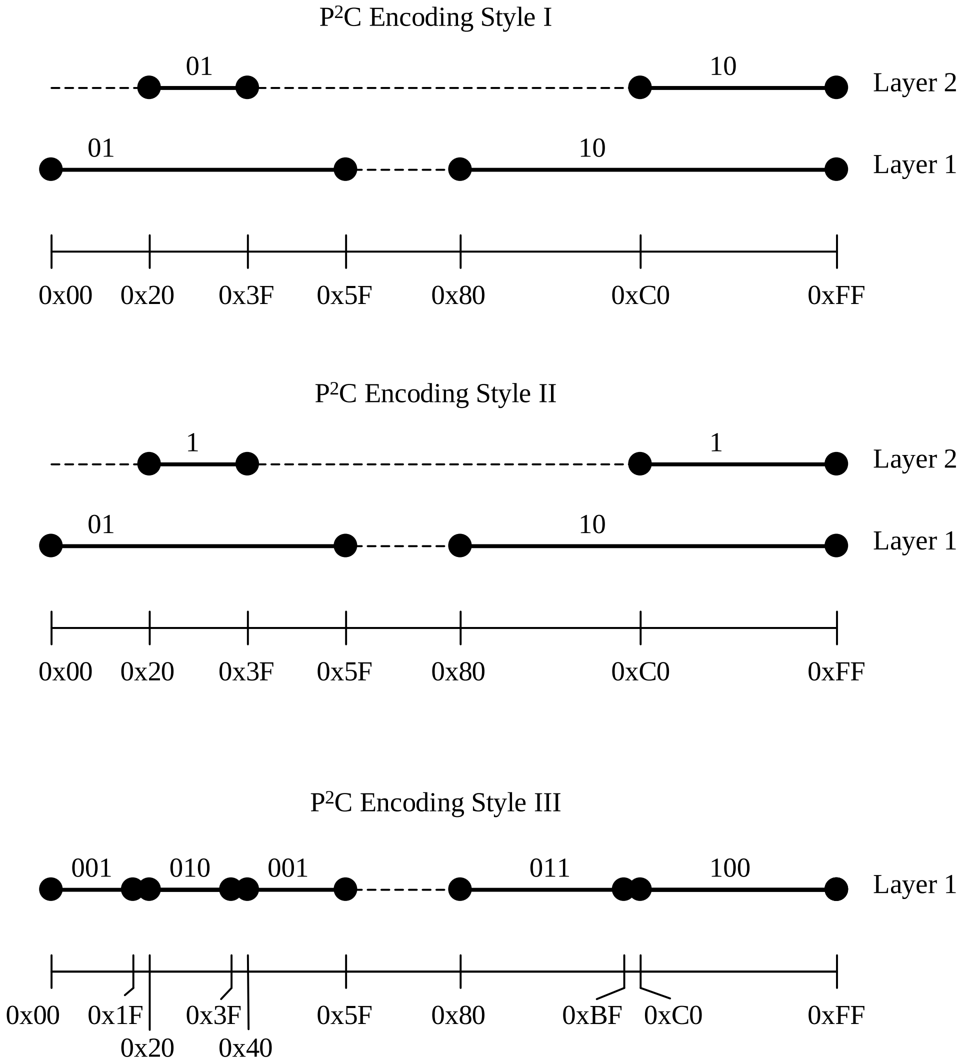


Figure 2.10: Example primitive range hierarchies

Note that the range corresponding to a fully wildcarded value need not be included in the primitive range hierarchy; wildcarding all bits for that field will work. Also, for the sake of efficiency, placing all exact-match entries in the same layer of the primitive range hierarchy is generally a good idea.

The second encoding style is similar to the first, but reduces the number of bits required for a layer when ranges within that layer can already be distinguished via bits of some other layer. For example, in the Encoding Style II section of Figure 2.10, the filters in Layer 2 can be distinguished by making use of one of the bits from Layer

Range in filter (hex)	*Encoded value (style I)*	*Encoded value (style II)*	*Encoded values (style III)*
0x00-0xFF	xxxx	xxx	xxx
0x00-0x5F	01xx	01x	001, 010
0x20-0x3F	xx01	0x1	010
0x80-0xFF	10xx	10x	011, 100
0xC0-0xFF	xx10	1x1	100

Table 2.13: Example encodings for match conditions (filter fields)

1. This can further reduce the bit width required of the TCAM, but adds substantial complexity to the creation and maintenance of the data structures (in our example, if the encodings of Layer 1 change, this affects the encodings of Layer 2 which make use of those.)

The third encoding scheme reduces the number of layers by combining layers together. When layers are combined, the ranges that overlap are split into nonoverlapping ranges; any rule using those ranges is thus required to span multiple TCAM entries (assuming hardware support for general range matching is not available) in order to correctly cover the range specified in the rule. The primitive range hierarchy can even be flattened into a single layer by repeated application of this technique. The bottom of Figure 2.10 shows the result of combining the two layers in our example together.

These three encoding schemes can be used in the same classifier for different rules, which can be useful since each encoding scheme has its own advantages and disadvantages. The data structures used in the first scheme are the easiest to create and maintain; the second scheme can reduce the number of bits needed for some layers, at the cost of increased complexity of the data structures used. The third scheme can reduce the number of bits of TCAM width needed by combining layers in the primitive range hierarchy, at the cost of using multiple TCAM entries for some rules.

The process of converting a search key to its encoded representation can be done in a variety of ways. Ultimately the encoded search key contains the identifiers of the regions matching the packet's header, but there are more efficient methods than looking up the value in each layer separately and combining the results. The authors of [53] suggest using a slightly modified version of BART [52] for encoding the search keys; this approach is fast (a lookup for a 32-bit field can be done in four memory accesses) and provides high storage efficiency while allowing fast updates.

In [53], results are shown for three databases, ranging from 37 rules to 1,733 rules in size. The encoding decreases the TCAM width required from 96 bits (for the 4-tuple used in that study) to 22 to 25 bits, with a corresponding decrease in total TCAM size required; the additional SRAM used for the BART lookup was quite small, ranging from 0.25 kB for the 37 rule database to 2.0 kB for the 1,733 rule database.

We include in Table 2.14 our own results of P^2C encoding style I applied to seven real-world filter databases. The table indicates how many bits are required to encode the 4-tuple used in [53], and the number of bits required for encoding the 5-tuple of Source Address, Destination Address, Source Port, Destination Port, and Protocol. The results for small databases appear consistent with the results in [53]. While the larger databases in this study do not appear to encode quite as well, the encoding still results in a significant decrease in the width required of the TCAM.

Number of Filters	*Bits Required for 4-tuple*	*Bits Required for 5-tuple*
160	29	33
184	25	29
192	16	20
280	31	36
613	38	43
1,556	49	56
2,396	60	65

Table 2.14: Encoding widths for some real-world filter databases, using P^2C encoding style I

2.4 Summary of Additional Related Work

TCAMs, RFC and P^2C are of particular interest with respect to this dissertation, but there are several additional classification techniques worthy of mention.

2.4.1 Classification in One Dimension

Classification in one dimension is equivalent to the IP route lookup problem; several algorithms have been developed for this special case that perform very well [7] [24]

[54] [18] [45] [26] [14]. The focus of this dissertation, however, is the more general multi-dimensional case, which is a much harder problem.

A recent paper [31] describes a method for performing IP routing lookups using a partitioned TCAM. By restricting the search to only relevant partitions in the TCAM, power requirements are greatly reduced. One may think of it as carving out portions of a routing tree, and placing each portion in a partition of the TCAM. The prefix for each carved portion is placed in the *index partition*, along with the best matching route covering that portion. A routing lookup, then, requires a search of only two partitions: first, the index partition is searched, to determine which portion of the tree is relevant to that query; secondly, the relevant partition itself (as indicated by the result from the index) is searched. If a route is not found in that partition, the best matching cover (which was obtained from the lookup in the index) is used.

2.4.2 Classification in Two Dimensions

Packet classification in two dimensions (e.g. classifying by Source Address and Destination Address) is more difficult than in just one dimension, but reasonable algorithms for this case exist also.

The Grid-of-Tries [46] algorithm constructs a search trie for the Destination Address, followed by several search tries for the Source Address. Clever use of switch pointers allows this algorithm to avoid backtracking; it also keeps storage requirements linear in terms of the number of filters, but this is only possible for the two dimensional case.

Area-based quadtrees (AQT) [8] are based on space decomposition techniques. In the two dimensional case, AQT requires $O(N)$ storage, takes $O(aw)$ memory accesses for query processing and $O(aN^{1/a})$ time for updates, where N is the number of filters, w is the width of the fields used for classification, and a is a tunable parameter.

Filters can also be organized using FIS-trees [16], which can support query processing with $O(\log w)$ memory accesses. This approach has reasonable storage requirements in the two dimensional case, but does not scale well beyond two dimensions.

2.4.3 General Multi-dimensional Classification

In the most general case, classifying packets in K dimensions is inherently hard for K greater than 2 [23] [46] [44] [19] [20]. The best algorithms at this time require

$O(\log^{K-1} N)$ time and linear space, or $\log N$ time and $O(N^K)$ space, where N is the number of rules [23].

Fortunately, the filters in real-world classifiers tend to exhibit certain properties; these properties allow clever algorithms to beat the worst-case bounds in most real-world applications. This idea has led to a number of classification algorithms which, in spite of the aforementioned bad worst-case bounds, tend to perform very well with "typical" filter databases.

Tuple-space search [44] is an early heuristic approach, and Entry Pruned Tuple Search [43] is a refinement of that technique. In these approaches, filters are categorized by the lengths of each specified field; this decomposes the classification problem into a number of exact-match problems, where hashing is used to probe for a match. In many cases this works well, but the hashing can lead to unpredictable search times.

HiCuts [20] constructs a decision tree to partition the classification space into regions until the number of filters in each region falls below a threshold. Thus classification is performed by walking the decision tree and then performing a short linear search through the filters in that region. HiCuts is not as fast as RFC, but its memory usage is not quite as bad.

Extended Grid-of-Tries (EGT) and EGT with Path Compression (EGT-PC) [3] use a slightly modified two dimensional Grid-of-Tries for classification on Source Address and Destination Address, and then a linear search of the filters which match the filters at that point. For "typical" databases, the storage requirements appear to grow linearly with the number of filters; the query time, however, is not as fast as HiCuts or RFC.

HyperCuts [37] is similar to HiCuts, but uses multidimensional cuts at each step and includes several other storage-related optimizations. Nodes containing the same rulesets are merged; any rule covered by a higher priority rule in the same node is not stored in that node; the region associated with each node is compacted to the minimum cover for the rules in that region; and, filters that are stored in many leaves of the decision tree are pushed "upward" (i.e. towards the root), to reduce the number of times these filters are stored. Furthermore, the heuristics used to build the decision tree are specifically designed with minimization of storage requirements in mind. HyperCuts appears to have excellent lookup performance (very close to RFC), but thus far results have not been shown for databases with more than 25,000 filters. It has excellent storage efficiency in many cases, but does not fare quite as well with heavily wildcarded filters.

2.4.4 Packet Classification Repository

A repository [2] has been established for retaining papers and source code for some of these algorithms. This repository allows researchers to share information, and to study and compare implementations of known algorithms, without the time-consuming and potentially error-prone process of re-implementing them.

2.5 The ClassBench Benchmark

Since many modern packet classification techniques employ heuristics to obtain good performance, the use of realistic test vectors is vital for proper performance analysis. Such input test vectors are difficult to obtain, however, since the network administrators who have access to real filter databases have strong concerns about privacy and security. Additionally, since the current technologies do not scale well for truly large filter databases, there are very few filter databases of great size.

ClassBench [1] [49] is a recent project addressing that problem. A key component of ClassBench is the Filter Set Generator, which produces a synthetic filter database using parameters measured from real databases. At this time, twelve filter set parameter files are available, each measured from a different real-world filter database. This allows the creation of filter sets having any size and retaining important properties (such as prefix length distributions, correlations between the specificity of each field, and maximum prefix nesting depth) that affect the performance of packet classification algorithms. In addition, the generator allows certain properties of the output filter set to be adjusted (e.g. alterations in filter scoping or the smoothness of tuple distributions) if desired.

ClassBench also includes a filter set analyzer and a packet trace generator. The filter set analyzer is used to produce filter set parameter files from real-world filter sets; the packet trace generator produces sequences of packet headers usable as test inputs to exercise a classifier.

Chapter 3

Compressed Data Structures for RFC

In the previous chapter, we saw that Recursive Flow Classification performs packet classification extremely quickly, but this speed comes at the cost of relatively high storage requirements. In this chapter we explore ways to reduce those storage requirements.

The chapter is organized as follows: a simple compression scheme is described in Section 3.1; Section 3.2 contains an improvement of the simple compression technique; the impact of selecting different reduction trees is explored in Section 3.4.2.

3.1 Compression of Data Structures

Recursive Flow Classification has excellent performance with respect to lookup time, but its memory requirements can be quite high when many filters are used. Here we describe a scheme using compressed data structures to reduce the memory requirements of the lookup algorithm.

Recall that, in RFC, classification is performed by first looking up individual header chunks, and then by combining the chunks in a series of crossproducting steps. An example of this is shown in Figure 3.1. The major data structures involved, then, are the data structures for looking up chunks in the initial step, and the crossproducting tables.

The proposed compression method is based on two observations: first, that much of the storage is required for the crossproducting tables (especially for large filter databases), and secondly, that these crossproducting tables tend to have many

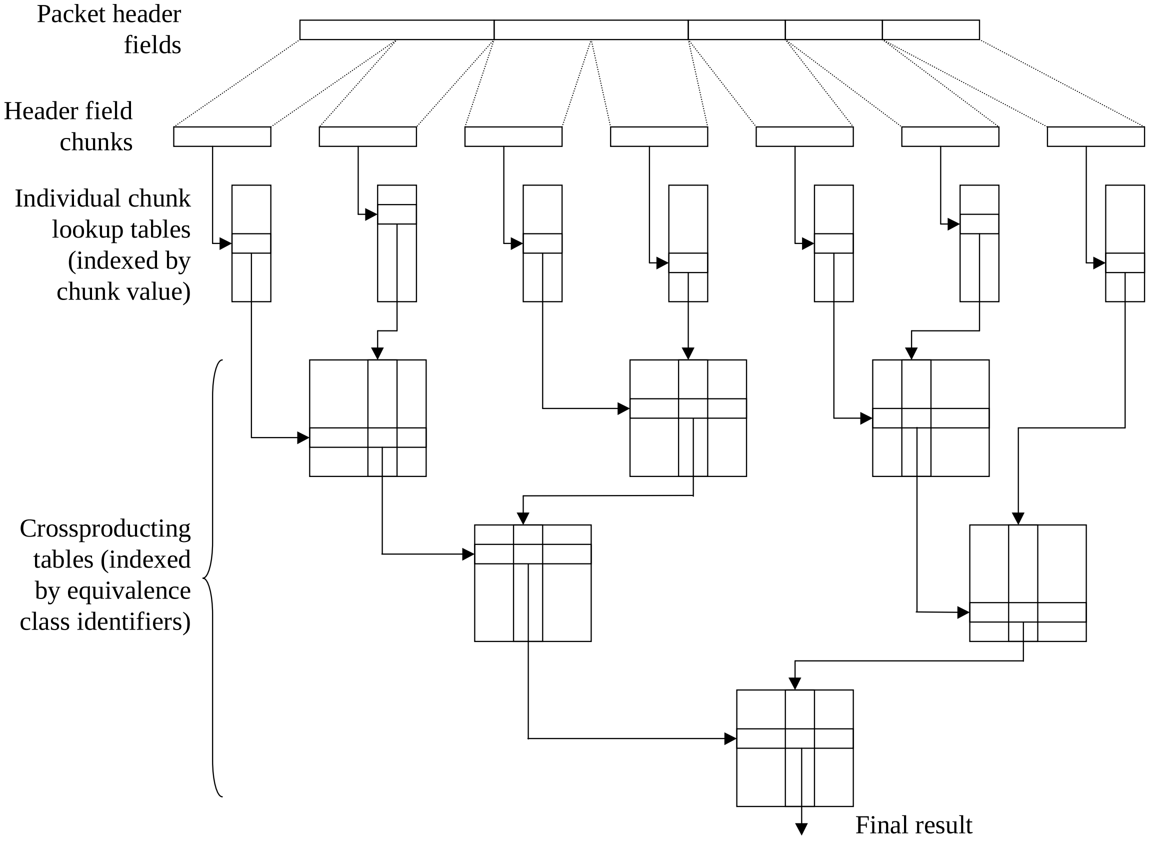

Packet header fields
Header field chunks
Individual chunk lookup tables (indexed by chunk value)
Crossproducting tables (indexed by equivalence class identifiers)
Final result

Figure 3.1: Example structure of RFC classifier.

contiguous elements repeated (again, especially true with large filter databases.) We can take advantage of this to compress the crossproducting tables, but this new representation must still allow fast lookups.

3.1.1 Simple Compression Algorithm

To see how we might compress the tables, let us consider a table stored in row-major order in an array. The original array can be represented by a compressed array and a bit vector, constructed in the following way: For each run of repeated elements, we store only one such element in the compressed array. Thus, the compressed array for $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$ would be $A\ B\ C\ B\ C$. The bit vector has a bit corresponding to each element in the original (uncompressed) array; this bit is a 0 if that element is the first element or is the same as the previous element, and 1 otherwise; thus, the bit vector for $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$ would be 0 0 0 0 1 0 0 1 1 0 1 0, as shown in Figure 3.2.

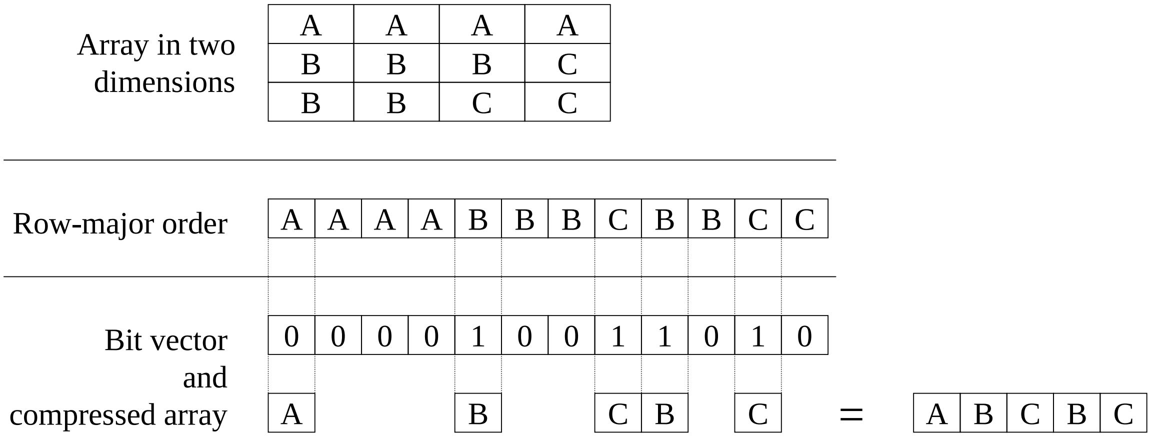


Figure 3.2: Example representations of a two-dimensional array.

The bit vector is used to find the results of a lookup in the compressed array. If we want to know what the ith item was in the original array, we count the number of 1s in bit vector elements zero through i, inclusive. If there are j 1s, then we can find the result by looking at the jth element of the compressed array. Continuing the example from the previous paragraph: To look up the 7th element of the original array $A\ A\ A\ A\ B\ B\ B\ C\ B\ B\ C\ C$, we count the 1s in bit vector elements 0 through 7; there are 2, so the answer is element 2 of the compressed array $A\ B\ C\ B\ C$, i. e. the answer is C.

Counting the number of 1s becomes expensive when the bit vector is large; to avoid performing much of this work at classification time, we use precomputation as follows: If we precompute the total of 1s set in the first W bits, the first $2W$ bits, the first $3W$ bits, etc., then at lookup time we only need to count at most $W-1$ bits in the bit vector.

Thus, each crossproducting table can be represented efficiently by a compressed array, a bit vector with a bit for each item in the original array, and a set of precomputed counts of ones.

3.1.2 Results

To evaluate the compression techniques, we construct a packet classifier using a real filter database with 159 rules. The classifier splits the header fields into 7 chunks

Chunk Number	*Chunk Contents*
0	First 16 bits of IP source address
1	Last 16 bits of IP source address
2	First 16 bits of IP destination address
3	Last 16 bits of IP destination address
4	Source port number
5	Destination port number
6	Transport protocol number and flags

Table 3.1: Chunks used in evaluating RFC compression scheme

as indicated in Table 3.1. We use 16-bit chunks as suggested in [19], and because the port fields require range matching, which precludes splitting them into smaller chunks. We perform classification on five header fields in this experiment, though in some papers it appears that RFC is only used for classification on four fields.

Since the choice of reduction tree affects the size of the crossproducting tables, and there is no immediately obvious “best choice” reduction tree, we consider all possible binary reduction trees for this database. For each possible reduction tree, we determine the size of the data structures in an uncompressed form and in the compressed form. In this way, we not only find the best choice of reduction trees, but we also gain insight into the effect of choosing non-optimal reduction trees as well.

Results are shown in Table 3.2. Rows 1-6 each correspond to a specific reduction tree; for example, row 1 shows the results for the reduction tree which resulted in the largest data structures when using the uncompressed form. Rows 7 and 8 do not correspond to specific reduction trees; instead they show the average and median for each metric across all reduction trees. In this experiment, compression typically reduced the crossproducting table storage requirements by 39%. The overall storage requirements were typically reduced by 22%, but we expect that larger classifiers would enjoy better results. With the relatively small (159 filter) classifier in this experiment, the individual field lookup tables (which are not compressed) occupied 2,621,440 bits; in some cases, this accounted for the majority of the data structure size. But with a larger classifier, we expect that the crossproducting tables occupy a much greater fraction of the overall storage needed.

Reductions of over 80% occurred in some cases, usually involving large crossproducting tables that compressed well (but, not all large tables compress well.) In a few cases, the compressed form actually required more storage than the uncompressed form; this occurs when the size reduction from the original array to the compressed array is less than the additional storage needed to hold the bit vector.

The choice of reduction tree has a significant effect, as well. In this experiment, the best choice outperforms the worst choice roughly by a factor of 30, and it outperforms the typical case by a factor of two to three. We explore the selection of reduction tree more in Section 3.4.

	Uncompressed size (bits)	*Compressed size (bits)*	*Overall size reduction*	*Reduction of crossproduct tables*
Largest uncompressed	89,380,018	19,926,831	77.7%	80.1%
Smallest uncompressed	3,113,672	2,914,527	6.4%	40.5%
Largest compressed	89,380,018	96,187,539	-7.6%	-7.8%
Smallest compressed	3,113,672	2,914,527	6.4%	40.5%
Most overall compression	77,233,034	15,235,167	80.3%	83.1%
Least overall compression	67,280,539	72,425,225	-7.6%	-7.9%
Average over all values	12,196,446	8,227,865	25.4%	36.7%
Median over all values	7,454,688	5,800,143	22.4%	39.5%

Table 3.2: Experimental compression results

A graph showing the overall compression ratios achieved in this experiment is shown in Figure 3.3; these figures represent the ratio of bits required for the compressed representation to the bits required for the uncompressed representation. The results have been sorted by compression ratio, so that one can easily see from the graph, for example, that for 10% of the reduction trees used the algorithm reduced overall storage requirements to 44% of the original amount. Since we are using a small classifier, a significant fraction of the overall storage is needed for the individual chunk lookup tables; therefore we also present the compression ratios for the crossproducting tables by themselves, shown in Figure 3.4.

In this study, the bitvectors account for 6.15% to 46.3% of the storage requirements for the compressed representation of the crossproducting tables, with the median value being 11.42%; thus it appears that, in the typical case, the majority of the storage needed for the compressed representation is due to the compressed arrays, rather than the bitvectors.

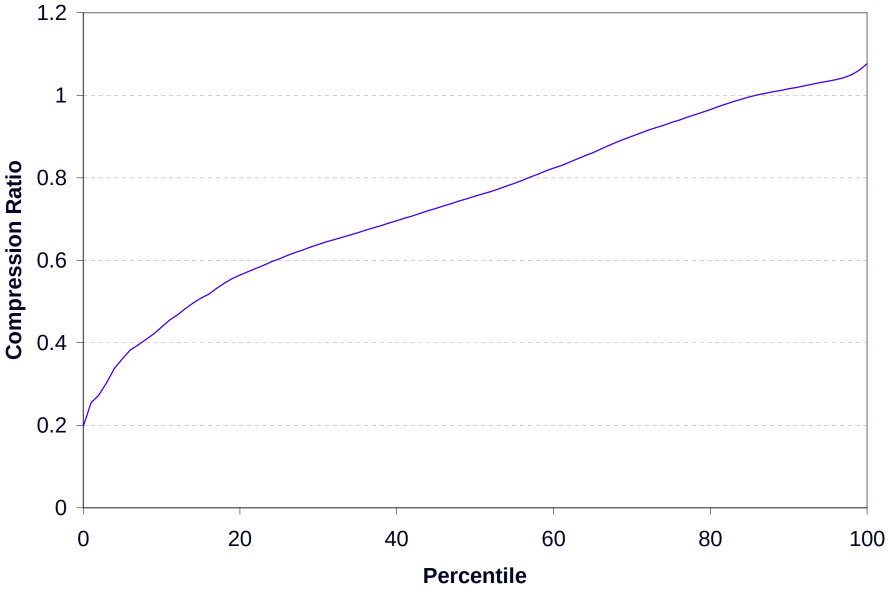


Figure 3.3: Overall compression ratios from experimental results.

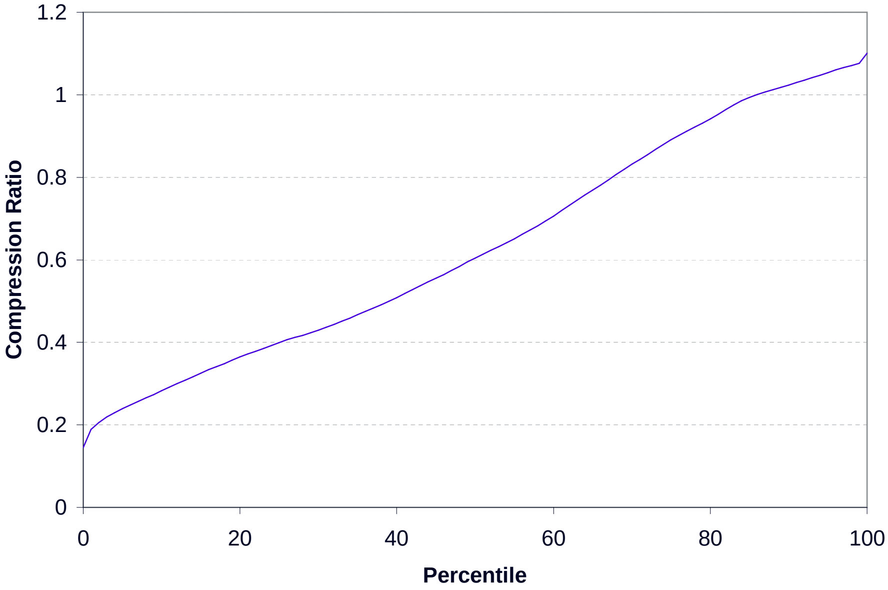


Figure 3.4: Compression ratios for crossproducting tables only, in experimental results.

3.2 Improving Compression

The compression technique just described relies on the tendency for adjacent table entries in the same row to have the same value. For this reason, some tables compress

	0	1	2	3	4	5	6	7
0	a	b	a	b	a	b	a	b
1	b	a	b	a	b	a	b	a
2	a	b	a	b	a	b	a	b
3	b	a	b	a	b	a	b	a
4	a	b	a	b	a	b	a	b
5	b	a	b	a	b	a	b	a

Table 3.3: Two-dimensional crossproducting table with poor compression

	0	1	2	3	4	5	6	7
0	a	a	a	a	b	b	b	b
1	b	b	b	b	a	a	a	a
2	a	a	a	a	b	b	b	b
3	b	b	b	b	a	a	a	a
4	a	a	a	a	b	b	b	b
5	b	b	b	b	a	a	a	a

Table 3.4: Two-dimensional crossproducting table with better compression

well, and others do not. For example, Table 3.3 does not compress particularly well; in fact, the "compressed" representation requires more storage than the uncompressed form.

Since the equivalence class identifiers are assigned in an arbitrary order, it is possible to re-arrange rows and/or columns of the tables by reassigning the identifiers. Thus it is possible to re-arrange rows and columns in a table to improve compression. If we re-order the columns of Table 3.3 we can produce Table 3.4, which will result in improved compression since it has more runs of repeated elements.

In general, there are too many ways re-arrange a large table to conduct an exhaustive search. However, there are some heuristics that usually produce good results. For example, if tables are stored in row-major order, then re-arranging rows will have little effect compared to re-arranging columns. This is because the compression works well when a value in memory is followed by the same value; re-arranging rows will only alter that at the wrap-around point (where the end of one row is followed by the beginning of the next row), whereas re-arranging columns will alter that at every position in the column.

Therefore, in this work we focus on re-arranging columns; in the following section we develop a heuristic for selecting orderings of columns with good compression.

3.2.1 TSP Heuristic

The question of how to re-arrange the columns for best compression can be transformed into a variation of the Traveling Salesman Problem, as follows: Let each column in the table be represented as a node in the TSP problem. The goal is to select an ordering (tour) of the columns (nodes) such that the number of elements in the compressed array (cost of the tour) is minimized.

The total cost of an ordering of columns is the number of elements in the compressed array. An element in column $i + 1$ is only added to the compressed array if it differs from the element in column i of the same row; thus, the cost contribution of placing column $i + 1$ immediately after column i is equal to the number of rows in which the two columns have differing entries. This way, the cost of a particular tour reflects the cost of using that ordering for columns, except for the cost of the first column itself (due to wrap-around from last column of a row to first column of the next row.)

With this definition of cost, note that cost(A,C) $\leq$ cost(A,B) + cost(B,C) (i.e. the triangle inequality applies.) Thus, TSP approximation algorithms based on a minimum spanning tree will work and can be used to find an ordering of columns that produces good results.

This generally produces better results than the naive compression scheme described earlier, but computing the TSP cost matrix can be expensive. For a 2-dimensional table with R rows and C columns, this requires $\mathrm{O}(C^2R)$ time and $\mathrm{O}(C^2)$ space.

3.2.2 Results

To evaluate the effectiveness of the TSP heuristic, we perform compression on the same filter set that is used in Section 3.1.2; again, we study the effects of compression on all the possible binary reduction trees for the header chunks defined in Table 3.1. This time, the TSP heuristic is used whenever practical, given the time constraints (in this case, as long as no crossproducting table exceeded 6,400 columns), and the simpler compression technique is used otherwise. The results are shown in Table 3.5. Rows 1-6 each correspond to a specific reduction tree; rows 7 and 8 are average and median for these values across all reduction trees.

In this study, we use a simple approach of finding a minimum spanning tree and traversing around it in preorder to determine the new order of columns, starting with the first column. The intent of this particular experiment is merely to discern whether re-arrangement of columns produces benefits sufficient to justify the cost of computing the new column ordering. Thus it is worth keeping in mind that small improvements on this (e.g. Christofides' approximation algorithm [10] for TSP, and intelligent selection of the start and end of the tour) are still possible.

In these results, compression with the TSP heuristic typically reduced the crossproduct table storage requirements by 62%. Reductions over 85% occurred in some cases, usually involving large tables that compressed well (but, not all large tables compress well.) There are still cases where the compressed form requires slightly more storage than the uncompressed form, but not as much as when using the naive compression method.

The overall storage requirements were typically reduced by 37%. But, again, that is mainly due to the individual field lookup tables occupying a significant fraction of the data structure, due to the small size of this classifier. We expect that larger classifiers will experience more overall compression, since their crossproduct tables will occupy a larger fraction of the overall data structure size.

	Uncompressed size (bits)	*Compressed size (bits)*	*Overall size reduction*	*Reduction of crossproduct tables*
Largest uncompressed	89,380,018	13,677,785	84.7%	87.3%
Smallest uncompressed	3,113,672	2,905,502	6.7%	42.3%
Largest compressed	67,963,198	73,138,696	-7.6%	-7.9%
Smallest compressed	3,113,672	2,905,502	6.7%	42.3%
Most overall compression	89,253,854	13,580,518	84.8%	87.3%
Least overall compression	73,052,072	67,868,399	-7.6%	-7.9%
Average over all values	12,196,446	5,976,058	38.1%	54.4%
Median over all values	7,454,688	4,931,011	37.7%	61.6%

Table 3.5: Experimental TSP-heuristic compression results

A graph showing the overall compression ratios achieved using the TSP heuristic when possible is shown in Figure 3.5; as before, these figures represent the ratio of bits required for the compressed representation to the bits required for the uncompressed representation. We also present the compression ratios for the crossproducting tables by themselves, shown in Figure 3.6. These results indeed are an improvement over the simple compression algorithm.

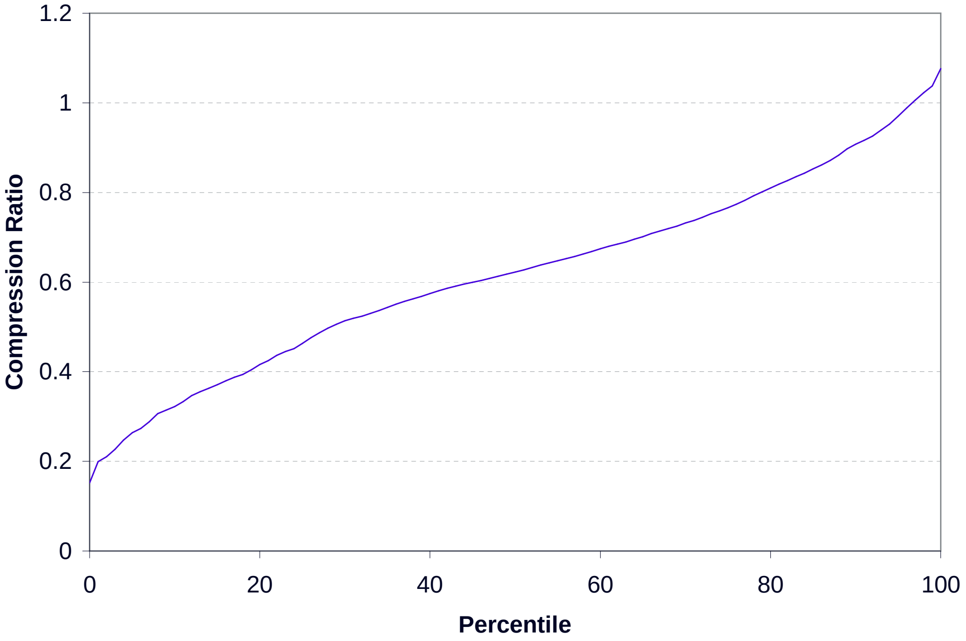


Figure 3.5: Overall TSP compression ratios from experimental results.

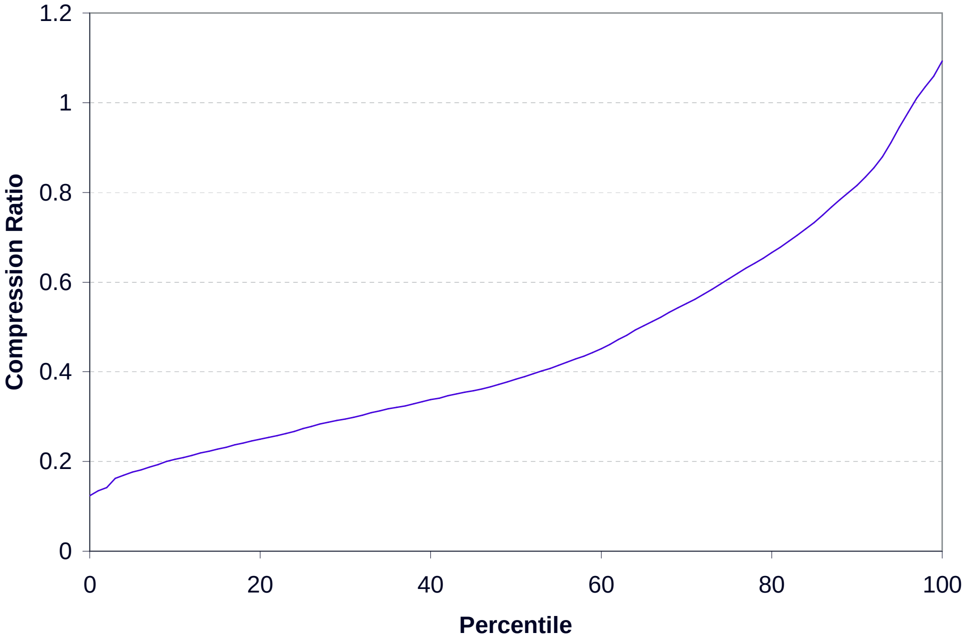


Figure 3.6: TSP compression ratios for crossproducting tables only, in experimental results.

For an indication of how much the TSP heuristic helps, we divide the number of bits needed with the TSP heuristic by the number of bits needed for the simple compression method. A graph of results is shown in Figure 3.7 with respect to overall compression, and in Figure 3.8 with respect to the sizes of the crossproducting tables only.

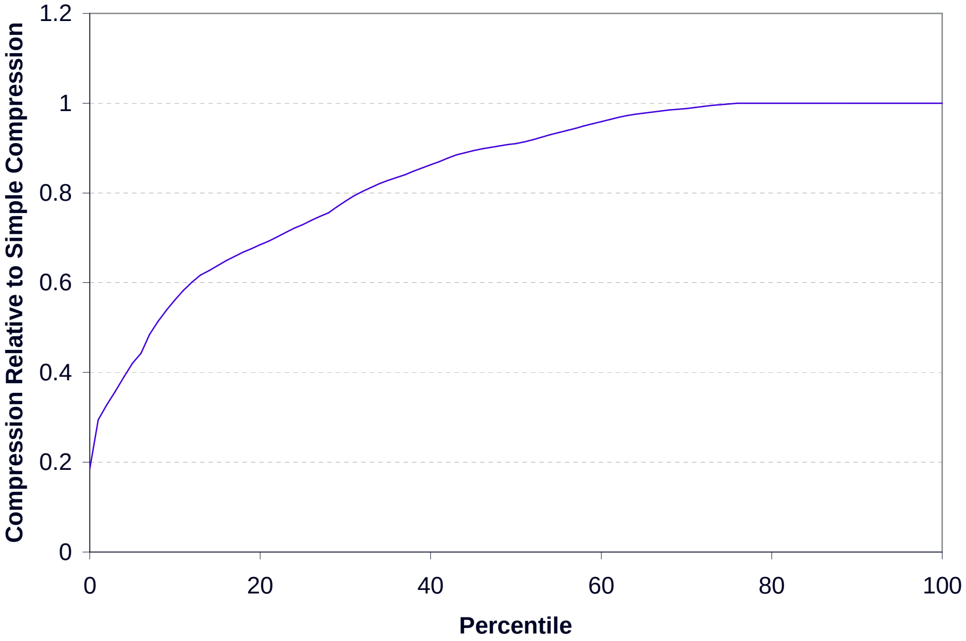


Figure 3.7: TSP compression ratio relative to simple compression, in terms of overall data structure size.

In both graphs, it can be seen that the TSP heuristic usually improves the compression ratio; unfortunately, this comes at the cost of additional computation needed to apply the TSP heuristic (which, for a table with R rows and C columns, requires $O(C^2R)$ time and $O(C^2)$ space for the TSP cost matrix).

With TSP compression, the bitvectors account for 6.15% to 54.1% of the storage requirements for the compressed representation of the crossproducting tables in this experiment, with the median value being 17.91%; the increase over the non-TSP case is due to the compressed arrays containing fewer elements with TSP, while the bitvectors have the same in both cases.

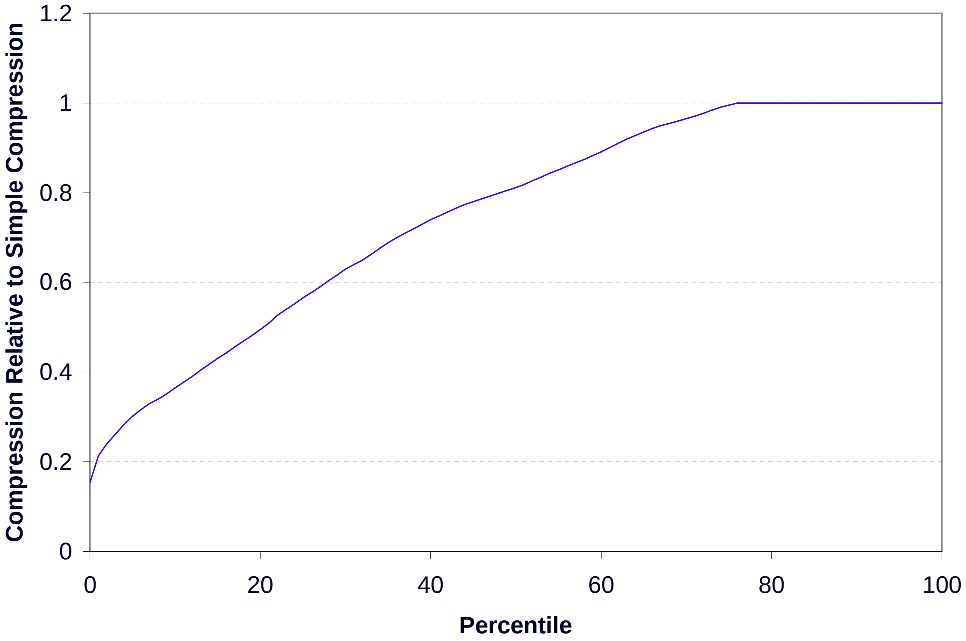


Figure 3.8: TSP compression ratio, relative to simple compression, in terms of size of crossproducting tables.

3.3 Performance on Larger Databases

In this section, we briefly explore the idea that larger classifiers may, in general, produce crossproducting tables that are more compressible than smaller classifiers. This would be a beneficial property, since it is typically with the larger classifiers that storage space becomes a concern.

To explore this idea, we apply the TSP-guided compression scheme to subsets of a larger filter database, varying the number of rules each time. In this experiment, we are considering only one reduction tree, selected arbitrarily from those having average results with the other database. The results are shown in Figure 3.9.

These results suggest that, for typical filter database, a larger number of rules tends to result in more compressible arrays. As the number of filters increases, the reduction in size of the crossproduct table tends to increase (exceeding 60% at around 250 filters.) The sharp jumps in the overall compression efficiency in Figure 3.9 appear to occur when the addition of a particular rule greatly increases the size of the crossproducting tables; note that the compression ratio of the crossproducting tables remains approximately the same, but the overall compression ratio jumps since the crossproducting tables are now a larger part of the whole.

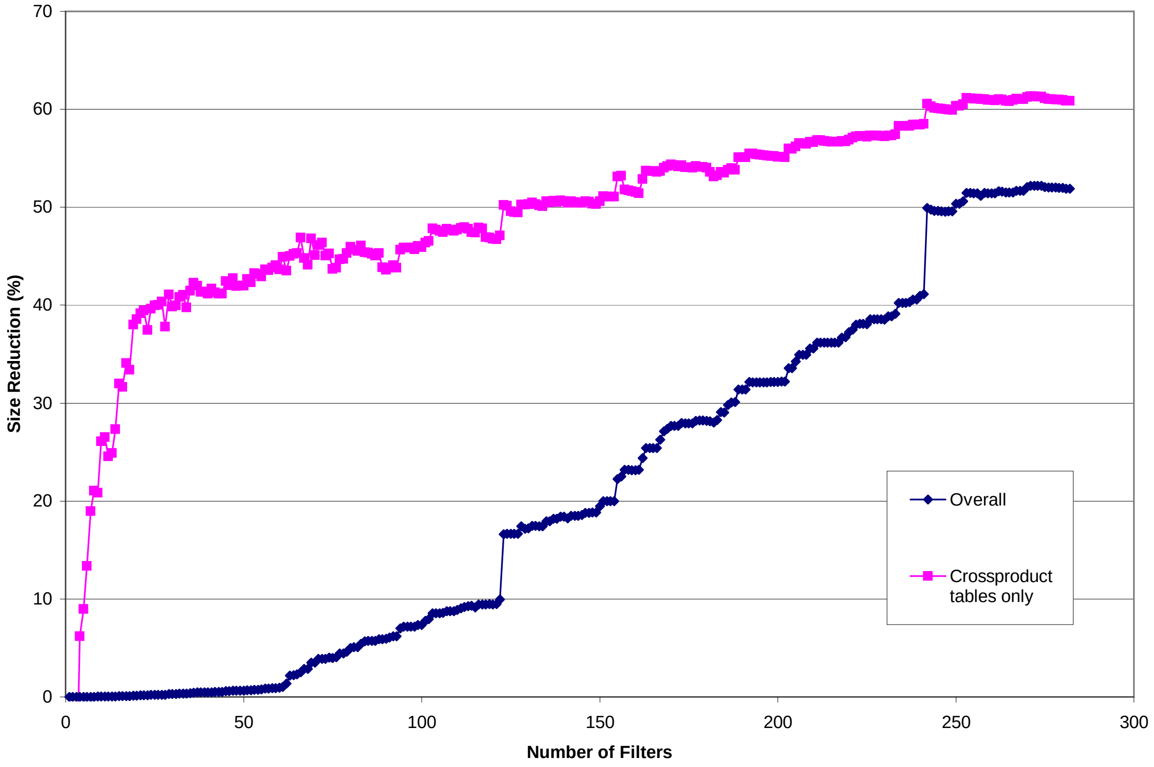


Figure 3.9: Compression Efficiency vs. Number of Filters.

3.4 Choice of Reduction Trees

The choice of reduction tree (explained in Section 2.2.3) affects the size of the data structure created. This is true for both the uncompressed representations and the compressed representations of the data structures.

In [19], two heuristics for selecting the reduction tree are mentioned: (i) combining chunks with the most "correlation," e.g. the two 16-bit chunks of IP source address, as soon as possible, and (ii) combining as many chunks as possible without causing unreasonable memory consumption. In our studies of RFC, we follow heuristic (ii) by only combining chunks two at a time. The "correlation" for heuristic (i) is not clearly defined, so we consider all possible combinations. This allows us to see exactly how much of a difference the choice of reduction tree can make, in terms of the RFC data structure size.

3.4.1 Impact of Reduction Tree Selection

The impact of reduction tree selection can be seen in the results summarized in Table 3.2 and Table 3.5. The average uncompressed data structure required 12,196,446 bits of storage, but the most efficient one required only 3,113,672 bits; this represents a 74.5% reduction in size. So, although compression can help, proper selection of the reduction tree can help more.

3.4.2 Finding an Optimal Reduction Tree

Now that we have seen how important the selection of the reduction tree can be, we turn our attention to the question of how to select a tree resulting in minimal storage requirements. Certainly, the exhaustive search technique of instantiating each possible tree and checking its size will find the optimal tree; that method, however, generally requires far too much compute time to be practical. For example, if the header is split into 7 chunks, and we restrict ourselves to only considering *binary* reduction trees (i.e. each node in the tree has exactly two children), then there are 665,280 different reduction trees that must be created.

We therefore consider a more efficient means for finding an optimal reduction tree; the approach described in this chapter uses dynamic programming techniques to do this.

This approach runs in a series of steps. In step i, the algorithm finds, for each unique set of $i+1$ header chunks, a optimal reduction tree with exactly those chunks as leaves.

Step 1 is fairly straightforward: for each pair of chunks (j, k), the algorithm compares the size of the crossproducting table with left child j and right child k against the size of the crossproducting table with left child k and right child j. The algorithm chooses the table with the smaller total size.

The uncompressed size of each of the two possible crossproducting tables will be the same; this is because the number of elements in each is simply the product of the number of equivalence classes for chunks j and k by themselves, and each element will require $\lceil \log_2 m \rceil$ bits, where m is the number of distinct equivalence classes produced by combining j and k.

The compressed sizes, however, may differ; this occurs because one case may involve more runs of repeated elements than the other, when stored in row-major order. Therefore, when seeking a reduction tree with minimal *compressed* storage requirements, all candidate crossproducting tables must be computed (since the compressed size will not be known until we can see how much compression can be done.) When seeking a reduction tree with minimal *uncompressed* storage requirements, only the table with minimal size is computed, since the uncompressed size of the candidate tables can be determined without filling them all out (i.e. it does not depend on compression.)

In the second and subsequent steps, the algorithm has more choices to make. For example, in step 2, for each trio of unique chunks (j, k, l) the algorithm searches for an optimal subtree containing exactly those chunks as leaves; if we are only considering binary reduction trees (because the resulting two-dimensional reduction tables will generally be smaller than tables with three or more dimensions) then it must consider every pair of subtrees from the previous step(s) that, when combined, have leaves i, j, and k. Over all of these, the algorithm selects the subtree with minimal total size for its tables.

Note that, before step i, the algorithm has already found optimal subtrees for all combinations of i chunks. Thus, the algorithm is not building subtrees from scratch in each step; it is building subtrees by combining the smaller subtrees built in the previous steps.

More formally, we can express the algorithm to find an optimal binary reduction tree as follows. Let C represent the set of header chunks used by the classifier, and let k represent the number of such chunks. Variables S_t, L_t, and R_t are subscripted t, where t is a subset of C indicating some collection of header chunks. One could instead think of t as a vector of bits, with one bit for each chunk used by the classifier; this would require more complicated notation in the algorithm's pseudocode, but may make implementation easier. The algorithm can then be expressed as follows:

```
for each $t \in C^*$ do
    $T_{|t|} \leftarrow T_{|t|} \cup t$
end for
for $i = 1$ to $k$ do
    $S_i \leftarrow 0$
end for
for $i = 1$ to $k - 1$ do
    for each $t \in T_{i+1}$ do
        $S_t \leftarrow \infty$
```

```
10:       $LC \leftarrow t^* - \{\emptyset \cup t\}$
11:       for each $l \in LC$ do
12:         $r \leftarrow t - l$
13:         $s \leftarrow$ size of table for $(l,\ r)$
14:         $newsize = S_l + S_r + s$
15:         if $newsize < S_t$ then
16:           $L_t \leftarrow l$
17:           $R_t \leftarrow r$
18:           $S_t \leftarrow newsize$
19:         end if
20:       end for
21:     end for
22: end for
```

The algorithm builds the following data structures in a dynamic programming fashion: S_t represents the total size of the crossproducting tables for the best subtree found having exactly the chunks in set t as leaves. L_t and R_t represent the left child and right child (respectively) of the root node, in the best subtree found having exactly the chunks in set t as leaves.

Lines 1 and 2 create sets T_i, where set T_i indicates all subtrees with exactly i leaves; this is useful so we can iterate through the small subtrees first, and then larger and larger subtrees.

Lines 3 through 5 initialize the "total size of crossproducting tables" measure (S_i) for the subtrees of size 1. These subtrees have no crossproducting tables, so the total size for them is zero. (We could instead initialize each of these to the size of its chunk's lookup table, in which case S_i becomes a measure of the *total* data structure size, i.e. not just the crossproduct tables.)

The **for** loop starting at line 6 iterates through the aforementioned *steps* of the algorithm. In step i, we find an optimal subtree for each possible combination of i header chunks; thus the **foreach** loop starting at line 8 iterates over all such subtrees.

Line 9 initializes the "total size" variable for S_t to infinity, since we have not yet processed any subtrees with leaves indicated by t.

Line 10 creates set LC, which is a set of candidates for the left child; for each candidate left child, the algorithm determines what the corresponding right child must be (line 12). It then computes the size of the crossproducting table for combining the left child and right child; that value is added to the size of the best subtree found for the left child and the best child found for the right subtree. If the total size is better than the (previous) best value found for this subtree, then we update that "best" value (S_t) and record the left and right children that produce this value.

In the final step (step $k-1$), the **foreach** loop in line 8 runs for only one iteration, since there is only one element in T_k (i.e. a set containing all k elements from the set C.) Thus in this step, it computes an optimal binary reduction tree and stores its size in S_C. The structure of the tree can be found by traversing L_t and R_t values, starting with L_C and R_C.

3.4.3 Results

To evaluate the dynamic programming algorithm just described, we apply it to several real-world filter databases of various sizes. The results are shown in Table 3.6.

This dynamic programming algorithm is interesting, at least from a theoretical perspective, but it should be noted that its memory requirements makes it less useful for practical application. The high memory requirements stem from the need to retain the equivalence classes resulting from each subtree considered by the algorithm; these are necessary for the determination of how many unique equivalence classes are produced in a given crossproducting table, which in turn helps determine the size of its parent table in the tree.

If a classifier has N rules and m equivalence classes, then associating a bitvector of N bits for each equivalence class (to indicate which filters are matched by the class) requires Nm bits; additional memory is typically required for the reverse mapping, i.e. the ability to look up equivalence classes by identifying the set of filters matched. If m grows proportionally to N (as observed in [19]), or faster, then this is at best an $O(N^2)$ memory requirement. Thus the memory requirements of the tree selection algorithm become unwieldy beyond several thousand filters; in such a case, a more practical approach would be to randomly select a small number (around ten or twenty) of reduction trees, and choose the one with the best performance.

3.5 Summary

Recursive Flow Classification [19] is a packet classification technique which performs classification very quickly (e.g. 12 memory accesses for classification using the typical

Database	*Number of Filters*	*Minimum Bits Required (total)*	*Bits Per Filter*	*Storage for individual chunk lookups*	*Storage for crossproduct tables*
ifw1	280	3,813,448	13,619.46	72.2%	27.8%
ifw3	184	2,946,976	16,016.17	86.73%	13.27%
acl1	814	3,615,296	4,441.40	76.1%	23.9%
acl2	613	7,721,637	12,596.47	37.3%	62.7%
acl3	2,396	35,106,066	14,651.95	9.33%	90.67%
ipc_fwd	1,556	16,036,551	10,306.27	20.0%	80.0 &
ipc_inp	192	2,087,180	10,870.73	97.34%	2.66%

Table 3.6: Results from optimal reduction tree selection for real-world filter sets

Internet 5-tuple) but has relatively high storage requirements. In this chapter we have studied methods for reducing the amount of storage required by RFC.

The simple compression technique described in Section 3.1 reduced storage requirements for the classifier's crossproduct tables by 37% on average in the experiment. The overall storage requirements were reduced by 25% on average, but it appears that larger classifiers will be more compressible.

The efficiency of the compression can be improved by the TSP heuristic described in Section 3.2. This reduced storage requirements by 54% on average in the experiment. The efficiency of compression appears to increase with larger filter databases, but for sufficiently large filter databases (where compression is really needed) the computational cost of this heuristic becomes excessive.

The most benefit, in terms of reducing storage requirements, comes from proper selection of the reduction tree. Optimal reduction tree selection reduced the storage requirements by 74% over the average size in our experiment. A dynamic programming algorithm can be used to select an optimal reduction tree, but the memory requirements for such an algorithm are steep.

Chapter 4

Extended TCAMs

This chapter discusses a hardware approach to packet classification called Extended TCAMs. This method achieves classification at a rate comparable to TCAMs, but without TCAM's high power consumption and inefficient handling of range match fields.

This chapter is organized as follows: Section 4.1 provides a brief overview of the ideas behind Extended TCAMs; Section 4.2 describes the partitioning techniques used to group filters. A CMOS implementation of hardware range checking is provided in Section 4.3. Results for Extended TCAMs using both the partitioning and range check support are given in Section 4.4. A technique for handling dynamic filter database updates is discussed in Section 4.5. Section 4.6 describes the used of multi-level indexing. Finally, a summary is presented in Section 4.7

4.1 Overview

TCAMs (described in Chapter 2) are the most popular practical approach to high performance packet classification, but TCAMs suffer from high power requirements and inefficient representation of range match fields. A large TCAM at the time of this writing can store 18 Mb and requires up to 20 watts or more; this is particularly problematic in high performance routers requiring one or more such chips on each port. Implementation of range matching in TCAMs requires replication of rules; this causes typical filter databases to expand by a factor of two to six [42], thus increasing both transistor count and power dissipation. To address these issues, we introduce a packet classification technique called Extended TCAMs.

The Extended TCAM concept is based on two main ideas. First is the idea that power dissipation can be reduced by dividing the TCAM into partitions, and only activating a small subset of these on each query; this idea was inspired by [31], which uses a partitioned TCAM on the much simpler problem of IP address lookup. The second idea is the use of range-matching logic in the hardware itself to solve the range-match storage problem.

4.2 Partitioning of the TCAM

The most serious problem with using standard TCAMs for packet classification is their excessive power consumption. It has been observed [31] that the main component of power consumption in TCAM search is proportional to the number of entries searched; thus, to reduce a TCAM's power consumption, we can partition it into blocks of words, and only search a small number of block(s) when performing a lookup. For this to work, of course, an index mechanism is needed to determine which block(s) of the TCAM to search for a given query.

In the IP lookup case [31], a lookup in the index identifies a single bucket that needs to be searched; thus the index can actually be implemented simply by using one of the blocks inside the TCAM. Multi-dimensional packet classification, on the other hand, may require searching several blocks for filters; thus we use an indexing mechanism that can support this. Each Extended TCAM block has an associated *index filter*, which consists of the same match circuitry as one word in the Extended TCAM device. When this filter is matched, it enables its corresponding TCAM block for search.

So, a search in an Extended TCAM device works as follows: first, the index filters are searched (in parallel) to determine which blocks to enable; then, the enabled blocks are searched (in parallel) for matching filters. Each block then returns its highest priority matching filter, and a final priority resolution step returns the highest priority filter of those.

An example of an Extended TCAM is shown in Figure 4.1; for simplicity, we perform two-dimensional classification in the example, with a four bit range field and a four bit bitmask field. To perform a lookup for a packet with header field values (2, 10), we first check the index filters and determine that the second and fourth index filters match the packet. The search then progresses to the second filter block and the fourth filter block. In the second block we find the matching filter (1-2, 1x1x), and in the fourth block we find the matching filter (0-14, 1010).

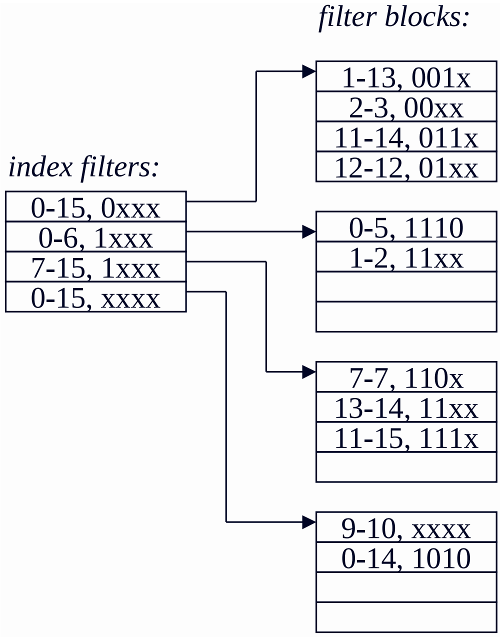


Figure 4.1: An example of filters and index filters in an Extended TCAM

When using a partitioned TCAM for packet classification, the key to making the search power efficient is to group the filters in such a way that only a few TCAM blocks must be searched for any given query. The current approach is to use a heuristic filter grouping algorithm to organize the filters. The lookup process itself is completely independent of the algorithm used to group the filters, i.e. the same Extended TCAM hardware will work regardless of what filter grouping algorithm is used.

This dissertation describes two specific algorithms for filter grouping. The first algorithm runs through a series of phases, and in each phase it divides the multidimensional classification space into subregions; it then uses these subregions to group the filters. The second algorithm organizes the filters by storing them in tries and then carving sections out of the tries; this approach is more directly inspired by [31] but includes modifications for application to the general multidimensional packet classification problem.

4.2.1 Region-Splitting Algorithm

The region-splitting algorithm runs in a series of phases. In each phase, a partitioning is made of the entire classification space; it is this partitioning that is used to group the filters. Later in this section we provide specifics for how the partitionings are chosen, but let us first build some intuition about what it means to split the regions.

Filter	*Source Port*	*Source Address*
a	1-13	001x
b	2-3	00xx
c	9-10	xxxx
d	11-14	011x
e	12-13	0xxx
f	0-14	1010
g	7	110x
h	0-5	1110
i	1-2	11xx
j	13-14	11xx
k	11-15	111x

Table 4.1: Example set of filters

Let us begin with a simple example using the filters shown in Table 4.1. These filters can be plotting in two dimensions, as shown in Figure 4.2. In this example, let us use an Extended TCAM with block size of four, i.e. each filter storage block can hold up to four filters.

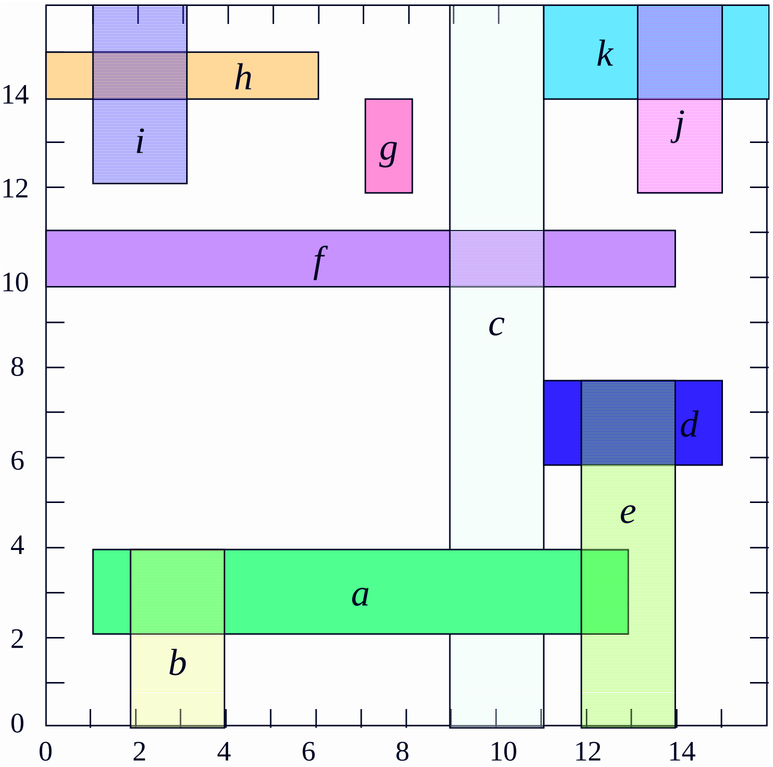


Figure 4.2: Example filters plotted in two dimensions

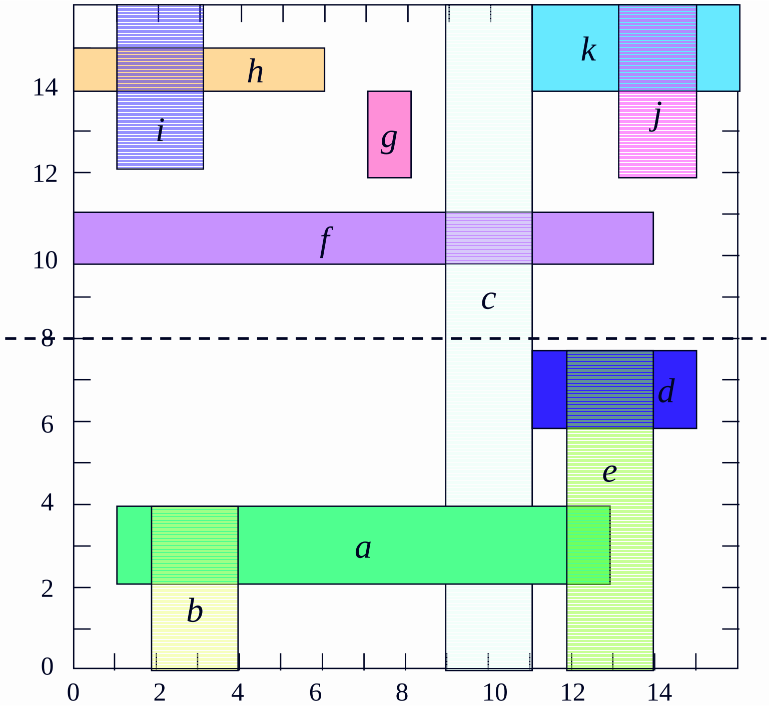


Figure 4.3: First split performed in region-splitting example

We begin the first phase considering the region spanning all of the two-dimensional classification space. For the sake of example, let us assume the algorithm decides to split that region along the dashed line shown in Figure 4.3. Note that the lower region (spanning source ports 0-15, and source addresses matching 0xxx) completely contains four filters (filters a, b, d, e), which will fit within a TCAM block. We can therefore allocate a TCAM block to correspond to that region, store these filters in the block, and set its index entry to the filter (0-15, 0xxx). And now we can remove filters a, b, d, e from consideration in future steps.

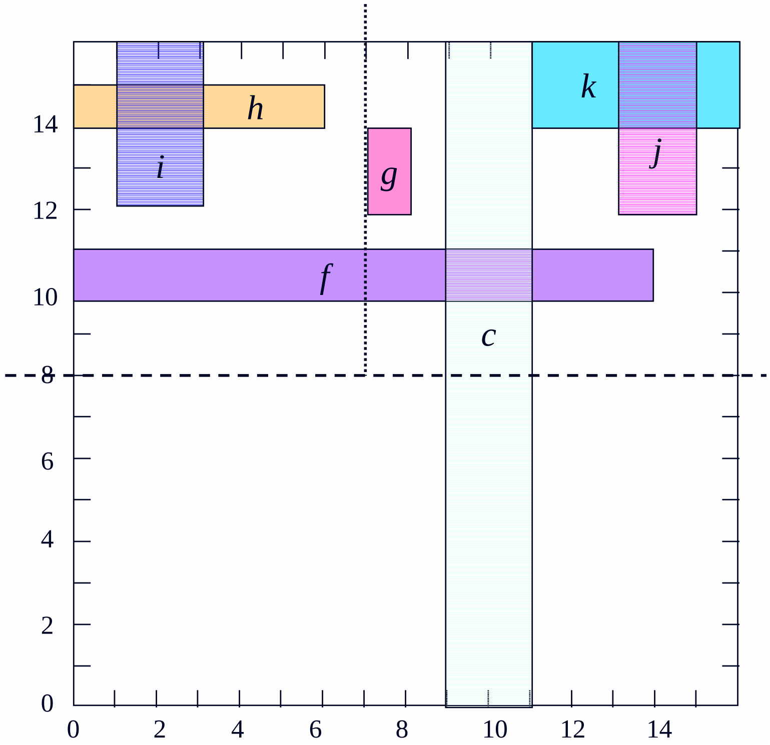


Figure 4.4: Second split performed in region-splitting example

We now turn our attention to the upper region. Suppose, for the sake of example, that the algorithm decides to split the upper region along the vertical dotted line shown in Figure 4.4. Note that the region in the upper left (spanning source ports 0-6 and source addresses matching 1xxx) contains two filters (h and i), which will fit in a TCAM block. We can therefore allocate a TCAM block to correspond to this region, store the filters in that block, and set its index entry to the filter (0-6, 1xxx). Now we remove filters h, i from consideration in future steps.

Now let us look at the upper right region spanning source ports 7-15 and source addresses matching 1xxx. This region contains three filters (g, j, k), which will fit in a TCAM block. We can therefore allocate a TCAM block to correspond to this region, store the filters in that block, and set its index entry to the filter (7-15, 1xxx). At this point we remove filters g, j, k from consideration in future steps.

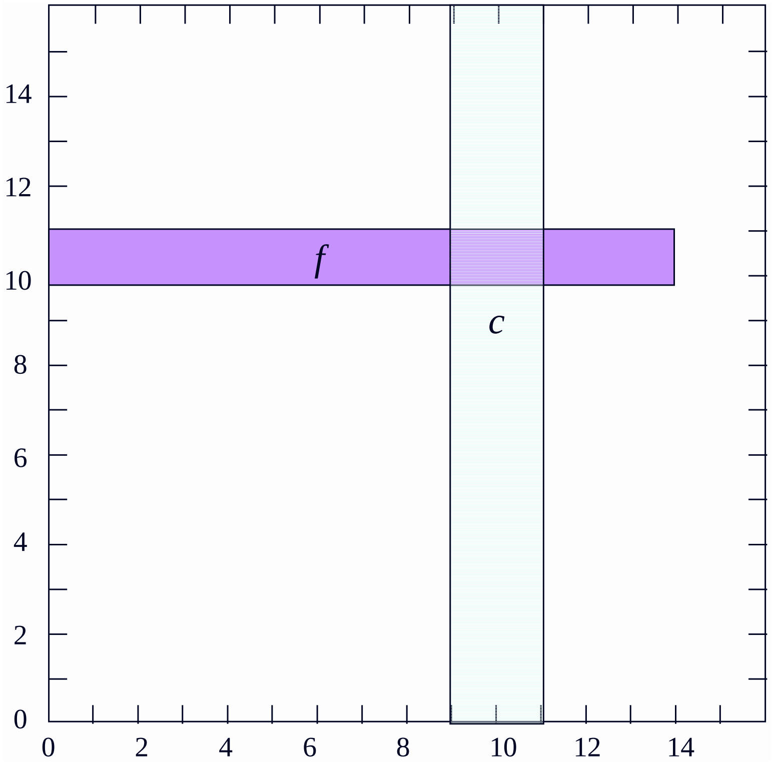


Figure 4.5: Second phase of region-splitting example

Filters c and f still remain (as shown in Figure 4.5), so we need to run the algorithm for another phase. We begin this phase considering the region spanning all of the two-dimensional classification space. This contains two filters (c and f), so there is no need to split it further. We allocate a TCAM block for this region, store filters c and f in that block, and set its index entry to the filter (0-15, xxxx).

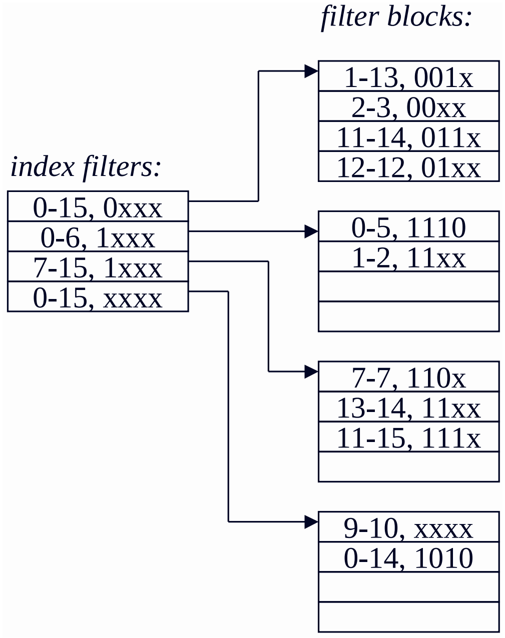


Figure 4.6: Contents of Extended TCAM for region-splitting example

After that, all filters have been assigned to storage blocks, so the algorithm is done. The contents of the Extended TCAM are shown in Figure 4.6. Note that a query corresponds to a point in the two-dimensional classification space, and thus any given query will fall into one of the first 3 regions and also into the last region; thus, for any query, we will need to search exactly two TCAM storage blocks (i.e. one of first three blocks, and also the last block). For example, suppose a packet arrives with a source port of 10 (decimal) and a source address of 0010 (binary). We first check the index filters and determine that the first and fourth index filters match; as a result, in the second step we activate the first and fourth filter storage blocks for searching. We find that filter (1-13, 001x) in the first block and filter (9-10, xxxx) in the fourth block match; we then perform priority resolution to determine that, of the matching filters, (1-13, 001x) occurred first in the original list.

As noted before, in each step of the algorithm, a region r is selected and cut into two sub-regions r_1 and r_2. A region can be defined by the filter specification which exactly covers that region. A cut along a bitmask dimension is performed by selecting one of the "don't care" bits and setting its value to 0 in one subregion and 1 in the other subregion; all other specifications are inherited from the original region. A range-matching dimension is split by dividing a range (lo, hi) into subranges (lo, m)

and $(m+1, hi)$ where $lo \leq m < hi$, and inheriting all other specifications from the original region.

A step of the algorithm, then, can be expressed as follows. Let F_i be the set of filters remaining to be processed at the start of phase i, and let S_i be the set of sub-regions created by the algorithm during phase i. At the start of phase i, S_i contains one region spanning the entire classification space. Let $\sigma(r)$ denote the set of filters in F_i that lie entirely within r. In all but the last phase, we repeat the following step, until no region r in S_i can be split into two sub-regions containing least k filters from F_i in each:

- Let r be a region, selected from S_i, with $|\sigma(r)| > k$.
- Consider cuts that divide r into two sub-regions r_1 and r_2 that satisfy $|\sigma(r_1)| \geq k$ and $|\sigma(r_2)| \geq k$
- Among all such candidate cuts, select one that maximizes $|\sigma(r_1) \cup \sigma(r_2)|$
- Remove r from S_i, and replace it with r_1 and r_2.

If no candidate cuts are found for a region, it is not split, and is not considered again in that phase. The phase terminates when no more candidate cuts can be found. At the end of the phase, a storage block is allocated for each region, and up to k filters from that region are placed in the block.

These splitting criteria differ from the criteria described in [42], which require tuning of parameters α and β to achieve a good partitioning. By splitting only when both sub-regions have at least k filters each, where k is the block size, we improve storage efficiency by avoiding the creation of a lot of blocks that are only partially filled.

During the final phase of the algorithm, the filters are allowed to span more than one region; if a filter spans more than one region, it must be stored in the blocks corresponding to each of those regions. This special handling in the last phase is not absolutely necessary, but can often reduce the number of phases needed (if there is a small set of hard-to-fit filters left towards the end) at a small cost of storage efficiency. Since a query will result in searching one block for each phase, the best power efficiency is achieved when the number of phases is minimized.

In the final phase, we also allow a region to be split even if the sub-regions do not contain at least k filters each. The basic step for the last phase can be expressed as follows, letting $\chi(r)$ denote the set of filters in F_i that intersect with r but are not completely contained within r:

- Let r be a region, selected from S_i, with $|\sigma(r) \cup \chi(r)| > k$.
- Consider the cuts that divide r into two sub-regions r_1 and r_2.
- Among all such candidate cuts, select one that maximizes $|\sigma(r_1) \cup \sigma(r_2)|$
- Remove r from S_i, and replace it with r_1 and r_2.

The last phase terminates when, for every region r in S_i, $|\sigma(r) \cup \chi(r)| \leq k$, or when a cut results in no decrease in $|\sigma(r) \cup \chi(r)|$. In the latter case, the final phase fails to include all filters; the algorithm must be re-run, specifying more phases. The algorithm can be re-run, increasing the number of phases each time until it complete successfully; or, most of the redundant computation can be avoided by rolling back to the start of the last phase, if it fails (or if storage efficiency in the last phase became undesirably low). Alternately, the algorithm can be run without the special handling of the last phase; this simplifies implementation, but the results in some cases are not quite as good.

4.2.2 Region-Splitting Results

We evaluate performance on filter databases which are synthetically generated (via ClassBench) using real filter databases as seeds. The seed database "acl1" is an Access Control List obtained from an ISP; "fw1" is a firewall database from another ISP; "ipc1" is derived from an IP Chains database used by another ISP.

To measure power efficiency, we define the *TCAM power fraction* as the ratio of TCAM bits searched using a partitioned TCAM to the number of TCAM bits searched using a nonpartitioned TCAM; this can be computed by the expression $(b + sk)/N$, where b is the number of filter storage blocks needed (and thus the number of index entries), s is the maximum number of filter blocks searched for any query (which equals the number of phases, when using the region-splitting algorithm), k represents the filter storage block size, and N is the number of filters in the database. For this metric, lower is better.

To measure storage efficiency, we define the *TCAM storage complexity* as the ratio of TCAM storage bits required, using a partitioned TCAM, to the number of TCAM storage bits required for a nonpartitioned TCAM; this can be computed by the expression $(b + bk)/N$. For this metric also, lower is better. Note that, in the case of the partitioned TCAM, index entries are to be included in these calculations, as well as the unused entries in any storage blocks only partially filled

Before we can obtain meaningful results, we must first decide how to select the filter storage block size. Figure 4.7 shows power fraction results for databases of varying sizes (8,000 filters to 64,000 filters) as we vary the block size. With a sufficiently large block size, we may only need to search one TCAM block for any given query. But, that TCAM block will likely contain a lot of filters that are not particularly relevant to the query received. Thus in that case we achieve poor power efficiency. On the other hand, if we use a very small block size, we will need a much larger total number of blocks; this means a larger index size, which again means poor power efficiency. The best choice lies somewhere between those two extremes, and appears to be proportional to the square root of the storage block size. Based on this data, we select a storage block of size equal to the largest power of two less than $\frac{1}{2}\sqrt{N}$, where N is the number of filters. Now that we have a method for selecting the storage block size, we can obtain some meaningful results regarding the algorithm's performance.

Power fraction results are shown in Figure 4.8. Even for the smaller filter databases, where a small TCAM can be used and thus power dissipation is a lesser concern, the results are good; in the case of the larger databases, where power dissipation is a critical issue, these results are *extremely* good.

For example, a standard TCAM large enough to hold one of the databases with 128,000 filters would consume 20 to 30 watts; an Extended TCAM implementation with a power fraction of 0.02 would only require 0.4 to 0.6 watts.

Storage complexity results are shown in Figure 4.9. In most cases the partitioning does not require much additional storage when compared to a standard TCAM implementation; given the substantial improvements in the power fraction, this is in most cases a very reasonable tradeoff to make. The two cases of high storage complexity in the chart are the result of the final phase allowing filters to span more than one block; in both cases a much lower storage complexity can be achieved (with a slight increase in power fraction) by allowing the algorithm to use one more phase.

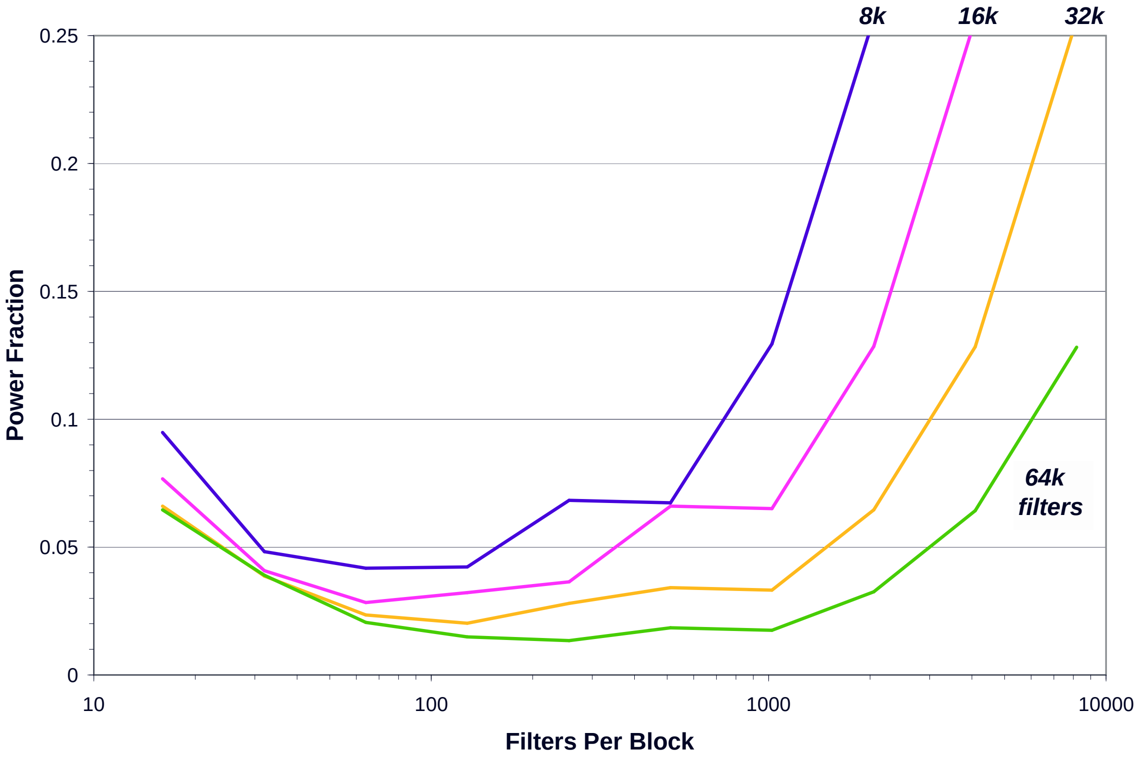


Figure 4.7: Power fraction results for different filter storage block sizes

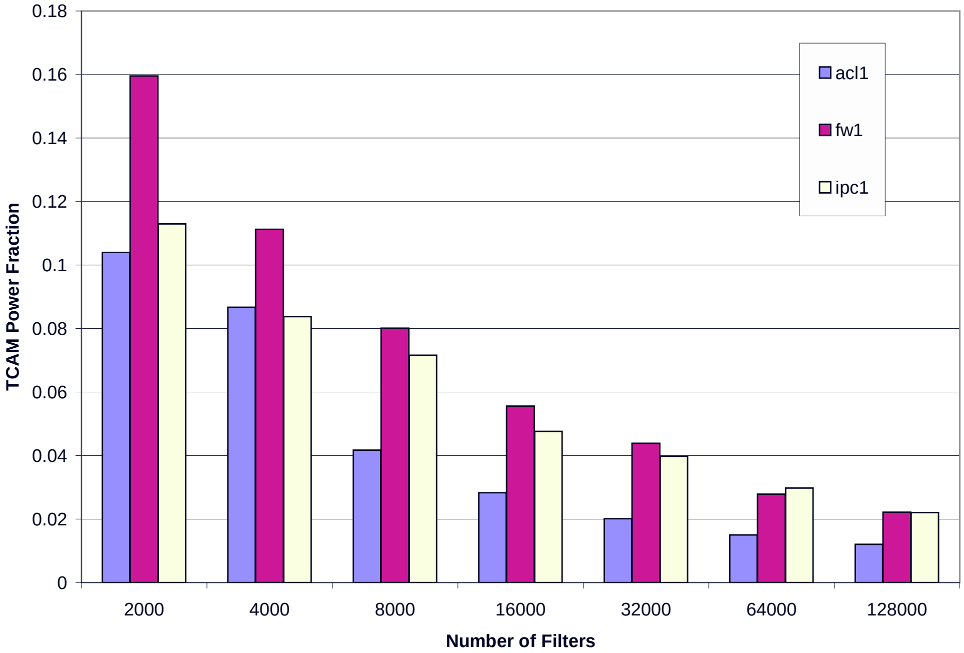


Figure 4.8: Power fraction results for region-splitting algorithm

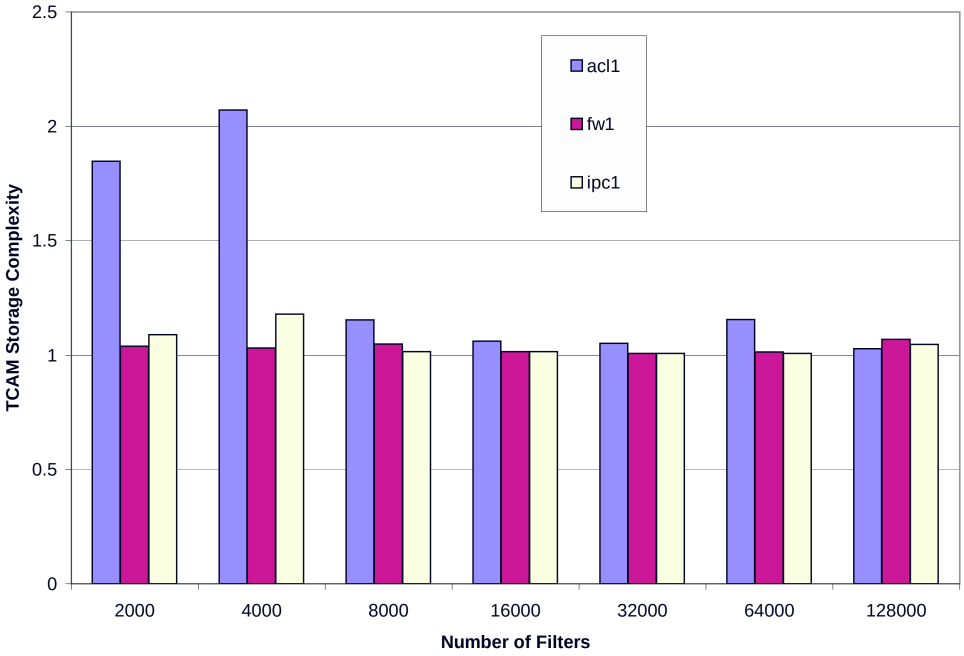


Figure 4.9: Storage complexity results for region-splitting algorithm

4.2.3 Trie-Carving Algorithm

In this section we present a trie-based filter grouping algorithm inspired by [31] as an alternative method for grouping filters. As we shall see, the trie-carving algorithm is simpler (thus more amenable to the handling of filter updates) and faster than the region-splitting algorithm, but not nearly as storage-efficient.

The trie-carving algorithm maintains a pair of tries containing filters; one trie is organized using the source address of the filters, and the other trie is organized using the destination address of the filters. Each filter is inserted into one of the two tries, and then filters are grouped by carving out sections of the trie. To decide into which trie a filter should be inserted, we examine the source and destination address fields of the filter and choose whichever one is more fully specified; in the case of a tie, the source address trie is used.

Filter	*Source Address*	*Destination Address*	*Protocol*
a	0010	1101	*
b	10*	0001	TCP
c	11*	10*	TCP
d	11*	*	*
e	00*	0*	UDP
f	01*	10*	TCP
g	01*	*	TCP
h	0*	10*	*
i	*	10*	UDP
j	1*	0*	*
k	*	0*	UDP
l	*	*	TCP
m	*	*	UDP

Table 4.2: Example set of filters

For example, consider the filters in Table 4.2, which are defined over a four bit wide source address, a four bit wide destination address, and a protocol value which can be either TCP or UDP. These filters would be placed in the Source Address trie shown in Figure 4.10 and the Destination Address trie shown in Figure 4.11. Each node is associated with a *count* value (shown below the node) which indicates the number of filters contained in the subtree rooted at that node.

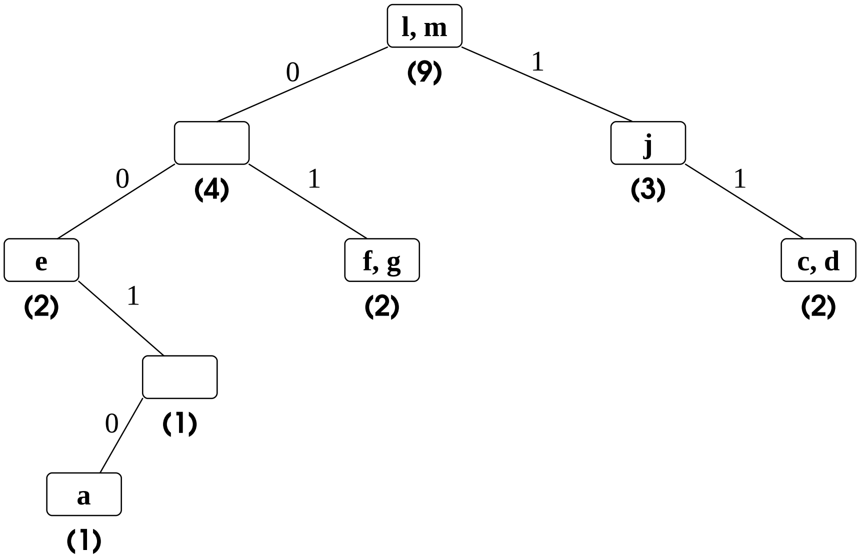


Figure 4.10: Source address trie for example filter set

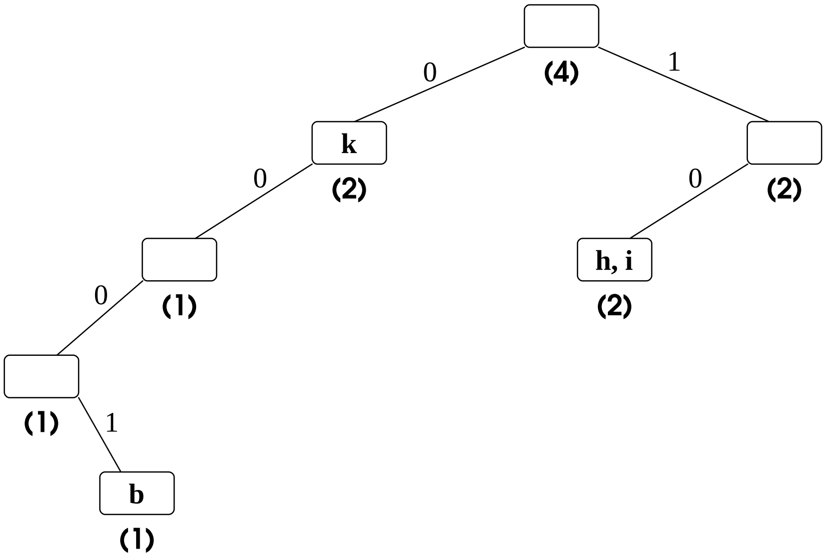


Figure 4.11: Destination address trie for example filter set

We use a carving algorithm which is similar to the *subtree-split* algorithm presented in [31]. The algorithm is run once for each of the two tries.

The carving algorithm performs a postorder traversal of the trie, considering each node as a possible *carving node*. A carving node, in this case, is a node whose count is at least $\lceil k/2 \rceil$ and whose nearest ancestor with greater count either has a count greater than k or does not exist.

When a carving node p is found, the algorithm carves out filters contained in the subtree rooted at p; the algorithm carves out as many filters as possible without resulting in a block at half capacity or less (except when there are no other filters left in the entire trie) and places those filters in TCAM blocks. The index entry for each such TCAM block is set to a filter wildcarded in all fields except the field to which the current trie corresponds; that field is set to the prefix which corresponds to the node p. Any filters remaining (in cases that would have produced underfull blocks) are stored at node p to be processed in further steps of the algorithm. Finally, the count of each ancestor of p is updated by subtracting the number of filters that were removed from the trie and placed into the TCAM.

The algorithm can be stated more formally as follows:

```
while there is a next node in post order do
  $p \leftarrow$ next node in post order
  $anc \leftarrow \text{parent}(p)$
  while $anc \neq$ null and $\text{count}(p) = \text{count}(anc)$ do
    $anc \leftarrow \text{parent}(anc)$
  end while
  if $anc =$ null or $(\text{count}(p) \geq \lceil k/2 \rceil$ and $\text{count}(anc) > k)$ then
    $FList \leftarrow \text{CarveOut}(p)$;
    $numToStore \leftarrow |FList|$
    if $\text{count}(p) \bmod k < \lceil k/2 \rceil$ then
      $numToStore \leftarrow numToStore - (\text{count}(p) \bmod k)$
    end if
    Remove $numToStore$ filters from $FList$ and store in TCAM blocks
    if $FList \neq \emptyset$ then
      Store the filters remaining in $FList$ at node $p$
    end if
    for each node $u$ along path from root to $p$, inclusive do
      $\text{count}(u) \leftarrow \text{count}(u) - numToStore$
      if $\text{count}(u) = 0$ then
        remove $u$
      end if
    end for
  end if
end while
```

The CarveOut procedure, which is called by the carving algorithm, consists of the following operations:

```
$retval \leftarrow$ filters($n$)
filters($n$) $\leftarrow \emptyset$
if leftchild($n$) $\neq$ null then
    $retval \leftarrow retval \cup$ CarveOut(leftchild($n$))
    delete leftchild($n$)
end if
if rightchild($n$) $\neq$ null then
    $retval \leftarrow retval \cup$ CarveOut(rightchild($n$))
    delete rightchild($n$)
end if
return $retval$
```

This algorithm has a total complexity of $O(N + NW/k)$ where N is the number of filters, W is the maximum prefix length, and k is the filter storage block size. This makes the trie-carving algorithm very appealing in terms of runtime.

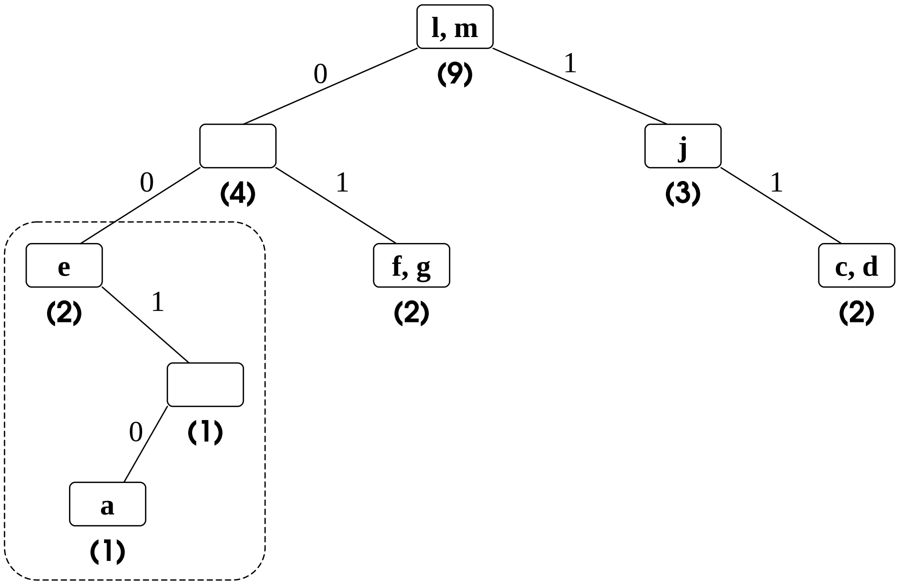


Figure 4.12: First carving step for source address trie

Let us apply this algorithm to the filters of Table 4.2, using left to right postorder traversals to traverse the tries; in this example, let us use a TCAM block size of three, i.e. $k = 3$. Figure 4.12 shows the first carving step applied to the source address trie. As a result of this carving step, filters a and e are stored in a TCAM block; that TCAM block's index filter is then set to the filter (00*, *, *), i.e. all fields wildcarded except source address, which is set to the prefix corresponding to the root of the section carved out.

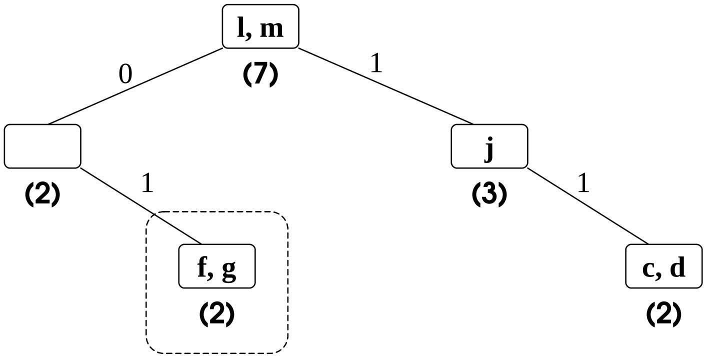


Figure 4.13: Second carving step for source address trie

The second carving step performed on the source address trie is shown in Figure 4.13. Here, filters f and g are placed in a new TCAM block, and its index entry is set to the filter (01*, *, *).

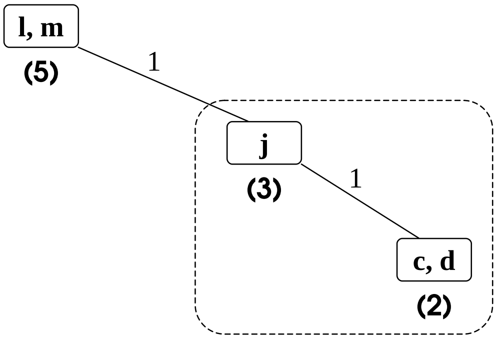


Figure 4.14: Third carving step for source address trie

Figure 4.14 shows the third carving step on the source address trie. In this step, filters c, d and j are stored in a new TCAM block; the index entry for the block is set to the filter (1*, *, *). Note that the *subtree-split* algorithm of [31] would perform the carving one level higher (at the node corresponding to 0* rather than 01*) in the trie. We shall say more about this in a moment.

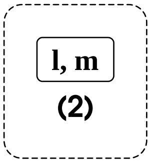


Figure 4.15: Fourth carving step for source address trie

The final carving step applied to the source address trie is shown in Figure 4.15. In this step, filters l and m are assigned to a new TCAM block, and the TCAM block's index entry is set to the filter (*, *, *).

Note that along any path through the original trie, we pass through nodes requiring activation of a total of 2 TCAM blocks. If we had used the original *subtree-split* algorithm for carving, a worst-case path through this trie would require activation of 3 TCAM blocks. The same groups of filters are produced, but by carving them lower in the trie we can often reduce the total number of blocks activated for any given search (thus lowering total power requirements).

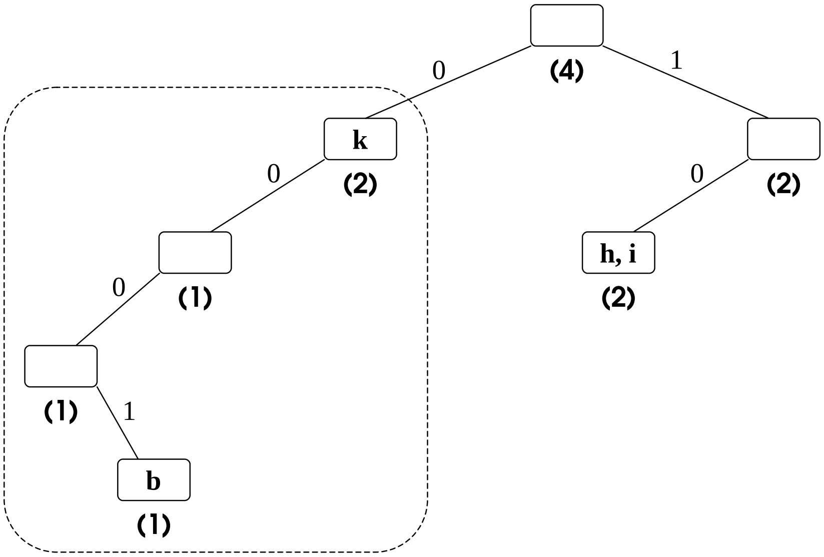


Figure 4.16: First carving step for destination address trie

Now that we have carved out all the filters from the source address trie, we shall process the destination address trie. The first carving step for the destination address trie is depicted in Figure 4.16. Filters b and k are stored in a new TCAM block, the index entry of which is set to the filter (*, 0*, *).

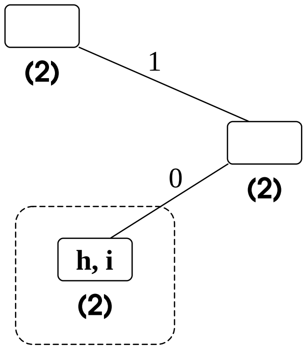


Figure 4.17: Second carving step for destination address trie

Figure 4.17 shows the final carving step applied to the destination trie. In this step filters *h* and *i* are placed in a new TCAM block with index filter set to (*, 1*, *). At this point, the algorithm has finished grouping the filters, which results in the Extended TCAM contents shown in Figure 4.18. The first four bits in each TCAM entry are used for the source address; the next four are for the destination address, and the last bit indicates the protocol (0=TCP, 1=UDP).

Unlike the IP lookup case, a query in the general multi-dimensional packet classification problem must search *all* buckets along the query's path in each trie; this is because we can not simply store a single covering filter for each bucket (to handle cases where the query matches a bucket but none of the filters in it) due to the multidimensional nature of the problem.

Therefore, to accomplish this, for each bucket we allocate a TCAM block and we store the bucket's filters in that block; we then set the block's index entry using the bucket's prefix (its location in its trie) in in the relevant field (source or destination address) and wildcards in all other fields. The worst-case power fraction depends on the maximum number of TCAM blocks activated for search, which equals the sum of the maximum number of filters along a path in each trie.

We should note that, under this approach, all filters that are wildcarded in both source and destination address are searched on all queries. While this is not a problem on the real-world filter databases studied in this dissertation (where only a limited number of filters are specified only on port and protocol information), it can nonetheless be addressed by introducing additional tries (or other carvable data structures) organized by the additional header fields.

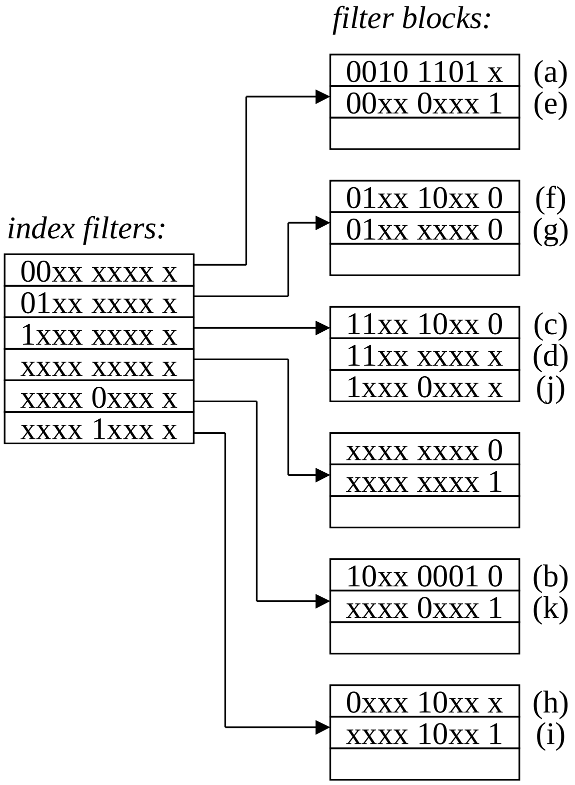


Figure 4.18: Extended TCAM contents for trie-carving example

4.2.4 Trie-Carving Results

We evaluate the Trie-Carving algorithm using the same performance metrics and input filter databases used for the Region-Splitting study in Section 4.2.2. To properly evaluate the Trie-Carving algorithm, we first determine how to select the block size. Figure 4.19 shows the power fraction obtained with four databases (of size from 8,000 filters to 64,000 filters) when varying the storage block size. Again we see that an excessively large or excessively small block size produces poor power fraction results, and again the largest power of two smaller than $\frac{1}{2}\sqrt{N}$ (where N is the number of filters) appears to be a good choice.

Power fraction results are shown in Figure 4.20. Again we see good results even for the smaller filter databases (where power dissipation is less of a concern) and excellent results for the larger databases. Despite this algorithm's lesser sophistication in grouping filters (i.e. that it only specifies one field in each index entry), its power fraction results compare well with the Region Splitting algorithm's results.

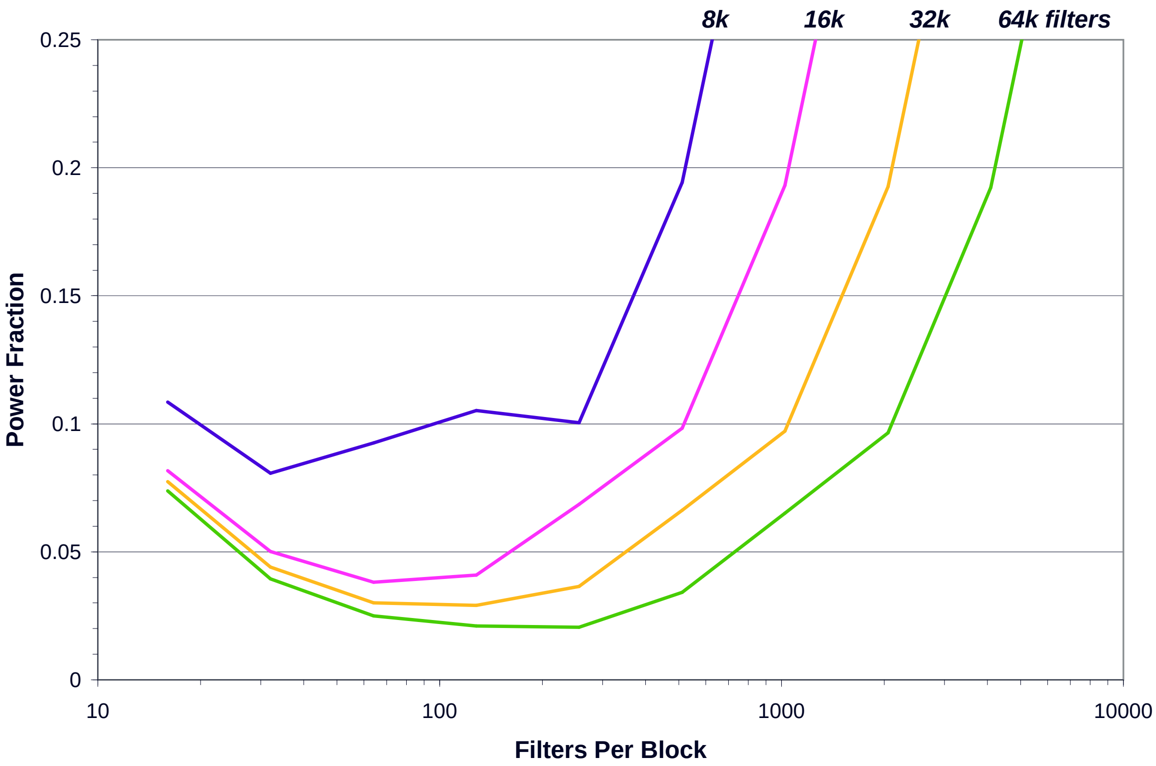


Figure 4.19: Power fraction results for different filter storage block sizes

Storage complexity results are presented in Figure 4.21; from this chart we can see that the Trie-Carving algorithm is not as efficient as the Region-Splitting algorithm in terms of filter storage needed. In fact it is because the Trie-Carving algorithm is not as concerned about storage efficiency that it is sometimes able to achieve better a power fraction than the more sophisticated Region-Splitting approach. But even with its higher storage complexity, trie-carving is still attractive due to its simplicity and running time.

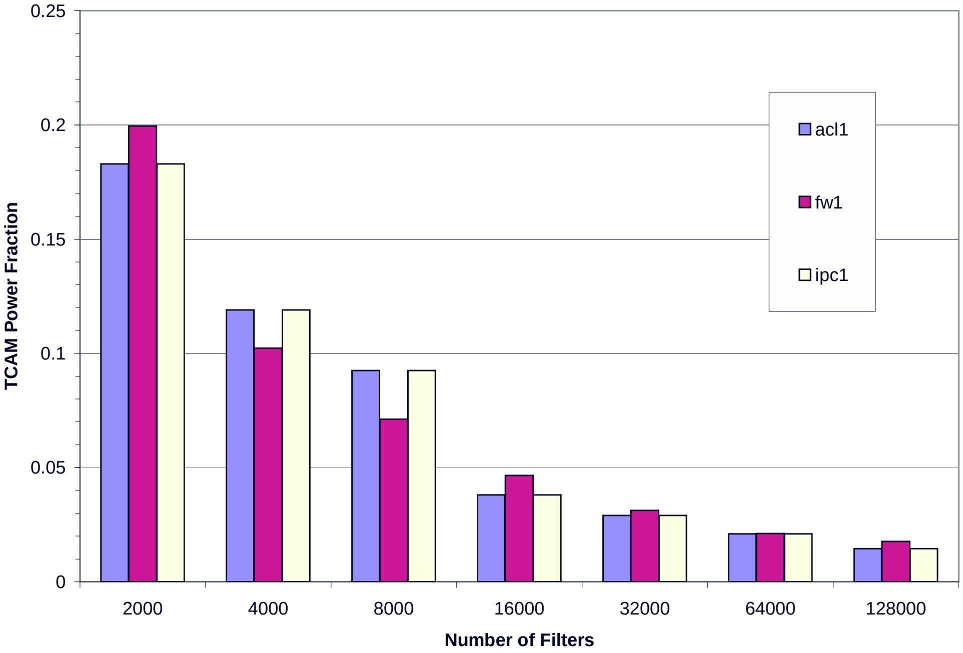


Figure 4.20: Power fraction results for trie-carving algorithm

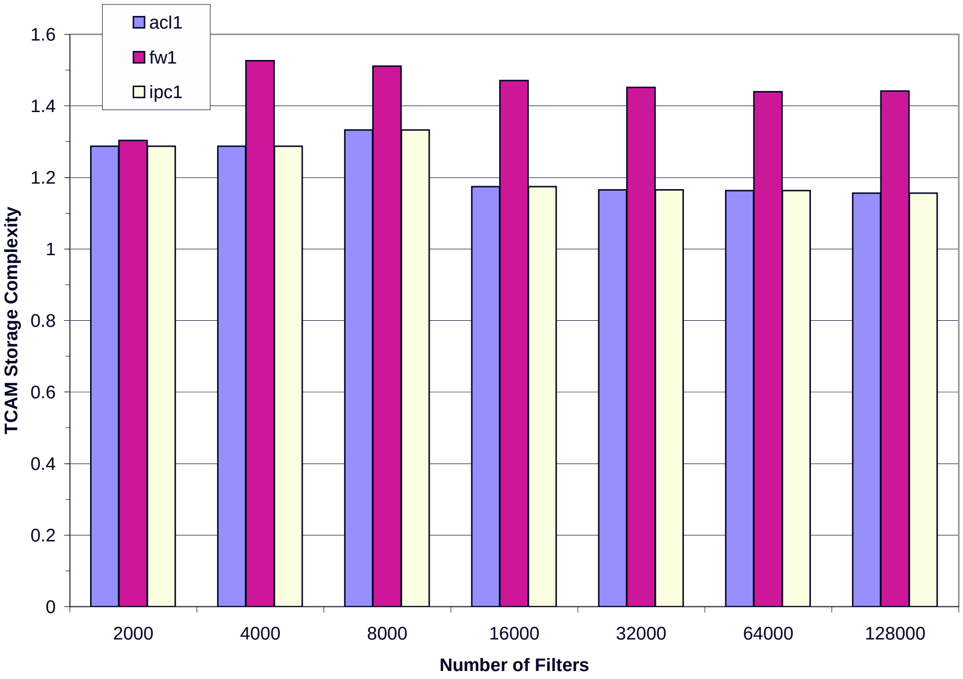


Figure 4.21: Storage complexity results for trie-carving algorithm

4.3 Range Check Hardware

In addition to the problem of high power consumption, TCAM-based packet classifiers are further encumbered by serious inefficiencies in representing range matches. Ranges are handled in a standard TCAM by replacing each filter with several filters, each using a prefix match that covers a portion of the desired range. For example, in a four bit wide field the range 2-10 can be expanded into the bit patterns 001*, 01*, 100* and 1010, which exactly cover that range.

In this manner, any sub-range of a k bit field can be represented as a set of prefixes, requiring up to $2(k-1)$ prefixes per sub-range. So, a 16-bit port field can require as many as 30 distinct TCAM entries. But if ranges are present in both the source and destination port fields, then we need a filter for *each combination* of the sub-ranges for the two fields. Thus a single packet filter may require 900 TCAM entries. In practice, real-world filter sets expand by a smaller yet still significant factor of about two to six times.

Therefore, in this section we describe range check circuits which avoid the need for rule replication. The most efficient of these designs allows classification on the standard IPv4 5-tuple with a 46% increase in transistor count, by using range check hardware for the port fields; this increase in transistor count, however, is more than offset by the fact that rules are no longer expanded to multiple entries. Also, since there is now a one-to-one correspondence between filters and TCAM entries, the range check support greatly simplifies creation and maintenance of the TCAM contents.

This is accomplished by using standard TCAM logic for the source and destination IP addresses, transport protocol number, and transport protocol flags, and range match logic for the transport layer port numbers. E.g. each Extended TCAM [42] word might have 88 bits of standard TCAM logic, and two 16-bit wide range match fields. Using 44 transistors per bit of range match, and 16 per bit of standard TCAM matching, this comes out to 2816 transistors per word vs. a standard TCAM's 1920 transistors per word.

It is also worth noting that range matching can also be applied to other fields in certain cases. For example, when using the third P^2C encoding scheme (see Section 2.3), efficient range match hardware would eliminate the need for storing multiple ternary match conditions per rule (i.e. rule replication), provided that the bits for each field are allocated in a contiguous fashion.

The implementations described in this section operate by storing the lower and upper bound for each range match, and using dedicated range check circuitry that performs the comparison in a set of stages. The difference in each design lies in how the range check sub-circuit is implemented.

Section 4.3.1 describes the most straightforward scheme, which uses four inter-stage signals. A design using three inter-stage signals is given in Section 4.3.2; a more efficient refinement of this design is described in Section 4.3.3. Finally, in Section 4.3.4 we show how to construct a more versatile device using range-check sub-fields that can be chained together as needed.

4.3.1 Implementation with four inter-stage signals

The first circuit presented is the most straightforward of the three designs. This circuit consists of a separate stage for each bit, and the comparison proceeds from the most significant bits to the least significant bits.

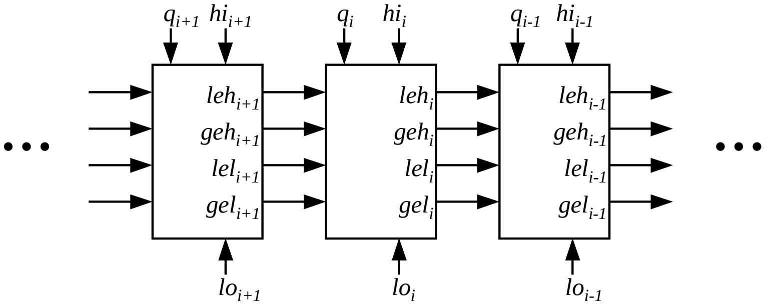


Figure 4.22: Iterative structure of range check circuit

The iterative structure of the circuit is shown in Figure 4.22, where hi_i, lo_i, and q_i represent bit i of the stored upper bound, the stored lower bound, and the query value, respectively. The four inter-stage signals geh_i, leh_i, gel_i, and lel_i are used to represent the following, where W is the width of a word:

geh_i: The quantity represented by the first $W-i$ bits of the query (i.e. q_{W-1} through q_i) is greater than or equal to the quantity represented by the first $W-i$ bits of the stored upper bound hi.

leh_i: The quantity represented by the first $W-i$ bits of the query is less than or equal to the quantity represented by the first $W-i$ bits of the stored upper bound hi.

gel_i: The quantity represented by the first $W - i$ bits of the query is greater than or equal to the quantity represented by the first $W - i$ bits of the stored lower bound lo.

lel_i: The quantity represented by the first $W - i$ bits of the query is less than or equal to the quantity represented by the first $W - i$ bits of the stored lower bound lo.

The names are abbreviations indicating that, up to bit i, the query is **g**reater (**l**ess) than or **e**qual to **h**i (**l**o).

The signals geh_W, leh_W, gel_W, and lel_W, which are inputs to the first stage, are always asserted. At that point, we can think of it as having compared the first zero digits (i.e. an empty string) from the query against the first zero digits of the upper and lower bounds (also empty strings.) The empty strings are equal; therefore those four signals are asserted.

The assertion of both leh_0 and gel_0 at the same time happens if and only if the query value q is within the range defined by hi and lo, inclusive; therefore, a "query is in range" signal can be formed by taking the logical AND of signals leh_0 and gel_0.

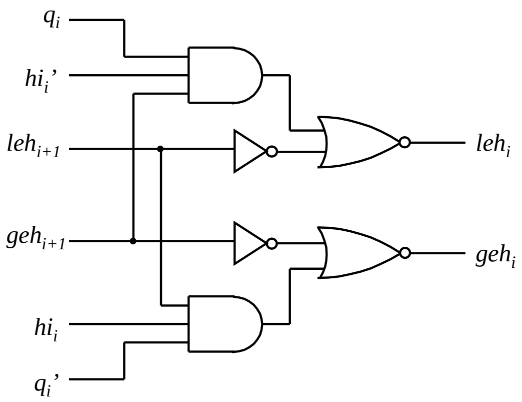


Figure 4.23: Range-check sub-circuit for upper bound comparison

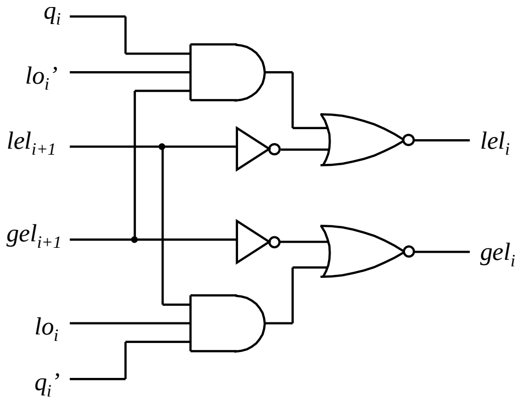


Figure 4.24: Range-check sub-circuit for lower bound comparison

The logic for the upper bound check of one stage is shown in Figure 4.23. If the query bits before this stage are greater than hi (i.e. leh_{i+1} is not asserted), then the query value is still greater than hi once this stage is included as well; therefore we ensure in this case that leh_i is not asserted. If, on the other hand, the query bits before this stage are not greater than hi, then there is only one condition under which the query value including this bit can be greater than hi. That condition is that the previous query bits equal hi (implied by geh_{i+1} and leh'_{i+1}, but we can omit leh'_{i+1} since that case already causes us to assert leh'_i), q_i is 1, and hi_i is 0; therefore in this other case we also ensure leh_i is not asserted. In all other cases we assert leh_i.

Each AND gate and OR gate pairing in the sub-circuit can be implemented as a compound gate using 8 transistors. Each inverter requires 2 transistors. Thus this part of the sub-circuit requires 20 transistors.

The logic for the lower bound check is shown in Figure 4.24; its operation is the same as the upper bound check, except that it uses lo_i (bit ith of the lower bound) instead of hi_i (bit i of the upper bound.) This part of the sub-circuit also requires 20 transistors.

Each stage requires an upper bound check (20 transistors), a lower bound check (20 transistors), and two SRAM storage cells (6 transistors each) for storing hi_i and lo_i. Thus 52 transistors are required for each bit in the query, using this design. The bounds checking logic for the first stage can actually be simplified somewhat, since geh_W, leh_W, gel_W, and lel_W are always asserted. Similarly, the final stage does not need to generate the signals geh_0 and lel_0. This can reduce the transistor count of those particular stages, but the middle stages still need 52 transistors each.

4.3.2 Implementation with three inter-stage signals

Using three inter-stage signals instead of four can allow us to reduce the transistor count, if we are sufficiently careful. This circuit also consists of a separate stage for each bit, similar to the previous circuit. The overall structure of the new circuit is shown in Figure 4.25.

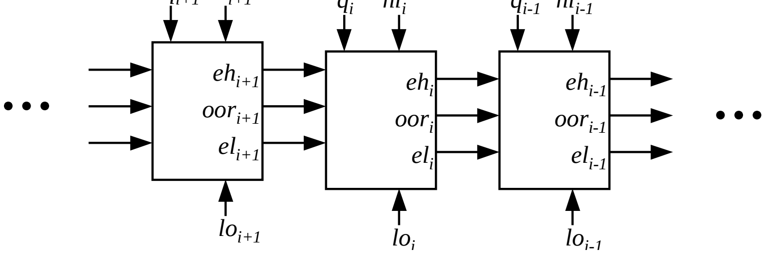


Figure 4.25: Iterative structure of range check circuit

The three inter-stage signals are:

eh_i: The quantity represented by the first $W-i$ bits of the query (i.e. q_{W-1} through q_i) is equal to the quantity represented by the first $W-i$ bits of the stored upper bound hi.

el_i: The quantity represented by the first $W-i$ bits of the query is equal to the quantity represented by the first $W-i$ bits of the stored lower bound lo.

oor_i: The quantity represented by the first $W-i$ bits of the query is out of range (i.e. is above hi or below lo.)

The names are abbreviations indicating that, up to bit i, the query is **e**qual to **h**i, **e**qual to **l**o, or **o**ut **o**f **r**ange.

Logic expressions for each inter-stage signal can be written as follows:

$$eh_i \equiv eh_{i+1} \wedge (q_i = hi_i)$$

$$el_i \equiv el_{i+1} \wedge (q_i = lo_i)$$

$$oor_i \equiv oor_{i+1} \vee (eh_{i+1} \wedge hi'_i \wedge q_i) \vee (el_{i+1} \wedge lo_i \wedge q'_i)$$

And we use as initial conditions that eh_W and el_W are asserted, and oor_W is not.

The final answer is determined simply by looking at the value of oor_0. This signal is asserted if and only if the query is out of range.

A circuit for generating the inter-stage signals is shown in Figure 4.26. This is the result of pulling out common subexpressions in the circuit, after rewriting the logic expressions as follows:

$$eh_i \equiv eh_{i+1} \wedge (q_i = hi_i) \equiv eh_{i+1} \wedge ((q_i \wedge hi_i) \vee (q'_i \wedge hi'_i))$$

$$el_i \equiv el_{i+1} \wedge (q_i = lo_i) \equiv el_{i+1} \wedge ((q_i \wedge lo_i) \vee (q'_i \wedge lo'_i))$$

Thus, this subcircuit requires only 36 transistors, including 6 for the inverters to generate eh'_{i+1}, el'_{i+1}, and oor'_{i+1}. Adding in the 12 transistors for the two SRAM storage cells (for storing hi_i and lo_i) brings the total to 48 transistors per bit. As before, the first stage circuitry can be somewhat simplified, given that the values of eh_W, el_W, and oor_W are fixed. And the last stage can be simplified, because the signals eh_0 and el_0 need not be generated.

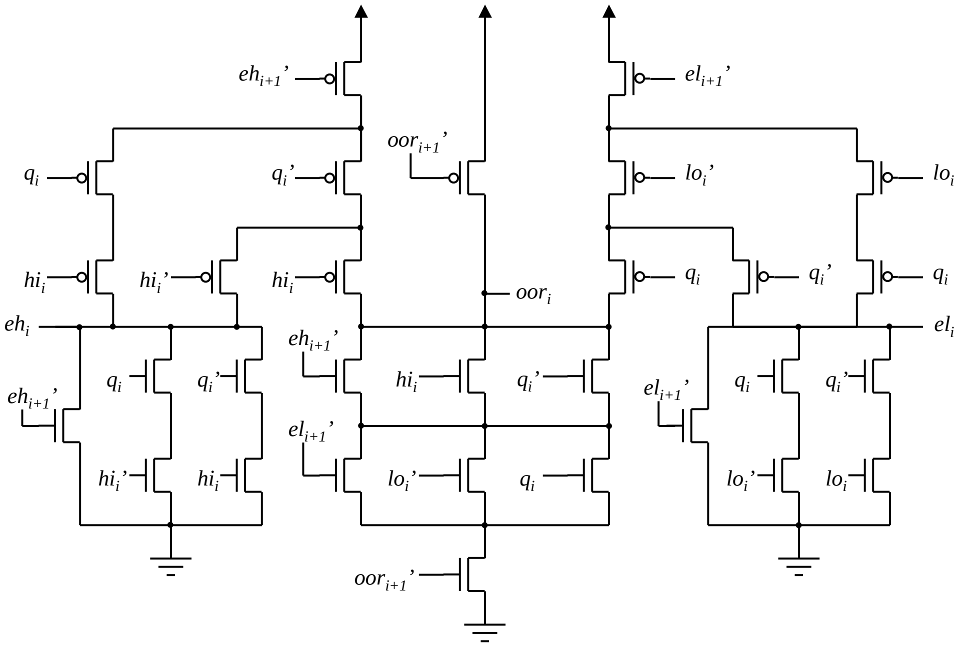


Figure 4.26: Range-check sub-circuit using three inter-stage signals

4.3.3 Revised implementation with three inter-stage signals

We can reduce the transistor count further by relaxing the constraints used to define two of the inter-stage signals in the previous circuit. The top-level structure of the new circuit, shown in Figure 4.27, is the same except for the labels of the inter-stage signals.

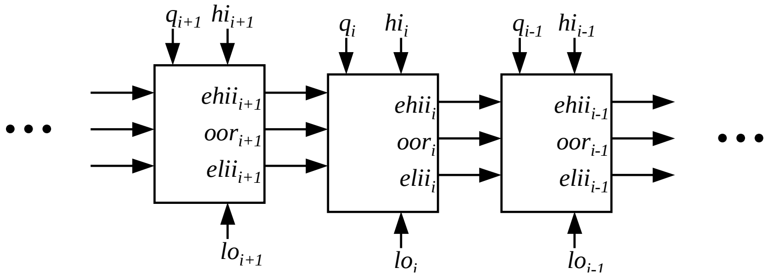


Figure 4.27: Iterative structure of revised range check circuit

The basic idea is this: once the query value is determined to be out of range, we no longer need the "equals *hi*" or "equals *lo*" signals. We can relax the constraints on the signals accordingly, as follows:

$$ehii_i \equiv \begin{cases} 0 & \text{if, up to and including bit } i\text{, } q \text{ is less than } hi \\ 1 & \text{if, up to and including bit } i\text{, } q \text{ is equal to } hi \\ \textit{undefined} & \text{if, up to and including bit } i\text{, } q \text{ is greater than } hi \end{cases}$$

$$elii_i \equiv \begin{cases} 0 & \text{if, up to and including bit } i\text{, } q \text{ is greater than } lo \\ 1 & \text{if, up to and including bit } i\text{, } q \text{ is equal to } lo \\ \textit{undefined} & \text{if, up to and including bit } i\text{, } q \text{ is less than } lo \end{cases}$$

$$oor_i \equiv \begin{cases} 0 & \text{if, up to and including bit } i\text{, } q \text{ is neither above } hi \text{ nor below } lo \\ 1 & \text{if, up to and including bit } i\text{, } q \text{ is either above } hi \text{ or below } lo \end{cases}$$

The inter-stage signal names are abbreviations as in Section 4.3.2, except *eh* and *el* have the string *ii* appended as a reminder that those signals are only valid **i**f the query is **i**n range. We use as initial conditions that $ehii_W$ and $elii_W$ are asserted, and oor_W is not.

As before, the final answer is determined simply by looking at the value of oor_0. This signal is asserted if and only if the query is out of range.

Since $ehii_i$ and $elii_i$ are undefined in some cases, we have more freedom in the implementation of this circuit than the previous one. The following set of logic expressions, for example, can be used:

$$ehii_i \equiv ehii_{i+1} \wedge (q_i \vee hi'_i)$$

$$elii_i \equiv elii_{i+1} \wedge (q'_i \vee lo_i)$$

$$oor_i \equiv oor_{i+1} \vee (eh_{i+1} \wedge hi'_i \wedge q_i) \vee (el_{i+1} \wedge lo_i \wedge q'_i)$$

The expression defining $ehii_i$ differs from the Section 4.3.2 definition of eh_i by replacing the quantity $(q_i = hi_i)$ with the quantity $(q_i \vee hi_i)$. The value of these quantities differ only when $q_i = 1$ and $hi_i = 0$, and this difference is only relevant when $ehii_{i+1} = 1$. But in that case, the query will be out of range (as detected by oor_i), which means that $ehii_i$ is undefined. Thus a value of 1 under those conditions is acceptable. A similar analysis applies to the expression for $elii_i$.

A circuit for generating oor_i is shown in Figure 4.28; it is essentially the portion of the previous design (Figure 4.26) that computes oor_i. This requires 14 transistors, plus 6 for the inverters used to generate $ehii'_{i+1}$, $elii'_{i+1}$, and oor'_{i+1}. Figure 4.29 shows the logic required to generate $ehii_i$, which requires 6 transistors. Figure 4.30 shows the logic required to generate $elii_i$, which also requires 6 transistors. Adding the 12 transistors for the two SRAM storage cells (for hi_i and lo_i) brings the total to 44 transistors per bit, using this design.

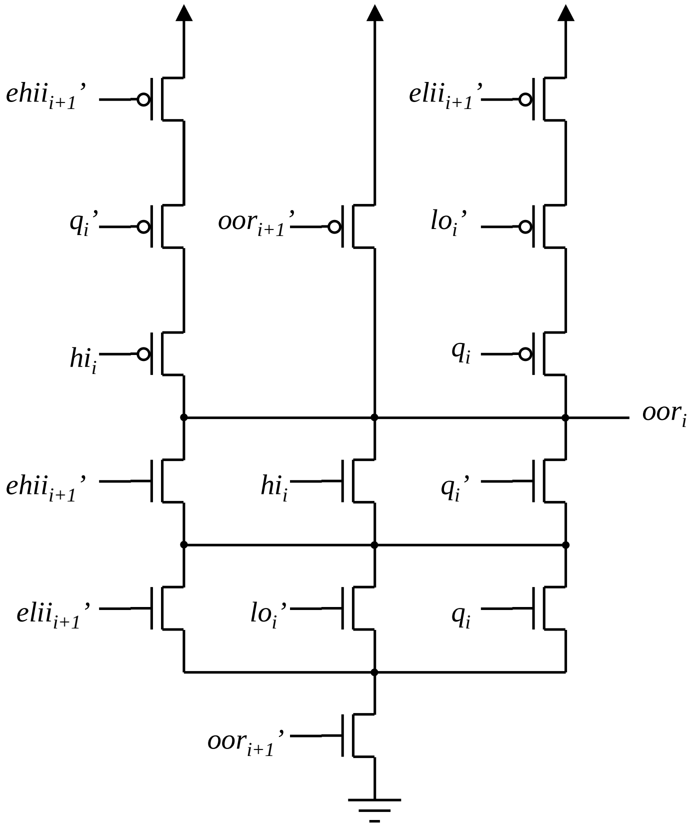


Figure 4.28: Sub-circuit for *out-of-range* signal

As in the previous design, the first stage circuitry can be simplified, given that the values of $ehii_W$, $elii_W$, and oor_W are fixed. The last stage can also be simplified, because the signals $ehii_0$ and $elii_0$ need not be generated.

4.3.4 Subfield Chaining

The aforementioned range match circuits are a good choice for a device that targets a particular application, where the number and width of range fields are known in advance. For situations where the manufacturer of a classification device does not know those parameters in advance, it would be nice to have some flexibility. One approach to this is to implement range matching via a set of smaller fields, chaining

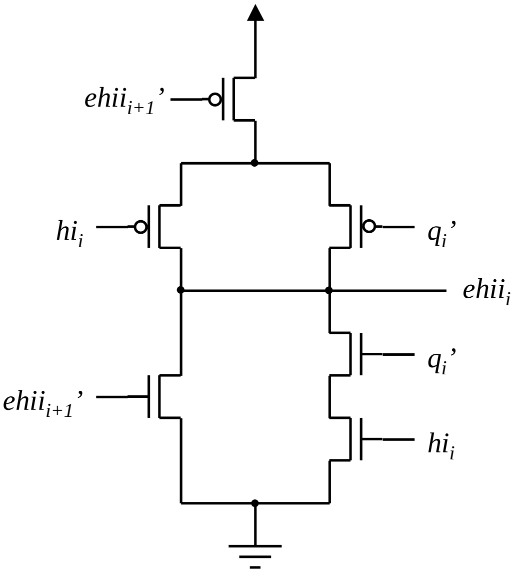


Figure 4.29: Sub-circuit for *equals-hi-if-in-range* signal

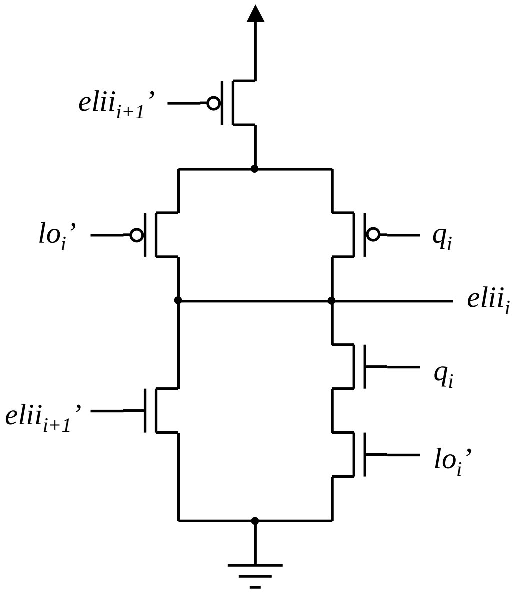


Figure 4.30: Sub-circuit for *equals-lo-if-in-range* signal

them together as needed. For example, if a device is built with subfields of 8 bits in width, then two such subfields can be chained together to perform a 16-bit range match, three can be chained to perform a 24-bit match, and so on.

This can be accomplished as follows: Consider a device with j subfields, each of width s and numbered $j-1$ through 0. Let the signal ch_i denote whether subfield i is chained to subfield $i-1$. Chaining subfield i involves an interaction between the stage for the least significant bit of subfield i and the stage for the most significant bit of subfield $i-1$. To accomplish this, each stage $si-1$ (for each integer i, where $1 \leq i \leq j$) uses signal ch_i to determine whether to use the inter-stage signals from stage si as inputs or not; if not, then the normal initial conditions are used (e.g. for the design in Section 4.3.3, use 1 in place of $ehii_{si}$ and $elii_{si}$, and 0 in place of oor_{si}.) An example of logic for generating oor_{si} for stage $si-1$ is shown in Figure 4.31. Figure 4.32 and figure 4.33 show the logic for $ehii_{si}$ and $elii_{si}$ respectively.

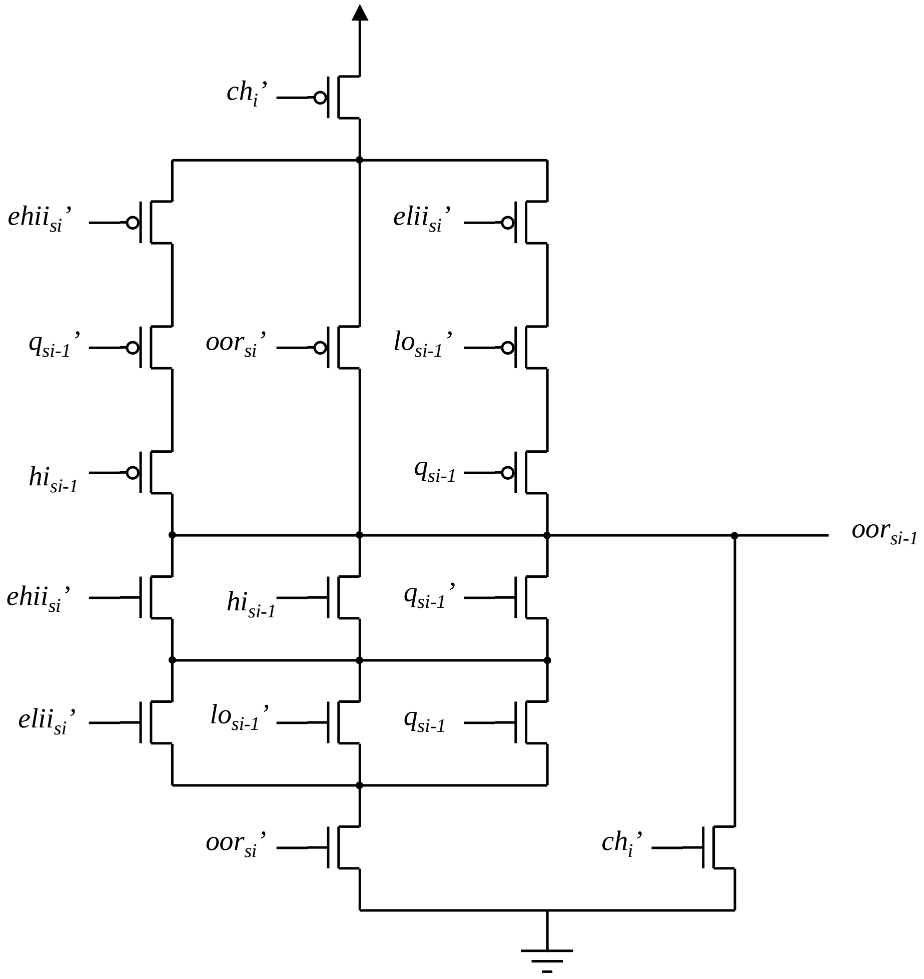


Figure 4.31: Sub-circuit for *out-of-range* signal for input to stage $si-1$

Whether all of the range fields in a word match can be determined by examining the *oor* signals for all subfields (i.e. oor_{si} for all integers i where $0 \leq i < j$), regardless of how the subfields are chained. A match is indicated when none of those signals are asserted. This works because, in a multi-subfield match, the first s bits must match, the first $2s$ bits, and so on, if the entire quantity matches.

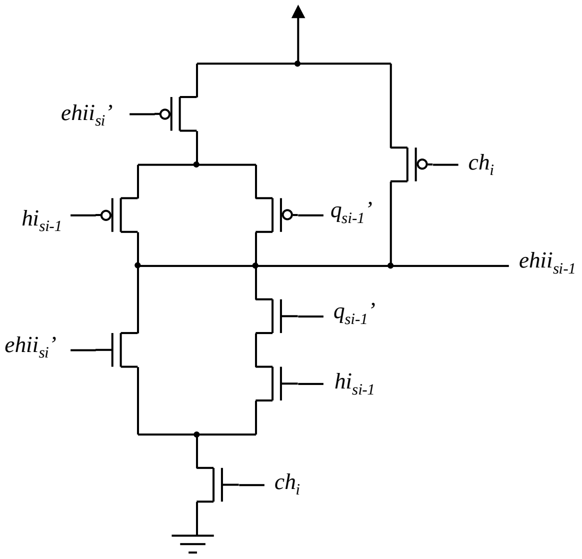


Figure 4.32: Sub-circuit for *equals-hi-if-in-range* signal for input to stage $si - 1$

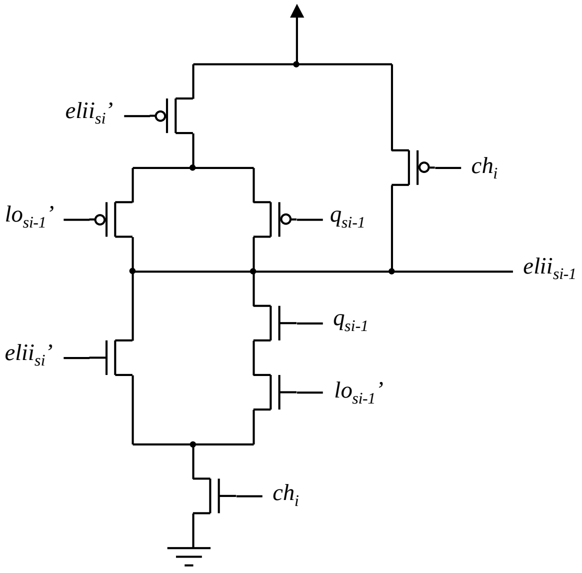


Figure 4.33: Sub-circuit for *equals-lo-if-in-range* signal for input to stage $si - 1$

There is a tradeoff between subfield width and versatility of the range matching device. A smaller subfield requires more overhead (for the additional "chaining enable" signals and logic), but results in finer granularity with respect to allocation of bits for range match fields; this can result in more efficient use of the available bits.

In the extreme case where subfield width equals one, range match fields can be created of any arbitrary width without wasting bits. Also in that case, range match bits can perform the same type of matching as standard TCAM bits, thus providing even greater flexibility. This style of matching is accomplished by using fields of width 1 and using lower and upper bounds of (0, 0) to represent 0, (1, 1) to represent 1, and (0, 1) to represent "don't care."

4.3.5 Summary of Range Check Hardware

In this section we have described three CMOS implementations of a range check circuit, the most efficient of which requires 44 transistors per bit. For a typical IPv4 application, this means a 46% increase in transistor count per word, but for typical filter databases it also means using a sixth to half as many words; in those cases, overall transistor count required is only 24% to 78% of the standard TCAM design, and power dissipation is reduced similarly. Also, the range check support greatly simplifies creation and maintenance of the TCAM contents, since there is now a one-to-one correspondence between filters and TCAM entries. In addition, we describe a means of chaining small range match subfields together; thus a range matching device can be configured for various numbers and sizes of range match fields, by chaining subfields as needed. In the most extreme case where subfields are one bit wide, the range match portion of a device can also perform standard TCAM-style matching.

4.4 Combined Results

Partitioning the TCAM results in greatly reduced power consumption, and using range-match hardware typically reduces both power consumption and overall storage complexity. In this section we show the results of using both extensions at the same time.

Figure 4.34 shows the power fraction obtained when using both extensions on the same input databases as before, for both the region-splitting (RS) and trie-carving (Trie) algorithms. In all cases the power fraction is below 20%, which is an excellent result; for larger databases, where power is of particular concern, we do even better, having power fraction in the one or two percent range. And, for databases beyond a certain size, the simpler and faster trie-carving algorithm produces power fraction results quite competitive with the region-splitting algorithm

Figure 4.35 shows the storage complexity results on the same databases. The storage complexity results depend very strongly on the database being used; the databases derived from fw1, for example, contain extensive use of port ranges and

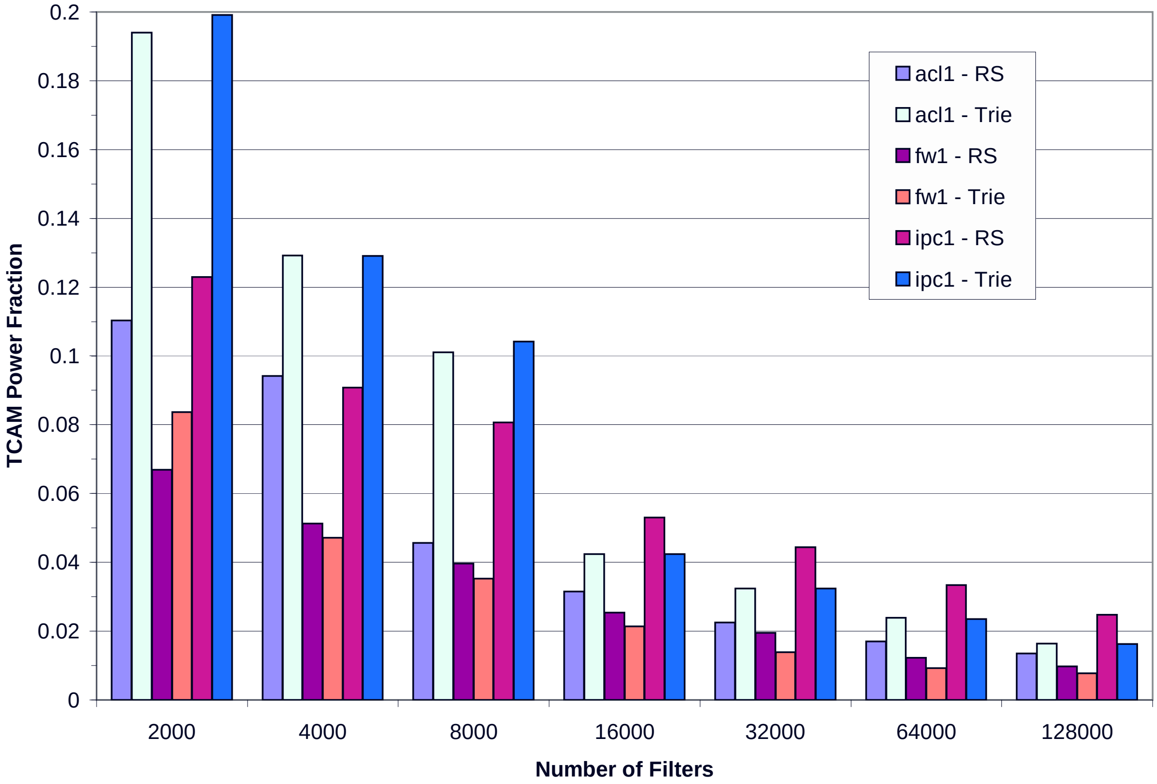


Figure 4.34: Power fraction results, including effects of range support

thus benefit greatly from the range checking hardware. The region-splitting cases with 2,000 and 4,000 filters derived from acl1, as noted before, have poor storage complexity which can be relieved by allowing the algorithm to run for another phase.

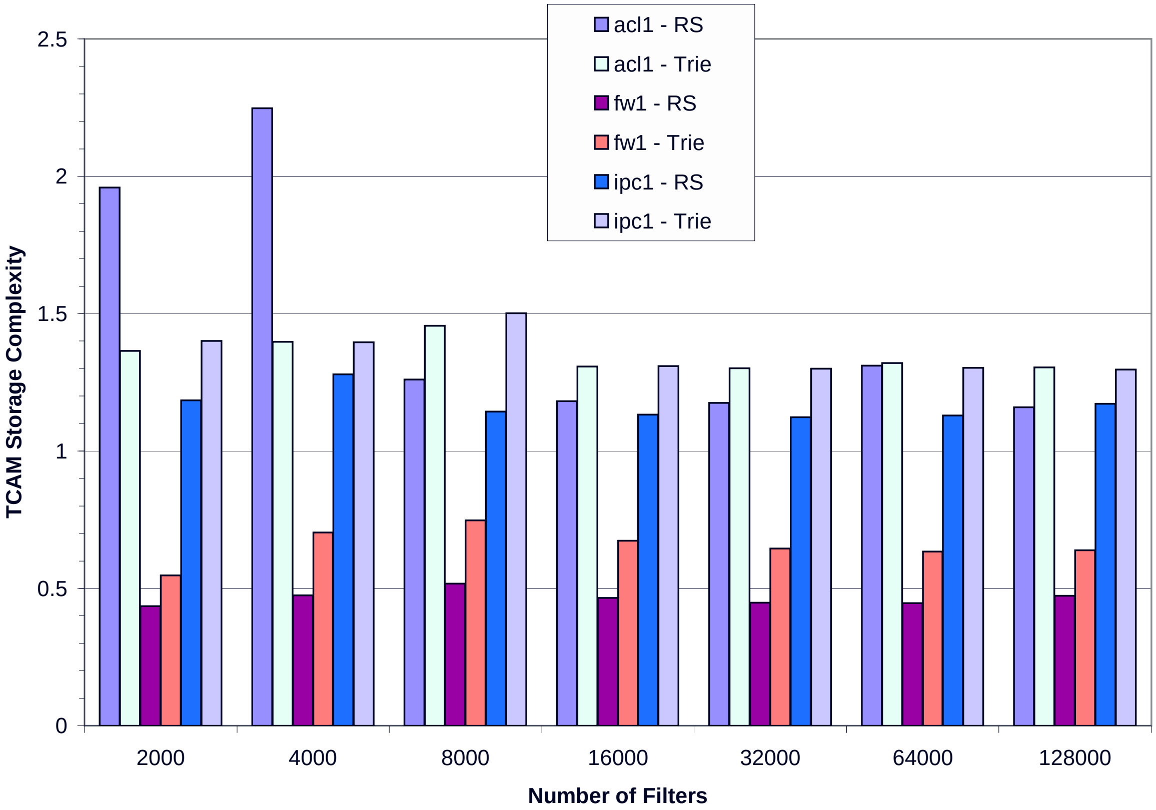


Figure 4.35: Storage complexity results, including effects of range support

4.5 Handling Filter Updates

In many packet classification applications, it is necessary to add or delete filters from the classifier, while maintaining high throughput in terms of packets classified per second; and we expect dynamic updates to be of particular importance to classifiers with truly large numbers of rules, since those are quite unlikely to be purely static filter databases. Thus it is highly desirable to devise methods for adding and removing filters from the Extended TCAM classifier efficiently.

Handling incremental updates is not a trivial thing in many of the modern high-performance classification schemes. HyperCuts, for example, does not at this time include an incremental update procedure. Woo's modular approach to packet classification [55] mentions that incremental updates are possible, but does not go into detail on the subject; furthermore, it turns out that it requires rebuilding the entire data structure in the worst case anyway.

If updates occur infrequently, and they need not be effected immediately, then it may be acceptable to recompute the entire data structure (TCAM entries, index entries, and the relationships created by the filter grouping algorithm). But, since the computation can take several seconds, this will not meet the performance needs of the more demanding packet classification applications. Also, it either precludes classification while the new data structure is loaded into the TCAM, or it requires a larger TCAM (to hold both new and old data) and a means of switching on the fly which data structure is used for classification. The latter approach could be accomplished by adding logic to the index entries, such that each index filter is also associated with either the new or the old data structure, and is enabled for search only when appropriate.

A slight improvement over the above is to reserve a few blocks of the TCAM for adding filters between recomputation of the data structure. Filter updates can then be effected immediately, either by removing the filter from the TCAM, or by adding it to the reserved blocks. Periodic rebuilding of the data structure can then clean out the reserved blocks for use later.

Unfortunately, that still requires either blocking all classification queries during update time, or the use of a TCAM large enough to hold both the old and new data structures at the same time. Those constraints are not acceptable for all applications, so we seek to develop a more dynamic update processing technique.

4.5.1 Trie-Carving Update Algorithm

We choose the Trie-Carving algorithm as a basis for supporting dynamic filter updates, due to the simplicity of the data structures used for Trie-Carving.

A naive approach could work as follows: filters are deleted simply by removing them from the TCAM, with no other updates made; filters are inserted either by placing them in a non-full block along the path for that filter in one of the tries, or in a newly allocated block if no non-full blocks are found. Such an approach is reasonable for a small number of updates; after a large number of updates, however, the filter grouping can become quite inefficient. There is nothing to ensure the TCAM blocks are used efficiently, or that the number of blocks searched on a worst-case query is kept small.

To address that problem, we would like to maintain the following invariants:

1. Filters are pushed down (moved from one block to another, towards the leaves) in the trie as far as is possible; filters more specific (in the dimension corresponding to the trie containing them) are swapped with filters that are less specific.

2. Filter storage blocks are pushed down in the trie (moved from one node to another) as much as possible, given the filters that they contain.

3. In each trie, there is at most one filter storage block containing fewer than $\lceil k/2 \rceil$ filters.

Invariant 2 aims to keep the power fraction low, by reducing the number of blocks searched along the worst-case path through each trie. Invariant 1 assists in this process by putting the most specific filters in the blocks which are farthest down. Invariant 3 is designed to keep the storage complexity low, by ensuring that the filter storage blocks are kept mostly full.

We define a set of procedures which are used to maintain the three invariants. The FilterPushdown operation, which helps maintain invariant 1, is defined as follows when performed on a node n:

```
for each filter $f$ stored in a block of the node $n$ do
  traverse the path for filter $f$ within the descendants of $n$, looking for an empty slot or a less-specific filter
  if an empty slot $s$ was found then
    move $f$ to slot $s$
  else if a less-specific filter $f'$ was found then
    swap $f$ with $f'$
  end if
end for
perform FilterPushdown on each child of $n$
```

The BlockPushdown operation, used to maintain invariant 2, is defined as follows when performed on a node n:

```
  if a single block $b$ is stored at node $n$ then
    if all filters in $b$ are compatible with the left child of $n$ then
      move $b$ to the left child of $n$
      perform a Consolidate operation on the left child of $n$, if needed
    else if all filters in $b$ are compatible with the right child of $n$ then
      move $b$ to the right child of $n$
      perform a Consolidate operation on the right child of $n$, if needed
    end if
  else if multiple blocks are stored at node $n$ then
    Rearrange the filters at $n$ so as to maximize the number of blocks pushable to the children of $n$
    move as many blocks to $n$'s children as possible
    if the rearrangement produced a block with $\lceil k/2 \rceil$ filters or fewer, and another such block already exists then
      Coalesce those two blocks
    end if
    if one more more blocks were moved to $n$'s left child, and it already had at least one block then
      Consolidate the blocks at $n$'s left child
    end if
    if one more more blocks were moved to $n$'s right child, and it already had at least one block then
      Consolidate the blocks at $n$'s right child
    end if
  end if
  perform BlockPushdown on each child of $n$
```

The Coalesce operation maintains invariant 3. When we have two blocks with $\lceil k/2 \rceil$ or fewer filters each, we merge the two together. The Coalesce operation is defined as follows when performed on blocks b_1 and b_2:

```
move all filters from $b_1$ to $b_2$
move $b_2$ to the nearest common ancestor of $b_1$ and $b_2$
delete $b_1$
perform FilterPushdown on the node now containing $b_2$
perform BlockPushdown on the same node
```

The Consolidate operation mentioned in the above procedures compacts the i filters stored at a node into $\lceil i/k \rceil$ blocks, in the event that the node should happen to have more than $\lceil i/k \rceil$ blocks.

Now that we have defined the basic operations for maintaining the invariants, we can describe how to insert and delete filters.

To insert a filter f into a trie, we perform the following actions:

```
traverse the trie along the path corresponding to filter $f$
let $b$ be the last non-full block found along that path
if $b \neq$ null then
  insert $f$ into $b$
  perform FilterPushdown on the entire trie (*)
else
  create a new block $b_1$ at the end of the path traversed
  insert $f$ into $b_1$
  if another block $b_2$ exists having $\lceil k/2 \rceil$ filters or fewer then
    Coalesce $b_1$ and $b_2$
  end if
end if
```

In practice, performing the FilterPushdown operation on the entire trie is not a trivial operation. Good results are still obtained with lower computational cost if we only perform the FilterPushdown every k times that section of the Insert routine is reached, where k is the filter storage block size. Thus, for the rest of this section, we use that approach.

To minimize the number of TCAM writes required to perform an insertion, we assume the use of a *weighted TCAM*. In a weighted TCAM, each entry has an associated weight which indicates its priority. Thus we can insert filters into any empty slot, rather than needing to move several filters (up to the number of disinct priority levels needed to disambiguate all filters in the database) to keep the block's contents in priority order.

To remove a filter from the trie, we perform these actions:

```
traverse the trie to find the block $b$ which contains filter $f$
remove $f$ from $b$
if $b$ is now empty then
  delete $b$
else
  let $n$ be the node containing $b$
  let $i$ be the number of filters stored at $n$
  let $j$ be the number of blocks stored at $n$
  if $j > \lceil i/k \rceil$ then
    Consolidate the blocks at $n$
  else if $i \leq \lceil k/2 \rceil$ then
    if another block $b'$ exists with at most $\lceil k/2 \rceil$ filters then
      Coalesce blocks $b$ and $b'$
    else
      Perform BlockPushdown at node $n$
    end if
  else
    Perform BlockPushdown at node $n$
  end if
end if
```

4.5.2 Results

Since the characteristics of typical filter database updates are not known, we instead use the following process to evaluate the effectiveness of our filter update procedure: Given two filter databases (one called the *initial filter set* and the other called the *final filter set*), each containing N filters, perform the following steps:

1. Run the Trie-Carving algorithm on the initial filter set
2. Repeat the following steps N times:
   - Remove one randomly selected filter that was in the initial filter set

- Insert one randomly selected filter from the final filter set.

3. Compute the power fraction and storage complexity of the result.

This tests the update procedure's ability to adapt the grouping structure to a completely different distribution of filters.

Before presenting the results, let us note a few things. The power fraction and storage complexity of the initial grouping is simply the result of applying the Trie-Carving algorithm; since these results are already presented in Section 4.2.3, we do not present them here. The power fraction and storage complexity of the data structures in between the initial and final states are also not presented here, since the data structures at that point (containing subsets of two completely different filter databases) do not reflect any meaningful real-world situation.

Also, it is important to note that the power fraction and storage complexity results presented in this section are with respect to the partitioning only; these results do not include the effects of adding range check support.

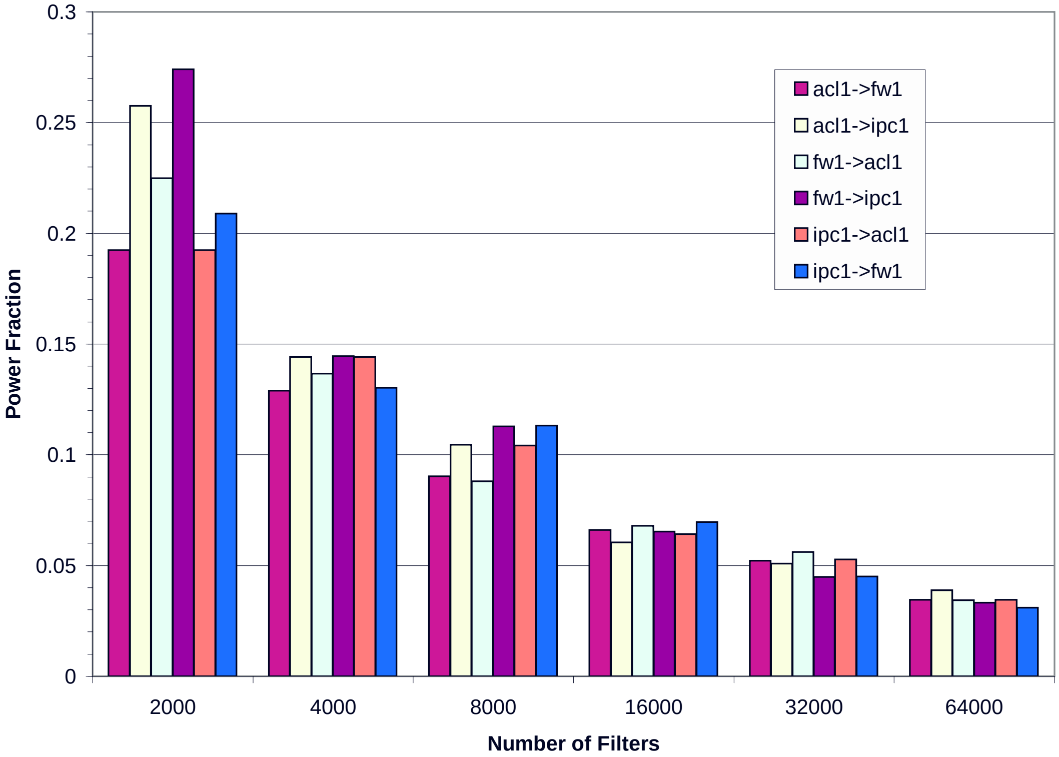


Figure 4.36: Power fraction after switching to the final filter set

Power fraction results for the data structures holding the final filter set are shown in Figure 4.36. These results are not quite as good as applying the static Trie-Carving algorithm directly on the final filter set, but the results still represent a substantial improvement in power usage relative to a standard TCAM solution.

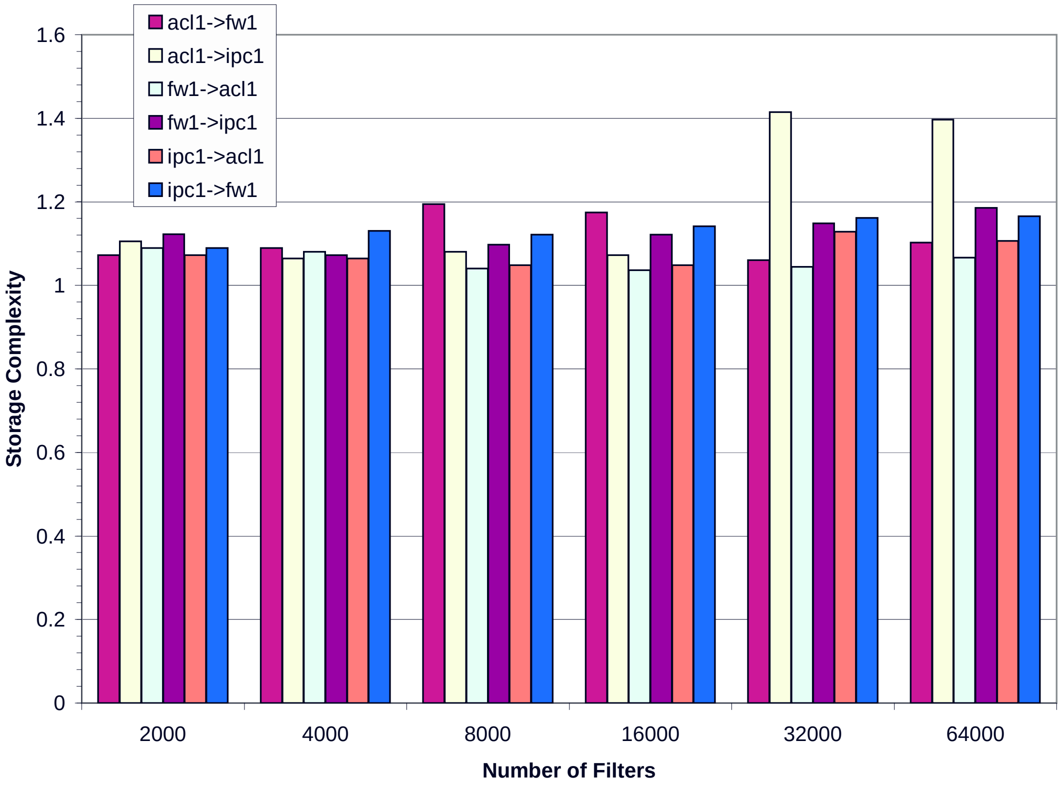


Figure 4.37: Storage complexity after switching to the final filter set

The TCAM storage complexity for the data structures holding the final filter set are shown in Figure 4.37. These results are quite good; in fact they are better than the results obtained by applying the static Trie-Carving algorithm directly to the final filter set.

The filter update procedure is not a completely trivial operation, so in Figure 4.38 we present for reference the average rate of update processing on a 700MHz Pentium III. Not all packet classification applications will have that much processing power available, and in those cases the sustainable update rates would be lower. But, for high performance routers that perform classification using the same rulesets on multiple ports, adding a single 35 watt CPU to the entire system makes much more sense than using a 20 or 30 watt TCAM on each port.

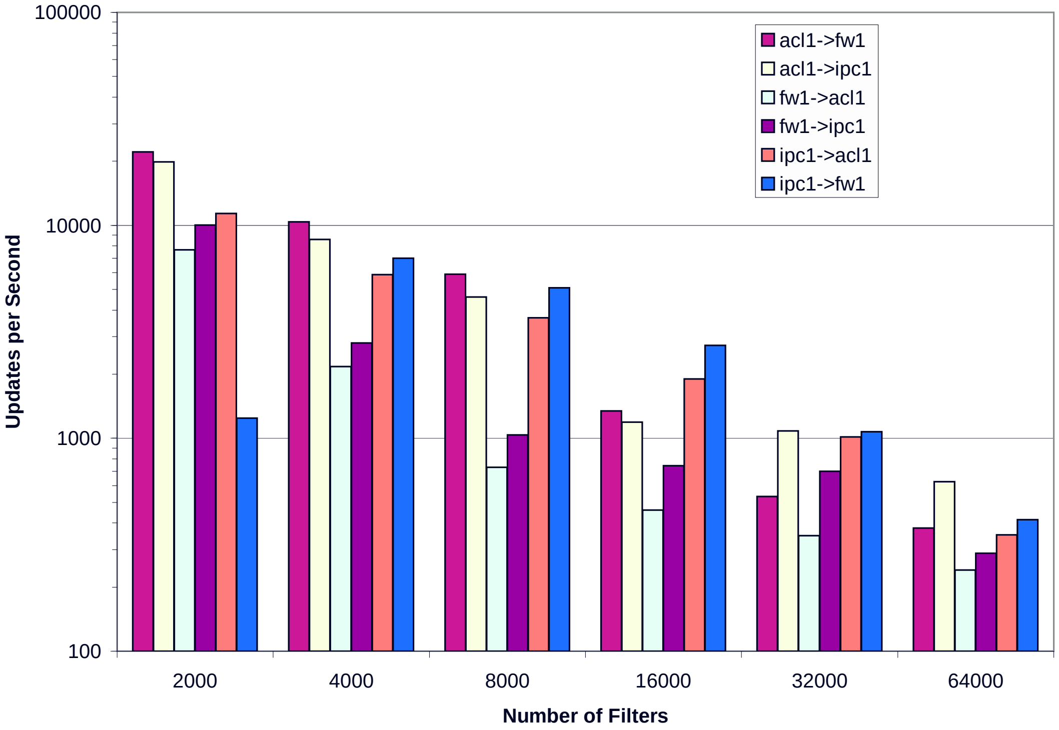


Figure 4.38: Average rate of update processing in software on 700 MHz Pentium III

One thing to note from Figure 4.38 is that, as the number of filters in the trie increases, the rate at which updates can be processed decreases. We can support high update rates by limiting the number of filters in each trie, since there is no reason why one cannot have multiple Source Address tries and/or multiple Destination Address tries. The only disadvantage is that the power fraction tends to be higher when the tries have fewer filters; thus there is a tradeoff between update performance and power efficiency.

Figure 4.39 shows the average number of TCAM writes needed per update, assuming the TCAM can be locked in such a fashion that two TCAM writes can be performed as an atomic operation; without such a capability, extra TCAM writes would be necessary in order to keep the TCAM contents consistent while lookups are being performed. On average, only 3 to 8 TCAM writes are needed per update, which is not bad.

In the worst case, however, an individual update may require more TCAM writes; such a case can occur if the data structure reaches a point where a single update can trigger pushdown operations on several blocks. Figure 4.40 shows the

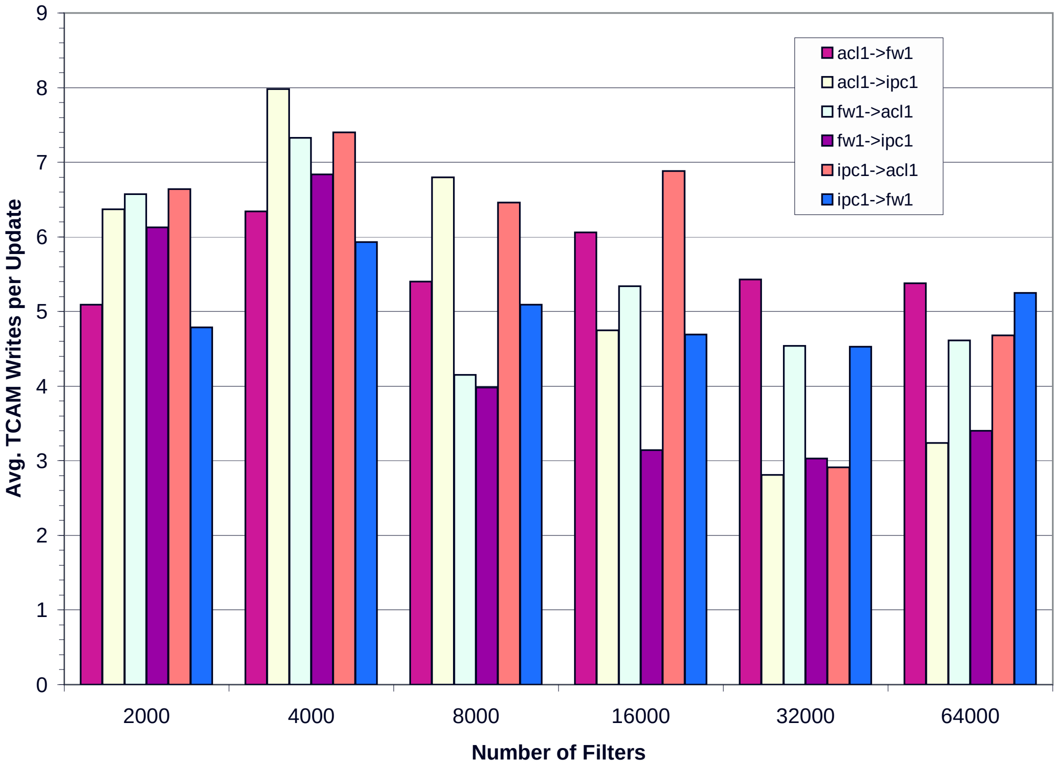


Figure 4.39: Average number of TCAM writes needed per update

maximum number of TCAM writes needed for an update. Although these numbers are much higher than the average number of writes needed per update, they are still not bad; a TCAM supporting millions of updates per second would still only require a few milliseconds to process the writes for one of these pathological updates.

Figure 4.41 shows a time history of performance for the case of starting with the 2,000 filter database derived from acl1 and ending with the 2,000 filter database derived from fw1. Neither the power fraction nor the storage complexity suffer any dramatic increases, despite the fact that the database contents during the middle of the experiment (i.e. half of one database and half of the other) does not reflect a typical real-world scenario. And, as noted before, the number of TCAM writes to effect an update is typically low, with the occasional spike.

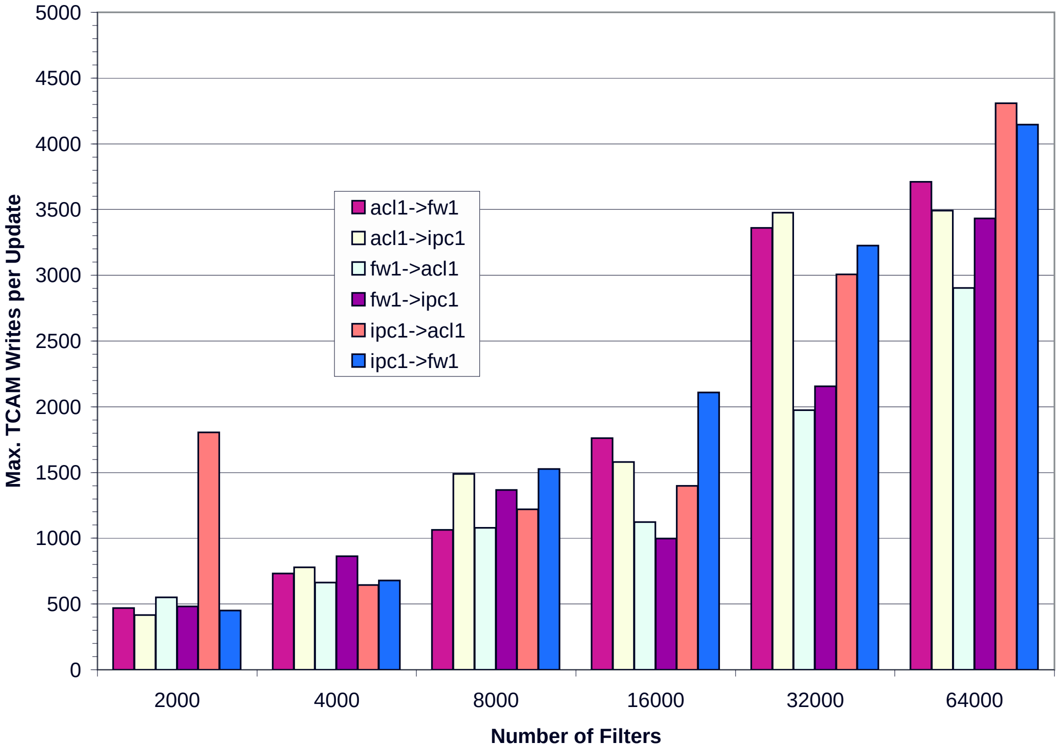


Figure 4.40: Maximum number of TCAM writes needed for an update

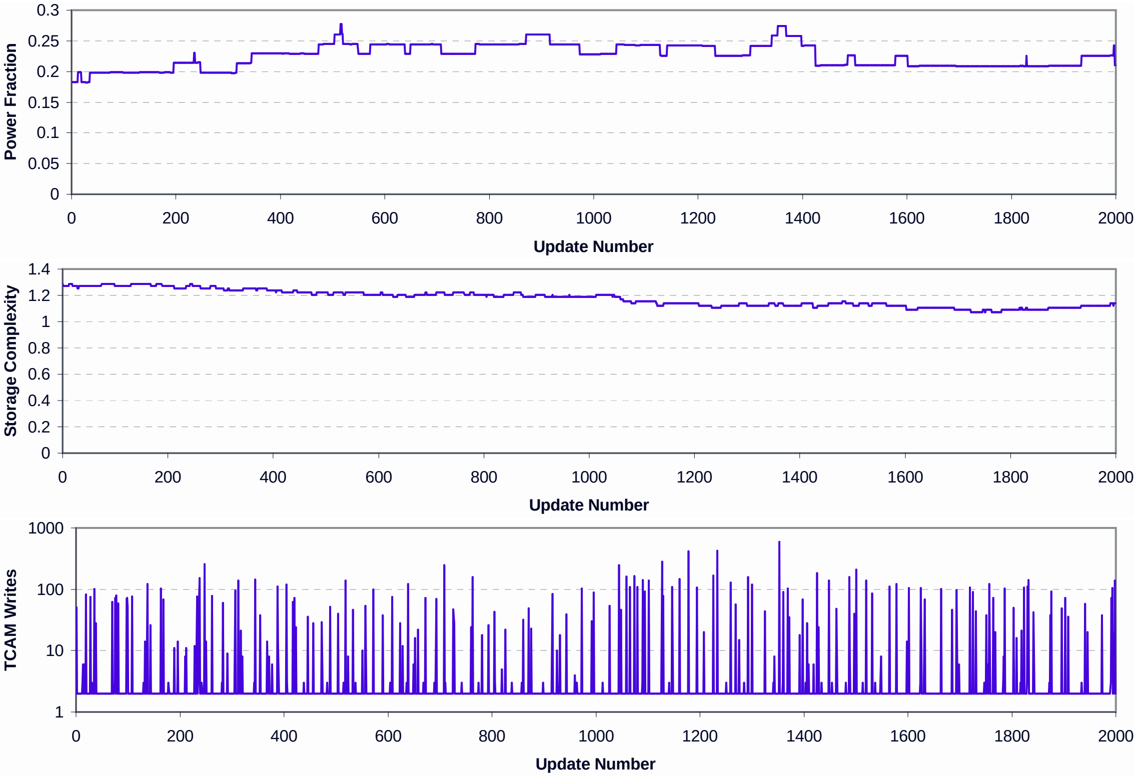


Figure 4.41: Time history of performance for updating from 2,000 filter acl1 to fw1

4.6 Multilevel Indexing

Power requirements can be further reduced by using a multi-level indexing structure; in a pipelined implementation this can be done with essentially no reduction in classifier throughput. In this section we explore the simplest case of multi-level indexing, namely a two-level index. An example of an Extended TCAM with a two-level index structure is shown in Figure 4.42.

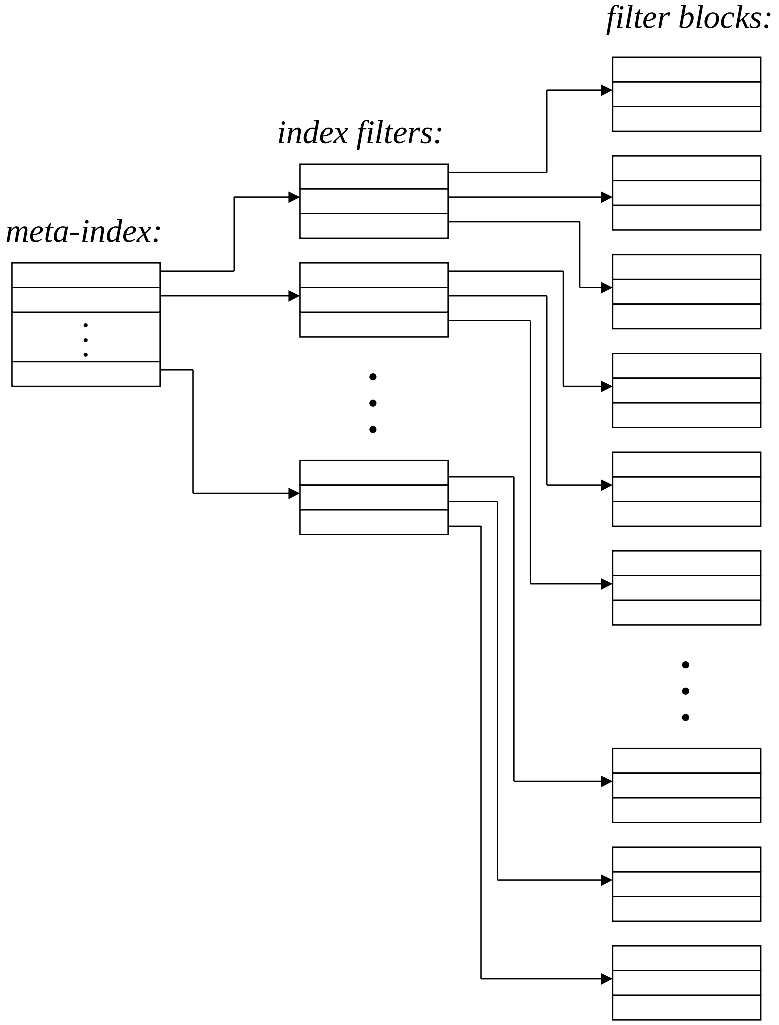


Figure 4.42: Example of a two-level indexing structure

The benefit of a multi-level index comes from the fact that, thanks to the meta-index, we only need to search a subset of the index for any given query. This improvement is particularly beneficial when using smaller block sizes, because a smaller block size means more blocks are required and thus more index entries are required.

A fairly straightforward way to group filters for an Extended TCAM with multiple index levels is to first run a grouping algorithm on the filters to produce the index entries, and then run the algorithm using the index filters as inputs to produce the meta-index filters.

To evaluate this approach we can use the concepts of power fraction and storage complexity discussed in Section 4.2.2. Note that the total number of TCAM bits activated for search, which is used to compute power fraction, now includes meta-index entries, index entries, and filter block entries. Power fraction can now be computed using the expression $(b_{ix} + (s_{ix} + s_{fs})k)/N$, where b_{ix} is the number of blocks in the index level (and thus the number of entries in the meta-index level), s_{ix} is the maximum number of index blocks searched on any given query, and s_{fs} is the maximum number of filter storage blocks searched on any query. Storage complexity can be computed via the expression $(b_{ix} + (b_{ix} + b_{fs})k)/N$, where b_{fs} is the number of blocks in the filter storage level.

4.6.1 Using Region-Splitting Algorithm

As before, we begin our analysis with a study of the effects of choosing different block sizes. The results of this study are shown in Figure 4.43. Intuitively we expect that the best choice for block size is proportional to $\sqrt[3]{N}$, where N is the number of filters. The data appear to support that, and lead us to select a block size of 2^i where i is the result of rounding $\log_2(\sqrt[3]{N})$ to the nearest integer.

For some of the smaller filter databases, this would mean using a block size which may not be practical to implement in the real world (due to the overhead involved in partitioning the TCAM). For that reason we exclude the smaller filter databases from this experiment, and include some 256,000 filter databases instead to ensure we still have a reasonable number of data points. We use a block size of 32 for the 16,000 filter databases, 32,000 filter databases and 64,000 filter databases; we use a block size of 64 for the 128,000 filter databases and 256,000 filter databases.

The power fraction obtained from region splitting with a two-level index is shown in Figure 4.44. In all cases, the power fraction is below 4.5%, which is a very positive result. In fact, for the largest databases in this study the power fraction is below 1%, which is extremely good.

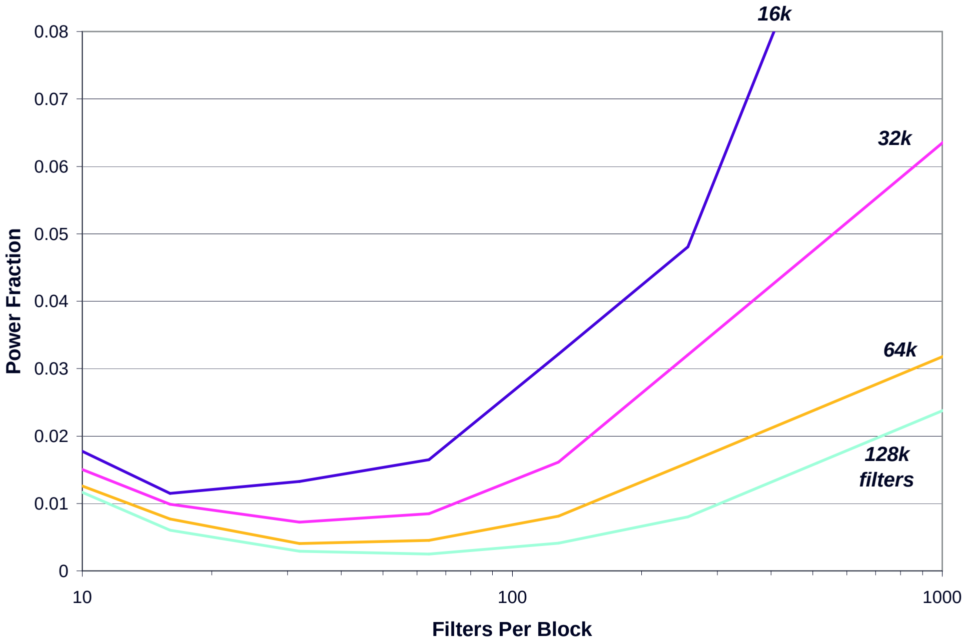


Figure 4.43: Effects of varying storage block size, for region-splitting with a two-level index

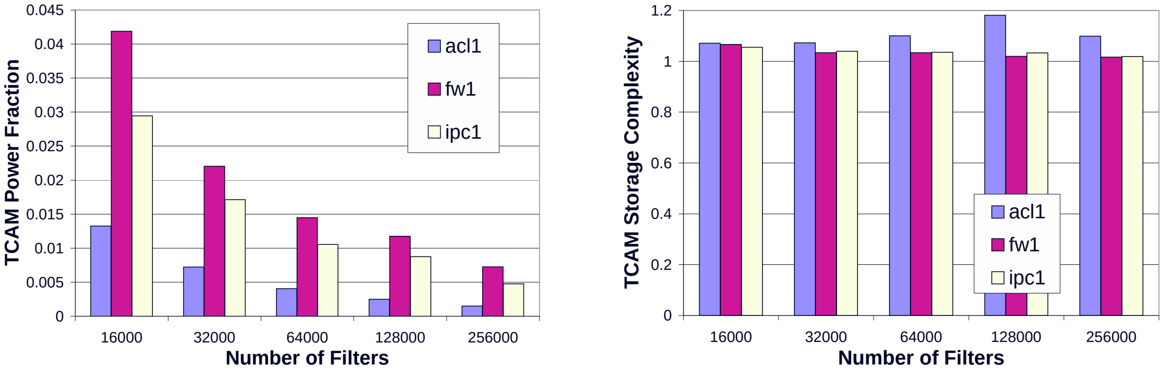


Figure 4.44: Power fraction for region-splitting with a two-level index

Figure 4.45: Storage complexity for region-splitting with a two-level index

The storage complexity for region-splitting with a two-level index is shown in Figure 4.45. In all cases the additional storage cost of using region-splitting partitioning is below 20% which is very acceptable considering the magnitude of power savings provided.

Figure 4.46 shows power fraction results which include the effects of range match circuitry, and Figure 4.47 shows storage complexity results which include the effects of range match circuitry. As noted before, the precise effect of using hardware

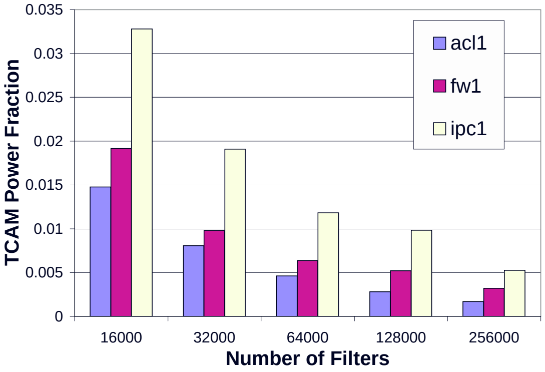


Figure 4.46: Power fraction for region-splitting with a two-level index, including effects of range-check circuit

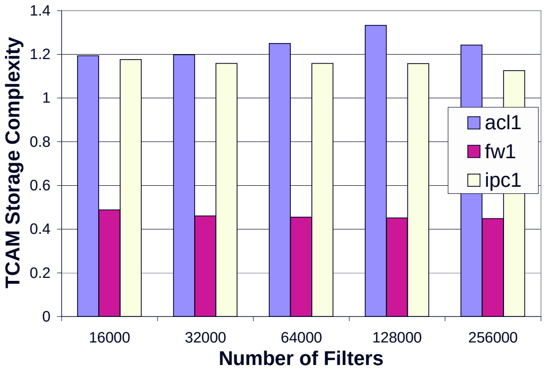


Figure 4.47: Storage complexity for region-splitting with a two-level index, including effects of range-check circuit

range-check support depends strongly on the nature of the database used. The firewall databases, for example, use port ranges often and thus benefit greatly from range-check support; with other databases the difference is smaller.

4.6.2 Using Trie-Carving Algorithm

We begin the analysis of the two-level trie-carving approach by determining how to choose the storage block sizes for this algorithm. Power fraction results with various storage block sizes are shown in Figure 4.48. From this we can see that the storage block selections proposed for the two-level region splitting case will work well here also. Thus for the remaining analysis of the two-level trie-carving approach we use the same block sizes as in Section 4.6.1.

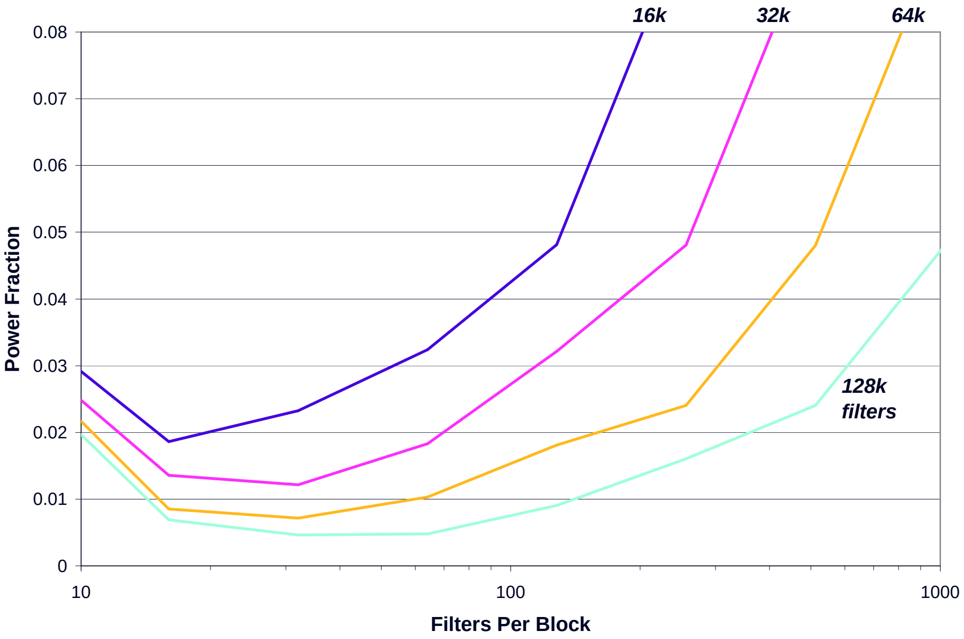


Figure 4.48: Effects of varying storage block size, for trie-carving with a two-level index

The power fraction obtained from trie-carving with a two-level index is shown in Figure 4.49. In all cases the power fraction is below 5% and for the larger filter databases it is below 1.5%, which is an extremely positive result. In terms of power fraction, trie-carving with a two-level index outperforms both one-level approaches and is competitive with the two-level region-splitting results.

The storage complexity for trie-carving with a two-level index is shown in Figure 4.50. This is where one pays a price for the simplicity and speed of the trie-carving algorithm. Still, in all cases the additional storage cost of using trie-carving partitioning is below 60%, which may be a very worthwhile tradeoff for many applications.

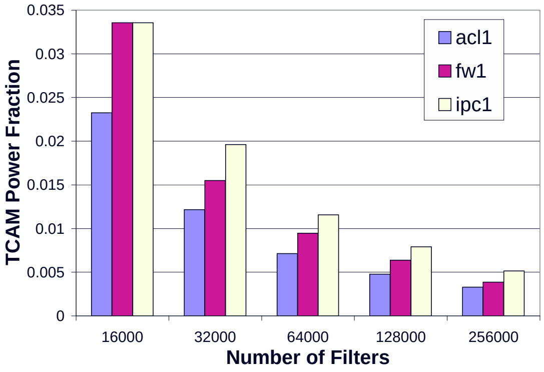


Figure 4.49: Power fraction for trie-carving with a two-level index

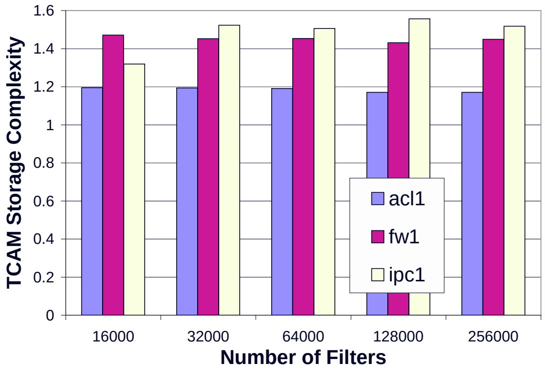


Figure 4.50: Storage complexity for trie-carving with a two-level index

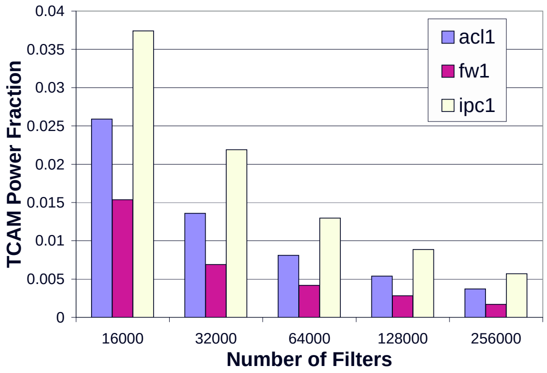


Figure 4.51: Power fraction for trie-carving with a two-level index, including effects of range-check circuit

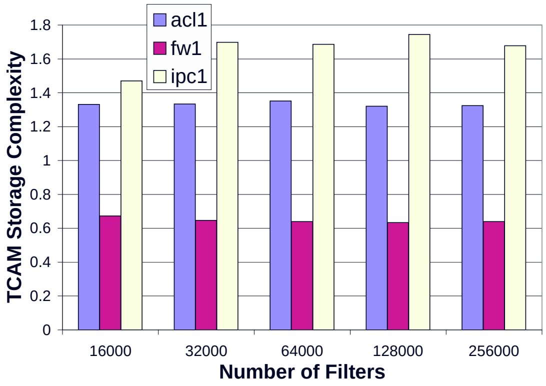


Figure 4.52: Storage complexity for trie-carving with a two-level index, including effects of range-check circuit

Figure 4.51 shows power fraction results which include the effects of range match circuitry, and figure 4.52 shows storage complexity results which include the effects of range match circuitry. As noted before, the precise effect of using hardware range-check support depends strongly on the nature of the database used. The firewall databases, for example, use port ranges often and thus benefit greatly from range-check support; the rules derived from the IP Chains database, on the other hand, use too few range specifications to make the range check circuitry worthwhile.

4.7 Summary

High-performance packet classification is crucial to the deployment of many advanced network services. Although the most popular approach in use is the TCAM, TCAM's usefulness is limited by their high power consumption and inefficient representation of range match fields.

Extended TCAMs provide the performance needed, while scaling up to hundreds of thousands of filters. Extended TCAMs allow classification at the same speed as TCAMs, usually with less than 5% of the power usage of a standard TCAM; they also avoid the storage efficiency problem that TCAMs have with range-match fields such as transport layer port numbers. These properties are achieved by using a partitioned TCAM and having hardware support for range matching.

Efficient use of a partitioned TCAM requires grouping filters into blocks such that only a small number of blocks need to be searched for any given query. One method to accomplish this is the region-splitting algorithm which has excellent power dissipation results (less than 5% of a standard TCAM for 32,000 filters or more) and very reasonable storage complexity (less than 10% more than standard TCAM in most cases). Another technique is the trie-carving algorithm, which is faster than region-splitting and produces competitive power fraction results at a small cost in terms of storage efficiency.

The range matching support can be implemented in CMOS using 44 transistors per bit; for a typical IPv4 application this means a 46% increase in transistor count per word, instead of a 2-6x increase in word count needed. This improves both power requirements and storage efficiency.

Dynamic filter updates can be supported using the Trie-Carving data structures. There is a tradeoff between update complexity and the power fraction obtained. For the databases studied in our experiments, it is possible to obtain a power fraction of 10% and support hundreds or thousands of updates per second by limiting trie sizes to 8,000 filters each.

For filter partitioning using either the Region-Splitting algorithm or the Trie-Carving algorithm, a multilevel indexing technique allows power requirements of large filter sets to be further reduced. for large databases this can result in power fractions below 1%.

Chapter 5

Partitioned Encoded Search of TCAMs

The ideas from Extended TCAMs (Chapter 4) can be combined with ideas from P^2C [53] to perform high speed packet classification even more efficiently. In this chapter we introduce a packet classification method called Partitioned Encoded Search of TCAMs (PEST). This technique is a novel combination of ideas from Parallel Packet Classification (P^2C) [53] and Extended TCAMS [42] with a modified partitioning algorithm. We also describe an update mechanism which can be used to eliminate fragmentation of the primitive range identifiers in P^2C-style encoding.

This chapter is organized as follows: Section 5.1 describes the PEST architecture, including a description of the encoding scheme and the use of partitioned TCAM. Section 5.2 describes the aforementioned TCAM update technique. An evaluation of the PEST system with experimental results is presented in Section 5.3. To help interpret the results, some lower bounds on the encoding width are derived in Section 5.4.

5.1 PEST Architecture

In this section we describe a classification scheme called Partitioned Encoded Search of TCAMs (PEST). This packet classification technique seeks to leverage the strengths of the P^2C approach and Extended TCAMs, by combining them and adapting them when appropriate. The result is a high speed packet classification method with lower power requirements than P^2C or Extended TCAMs alone. In the following subsections we describe the techniques used and walk through a simple example.

For understanding how the pieces of PEST fit together into the whole, the lookup architecture is shown in Figure 5.1. A packet classification query is first converted to an encoded form (using the P^2C encoding process); this encoded form is used to consult an index table and perform a lookup in the encoded filter storage blocks. As in Extended TCAMs, a final priority resolution stage is needed to handle cases where matches are found in more than one block.

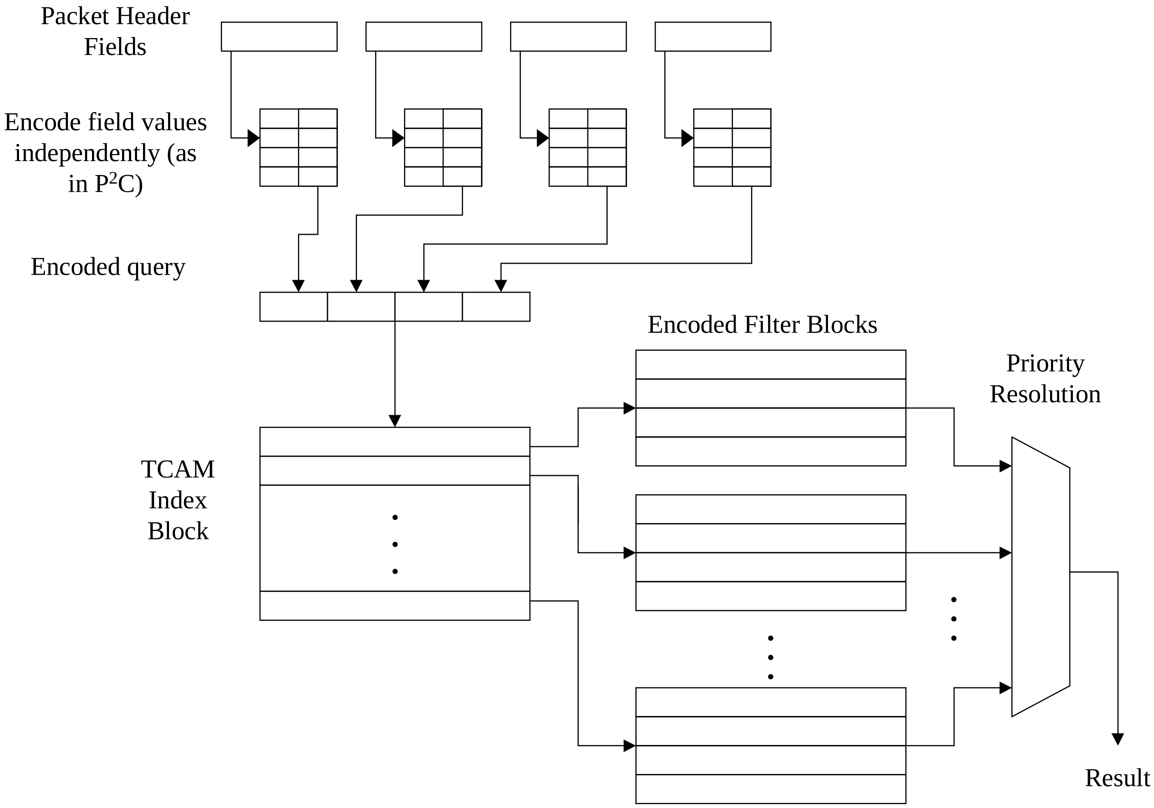


Figure 5.1: PEST lookup architecture.

So, the packet header values for a query are first converted to an encoded representation, and then this encoded form is used to query a partitioned TCAM. The encoding system is described in Section 5.1.1 in detail; the use of a partitioned TCAM (and modifications to the Extended TCAMs partitioning algorithm) are described in section 5.1.2.

5.1.1 Encoding of Field Values

Field values are encoded as in the P^2C [53] approach. To reduce the complexity of creating and updating data structures, we choose to use the first encoding style exclusively.

The use of encoding significantly reduces the width required of the TCAM; in fact, it is typically more effective than the dedicated range check hardware with respect to reducing power consumption and number of transistors required (dedicated range check hardware, however, still has merit if one does not wish to use this means of encoding.) We expect encoding to be even more useful for IPv6 [13] applications, which could otherwise require TCAM search keys exceeding 300 bits in width.

Another consequence of the encoding is that we must convert query values into encoded search keys in order to perform classification. The modified BART scheme described in [53] appears to be an excellent choice for this, since it performs the encoding quickly and does so with very modest SRAM requirements. Other techniques, such as a modified version of the approach in [14], are also possible.

Filter	*Source Address*	*Destination Address*
f_1	0100*	0100*
f_2	0100*	0111*
f_3	0111*	0100*
f_4	0100*	10*
f_5	0111*	*
f_6	*	0111*
f_7	0011*	01000101
f_8	1*	11010001-11011010
f_9	*	0100*
f_{10}	*	11010001-11011010
f_{11}	0100*	*
f_{12}	1*	10*

Table 5.1: Example set of filters

Consider as an example the set of filters shown in Table 5.1. Recall from Chapter 2 that the P^2C encodings are created by taking the set of primitive ranges used in the filters, grouping those primitive ranges into layers, and assigning labels to the ranges. The primitive ranges used in the source address field in our example are the prefixes 0011*, 0100*, 0111* and 1*, and the wildcard value of *. Since none of these primitive ranges overlap, they can all coexist in the same layer. Each range in the layer is assigned a label as indicated in Table 5.2. This layer requires three bits in the encoded representation.

Source Address	*Encoded Representation*
*	xxx
0100*	001
0111*	010
1*	011
0011*	100

Table 5.2: Encoding for source addresses in example rules

The primitive ranges used in the destination address field are 0100*, 01000101, 0111*, 10*, and 11010001-11011010, in addition to the * wildcard. Since the primitive ranges 0100* and 01000101 overlap, they must be stored in different layers; suppose we use the primitive range hierarchy shown in Figure 5.2. An assignment of Layer 1 requires three bits in the encoded representation, and layer 2 requires one bit; encoded representations are shown in Table 5.3.

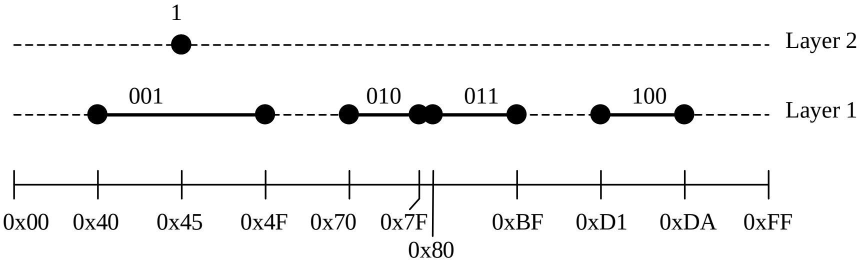


Figure 5.2: Destination address primitive range hierarchy from example

Destination Address	*Encoded Representation*
*	xxx x
0100*	001 x
0111*	010 x
10*	011 x
11010001-11011010	100 x
01000101	xxx 1

Table 5.3: Encoding for destination addresses in example rules

The encoded representation of a rule or query in this example requires seven bits total. In this example, we choose to concatenate the layers together in order (for the sake of readability). In practice, the bits used for a layer need not be contiguous in the encoded representation; this flexibility can simplify the update process, in situations where adding a range to a layer causes it to require another bit for its identifiers.

Filters are encoded using the identifiers for the primitive ranges in the filter in their respective bit positions, and wildcard bits in all other bit positions. A fully wildcarded field is encoded as wildcard bits in all layers for that field. For example, filter f_1 uses source address prefix 0100* (identifier 001 in the source address layer) and destination address prefix 0100* (identifier 010 in destination address layer 1), so it is encoded as 001 001 x (spaces included for the sake of readability.) In this manner, we can produce the encoded representation of filters as shown in Table 5.4.

Filter	*Encoded Representation*
f_1	001 001 x
f_2	001 010 x
f_3	010 001 x
f_4	001 011 x
f_5	010 xxx x
f_6	xxx 010 x
f_7	100 xxx 1
f_8	011 100 x
f_9	xxx 001 x
f_{10}	xxx 100 x
f_{11}	001 xxx x
f_{12}	011 011 x

Table 5.4: Encoded representation of example filters

Search queries are encoded using the identifiers for the query value in each layer; if the query value does not fall within any primitive ranges in a particular layer, then the identifier 0 is used for that layer. For example, the (source address, destination address) query (01010101, 01000101) is encoded as 000 001 1, and the query (10101010, 01110101) is encoded as 011 010 0. Efficient encoding of search keys can be done via a modified version of the BART [52] scheme.

5.1.2 Partitioning of TCAM

Power consumption is further reduced by the use of a partitioned TCAM. As described in Chapter 4, a partitioned TCAM consists of a TCAM divided into blocks of k entries each, and a set of index filters (one per block). When a query is processed, it is first compared against the index filters; in the next step, the blocks whose index filters matched are searched. This reduces power requirements by reducing the number of TCAM entries searched.

PEST uses a similar partitioning algorithm to the Extended TCAMs approach, but there are no range match fields involved at the partitioning step. That is, the encoding step transforms range match fields into bitmask fields. The partitioning algorithm runs on the *encoded* representations of the filters, and therefore only deals with bitmask fields.

The PEST partitioning algorithm runs in a series of phases. In each phase, a partitioning is made of the entire classification space. Each phase consists of a series of steps; in each step, one region in the space is selected and split into two parts.

At the end of each phase, the partitioning is used to assign encoded filters to blocks in the partitioned TCAM. There is one block corresponding to each region. Up to k encoded filters contained within a region are stored in the corresponding block, and removed from consideration in further phases; if the region contained more than k encoded filters, then we select the k encoded filters that have the most wildcard bits. The index entry for a block is set to correspond to the range of encoded classification space covered by that block's region; that way, any query that would match one of those encoded filters will activate the block during a search.

To help build intuition, consider the set of filters in Table 5.5. A two-dimensional representation of this set of filters is shown in Figure 5.3. In this simple example, let us pretend we would like to partition these filters into blocks with no more than 2 filters per block (i.e. $k = 2$).

	Source Address	*Destination Address*
a	110x	xxxx
b	0110	1110
c	x0xx	110x
d	1001	x0xx
e	0xxx	1000
f	xx1x	0101
g	00xx	001x
h	0101	00xx

Table 5.5: Filters for partitioning example

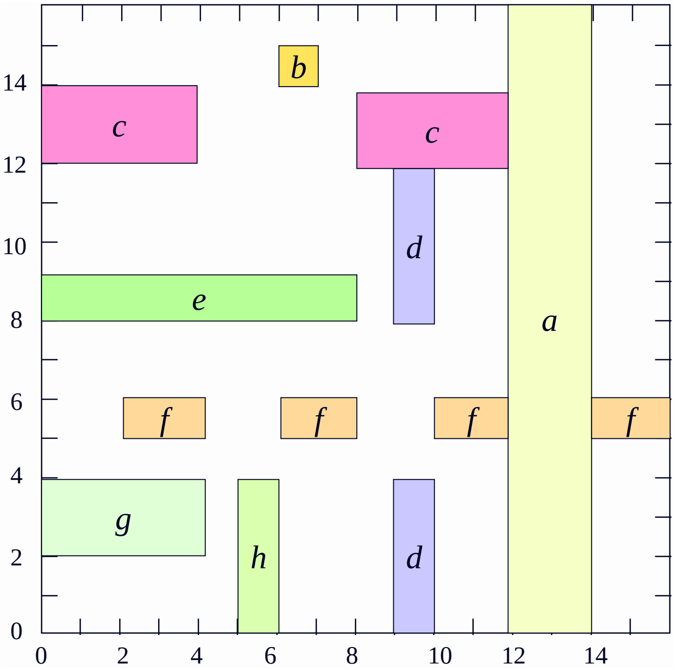


Figure 5.3: Filters for partitioning example

The algorithm starts with one region, spanning the entire classification space. For the sake of example, suppose it decides to split the region as shown in Figure 5.4. Now the algorithm has two regions. The right most of the two regions completely contains filters a and d, so, perhaps we don't want to split that region any more.

In the next step, suppose the algorithm splits the leftmost of the two regions as shown in Figure 5.5; there are now three regions. The upper left region completely contains filters b and e, and the lower left region contains filters g and h; this is probably a good place to stop the first phase, so, let's assume the algorithm does that.

At the end of the first phase, then, a TCAM block is allocated corresponding to the upper left region; its index entry is set to 0xxx 1xxx (i.e. the region itself), and filters b and e are placed in that block. Another TCAM block is allocated for the lower left region; its index entry is set to 0xxx 0xxx and filters g and h are placed in that block. One more TCAM block is allocated, this time for the rightmost region; its index entry is set to 1xxx xxxx, and filters a and d are placed in that block.

Filters a, b, d, e, g and h have been stored in the TCAM, and are thus removed from consideration in future phases. Thus we are left with the filters c and f, shown in Figure 5.6.

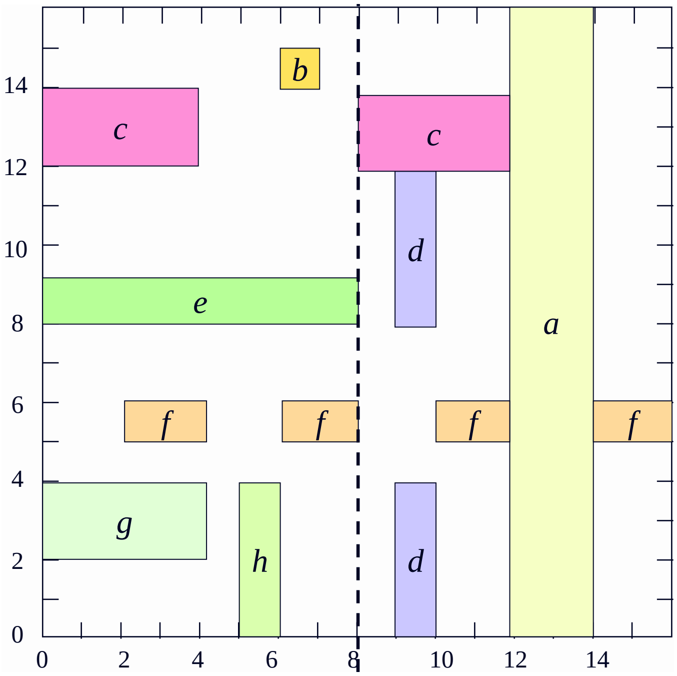


Figure 5.4: Partitioning after one step

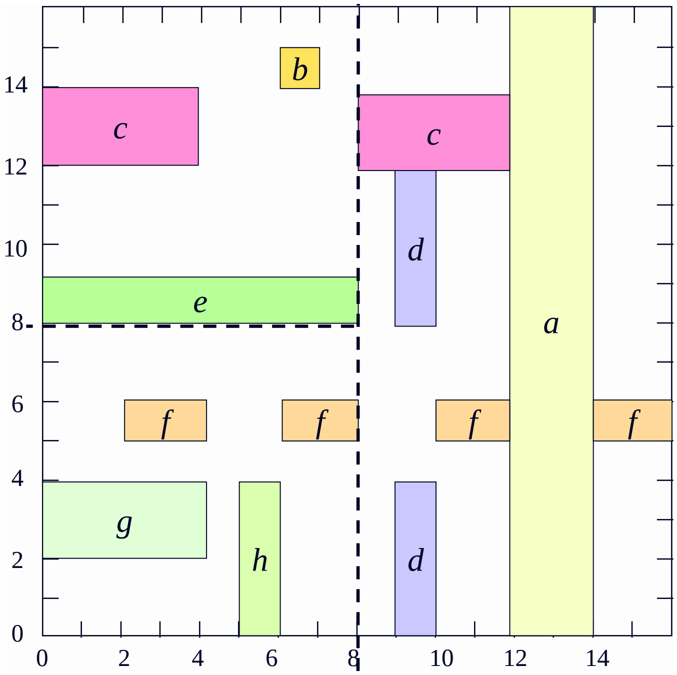


Figure 5.5: Partitioning after two steps

We begin the second phase of the algorithm with one region, spanning the entire classification space. This region completely contains the two remaining filters

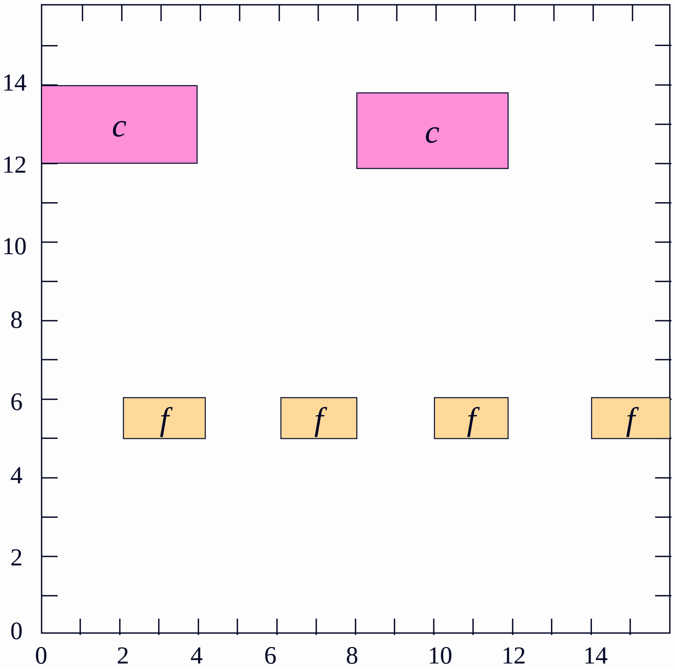


Figure 5.6: Partitioning during second phase

(c and f), so we can end the phase here. A TCAM block is allocated for this region; its index entry is set to xxxx xxxx, and filters c and f are stored in that block.

All filters have been stored in TCAM blocks at this point, so, the algorithm terminates; the contents of the Extended TCAM are shown in Figure 5.7.

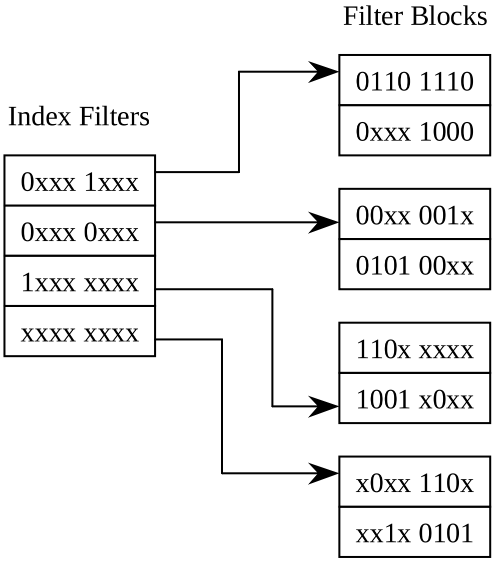


Figure 5.7: Extended TCAM contents for simplified partitioning example

As noted before, in each step of the algorithm, a region r is selected and cut into two sub-regions r_1 and r_2. Each region can be represented by the bitmask specification that exactly covers that region; using this representation, a region is cut by selecting one of the "don't care" bits in its bitmask representation. In one sub-region, we set that bit to 0, and in the other, we set it to 1. All other bits are inherited from the original region r.

A step of the algorithm, then, can be expressed as follows. Let F_i be the set of filters remaining to be processed at the start of phase i, and let S_i be the set of sub-regions created by the algorithm during phase i. At the start of phase i, S_i contains one region spanning the entire encoded classification space. Let $\sigma(r)$ denote the set of filters in F_i that lie entirely within r. In all but the last phase, we repeat the following step, until no region r in S_i can be split into two sub-regions containing least k filters from F_i in each:

- Let r be a region, selected from S_i, with $|\sigma(r)| > k$.
- Consider cuts that divide r into two sub-regions r_1 and r_2 that satisfy $|\sigma(r_1)| \geq k$ and $|\sigma(r_2)| \geq k$
- Among all such candidate cuts, select one that maximizes $|\sigma(r_1) \cup \sigma(r_2)|$
- Remove r from S_i, and replace it with r_1 and r_2.

If no candidate cuts are found for a region, it is not split, and is not considered again in that phase. The phase terminates when no more candidate cuts can be found. At the end of the phase, a storage block is allocated for each region, and up to k filters from that region are placed in the block.

During the final phase of the algorithm, the encoded filters are allowed to span more than one region; if a filter spans more than one region, it must be stored in the blocks corresponding to each of those regions. This special handling in the last phase is not absolutely necessary, but can often reduce the number of phases needed (if there is a small set of hard-to-fit encoded filters left towards the end) at a small cost of storage efficiency. Since a query will result in searching one block for each phase, the best power efficiency is achieved when the number of phases is minimized.

In the final phase, we also allow a region to be split even if the sub-regions do not contain at least k filters each. The basic step for the last phase can be expressed as follows, letting $\chi(r)$ denote the set of filters in F_i that intersect with r but are not completely contained within r:

- Let r be a region, selected from S_i, with $|\sigma(r) \cup \chi(r)| > k$.
- Consider the cuts that divide r into two sub-regions r_1 and r_2.
- Among all such candidate cuts, select one that maximizes $|\sigma(r_1) \cup \sigma(r_2)|$
- Remove r from S_i, and replace it with r_1 and r_2.

The last phase terminates when, for every region r in S_i, $|\sigma(r) \cup \chi(r)| \leq k$, or when a cut results in no decrease in $|\sigma(r) \cup \chi(r)|$. In the latter case, the final phase fails to include all filters; the algorithm must be re-run, specifying more phases. The algorithm can be re-run, increasing the number of phases each time until it complete successfully; or, most of the redundant computation can be avoided by rolling back to the start of the last phase, if it fails (or if storage efficiency in the last phase became undesirably low). Alternately, the algorithm can be run without the special handling of the last phase; this simplifies implementation, but the results in some cases are not quite as good.

Within each block, filters are stored in order of priority; thus, the TCAM's priority resolution will find the best match within each block. A further priority resolution stage is needed, of course, for cases where matching filters are found in multiple blocks.

As a more concrete example of the partitioning algorithm in action, let us continue the example from Section 5.1.1, and let us consider storing the encoded filters in a partitioned TCAM with block size of 3. The encoding process tends to result in a lot of noncontiguous bitmasks; this does not present a problem for the partitioning algorithm, but it makes visual representation of the partitionings too unwieldy to provide any intuition. We recommend that readers grit their teeth and refer to the bitmask representations in Table 5.4 while walking through the following example.

We begin the first phase with one region spanning the entire encoded classification space. We consider splitting this region on each bit in turn. If we split on the first bit, we get sub-regions specified by 0xx xxx x (containing encoded filters f_1, f_2, f_3, f_4, f_5, f_8, f_{11}, f_{12}) and 1xx xxx x (containing filter f_7); the latter sub-region does not contain enough encoded filters to fill a block, so this bit is not a candidate for splitting. If, instead, we split on the second bit, we get sub-regions specified by x0x xxx x (containing encoded filters f_1, f_2, f_4, f_7, f_{11}) and x1x xxx x (containing encoded filters f_3, f_5, f_8, f_{12}); each sub-region has enough encoded filters to fill a block, so this bit is a candidate for splitting. The third, fifth, and sixth bits are also also turn out to be candidates. The algorithm selects the one with the most filters contained in the sub-regions; since there is a four way tie in this case, suppose we break the tie by selecting the first candidate.

Thus, in the first step, we have split the region xxx xxx x into regions x0x xxx x and x1x xxx x. We now consider splitting region x0x xxx x into subregions. Splitting on the first bit would produce subregions 00x xxx x and 10x xxx x, which contain 3 encoded filters and 1 encoded filter respectively, i.e. it is not a candidate for splitting. In fact, no split will meet the requirement that both sub-regions contain at least k encoded filters, so we do not split region x0x xxx x.

Since region x1x xxx x also does not get split (for the same reason), we end the first phase with regions x0x xxx x and x1x xxx x. A block of TCAM storage is allocated for region x0x xxx x; we store encoded filters f_1, f_7, and f_{11} in that block, and set its index entry to x0x xxx x. Another block is allocated, this time for region x1x xxx x; we store encoded filters f_3, f_5, and f_8 in that block, and set its index entry to x1x xxx x.

We begin the second phase with a new region spanning the entire encoded classification space. Again, we consider splitting this region on each bit in turn; the best split found by the algorithm is a split on the sixth bit, producing region xxx xx0 x (containing f_2, f_6, and f_{10}) and region xxx xx1 x (containing f_4, f_9, and f_{12}). The algorithm considers splitting each of these into subregions, but finds no acceptable choices in either case. Thus we end the second phase, assigning f_2, f_6, and f_{10} to a block with index entry xxx xx0 x, and assigning f_4, f_9, and f_{12} to a block with index entry xxx xx1 x.

At this point, all filters have been assigned to blocks, so the algorithm terminates. The result is shown in Figure 5.8. Suppose a query arrives with header value (00110011, 01000101). First, we create the encoded representation of the query, which is 100 001 1. Then, this is compared against the index entries. The first and fourth index entries match, so the first and fourth filter storage blocks are activated for search. This results in a match for filter 100 xxx 1 (i.e. f_7), which is the correct answer.

Note that any search will match exactly two of the index filters, so, on any query only half of the filter storage blocks need to be searched.

Note also that the partitioning is actually applied to the encoded representation of the rules. The partitioning algorithm does not need to know anything about the

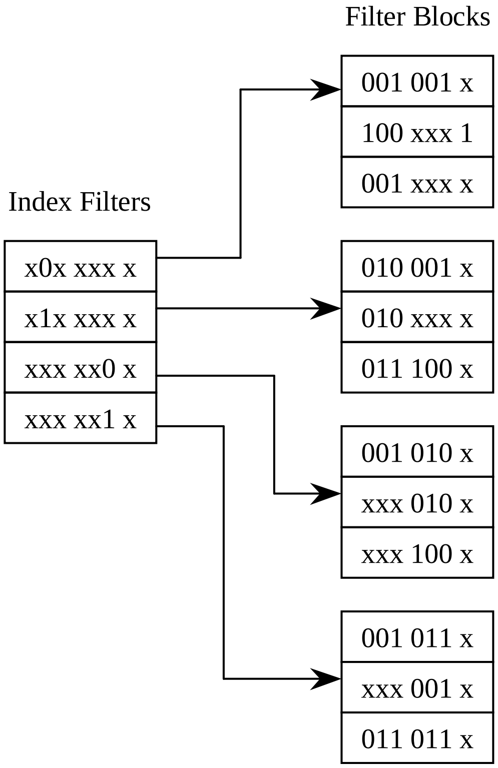


Figure 5.8: Extended TCAM contents for example

existence of layers or which bits are associated with which layers; it only needs to know how wide the encoded search key is. This is because the partitioning algorithm considers each bit in the search key as a candidate on which to split the regions; therefore it does not need to know which bits represent which fields or anything of that sort.

5.2 TCAM Update Technique

In this section we note that filter database update operations (insertion/deletion of a filter) can cause fragmentation of the primitive range identifier space, which results in encodings that are wasteful in terms of TCAM bits used for the encodings; following that, we show a technique for changing an encoding scheme on the fly. Without such a technique, a change to the encoding scheme would require blocking all queries to the TCAM until every rule in the TCAM had been updated.

Insertion of a rule with a new primitive range requires either the creation of a new layer in the primitive range hierarchy or the insertion of the new primitive range into an existing layer. Creation of a new layer requires one more bit in the encoded representation of filters. Insertion of a new primitive range into an existing layer can also require an extra bit (e.g. if we add a new range into a layer with pre-existing ranges labeled 01, 10, and 11, then we relabel those ranges as 001, 010, 011 and then insert the new range as 100). When a new bit is required, it need not be contiguous with the other bits for that field; thus allocation of a new bit is not difficult, as long as a free bit exists (i.e. as long as the encoded width does not exceed the width of a TCAM word).

Bits allocated in this fashion can be deallocated upon deletion of filters, if certain conditions are met. When the last filter to use a particular primitive range is deleted, that primitive range is removed from the primitive range hierarchy, and its identifier is freed. This decreases the number of primitive range identifiers in use. Ideally, a layer with r ranges should require $\lceil \log_2(r+1) \rceil$ bits, but in practice we can encounter fragmentation which results in wasteful allocation of TCAM bits.

For example, suppose we have a layer with range identifiers 01, 10, and 11. Now suppose we insert a rule with a new range in that layer. We now have range identifiers 001, 010, 011, and 100. If a subsequent delete operation causes us to deallocate range identifier 010, we are left with 001, 011, and 100. So we find ourselves using three bits to represent a set of identifiers that should only require two bits.

Thus we propose the following extension which allows the encodings to be changed without blocking queries to the TCAM. Updates to the encoding are not expected to occur frequently, but when they occur, they require rewriting the entire TCAM. Thus it is desirable to retain the ability to process lookups during the transition from one encoding to another.

The extension works as follows: We use one bit in each TCAM entry as an encoding version indicator; initially, TCAM entries are created with this bit set to 0. When the use of a new encoding scheme is desired (e.g. when defragmentation of primitive range identifier space is desired), one simply rewrites the TCAM entries, using the new encoding, and with the encoding version bit set to 1.

Classification of a packet during the encoding update requires two queries, one in the old encoding (including the version bit being set to 0) and one in the new encoding (with version bit set to 1); fortunately, TCAM search speeds are high enough that this should not be a problem for most applications.

Applying this technique in a Extended TCAM requires that the index be kept consistent with the TCAM block contents. One technique is to build an Extended

TCAM where each block has two index filters, either of which is capable of activating the block.

A different approach would be to update one block at a time as follows: allocate a new block; copy the filters from the source block (which uses the old encoding) into the new block (using the new encoding); deallocate the old block when done. This method requires having one extra block of TCAM space available, but avoids requiring each block to have multiple index entries (and the corresponding increase in power consumption and transistor count).

5.3 Evaluation and Results

To evaluate our packet classification scheme, we study its performance using a collection of synthetic filter sets of various sizes created by the *ClassBench* tool, described in Section 2.5. The synthetic filter sets used in this study range in size from 2,000 filters to 128,000 filters, and are generated from the *ClassBench* parameter files *acl1*, *fw1*, and *ipc1*. In generating these databases, we enabled address prefix scaling with database size; smoothness, address scope, and port scope adjustments were set at 0.

In evaluating the results for this packet classification scheme, we are primarily interested in two things: power consumption and storage efficiency. These are affected by the reduction in TCAM word width due to encoding, by having an efficient representation of range matches, and by the use of a partitioned TCAM.

First, let us consider the gain resulting solely from the reduction of the required TCAM width due to encoding. Table 5.6 shows the number of bits needed to represent each filter, for the various databases used in this study; these results are from the use of P^2C encoding style I. In this study, the encodings were created with all exact match conditions in the first layer, and then adding all other match conditions in the order in which they appear in the database, creating new layers as needed. In the cases marked with an asterisk (*), one or more of the address fields required more than 32 bits in encoded form; thus in those cases we use the unencoded address field instead. An unencoded representation of the 5-tuple used in these databases requires 104 bits; by reducing it to 55 to 91 bits, we can reduce power consumption (and storage complexity, in terms of silicon area or transistor count) to 53% to 88% of the original requirements.

The results in Table 5.6 might not be as good as one would expect. To get a better understanding of the results, let us consider the figures shown in Table 5.7;

Size	*acl1*	*fw1*	*ipc1*
2,000	57	59	74
4,000	62	67	79*
8,000	70	47	86*
16,000	71	55	90*
32,000	62	60	91*
64,000	64	64	68
128,000	64	66*	90*

Table 5.6: TCAM width required, in bits

these figures indicate the minimum number of bits required to represent all ranges in the primitive range hierarchies, assuming that none of them overlap (i.e. if all primitive ranges can be placed in one layer). This lower bound on encoding width is described in [53] (we will derive some improved lower bounds in Section 5.4). These numbers are much lower than the actual number of bits required, so, the magnitude of the numbers in Table 5.6 is not simply the result of having a large number of primitive ranges.

Size	*acl1*	*fw1*	*ipc1*
2,000	26	30	29
4,000	28	33	31
8,000	31	37	34
16,000	34	39	36
32,000	39	41	41
64,000	41	43	44
128,000	43	45	46

Table 5.7: Lower bound on TCAM width required

The difference between this lower bound and the actual number of bits required results from needing multiple layers. The number of layers needed to encode each field is shown in Table 5.8. Note that the PR column refers to encoding of both the transport protocol number and protocol-specific information such as TCP flags and ICMP message types.

Indeed, most of the databases studied in fact required multiple layers for most fields. It is not unexpected that the cases with 13 and 14 layers did not fit within 32 bits, but the cases where 3 layers do not fit are more of a surprise. These cases result when databases have a large number of primitive ranges, especially when the additional layers contain more than just a few ranges each. For example, the fw1-derived database with 128,000 entries has 50,374 primitive ranges in the source address field; the first layer has 49,722 (requiring 16 bits), the second has 472 (requiring 9 bits) and the third has 180 (requiring 8 bits).

Seed	*Size*	*SA*	*DA*	*SP*	*DP*	*PR*
acl1	2,000	3	5	0	4	3
acl1	4,000	3	5	0	4	3
acl1	8,000	3	6	0	4	3
acl1	16,000	3	5	0	4	3
acl1	32,000	2	5	0	4	3
acl1	64,000	2	4	0	4	3
acl1	128,000	2	3	0	4	3
fw1	2,000	3	3	2	2	4
fw1	4,000	3	3	2	2	4
fw1	8,000	2	1	2	2	4
fw1	16,000	3	1	2	2	4
fw1	32,000	3	1	2	2	4
fw1	64,000	3	1	2	2	4
fw1	128,000	3*	1	2	2	4
ipc1	2,000	3	5	2	4	6
ipc1	4,000	3	4*	3	4	6
ipc1	8,000	3	5*	3	4	6
ipc1	16,000	3	5*	3	4	6
ipc1	32,000	3*	5*	3	4	6
ipc1	64,000	2	2	3	4	6
ipc1	128,000	3*	2	3	4	6

Table 5.8: Layers required for encoding fields

Further savings are obtained by eliminating the need to expand range-match entries into multiple prefix entries in the TCAM. As noted before, this could be done via hardware support for range checking [40] but that is not necessary when using the P^2C encoding scheme. P^2C encoding style I requires one TCAM entry per rule; by comparison, the standard approach of expanding ranges into prefixes typically requires many more entries than rules, as shown in Table 5.9. The ACL and IP Chains databases did not expand by as much as the databases in [42], since the ACL and IP Chains databases have less frequent use of range matches; on the databases used in [42] we would expect PEST to produce even better results.

The greatest power savings, however, come from the partitioning of the TCAM; as noted earlier, the power reduction thus derived comes at a small cost of storage

Seed	*Number of Filters*	*TCAM Entries*
acl1	2,000	2,902
acl1	4,000	5,669
acl1	8,000	11,267
acl1	16,000	22,096
acl1	32,000	44,054
acl1	64,000	86,714
acl1	128,000	174,523
fw1	2,000	7,335
fw1	4,000	13,345
fw1	8,000	24,900
fw1	16,000	53,795
fw1	32,000	110,620
fw1	64,000	223,615
fw1	128,000	444,815
ipc1	2,000	2,827
ipc1	4,000	5,673
ipc1	8,000	10,927
ipc1	16,000	22,088
ipc1	32,000	44,149
ipc1	64,000	87,923
ipc1	128,000	175,737

Table 5.9: Results of expanding ranges for standard TCAM representation

efficiency. To measure power efficiency, we again use the *TCAM power fraction*, defined as the ratio of TCAM bits searched using a partitioned TCAM to the number of TCAM bits searched using a nonpartitioned TCAM; for this metric, lower is better. To measure storage efficiency, we use the *TCAM storage complexity*, defined as the ratio of TCAM storage bits required, using a partitioned TCAM, to the number of TCAM storage bits required for a nonpartitioned TCAM; for this metric also, lower is better. In the case of the partitioned TCAM, index entries are to be included in these calculations, as well as the unused entries in any storage blocks only partially filled.

In Chapter 4, we observe that a TCAM block size of the largest power of two smaller than $(1/2)N^{1/2}$ appears to be the best choice. However, as the filter block size becomes smaller and smaller, the overhead involved in the partitioning of the device becomes significant. Thus we use slightly larger block sizes in this study, as indicated in Table 5.10.

Number of Filters	*Block Size*
2,000	64
4,000	64
8,000	128
16,000	128
32,000	256
64,000	256
128,000	512

Table 5.10: Block sizes used for partitioned TCAM

The power fraction resulting from the use of a partitioned TCAM, then, is $w(b + sk)/wN$, where b is the number of storage blocks used by the partitioning algorithm, s is the number of phases used by the partitioning algorithm (and thus the number of blocks that must be searched for any packet), k is the storage block size (in TCAM entries), w is the width of an encoded filter (in bits), and N is the number of filters in the database. Results for the databases studied are shown in Figure 5.9. The power fraction resulting from partitioning ranges from 1.5%, which is extremely good, to 21%, which is still substantial. Also, note that the greatest power reductions occur with the larger databases, which tend to be the cases where power reduction is of particular importance.

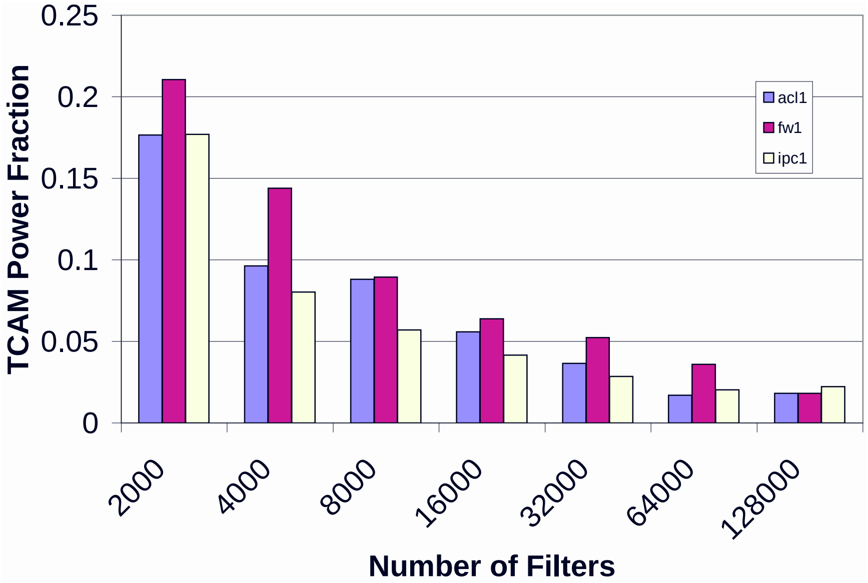


Figure 5.9: TCAM power fraction resulting from partitioning

The storage complexity of using the partitioned TCAM is $w(b{+}bk)/wN$; results for the databases studied are shown in Figure 5.10. The partitioned TCAM results in an increase in storage complexity, due to the need for index entries, and due to any storage blocks that are not entirely full. This increase, however, is fairly small (2% to 33% for the databases studied) compared to the improvement in power fraction; furthermore, it will be more than offset by the gains from encoding and the use of an efficient range representation, as we see in the following metric.

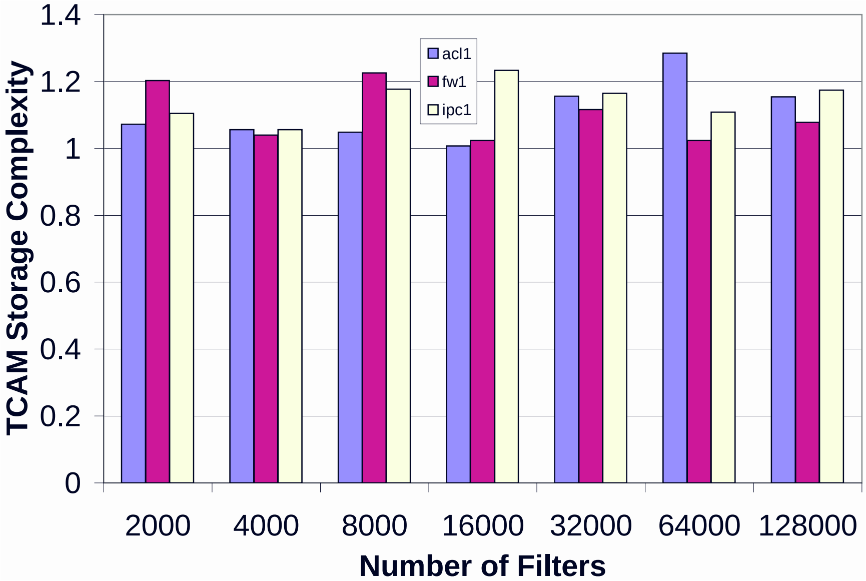


Figure 5.10: Storage complexity factor resulting from partitioning

To evaluate the overall efficiency of packet classification via Partitioned Encoded Search of TCAMs, we use the same power fraction and storage complexity measures, relative to the use of a standard TCAM implementation. Thus, the overall power fraction is $w(b + sk)/(104N_e)$, where N_e is the number of TCAM entries required after expanding ranges into prefixes (Table 5.9) and 104 is the number of bits required to represent the header fields of the 5-tuple in unencoded form. The overall TCAM power fractions, shown in Figure 5.11, range from 0.33% to 8.9%, which represents a substantial reduction in power consumption.

The overall TCAM storage complexity is $w(b{+}bk)/(104N_e)$. Figure 5.12 shows the results for the databases used in this study. The overall TCAM storage complexity for these databases ranges from 16% to 74%, which is quite good.

Of course, this approach requires the use of a lookup mechanism such as BART to convert classification queries into their encoded forms. This lookup mechanism

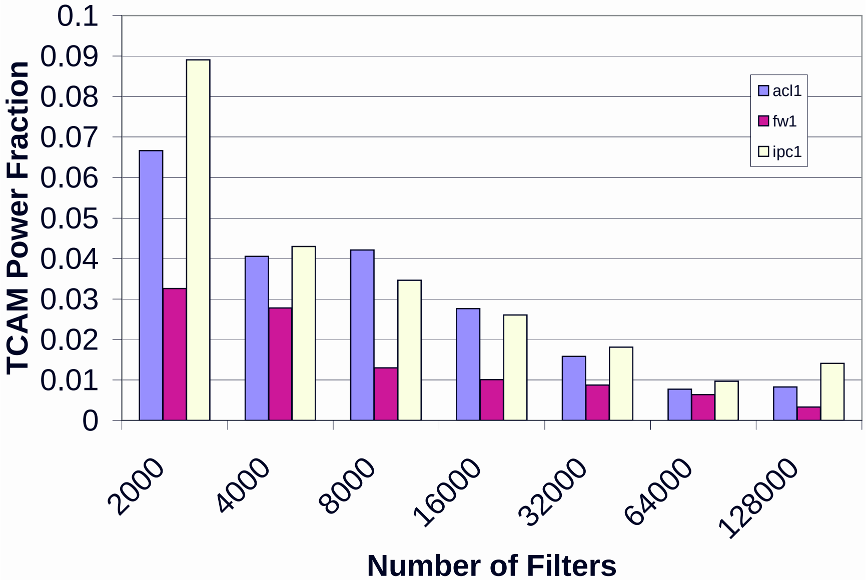


Figure 5.11: Overall TCAM power fraction

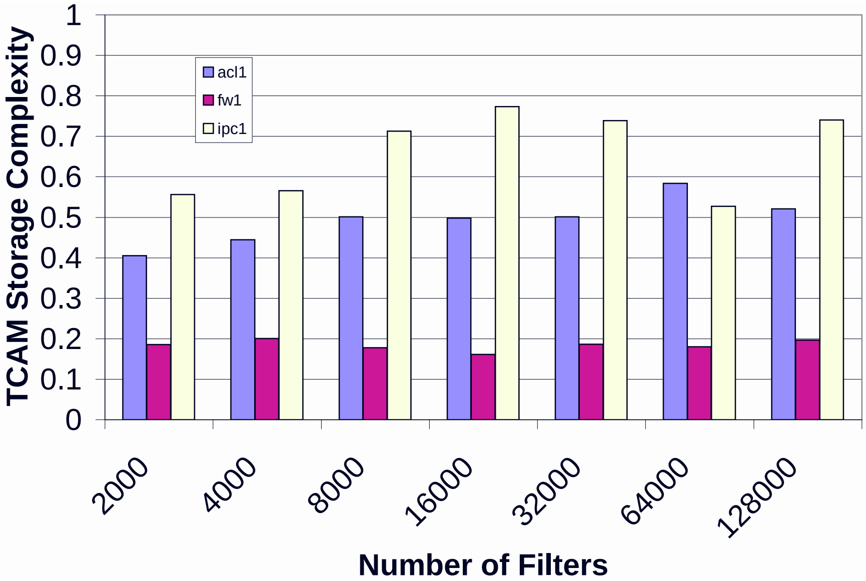


Figure 5.12: Overall TCAM storage complexity

will add its own cost (e.g. SRAM storage and accesses to SRAM), but, in the case of BART, these costs are quite low [53]. And, if the encoding steps are pipelined, it comes at little or no cost in terms of classifier throughput.

5.4 Lower Bounds on Encoding Width

The previously mentioned lower bounds (calculated as described in [53]) are somewhat far away from the encoding width results in the experiment; this brings up the question of whether this discrepancy is due to a loose lower bound or if our encoding implementation is simply not performing well. To answer this question, we derive some improved lower bounds on the encoding width requirements.

5.4.1 Initial Lower Bounds on $\mathrm{P}^2\mathrm{C}$ Encoding Widths

Here we examine two initial lower bounds for the encoding width. Not only do these serve as lower bounds, but also their derivation will build intuition to guide us towards a more generalized lower bound discussed later.

In the first lower bound, we consider only that there are n intervals and they are assigned to k layers; for this simplified problem we do not require overlapping intervals to be assigned to different layers. Note that any solution to a real $\mathrm{P}^2\mathrm{C}$ encoding must also be a valid solution to the greatly simplified problem, described below.

Simplified Layer Assignment Problem: Assign n intervals to exactly k layers, such that the total encoding width (i.e. $\sum_{i=1}^{k}\lceil\log_2(|L_i|+1)\rceil$, where L_i is the set of intervals assigned to layer i) is minimized.

Solution: An optimal assignment (i.e. assignment requiring the fewest bits) to the Simplified Layer Assignment Problem exists with $n-(k-1)$ intervals in the first layer, and 1 interval in each of the remaining layers; a proof is presented below. Therefore, the number of bits needed for a Simplified Layer Assignment grouping of n intervals in k layers is at least $\lfloor\log_2(n-(k-1)+1)\rfloor+(k-1)$. This is a lower bound on the number of bits needed for a $\mathrm{P}^2\mathrm{C}$ encoding of n intervals in k layers, since all valid $\mathrm{P}^2\mathrm{C}$ encodings also fit the Simplified Layer Assignment Problem. After simplification, this bound becomes: $\lfloor\log_2(n-k+2)\rfloor+k-1$

Furthermore, such an optimal solution to the Simplified Layer Assignment Problem with n intervals and i layers has a bit width equal or less than an optimal solution with n intervals and $i+1$ layers, since we can simply take the interval from layer $i+1$ and insert it into layer 1. This eliminates the bit needed for layer $i+1$, and increases the number of bits needed for layer 1 by at most one. Repeated application of this implies that $\lfloor \log_2(n-k+2) \rfloor + k - 1$ is also a lower bound on bit widths for Simplified Layer Assignment Problem solutions and P^{2}C encodings of n intervals in more than k layers.

Proof: Each of the k layers must contain at least one interval (see definition of k). Furthermore, a total of n intervals must be assigned to the various layers (definition of n).

Let G_i represent an optimal grouping of intervals into layers, with i of the k layers having more than one interval each. For simplicity, let us order the layers in decreasing order of size, i.e. $G_i = \{L_1, L_2, ...L_k\}$ where $L_m < L_n$ where $m < n$. In this case, L_i is the smallest layer containing more than one interval.

For any G_i where $i > 2$, we can create a grouping G_{i-1} with equal or smaller coding width (and $i-1$ of the k layers containing more than one interval) as follows: Begin with the grouping G_i, but move $|L_i| - 1$ intervals from L_i into L_1. This results in layer 1's encoding increasing in size by at most one bit (since L_1 already has at least $|L_i|$ items, adding one extra bit to its encoding size allows at least an extra $|Li| + 1$ items to fit), and the encoding for layer i decreasing in size by at least one bit. Since $i-1$ of its k layers contain more than one interval each, we can call it G_{i-1}. And, because G_{i-1} requires the same or fewer number of bits for its encoded representation, G_{i-1} is also an optimal solution to the Simplified Layer Assignment Problem.

By applying this repeatedly to an optimal solution G_i, we eventually obtain an optimal solution G_1 with one interval in each of layers 2 through k, and the remaining $n - (k-1)$ intervals in the first layer.

Improving the initial lower bound: The act of assigning n intervals to k layers can be compared to the act of k-coloring n vertices in a graph (i.e. assigning one of k possible colors to each vertex, such that no pair of vertices connected by an edge are assigned the same color), where an edge between two vertices exists if and only if the primitive ranges corresponding to those vertices overlap. As a first step towards a more general approach, we use the size of the largest 1-colorable subgraph to create a tighter bound than the first bound.

- Let n_i = size of largest i-colorable subgraph.
- Let n = number of intervals.

- Let k = minimum number of colors needed to color all n intervals.

Note that $n - n_1 \geq k - 1$, because if a layer contains n_1 intervals, we still need at least $k - 1$ layers which must contain at least one interval each.

Refined Assignment Problem: Assign n intervals to exactly k layers, with no layer containing more than n_1 intervals, and such that the encoding width (i.e. $\sum_{i=1}^{k} \lceil \log_2(|L_i| + 1) \rceil$, where L_i is the set of intervals assigned to layer i) is minimized.

Lower Bound: An optimal solution to the Refined Assignment Problem requires **at least** $\lfloor \log_2(n_1 + 1) \rfloor + \lfloor \log_2(n - n_1 - (k-2) + 1) \rfloor + k - 2$ bits, which is also a lower bound on the number of bits for a P^{2}C encoding of n intervals into k layers where n_i is the size of the largest i-colorable subgraph.

Proof: First let us show that an optimal solution to the Refined Assignment Problem exists with at least one layer requiring at least $\lfloor \log_2(n_1 + 1) \rfloor$ bits. If an optimal solution exists with none of its layers requiring that many bits, we can transform it into a solution (of equal or better optimality) with a layer requiring $\lfloor \log_2(n_1 + 1) \rfloor$ bits as follows:

Suppose we have an optimal solution where the largest layer contains i intervals. If $2i + 1 \leq n_1$, then we can move $i + 1$ more intervals from other layers into the largest layer, increasing its encoding size by 1 bit. Let L represent a layer (other than the largest one) which, prior to the move, contains more than one interval; let j denote the number of intervals it contains. In the step of moving the $i+1$ intervals to the largest layer, if we select those intervals such that $j - 1$ of them are removed from L, then we shrink the encoded size of L by at least one bit. Thus the new solution has equal or lesser encoding length, and its largest field requires one more bit for encoding than the previous solution.

We can repeat this step until $2i + 1 > n_1$; at that point, the encoded representation for the largest layer requires $\lceil \log_2(i + 1) \rceil$ bits. Since $2i + 1 > n_1$ implies $i > \frac{n_1 - 1}{2}$, we can say $\log_2(i+1) > \log_2(\frac{n_1-1}{2} + 1) = \log_2(\frac{n_1+1}{2}) = \log_2(n_1 + 1) - 1$; Since $\log_2(i + 1)$ is strictly greater than $\log_2(n_1 + 1) - 1$, we can conclude that the encoded representation for the largest layer in fact requires at least $\lfloor \log_2(n_1 + 1) \rfloor$ bits.

Thus, an optimal solution exists requiring $\lfloor \log_2(n_1+1) \rfloor$ bits for one of its layers, with at most n_1 intervals in that layer. At least $n - n_1$ intervals must be assigned to the remaining $k - 1$ layers; thanks to our study of the Simplified Layer Assignment

Problem, we know that this takes at least $\lfloor \log_2((n-n_1)-(k-1)+2)\rfloor + (k-1)-1$ bits. Therefore, our lower bound on the total number of bits needed for the Refined Assignment Problem is $\lfloor \log_2(n_1+1)\rfloor + \lfloor \log_2((n-n_1)-(k-1)+2)\rfloor + (k-1)-1$. After simplification, this becomes: $\lfloor \log_2(n_1+1)\rfloor + \lfloor \log_2((n-n_1)-k+3)\rfloor + k-2$.

In fact, this lower bound also applies when more than k layers are used, since the lower bound for the Simplified Layer Assignment Problem applies in those cases as well.

5.4.2 Generalized Lower Bound

In this section we generalize the lower bound to include constraints on the maximum size i-colorable subsets. To establish the lower bound, let us first define a generalized version of the interval assignment problem as follows.

Generalized Assignment Problem with C-Colorability Constraint: Assign n intervals to exactly k layers, where for each integer i from 1 to C, no i layers contain more than n_i intervals, and such that the encoding width (i.e. $\sum_{i=1}^{k}\lceil |L_i|+1\rceil$, where L_i is the set of intervals assigned to layer i) is minimized.

In this section we show that the encoding representation requires at least a number of bits equal to the expression

$$\lfloor \log_2(n_1+1)\rfloor + \sum_{j=2}^{i}\lfloor \log_2(n_j-n_{j-1}+1)\rfloor + \lfloor \log_2(n-n_i-(k-i))\rfloor + k-i-1$$

Proof: This is an inductive proof; we begin by showing that the bound holds for the base case of $C=1$, and then show inductively that it holds for each integer C greater than that.

Base Case: In the case of the Generalized Assignment Problem with $C=1$, i.e. only the 1-colorability constraint, the problem is equivalent to the Refined Assignment Problem. From Section 5.4.1 we know that $\lfloor \log_2(n_1+1)\rfloor + \lfloor \log_2((n-n_1)-k+3)\rfloor + k-2$. is a lower bound on the number of bits needed for the encoded representation, which is actually a slightly tighter lower bound than the $\lfloor \log_2(n_1+1)\rfloor + \lfloor \log_2(n-n_1-k+1)\rfloor + k-2$ which we needed to prove for this step.

Inductive Step: Here we show that, if our lower bound holds for Generalized Assignment Problem with $(i-1)$-Colorability Constraint, it also holds for the Generalized Assignment Problem with i-Colorbility Constraint, where $i \geq 2$.

First let us show that at least one optimal solution to the Generalized Assignment Problem with i-Colorability Constraint contains a layer L_1 requiring at least $\lfloor \log_2(n_1+1) \rfloor$ bits for its encoded representation. If an optimal solution exists with none of its layers requiring that many bits, we can transform it into a solution with a layer requiring $\lfloor \log_2(n_1+1) \rfloor$ bits as follows:

Suppose we have an optimal solution where the largest layer contains j intervals. If $2j+1 \leq n_1$, then we can move $j+1$ more intervals from other layers into the largest layer, increasing its encoding size by 1 bit. Let L represent a layer (other than the largest one) which, prior to the move, contains more than one interval; let l denote the number of intervals it contains. In the step of moving the $j+1$ intervals to the largest layer, if we select those intervals such that $l-1$ of them are removed from L, then we shrink the encoded size of L by at least one bit. Thus the new solution has equal or lesser encoding length, and its largest field requires one more bit for encoding than the previous solution.

We can repeat this step until $2j+1 > n_1$; at that point, the encoded representation for the largest layer requires $\lceil \log_2(j+1) \rceil$ bits. Since $2j+1 > n_1$ implies $j > \frac{n_1-1}{2}$, we can say $\log_2(j+1) > \log_2(\frac{n_1-1}{2}+1) = \log_2(\frac{n_1+1}{2}) = \log_2(n_1+1) - 1$; Since $\log_2(j+1)$ is strictly greater than $\log_2(n_1+1)-1$, we can conclude that the encoded representation for the largest layer in fact requires at least $\lfloor \log_2(n_1+1) \rfloor$ bits.

For a lower bound on the number of bits needed to encode the remaining layers, we can consider the problem of assigning $n-n_1$ of the remaining intervals in an instance of Generalized Assignment Problem with $(C-1)$-Colorability Constraints. We set up the sub-problem as follows:

- The number of intervals to be assigned in the subproblem is $n-n_1$. In cases where fewer than n_1 intervals were assigned to L_1, combining L_1 with the results of the subproblem will not yield a correct solution to the original problem; this is acceptable because we are merely looking for a lower bound on the encoding width.

- The i-coloring set size constraints n'_i for the subproblem are defined in terms of the n_i constraints from the original problem as follows: $n'_i = n_{i+1} - n_1$. This ensures that the original problem's constraints (i.e. that no i layers contain more than n_i intervals) will still be met, when we consider the subproblem and plus the layer L_1.

- The number of layers in the subproblem definition is $k-1$.

In the inductive step, we're given that the expression holds for the previous step, i.e. that our lower bound expression is correct for the Generalized Assignment Problem with $(C-1)$-Colorability Constraints. Thus we can use that expression to write a lower bound of

$$\lfloor \log_2(n'_1+1)\rfloor + \sum_{j'=2}^{(i-1)} \lfloor \log_2(n'_{j'} - n'_{j'-1} + 1)\rfloor$$
$$+ \lfloor \log_2((n-n_1) - n'_{i-1} - ((k-1)-(i-1)))\rfloor + (k-1)-(i-1)-1$$

on the number of bits required for the encoded representation of the subproblem. By substituting for the n'_i constraints, it can be rewritten as

$$\lfloor \log_2(n_2 - n_1 + 1)\rfloor + \sum_{j'=2}^{(i-1)} \lfloor \log_2((n_{j'+1} - n_1) - (n_{j'-1+1} - n_1) + 1)\rfloor$$
$$+ \lfloor \log_2((n-n_1) - (n_{i-1+1} - n_1) - ((k-1)-(i-1)))\rfloor + (k-1)-(i-1)-1$$

By simplifying, and rewriting the summation in terms of j where $j = j'+1$, we get

$$\lfloor \log_2(n_2 - n_1 + 1)\rfloor + \sum_{j=3}^{i} \lfloor \log_2(n_j - n_{j-1} + 1)\rfloor + \lfloor \log_2(n - n_i - (k-i))\rfloor + k - i - 1$$

The first term can now be folded into the summation, yielding the expression

$$\sum_{j=2}^{i} \lfloor \log_2(n_j - n_{j-1} + 1)\rfloor + \lfloor \log_2(n - n_i - (k-i))\rfloor + k - i - 1$$

Adding in the $\lfloor \log_2(n_1+1)\rfloor$ bits for L_1 gives us a final result of

$$\lfloor \log_2(n_1+1)\rfloor + \sum_{j=2}^{i} \lfloor \log_2(n_j - n_{j-1} + 1)\rfloor + \lfloor \log_2(n - n_i - (k-i))\rfloor + k - i - 1$$

as a lower bound on the number of bits required, which completes the proof.

This lower bound on the number of bits required for the Generalized Assignment Problem is also a lower bound on the number of bits required for a P^{2}C encoding of n intervals with the n_i colorability constraints.

Finding Maximal i-Colorable Sets: Here we discuss an approach for finding the n_i values.

The basic idea is that we sort the intervals in increasing order of right endpoint, and then process intervals one at a time. We use variables e_j to denote the endpoint of the interval most recently added to layer j; initially, each e_j can be considered to be set to -1, or any other value less than the minimum left endpoint of all the intervals.

When an interval (a, b) is considered, we look for layers that can accommodate that interval. If $a \leq e_j$ for all j, then the interval can not be added to any of the layers, so we skip it. Otherwise, for each interval j with $e_j > a$, we select the layer m with maximum e_j, and place the interval (a, b) in that layer and update e_m by setting it to b.

After all intervals have been processed, n_i can be determined by summing the sizes of all layers. Alternatively, the program can keep a running total of layers added, rather than actually inserting intervals into layers.

Proof: This is a greedy algorithm, so we begin by showing that an initial greedy choice is correct. Then we show that, once the greedy choice is made, the remaining solution is an optimal solution to an instance of the problem with the remaining intervals. Therefore, by induction on the number of choices made, making the greedy choice at every step results in an optimal solution.

First let us show that the choice to add interval (a, b) is correct, given that it fits into at least one layer. Suppose we have an an optimal solution which does not use (a, b); the next interval it includes, when sorted by right endpoint, is instead some interval (c, d), where $d \geq b$. This interval can be swapped with (a, b), while keeping the rest of the solution the same, to produce an optimal solution containing (a, b).

Next let us show that adding (a, b) to the layer with maximum e_j (out of those with $e_j > a$) is correct. Suppose we have an optimal solution which puts (a, b) in some other layer l, rather than layer m which has maximum e_j of those with $e_j > a$; let us use (c, d) to represent the next layer which this optimal solution places in layer m (and if there is no such interval, the remainder of the proof becomes trivial). Note that $d > b$ since (c, d) occurs later in the sorted order than (a, b). We can swap (a, b)

and all the intervals following it in layer l with (c, d) and all intervals following it in layer m, because we already know that $a > e_m$, and because $e_l < e_m$ and $e_m < c$. This results in an optimal solution where (a, b) is in the layer with maximum e_j, out of those with $e_j > a$.

Now we extend our proof beyond the first greedy choice. We will show that the solution to the remaining problem is an optimal solution; thus, by induction on the number of choices made, making the greedy choice at every step will produce an optimal solution. First we consider the case where the greedy choice was to skip interval (a, b); then we consider the case where interval (a, b) in included.

If S is an optimal solution to finding the maximum i-colorable subset of the set I of intervals, given initial constraints $e_1, e_2, ...e_i$, and the initial greedy choice was that interval (a, b) is not added, then S must be an optimal solution to the remaining problem consisting of I intervals with the same set of initial constraints. Proof by contradiction: Suppose instead there is a solution S_2 containing more intervals than S; S_2 would also be a valid solution to the original problem, and it contains more intervals than S, thereby contradicting the optimality of S with respect to the original problem.

Let us now consider the case where the initial greedy choice was to add interval (a, b) to some layer m. If S is an optimal solution to the original problem, then S' is an optimal solution to the maximal i-coloring problem with the set $I' = I - (a, b)$ intervals and initial constraints $e'_j = e_j$ for all $j \neq m$ and $e_m = b$. Again, we use a proof by contradiction. Suppose instead there is a solution S_2 containing more intervals than S and meeting the same constraints; if that were true, we could take that solution and add interval (a, b) to produce a valid solution to the original problem with more intervals than S contains, thus contradicting the optimality of S regarding the original problem.

5.4.3 Evaluation of P^{2}C Encoding Results

Armed with the lower bounds and an algorithm for finding n_i values, we can now compute some lower bound figures for various filter databases. These numbers allow us to evaluate the efficiency of our P^{2}C encoding algorithm, in terms of bits required for the encoded representations.

Table 5.11 shows results for encoding the Source Address field of a set of synthetic databases. “Encoding Size” refers to the number of bits actually used by

Database Seed	*Database Size*	*Encoding Size*	*Lower Bound 1*	*Lower Bound 2*
acl1	2,000	15	13	13
acl1	4,000	21	17	18
acl1	8,000	28	21	25
acl1	16,000	30	25	28
acl1	32,000	21	20	20
acl1	64,000	23	22	22
acl1	128,000	24	23	23
fw1	2,000	22	16	20
fw1	4,000	26	20	24
fw1	8,000	17	16	16
fw1	16,000	24	20	22
fw1	32,000	28	22	26
fw1	64,000	31	24	29
fw1	128,000	(33)	26	31
ipc1	2,000	18	14	17
ipc1	4,000	22	16	19
ipc1	8,000	27	20	24
ipc1	16,000	31	23	29
ipc1	32,000	(39)	28	37
ipc1	64,000	24	23	23
ipc1	128,000	(35)	30	33

Table 5.11: Encoding width results for Source Address field

the encoding algorithm, whereas "Lower Bound 1" and "Lower Bound 2" refer to the lower bounds based on n_1, and on n_1 and n_2, respectively.

Table 5.12 shows results for encoding the Destination Address field of a set of synthetic databases. Again, "Encoding Size" refers to the number of bits actually used by the encoding algorithm. "Lower Bound i" refers to the lower bound based on colorability constraints n_1 through n_i; an entry of "n/a" indicates a case where $i > k$ and thus that particular lower bound does not apply.

Overall it appears that the majority of the bits used by our encoding implementation are actually necessary, based on the lower bounds found in this study.

We expect, however, encoding of databases using IPv6 addresses to be much more efficient. Encoding inefficiencies result from address intervals that overlap (and thus can not be put in the same layer). IPv6 addresses are four times the width of IPv4 addresses, but we do not expect to have a proportional increase in the maximum

Database Seed	*Database Size*	*Encoding Size*	*Lower Bound 1*	*Lower Bound 2*	*Lower Bound 3*	*Lower Bound 4*
acl1	2,000	24	18	20	20	19
acl1	4,000	23	17	19	19	19
acl1	8,000	24	19	20	20	20
acl1	16,000	23	20	20	19	20
acl1	32,000	23	21	21	22	22
acl1	64,000	24	20	20	19	20
acl1	128,000	24	20	19	20	n/a
fw1	2,000	20	15	17	18	n/a
fw1	4,000	24	19	22	22	n/a
fw1	8,000	13	13	13	n/a	n/a
fw1	16,000	14	14	14	n/a	n/a
fw1	32,000	15	15	15	n/a	n/a
fw1	64,000	16	16	16	n/a	n/a
fw1	128,000	17	17	17	n/a	n/a
ipc1	2,000	31	20	25	28	28
ipc1	4,000	(33)	20	25	30	30
ipc1	8,000	(39)	23	29	35	35
ipc1	16,000	(44)	25	33	39	40
ipc1	32,000	(51)	29	39	48	48
ipc1	64,000	17	17	17	17	n/a
ipc1	128,000	31	30	30	30	n/a

Table 5.12: Encoding width results for Destination Address field

depth of address prefix nesting (and thus the number of layers needed). We therefore expect the encoding efficiency (relative to the number of bits in the unencoded representation) to improve significantly.

5.5 Summary

By combining ideas from P^2C and Extended TCAMs, we can build a new packet classification technique which is more power-efficient than either P^2C or Extended TCAMs alone. This new classification technique reduces the power requirements by a factor of ten to a hundred, while simultaneously reducing the amount of TCAM storage needed; this all comes at the small cost of the encoding as noted in [53]. With its low power consumption and high throughput (with pipelined encoding stages), this scheme appears to be a promising solution for high performance packet classification.

Additional research contributions in this chapter include a means for defragmenting the primitive range identifiers space without the need to block lookups while rewriting the entire TCAM, and improved lower bounds on encoding width requirements.

Chapter 6

Conclusion

High-performance packet classification is crucial to the deployment of several emerging network services; as network speeds and the number of flows to be classified increase, it only becomes more difficult to solve. This dissertation makes several contributions to the field, as summarized below, and also provides interesting directions for future work.

6.1 Contributions

This dissertation describes several advances made in the field of high performance packet classification. These include: a means of reducing data structure size in RFC, which has the fastest and most deterministic lookup performance of all software-based schemes; a new TCAM architecture which retains the high lookup rate of TCAM while greatly reducing the power requirements; and a new classification method leveraging some of the strengths of both Extended TCAMs and P^2C.

Chapter 3 describes ways to reduce the size of data structures used in Recursive Flow Classification. A simple compression technique reduces storage requirements for the classifier's crossproduct tables by 37% on average in the cases studied, and it appears that larger classifiers are more compressible. Efficiency of the compression can be improved by a heuristic which rearranges the tables in accordance with the solution to a traveling salesman problem; this reduces storage requirements by 54% on average in those experiments. But the most benefit, in terms of reducing storage requirements, comes from proper selection of the RFC reduction tree. Optimal reduction tree selection reduces the storage requirements by 74% compared to the average size in the experiments conducted. A dynamic programming algorithm is presented for selecting an optimal reduction tree.

A new TCAM architecture is presented in Chapter 4. This design solves the two main scalability problems with TCAMs (currently the most popular method for high performance packet classification) by using a partitioned TCAM (to reduce power requirements) and by adding hardware support for range-matching; this allows classification at the high rates achievable by TCAMs, while scaling up to hundreds of thousands of filters.

Efficient use of a partitioned TCAM requires grouping filters into blocks such that only a small number of blocks need to be searched for any given query; we present two algorithms which accomplish this: Region-Splitting, which achieves excellent power dissipation results (less than 5% of a standard TCAM for 32,000 filters or more) and very reasonable storage complexity (less than 10% more than standard TCAM in most cases), and Trie-Carving, which is faster than region-splitting and produces competitive power fraction results at a small cost in terms of storage efficiency.

We describe an implementation of hardware hange matching support in CMOS using 44 transistors per bit; for a typical IPv4 application this means a 46% increase in transistor count per word, instead of a 2-6x increase in word count needed. This improves both power requirements and storage efficiency.

Additionally, we describe a means of supporting dynamic filter updates using the Trie-Carving data structures. There is a tradeoff between update complexity and the power fraction obtained. For the databases studied in our experiments, it is possible to obtain a power fraction of 10% and support hundreds or thousands of updates per second by limiting trie sizes to 8,000 filters each.

Additionally, we demonstrate a means of further reducing the power requirements by using a multilevel indexing technique; this technique works with any filter grouping algorithm, and for large databases it can result in power fractions well below 1%.

Chapter 5 introduced a novel packet classification technique which combines ideas from P^2C and Extended TCAMs to produce something which is more power-efficient than either P^2C or Extended TCAMs alone. In addition, we present a means for on-the-fly defragmentation of the primitive range identifier space and some improved lower bounds on the encoding width requirements, both of which can be applied to P^2C as well. The new classification technique reduces the power requirements by a factor of ten to a hundred, while simultaneously reducing the amount of

TCAM storage needed. With its low power consumption and high throughput (with pipelined encoding stages), this scheme is a promising solution for high performance packet classification. Also, due to its use of encoded field representations, this technique becomes much more attractive (in terms of power requirements) with the use of larger search keys, such as those obtained from IPv6 headers.

6.2 Future Directions

While this dissertation presents advancements in the field of packet classification, it also opens promising opportunities for future research.

One such opportunity is the development of filter grouping algorithms for Extended TCAMs and PEST with different strengths. The current algorithms represent a particular compromise between algorithm execution speed, TCAM power efficiency, and TCAM storage complexity; other algorithms may seek to lean more towards one of those goals, perhaps at the expense of the others. Bounds on how well an algorithm can perform, in terms of power fraction, are not even known at this time; thus that in itself is an opportunity for research. Another area for future work is further refinement of filter update handling in Extended TCAMs and PEST.

Also, it may be possible to apply some of the ideas from Extended TCAMs and PEST to other classification frameworks. With the properties of modern DRAM, a very low power classifier could be built with each row of DRAM corresponding to a filter storage block; the indexing mechanism itself can still be TCAM, and when filters are retrieved from DRAM, they can be subjected to a linear search or fed into a single dedicated filter comparator circuit. But this is meant merely as an example; there are many other possibilities in this area.

References

[1] ClassBench web site. <http://www.arl.wustl.edu/˜det3/ClassBench/>.

[2] Packet classification repository. <http://www.ial.ucsd.edu/classification>.

[3] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to cams? In *Proc. of IEEE INFOCOM 2003*, San Francisco, CA, 2003.

[4] F. Baboescu and G. Varghese. Scalable packet classification. In *Proc. of ACM SIGCOMM 2001*, San Diego, CA, August 2001.

[5] S. Blake, D. Black, M. Carlson, E. Davies, and Z. Wang. An architecture for Differentiated Services. *RFC 2475*, December 1998.

[6] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an overview. *RFC 1633*, June 1994.

[7] A. Brodnik, S. Carlsson, M.Degemark, and S. Pink. Small forwarding tables for fast routing lookups. In *Proc. of ACM SIGCOMM 97*, Cannes, France, August 1997.

[8] M. Buddhikot, S. Suri, and M. Waldvogel. Space decomposition techniques for fast layer-4 switching. In *Proc. of Protocols for High-Speed Networks 99*, Salem, MA, August 1999.

[9] A. Campbell, H. De Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela. A survey of programmable networks. *Computer Communication Review*, 29(2):7–23, April 1999.

[10] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[11] P. Crowley, M. Franklin, H. Hadimioglu, and P. Onufryk. *Network Processor Design: Issues and Practices*. Morgan Kaufmann.

[12] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner. A scalable, high performance active network node. *IEEE Network*, January 1999.

[13] S. Deering and R. Hinden. Internet Protocol version 6 (IPv6) specification. *RFC 2460*, December 1998.

[14] W. Eatherton. Hardware-based internet protocol prefix lookups. Master's thesis, Washington University in St. Louis, May 1999.

[15] K. Egevang and P. Francis. The IP network address translator (NAT). *RFC 1631*, May 1994.

[16] A. Feldman and S. Muthukrishnan. Tradeoffs for packet classification. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[17] G. Gibson, F. Shafai, and J. Podaima. Content addressable memory storage device. United States Patent #6,044,005, March 2000.

[18] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proc. of IEEE INFOCOM 98*, San Francisco, CA, April 1998.

[19] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proc. of ACM SIGCOMM 99*, Cambridge, MA, September 1999.

[20] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Proc. of Hot Interconnects 1999*, Stanford, August 1999.

[21] R. Kempke and A. McAuley. Ternary CAM memory architecture and methodology. United States Patent #5,841,874, November 1998.

[22] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. Campbell. Directions in packet classification for network processors. In *Second Workshop on Network Processors (NP2)*, February 2003.

[23] T. V. Lakshman and D. Stidialis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proc. of ACM SIGCOMM 98*, Vancouver, BC, September 1998.

[24] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. In *Proc. of IEEE INFOCOM 98*, San Francisco, CA, April 1998.

[25] C. Matsumoto. Netlogic takes narrow lead with 18-mbit cam shipment. *EE Times*, May 2002.

[26] A. McAulay and P. Francis. Fast routing table lookups using CAMs. In *Proc. of IEEE INFOCOM 93*, San Francisco, Canada, March 1993.

[27] Micron Technology Inc. 36Mb DDR SIO SRAM 2-Word Burst. Datasheet. December 2002.

[28] Micron Technology Inc. Harmony TCAM 1Mb and 2Mb. Datasheet. January 2003.

[29] J. Mogul. Simple and flexible datagram access controls for unix-based gateways. In *USENIX Conference Proceedings*, Baltimore, MD, June 1989.

[30] R. Montoye. Apparatus for storing don't care in a content addressable memory cell. United States Patent #5,319,590, June 1994.

[31] G. Narlikar, A. Basu, and F. Zane. Coolcams: Power-efficient tcams for forwarding engines. In *Proc. of IEEE INFOCOM 2003*, San Francisco, CA, 2003.

[32] J. Postel. User Datagram Protocol. *RFC 768*, August 1980.

[33] J. Postel. Internet Control Message Protocol - DARPA Internet Program Protocol Specification. *RFC 792*, September 1981.

[34] J. Postel (ed.). Internet Protocol - DARPA Internet Program Protocol Specification. *RFC 791*, September 1981.

[35] J. Postel (ed.). Transmission Control Protocol - DARPA Internet Program Protocol Specification. *RFC 793*, September 1981.

[36] N. Shah. Understanding network processors. Technical report, University of California, Berkeley, September 2001.

[37] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proc. of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.

[38] H. Song and J. Lockwood. Efficient packet classification for network intrusion detection using FPGA. In *International Symposium on Field-Programmable Gate Arrays (FPGA'05)*, Monterey, CA, February 2005.

[39] E. Spitznagel. Compressed data structures for recursive flow classification. Technical Report WUCSE-2003-65, Department of Computer Science and Engineering, Washington University in St. Louis, May 2003.

[40] E. Spitznagel. Cmos implementations of a range check circuit. Technical Report WUCSE-2004-39, Department of Computer Science and Engineering, Washington University in St. Louis, July 2004.

[41] E. Spitznagel and D. Taylor. On using content addressable memory for packet classification. Technical Report WUCSE-2005-9, Department of Computer Science and Engineering, Washington University in St. Louis, March 2005.

[42] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using Extended TCAMs. In *Proc. of ICNP 2003*, Atlanta, GA, November 2003.

[43] V. Srinivasan. A packet classification and filter management system. In *Proc. of IEEE INFOCOM 2000*, Anchorage, AK, April 2001.

[44] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. of ACM SIGCOMM 99*, Cambridge, MA, September 1999.

[45] V. Srinivasan and G. Varghese. Fast IP lookups using controlled prefix expansion. In *Proc. of ACM SIGMETRICS*, June 1998.

[46] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer 4 switching. In *Proc. of ACM SIGCOMM 98*, Vancouver, BC, September 1998.

[47] D. Taylor. *Models, Algorithms, and Architectures for Scalable Packet Classification.* PhD thesis, Washington University in St. Louis, August 2004.

[48] D. Taylor. Survey & taxonomy of packet classification techniques. Technical Report WUCSE-2004-24, Department of Computer Science and Engineering, Washington University in St. Louis, May 2004.

[49] D. Taylor and J. Turner. ClassBench: A packet classification benchmark. Technical Report WUCSE-2004-28, Department of Computer Science and Engineering, Washington University in St. Louis, May 2004.

[50] D. Taylor and J. Turner. Scalable packet classification using distributed crossproducting of field labels. Technical Report WUCSE-2004-38, Department of Computer Science and Engineering, Washington University in St. Louis, June 2004.

[51] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[52] J. van Lunteren. Searching very large routing tables in wide embedded memory. In *Proc. of IEEE GLOBECOM 2003*, San Francisco, CA, December 2003.

[53] J. van Lunteren and T. Engberson. Fast and scalable packet classification. *IEEE Journal on Selected Areas of Communication*, 21(4):560–571, May 2003.

[54] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed IP routing lookups. In *Proc. of ACM SIGCOMM 98*, Vancouver, BC, September 1998.

[55] T. Woo. A modular approach to packet classification: Algorithms and results. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

Vita

Edward W. Spitznagel

Date of Birth	January 5, 1974
Place of Birth	Saint Louis, Missouri
Degrees	**Washington University in Saint Louis** B.S. Electrical Engineering, August 1996 M.S. Computer Science, August 2003 D.Sc. Computer Science, anticipated December 2005
Experience	Applied Research Laboratory, Washington University in St. Louis (1998–2005) Microsoft Corporation (1996–1997) Medical Informatics, Washington University in St. Louis (1993–1995)
Scholarships and Awards	Research Assistantship (1998–2005) ACM International Collegiate Programming Contest: 18th place team (1995) Missouri Higher Education Scholarship (1992 – 1996) National Merit Scholarship (1992)
Publications	E. Spitznagel, D. Taylor and J. Turner. Packet classification using Extended TCAMs. In *Proc. of ICNP 2003*, Atlanta, GA, November 2003. S. Choi, J. DeHart, R. Keller, F. Kuhns, J. Lockwood, P. Pappu, J. Parwatikar, W. D. Richard, E. Spitznagel, D. Taylor, J. Turner and K. Wong. Design of a High Performance Dynamically Extensible Router. In *Proc. of the DARPA Active Networks Conference and Exposition, May 2002*.
Technical Reports	E. Spitznagel and D. Taylor. On Using Content Addressable Memory for Packet Classification. Technical Report WUCSE-2005-9, Department of Computer Science and Engineering, Washington University in St. Louis, March, 2005.

E. Spitznagel. CMOS Implementations of a Range Check Circuit. Technical Report WUCSE-2004-39, Department of Computer Science and Engineering, Washington University in St. Louis, July, 2004.

E. Spitznagel. High Performance Packet Classification. Technical Report WUCSE-2004-14, Department of Computer Science and Engineering, Washington University in St. Louis, March, 2004.

E. Spitznagel. Compressed Data Structures for Recursive Flow Classification. Technical Report WUCSE-2003-65, Department of Computer Science and Engineering, Washington University in St. Louis, May, 2003.

F. Kuhns, J. DeHart, R. Keller, J. Lockwood, P. Pappu, J. Parwatikar, W. D. Richard, E. Spitznagel, D. Taylor, J. Turner and K. Wong. Implementation of an Open Multi-Service Router. Technical Report WUCSE-2001-20, Department of Computer Science and Engineering, Washington University in St. Louis, August, 2001.

J. DeHart, W. D. Richard, E. Spitznagel and D. Taylor. The Smart Port Card: An Embedded Unix Processor Architecture for Network Management and Active Networking. Technical Report WUCSE-2001-18, Department of Computer Science and Engineering, Washington University in St. Louis, August, 2001.

M. Franklin, E. Spitznagel and T. Wolf. Design Tradeoffs for Embedded Network Processors. Technical Report WUCSE-2000-24, Department of Computer Science and Engineering, Washington University in St. Louis, July, 2000.

December 2005