"Good morning," said the little prince.
"Good morning," said the merchant.
This was a merchant who sold pills that had been invented to quench thirst. You need only swallow one pill a week, and you would feel no need of anything to drink.
"Why are you selling those?" asked the little prince.
"Because they save a tremendous amount of time," said the merchant. "Computations have been made by experts. With these pills, you save fifty-three minutes in every week."
"And what do I do with those fifty-three minutes?"
"Anything you like..."
"As for me," said the little prince to himself, "if I had fifty-three minutes to spend as I liked, I should walk at my leisure toward a spring of fresh water."

– Antoine de Saint-Exupery [13].

University of Alberta

Improving Packet Classification: Learning From Traffic

by

Kevin Lyle Andrusky ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science.**

in

Department of Computing Science

Edmonton, Alberta
Fall 2005

University of Alberta

Library Release Form

Name of Author: Kevin Lyle Andrusky

Title of Thesis: Improving Packet Classification: Learning From Traffic

Degree: Master of Science

Year this Degree Granted: 2005

Kevin Lyle Andrusky

Date: ____________________

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Improving Packet Classification: Learning From Traffic** submitted by Kevin Lyle Andrusky in partial fulfillment of the requirements for the degree of **Master of Science**.

M.H. (Mike) MacGregor

Chintha Tellambura

Martin Müller

Date: ____________________

To my parents.

Abstract

Some rules in a packet classifier's ruleset are more likely to be matched than others. Which of these rules are most likely to be matched cannot be known a priori, however. As a result, a classifier should be able to adapt itself to the traffic that it sees. This thesis explores the idea of using the traffic a packet classifier is seeing to tune that classifier's search structure in order to improve its average response time. To test this idea, HICuts, a heuristic-based packet classifier recently proposed by Gupta and McKeown, was modified to periodically restructure itself to best match the traffic it is seeing. In addition, a Ternary Content Addressable Memory (TCAM) was added to the classifier to act as a form of transposition table. The classifier was also given the ability to maintain a worst-case bound on its performance, as, by the nature of Internet traffic, classifiers must ensure that all possible packets be matched in a reasonable amount of time.

In order to describe the effects of restructuring, several modifications were made to an Interval Decision Diagram (IDD) notation. Such notation can be used as a HICuts tree is essentially an IDD tree with a slightly different condition for creating leaf nodes. Experimental results drawn from synthetic tests designed to test specific situations and extreme cases, and from real-world tests using actual packet traces are shown. In experimentation, compared to the original static HICuts classifier, the new dynamic classifier performed significantly better in terms of the amount of work performed during classification, while maintaining a strict bound on its worst-case performance.

Acknowledgements

I would like to express my gratitude to all those who aided me in completing this thesis. I am deeply indebted to my supervisor, Dr. M.H. (Mike) MacGregor, whose help, suggestions and encouragement guided me throughout both the research required for and the writing of this thesis. Writing this thesis has been one of the most enjoyable and rewarding experiences of my life and would not have been possible without Mike's guidance and support.

I would also like to thank Dr. Martin Müller who provided the impetus for this research, as well as many very helpful initial suggestions as to the course the research should take. In addition, I acknowledge the contributions of several of my peers, particularly Paul Berube, who offered both suggestions and support in aid of this research. I am also grateful for the questions and comments I received from many members of the Communication Networks Research Group over the course of my research. In particular, I would like to thank Baochun Bai, Shubhankar Chatterjee, Yuxi Li, Qiang Ye, and Drs. Pawel Gburzynski and Ioanis Nikolaidis.

I would like to acknowledge the University of Alberta's Computing and Network Services (CNS) for providing the packet traces which were used for this research. In particular, I would like to thank Josh Ryder of CNS for his invaluable assistance in determining a proper real-life ruleset on which to test my algorithm. In addition, I'd like to express my gratitude to Stephen Curial, who oversaw the final printing and submission of this document for me.

And, of course, I would like to thank my family, especially my parents Lyle and Vivian. Without everything I learned from them from day one, without their support, encouragement and nagging, without the wisdom they attempted to pass on to me, this document would not exist.

Table of Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Problem Definition . . . 2
- 1.3 Worst vs. Average Case: Motivation . . . 3
  - 1.3.1 Sorting . . . 4
  - 1.3.2 Optimal Binary Search Trees . . . 4
  - 1.3.3 Packet Classification . . . 6
- 1.4 Applicability and Impact . . . 7
- 1.5 Thesis Overview . . . 9

2 Current Classification Algorithms **12**
- 2.1 General Overview . . . 12
  - 2.1.1 Linear Search . . . 13
  - 2.1.2 Trie Based Methods . . . 13
  - 2.1.3 Geometric Approaches . . . 15
  - 2.1.4 Recursive Flow Classification . . . 17
  - 2.1.5 Use of Caching . . . 17
- 2.2 HICuts in Detail . . . 19
  - 2.2.1 HyperCuts . . . 22

3 Notation and Metrics **23**
- 3.1 IDD Notation . . . 23
- 3.2 Mathematical Notation . . . 25
- 3.3 Metrics . . . 26

4 The Dynamic Classifier - Design **29**
- 4.1 Range Reorders . . . 30
- 4.2 Node Promotion . . . 31
- 4.3 Range Combination . . . 34
- 4.4 Enhanced Cuts . . . 36
- 4.5 TCAM Caching . . . 37
- 4.6 Remembering and Forgetting . . . 39
- 4.7 Update Frequency . . . 40
  - 4.7.1 Update Frequency - Forgetting . . . 40
  - 4.7.2 Update Frequency - Major Restructurings . . . 41
- 4.8 Worst-Case Bound . . . 42

5 The Dynamic Classifier - Implementation **44**
- 5.1 Implementation of Static HICuts . . . 44
- 5.2 Implementation of the Dynamic Improvements . . . 46

6 Performance **51**
- 6.1 Experiment Design . . . 51
  - 6.1.1 Rulesets . . . 52
  - 6.1.2 Packet Traces . . . 53
- 6.2 Results and Commentary . . . 53
  - 6.2.1 Overall Improvements in Classification . . . 53
  - 6.2.2 Worst-Case Performance . . . 57
  - 6.2.3 Effects of the Individual Improvements . . . 60
  - 6.2.4 Memory Use . . . 63
  - 6.2.5 TCAM Hits - Locality of Traffic . . . 66

6.2.6 Effects of the Rate of Forgetting on Learning 68
6.3 Summary of Results 70

7 Conclusions **71**
7.1 Conclusions 71
7.2 Future Work 72

Bibliography **74**

A Additional Results **76**

List of Tables

1.1 The relative frequency of search for the nodes in Figure 1.1. Reproduced from [11]. . . . 5
1.2 The relative frequency of search for the nodes in Figure 1.2. 5
1.3 The Ruleset used for the motivating example. 7
1.4 The relative frequencies of hitting each region in our search space. Because rules 1 and 3 are found at two different depths, two frequencies are given for them 8

6.1 Best path length results in terms of L. 54
6.2 The average value of L, and 2 times the standard deviation for all tests. 54
6.3 Actual worst case results for all tests. 57
6.4 Maximum worst case results for all tests. 58
6.5 Effects of Node Promotion on classification 61
6.6 Effects of Range Reorders on classification 61
6.7 Effects of Range Combinations on classification 62
6.8 Effects of the TCAM on classification 62
6.9 Effects of Enhanced Cuts on classification 63
6.10 Effects of Node Promotion and Range Combination on classification 63
6.11 Average memory use results for all tests. 64
6.12 Worst case memory use results for all tests. 65
6.13 The average number of hits in the TCAM, as a percentage of packets classified. . . 67
6.14 The effects of the rate of forgetting on the value of L for the Single trace on the (a) Checkerboard and (b) Snort ruleset. 68
6.15 The effects of the rate of forgetting on the value of L for the Telus trace on the (a) Checkerboard and (b) Snort ruleset. 68
6.16 The effects of the rate of forgetting on the value of L for the Uniform trace on the (a) Checkerboard and (b) Snort ruleset. 69

A.1 The effects of the rate of forgetting on the value of L for the Clumped1 trace on the (a) Checkerboard and (b) Snort ruleset. 84
A.2 The effects of the rate of forgetting on the value of L for the Clumped10 trace on the (a) Checkerboard and (b) Snort ruleset. 84
A.3 The effects of the rate of forgetting on the value of L for the SetSource1 trace on the (a) Checkerboard and (b) Snort ruleset. 84
A.4 The effects of the rate of forgetting on the value of L for the SetSource10 trace on the (a) Checkerboard and (b) Snort ruleset. 84

List of Figures

1.1 Two search trees for translation. (a) is balanced while (b) is not balanced, but designed to take advantage of the expected frequency of search for each key. Reproduced from [11]. 5
1.2 A search tree which is linear in its worst case search depth, produced by the algorithm from [11]. The values used to produce this tree are given in Table 1.2. 6
1.3 A balanced binary search tree developed by a process similar to the one described in [18]. 7
1.4 A new binary search tree which takes the frequencies into account, while bounding the worst case search depth. 8

2.1 The ruleset used for the examples in Section 2.1.2. 15
2.2 An example of a Hierarchical Trie on the ruleset given in Figure 2.1 15
2.3 An example of a Set-Pruning Trie on the ruleset given in Figure 2.1 16

3.1 A sample path through a small HICuts tree. 24

4.1 A HICuts tree (a) before and (b) after range reordering. 31
4.2 A HICuts tree (a) before and (b) after node promotion. 34
4.3 A HICuts tree (a) before and (b) after range combination. 36

5.1 The structure of a node. 45
5.2 The structure of a rule. 45
5.3 The structure of an enhanced cuts node. 49

6.1 Average values of L for Single 56
6.2 Average values of L for Telus 56
6.3 Average values of L for Uniform 56
6.4 (a) Actual and (b) maximum worst-case results for Single on Checkerboard 59
6.5 (a) Actual and (b) maximum worst-case results for Telus on Checkerboard 60
6.6 (a) Actual and (b) maximum worst-case results for Uniform on Checkerboard . . . 60
6.7 (a) Average and (b) maximum memory use results for Single on Checkerboard . . . 65
6.8 (a) Average and (b) maximum memory use results for Telus on Checkerboard . . . 66
6.9 (a) Average and (b) maximum memory use results for Uniform on Checkerboard . 66

A.1 Average values of L for Clumped1 76
A.2 Average values of L for Clumped10 77
A.3 Average values of L for SetSource1 77
A.4 Average values of L for SetSource10 77
A.5 (a) Actual and (b) maximum worst-case results for Clumped1 on Checkerboard . . 78
A.6 (a) Actual and (b) maximum worst-case results for Clumped10 on Checkerboard . 78
A.7 (a) Actual and (b) maximum worst-case results for SetSource1 on Checkerboard . . 78
A.8 (a) Actual and (b) maximum worst-case results for SetSource10 on Checkerboard . 78
A.9 (a) Actual and (b) maximum worst-case results for Clumped1 on Snort 79
A.10 (a) Actual and (b) maximum worst-case results for Clumped10 on Snort 79
A.11 (a) Actual and (b) maximum worst-case results for SetSource1 on Snort 79
A.12 (a) Actual and (b) maximum worst-case results for SetSource10 on Snort 79
A.13 (a) Actual and (b) maximum worst-case results for Single on Snort 80
A.14 (a) Actual and (b) maximum worst-case results for Telus on Snort 80
A.15 (a) Actual and (b) maximum worst-case results for Uniform on Snort 80
A.16 (a) Average and (b) maximum memory use results for Clumped1 on Checkerboard 81
A.17 (a) Average and (b) maximum memory use results for Clumped10 on Checkerboard 81

A.18 (a) Average and (b) maximum memory use results for SetSource1 on Checkerboard 81
A.19 (a) Average and (b) maximum memory use results for SetSource10 on Checkerboard 81
A.20 (a) Average and (b) maximum memory use results for Single on Snort 82
A.21 (a) Average and (b) maximum memory use results for Telus on Snort 82
A.22 (a) Average and (b) maximum memory use results for Uniform on Snort 82
A.23 (a) Average and (b) maximum memory use results for Clumped1 on Snort 82
A.24 (a) Average and (b) maximum memory use results for Clumped10 on Snort 83
A.25 (a) Average and (b) maximum memory use results for SetSource1 on Snort 83
A.26 (a) Average and (b) maximum memory use results for SetSource10 on Snort 83

Chapter 1

Introduction

1.1 Overview

Packet classifiers work by matching the headers of packets against a large set of rules. Each rule contains instructions dictating how the packet (and the associated network flow) is to be dealt with by the router. With both increasing bandwidth and increasing demand for classifiers which can work on exceedingly large sets of rules over many dimensions (each dimension being a field in the packet header) classifiers which can more efficiently search such a space are always needed. This document explores the idea of using the traffic a packet classifier is seeing to tune the classifier's search structure in order to improve its average response time, while ensuring that the worst case response time can be bounded to ensure proper functioning of the router (or other hardware) the classifier is running on.

It is a given that some rules in a packet classifier are more likely to be matched than others, however, it cannot be known with any certainty which rules are most likely to be matched a priori. A classifier should be able to adapt itself to the traffic that it sees. To test this idea, HICuts [17], an algorithm for packet classification which uses heuristics to construct a search which balances the search times for all rules, has been modified to periodically restructure itself to best match the patterns of the traffic it is seeing. Several distinct methods of restructuring the search space are described, and techniques for ensuring that worst case times can be bounded are given for each method. In addition, means for determining the optimum interval between restructurings (to best counter the costs of restructuring versus the savings on subsequent searches) are detailed.

The effects of restructuring are described via a modified Interval Decision Diagram (IDD) notation (which can be used since a HICuts tree is essentially an IDD tree which a slightly different condition for creating leaves). This notation describes the path of a single packet through the classifier. Combining the notations of the paths for all packets allows for full representation of the HICuts (or IDD) tree. The IDD notation gives the ability to directly and accurately estimate the amount of work performed by the new, dynamic HICuts algorithm. By computing the sum of the individual costs of a large number of searches, the average search time over a large number of packet classifications can be computed. By looking at the maximum value of the estimated costs for classification, it can be guaranteed that the classifier is not exceeding any worst case bound placed on it.

The effects of the dynamic improvements on classification are then tested via experiments using synthetic tests designed to test specific situations and extreme cases, and real-world tests using actual packet traces and a subset of the classification rules used by Snort, an intrusion detection system which works via packet classification. Compared to a static HICuts classifier, the dynamic classifier performed significantly better in terms of both search depth and amount of work performed during classification.

1.2 Problem Definition

Packet classifiers work from the base of a set of rules. Each rule is defined over d components, d being the number of dimensions in the search. Each component corresponds to a particular entry in a packet header (such as the destination or source address, or the destination or source port). These components can take various forms, including regular expressions, ranges or simple numbers. For every packet that a classifier sees, the rule that matches that packet is found - and relates the proper action for the device running the classifier. A rule is deemed to have matched a packet if, on all of the d dimensions being classified, the values stored in the packet header match the corresponding component of the rule. If more than one rule is matched by a packet, some technique must be used to determine which rule has the highest priority (i.e. takes precedence over the other rules). This determination could be due, for example, to the position of the rules in the classifier - those rules that come first are considered to have higher priority.

There are many different criteria on which to base the performance of classifiers. Some of these criteria, including several adapted from [14] are listed below.

Above all, packet classifiers must ensure that the correct rule is always found for any given packet. Once this is guaranteed, then we must consider the performance of a packet classifier on other criteria. Packet classifiers must work quickly and must be able to work within whatever memory they are given. Several different metrics can be used to gauge the performance of a packet classifier.

A 10Gb/s line can bring in up to 31.25 million packets every second [14]. Searches must be performed quickly enough to classify this many packets. A general rule of thumb is to consider only the worst case search speed, as we must guarantee that *all* packets must be matched in a reasonable amount of time. It will be argued below, however, that average case performance is also an important measure of a classifier's performance.

Smaller storage requirements are generally better in terms of costs (though not if using a small amount of memory degrades the speed of classification by a large amount). In particular, techniques that use specialized memory hardware such as ternary Content Addressable Memories (TCAMs) need to be very careful about the amount of memory they use in order to limit the costs of the hardware needed.

A good packet classification algorithm should be scalable to any number of rules with little increase in the average and worst case search time. With the advent of faster packet classification schemes, ideas have been put forward that would have looked impossible only a few years ago. User driven rule sets for ISP routers could drive the number of rules searched for some routers into the hundreds of thousands [3]. Also, there should be no limit on the size of d. Since multi-dimensional searches are often quite difficult to perform this constraint can be relaxed slightly to ensure that d at least can be large enough to satisfy the most-used entries in a packet header. Finally, the operators for checking to see if a packet matches a field should be as general as possible (perhaps even full regular expressions) in order to help reduce the number of rules that a classifier needs.

Updates to the classifier - be they insertions, deletions or changes to the rules themselves - should be as quick as possible. In addition, there should be no need to completely recreate a search structure every time a change is made to the ruleset.

Currently, packet classifiers are designed to ensure that all rules can be matched in approximately the same amount of time. Since traffic patterns are often not known beforehand, it is generally assumed that this is the most effective way of ensuring that, at least, the worst case search time of the algorithm can be controlled, to ensure that packets are matched at an appropriate rate. However, Internet traffic is never uniform, so balancing all rules so that they are equally likely to be matched will not result in optimal performance. For many classifiers it would be extremely likely that certain regions in the search space will be visited more frequently than others. The easier it is to match the rules in these regions, the better the overall performance of the classifier.

At the same time, a classifier must never allow any rule to require an excessive amount of time to match. Classifiers must react to traffic as it comes in. If a packet takes a significant amount of time to be matched we must queue up or drop other packets. This will result in poorer performance of the classifier (and, worse, poorer performance of the network), and must be avoided at all costs.

These two requirements present a dilemma. The rest of this document details the ways that the average case search time of a classifier can be improved by decreasing the search time for rules in frequently searched regions, while the worst case search time can be bounded to ensure that all rules can be matched in a reasonable amount of time.

1.3 Worst vs. Average Case: Motivation

It is important to emphasize exactly why average case performance is a metric which should be considered alongside worst case performance. To do this, three examples are given below. The first two show the improvements we can expect when considering average case performance, along with the possible dangers of focusing on improving the average case. The final example shows, explicitly, the applicability of improving average case search time in packet classifiers.

1.3.1 Sorting

Consider Quicksort and Insertion Sort. Both of these sorting algorithms work in quadratic time in the worst case. It would be impossible to choose the "better" of these two algorithms based solely on this information - in fact, given that Quicksort is heavily recursive, while Insertion Sort is iterative, one might be tempted to choose the latter over the former. However the average case complexity of Quicksort is sub-quadratic ($O(n \log n)$), while the average case complexity of Insertion Sort remains quadratic. Since it is known that Quicksort will never perform worse than Insertion Sort and that Quicksort performs better than Insertion Sort in the average case, Quicksort is the better choice, given no other constraints on the choice.

1.3.2 Optimal Binary Search Trees

A slightly more interesting example is that of building a search tree for looking up words for a translation program [11]. A perfectly balanced binary search tree would certainly work for such a purpose, however such a tree would make searching for frequently used words in a language as difficult as searching for infrequently used words. Translating English to French, it would make sense for a dictionary to quickly be able to find words such as *the* or *I*, at the expense of other words which are likely to be infrequently searched.

It is well known that, given a set of words and a set of frequencies for the use of those words, a search tree can be built which will provide optimal expected lookup performance (given that the frequencies used in constructing the search tree are, in fact, correct). This can be done via a simple dynamic programming technique, which runs in $\Omega(n^3)$ time [11], where n is the number of words to be placed in the search tree.

[11] gives an example of a balanced binary search tree which, with the given word frequencies, is not optimal. This is reproduced as Figure 1.1. The frequencies by which nodes are searched for are given in Table 1.1. The nodes themselves are labeled p_i and q_i and follow the rules for searching that $p_i < p_{i+1}$, $q_i < q_{i+1}$, and $q_i < p_{i+1} < q_{i+1}$. So, for example, p_i will be found to the left of p_{i+1} in the binary search tree. The tree constructed via the dynamic programming technique is no longer balanced, but has a lower expected search time than the balanced search tree. This is exactly the expected benefit of considering rules frequencies in a classifier. Rather than a balanced splitting of rules, a splitting which divides the rules up such that the overall expected search time is lower than in a balanced situation is preferable. However, two differences between the lookup problem and the classification problem need to be highlighted.

First, there is no way to bound a worst case in the lookup algorithm. Since this code is probably not having to deal with input rates it cannot control, this may not be an issue. The example given in Figure 1.2 shows how some search frequencies could result in the search tree's performance degrading from logarithmic to linear in the worst case. A packet classification algorithm could never allow such performance degradation.

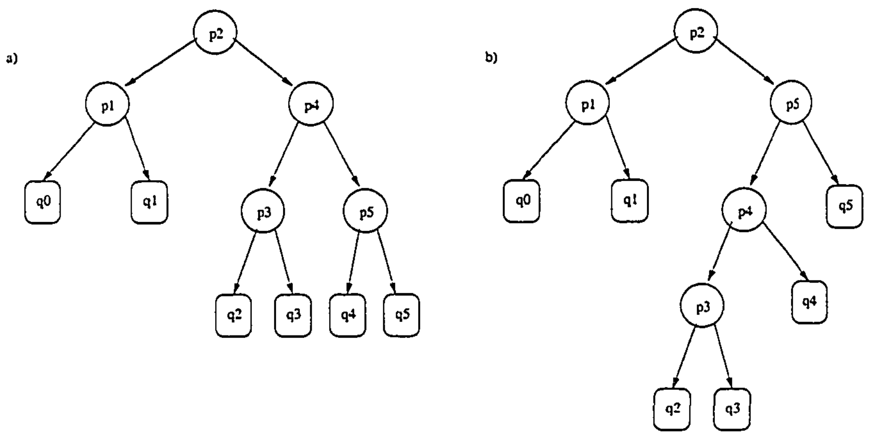

Figure 1.1: Two search trees for translation. (a) is balanced while (b) is not balanced, but designed to take advantage of the expected frequency of search for each key. Reproduced from [11].

i	0	1	2	3	4	5
p_i		0.150	0.100	0.050	0.100	0.200
q_i	0.050	0.100	0.050	0.050	0.050	0.100

Table 1.1: The relative frequency of search for the nodes in Figure 1.1. Reproduced from [11].

Second, one must consider what happens if the frequencies given for the words searched are wrong. Consider that the translation tree has been built for translating standard spoken English to standard spoken French. If it is used for this purpose it will probably perform as expected. If, however, it is then put to the task of translating formal technical documents between the two languages, it will probably exhibit significantly worse performance. Given the example in Figure 1.1, consider if p_3 is, in fact, the most frequently searched. The search performance is no longer even remotely optimal, and the search performs worse than the expected search time on the balanced tree. If the dictionary were able to adapt itself to the circumstances it sees, it would likely perform significantly better in many cases where incorrect assumptions about word frequency have been made.

In fact, since the algorithm runs quite quickly, it would be possible to actually generate word use statistics at runtime, and periodically rebuild the search tree to reflect these statistics.

i	0	1	2	3	4	5
p_i		0.250	0.100	0.050	0.025	0.030
q_i	0.250	0.100	0.040	0.030	0.015	0.010

Table 1.2: The relative frequency of search for the nodes in Figure 1.2.

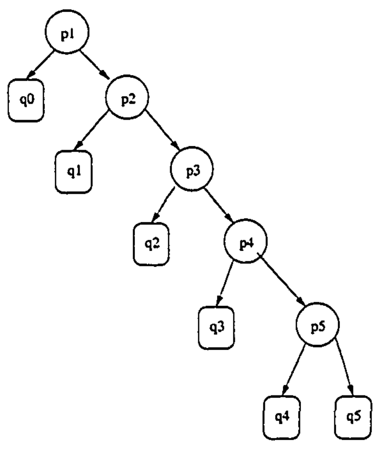

Figure 1.2: A search tree which is linear in its worst case search depth, produced by the algorithm from [11]. The values used to produce this tree are given in Table 1.2.

1.3.3 Packet Classification

The examples given above illustrate why average case is an important metric to consider. The importance of the average case to packet classification itself will now be illustrated.

Given in Table 1.3 is the ruleset which will be used for this example. This ruleset is defined over a 256×256 region. The packet classification algorithm used in this case is a modification of the trie-based classifier described in [18]. Here, at every level in the tree, one comparison is performed against a range. If the comparison succeeds, the search follows the right child, otherwise it follows the left child. The result is a trie-like binary search structure which can be used for packet classification.

One possible configuration of the search tree to correspond to the rules given in Table 1.3, which tries to balance the search tree as much as possible, is given in Figure 1.3. While values are given for both arcs leaving a node, it is important to remember that only one comparison is performed at each node in the search tree. All rules are either three or four nodes from the root of the tree - since all that is performed at each level is one comparison, the depth of a rule in the tree is a valid metric for the amount of work performed by the classifier to classify a packet which matches that rule.

If it is assumed that all rules are to be matched equally (and that rules 1 and 3, which are found at two depths in the search tree, split their hits equally between those two depths) then we can easily see that the expected required amount of work for any given packet header this classifier will see is approximately 3.28. If, however, the distribution of packet headers is not uniform, but instead follows the distribution given in Table 1.4, such expected performance would no longer be seen. In that case, the expected work performed would now be 3.52, slightly worse than if the rules were equally distributed.

However, classifiers can perform better than this. Even if we keep a tight bound on the worst case search depth (in this case, requiring it to be no more than the worst case depth of 4 in the initial

Rule	X Range	Y Range
1	0 - 95	192 - 255
2	96 - 127	192 - 255
3	0 - 127	0 - 191
4	128 - 255	128 - 255
5	128 - 254	64 - 127
6	128 - 254	0 - 63
7	255 - 255	0 - 127

Table 1.3: The Ruleset used for the motivating example.

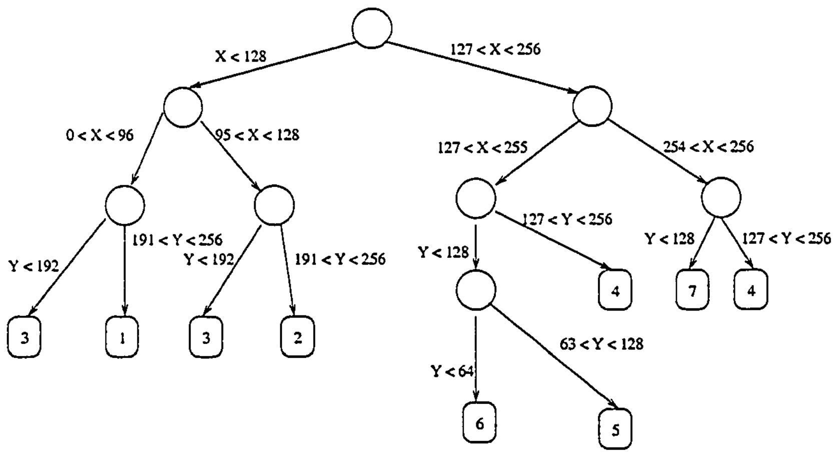

Figure 1.3: A balanced binary search tree developed by a process similar to the one described in [18].

search structure) we can redesign the search space from the configuration in Figure 1.3 into the one shown in 1.4. Here the worst-case bound is maintained. It is easy to calculate that the expected work performed for any packet header, given that the headers are distributed following Table 1.4, is now only 3.03. This is significantly better than the performance on the balanced tree.

The benefits of considering average case are shown clearly here. It is possible to create a search structure for packet classification which can reduce the amount of work performed by a classifier on average while not affecting the worst-case performance of the classifier. This thesis is dedicated to describing how a classifier could be built to consider traffic patterns and use that data to create a better (on average) search structure.

1.4 Applicability and Impact

It has long been asserted that packet classifiers must only consider their worst case search time when ascertaining their time performance. That was originally proposed in [20], and has been echoed throughout the literature on packet classification ever since. The claim that packet classifiers must consider the worst case is certainly valid, as classifiers deal with traffic which is not completely

Region (Rule)	Hit Frequency	Work
1	10%	3
2	3%	3
3	4%	3
4	30%	3
5	50%	4
6	2%	4
7	1%	3

Table 1.4: The relative frequencies of hitting each region in our search space. Because rules 1 and 3 are found at two different depths, two frequencies are given for them

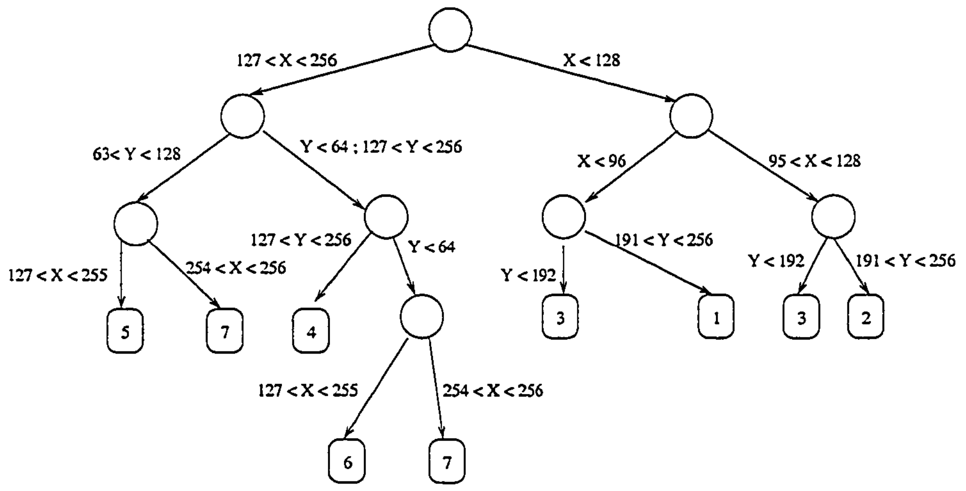

Figure 1.4: A new binary search tree which takes the frequencies into account, while bounding the worst case search depth.

predictable, and which arrives at quick, and difficult to control, rates. If classifiers had excellent average-case performance, but poor worst-case performance, it is easy to envision the problems that could crop up should a large number of packets which all can only be classified in the worst-case time require classification at once: long queues, dropped packets, and reduced network throughput, among others.

However, this does not mean, as many in the field of packet classification assert, that we should not consider metrics other than worst case when evaluating the time performance of classifiers. Certainly, as shown above, it is possible to improve the average case of classifiers, while ensuring that the worst case is bounded, and thus cannot detrimentally impact the network itself based upon the 'wrong' traffic showing up at the classifier at any given moment. That is the primary appeal of this research. Even more than simply showing the benefits of a dynamic classifier, over a static one, it proves that such a classifier can be made, and that while we must always consider the worst-case search time of our algorithms, that does not render us incapable of improving average-case search time.

Improving the average case search time has several advantages. Many classifiers are not run on their own hardware. Intrusion detection programs (such as Snort), firewalls and other programs which aid in ensuring the security of personal computers often run on those computers themselves. In these cases, the less work the classifier performs, the more CPU time is available for the other processes running on that system. Certainly, considering only worst-case time in such a situation would place (relatively) excessive demands on the CPU. While a classifier can guarantee that it reacts to packets in time, there is no guarantee that it is doing so in an efficient manner, friendly to other processes running on the same hardware.

Improving average search time is also useful for dedicated routing hardware. As quality of service becomes increasingly important, the number of rules over which classification is performed, as well as the number of dimensions over which classification is performed continue to increase [3], Any technique which can speed up the average time needed for classification would be extremely beneficial for providing such service. While no speed-up can be expected in the worst case, the fact that, on average, packets are classified faster than they would otherwise be certainly contributes to the overall quality of service provided by a network.

1.5 Thesis Overview

Throughout this document, several techniques which allow the HICuts algorithm (presented in Section 2.2) to adapt itself to the traffic which it is seeing are presented. These techniques provide significant improvements in the average case performance of HICuts, while ensuring that the worst-case performance can be bounded. This same adaptivity could be applied to many other classification algorithms, some of which are detailed in Section 2.1. Whether or not general techniques which can be applied to all classifiers exist is an interesting question which is not investigated in this work.

In order to best describe the effects of any changes to a classification structure, a formal, mathematical notation would be beneficial. A notation based upon the notation for Interval Decision Diagram (IDD) based firewall classifiers by Christiansen and Fleury [7] is developed for this purpose and is described in Chapter 3. The notation allows for the description of the path through the search structure taken when classifying a packet header. In doing so, it gives a quick way of estimating the amount of work done by the classifier for a single packet, as well as for determining the total amount of work performed by the classifier. Additionally, Chapter 3 describes the notation which will be used for the formulas which make use of the IDD based path description, and describes the ways in which some of the metrics described in Section 1.2 are measured.

Using this formal notation, along with algorithm excerpts and informal diagrams, the actual processes involved in redesigning the search tree are then outlined. General designs for all the restructuring techniques, along with the motivation for those techniques provide the bulk of Chapter 4. First, four techniques which actually affect the search structure - node promotion, range reorders, range combinations and enhanced cuts - are described in Sections 4.1 through 4.4, along with explanations of how they would improve the overall performance of the search structure.

Next, the idea of using a TCAM as a cache for rules which have been recently matched is explored. Since a TCAM can be used with any classifier (in fact, given a small enough ruleset, and a large enough TCAM, a TCAM can serve by itself as a classifier) the effects of the TCAM are first described in relation to all classifiers, static and dynamic. Then the specific advantages of using a TCAM with a dynamic classifier are outlined. This discussion is found in Section 4.5.

There are several other questions with regards to the performance of a dynamic packet classifier. In particular, it is important to know that a worst-case bound on the classifier's performance can be enforced. Techniques for dealing with this case are given below as Section 4.8. In addition, it is important to determine how often updates to the search structure should be performed (they do take a significant amount of resources themselves) and how quickly the classifier should forget the information it is remembering about the traffic it is seeing. If reorganized frequently, the classifier would probably benefit - but at the risk of occupying too many resources which would better be used by the classifier or other processes also running on the same hardware. Section 4.7 details the effects of varying the rate at which the classifier restructures the tree and at which it forgets the traffic it has seen.

After the overall design of the classifier has been explained, it is important to describe exactly how it was implemented. Many choices, such as the algorithm used for sorting and the use of static and dynamic memory affect the performance of the classifier. As a result, it is important to detail and justify the actual implementation of the dynamic classifier. Chapter 5 explains exactly how the static classifier (5.1) and the dynamic improvements to it (5.2) were actually implemented.

Chapter 6 deals with the design of the experiments used to test the effectiveness of the techniques described in this document. The selection and generation of packet traces is discussed, as is the selection and generation of rulesets. The process by which the various possible improvements to the classifier are tested is given in detail, along with explanations for the choices used in experimentation for the given parameters. The results of these experiments are also given in Chapter 6. These results are discussed in detail, using both informal explanations of the effects of the classifier's decisions, along with formal descriptions which make use of the modified IDD notation.

The overall conclusions of this research are then presented in Chapter 7.

Chapter 2

Current Classification Algorithms

As described in Section 1.2, packet classification is the problem of determining which of a large number of rules best matches a given packet header. A *rule* is simply a statement of certain properties that a packet header must show. For example, a rule may state that a header may come from any source address, may be heading to any address in a particular subnet, and must have a source and destination port of 80. In addition, a rule contains an action which is to be performed is a packet header does have all the properties defined in the rule. Common actions include dropping, accepting and forwarding to other devices. Rules are commonly grouped into *rulesets*.

Rules can be defined over multiple dimensions, where each dimension corresponds to a particular field in a packet header (such as the source address or the destination port). The term *field* is used not only to describe a component of a packet header, but also to describe the particular component of a rule used to determine if the packet header matches the rule on that component of the packet header.

The goal of any packet classifier is to find the rule which 'best' matches a given packet header. This is done by finding the rule which matches a packet header on every field defined in the rule. If multiple rules match, the classifier must choose the 'best' rule. Often this is accomplished by encoding a priority into rules. When given the choice between rules which all match a packet header, the classifier chooses the rule with the highest priority.

2.1 General Overview

The problem of packet classification has been around for as long as networks have existed. In that time, many different techniques for classifying packets have been proposed. Given below is an overview of several different classification techniques. The algorithms chosen below are illustrative of the various ways in which packets can be classified. Many variations of these techniques exist, and there are a few rarely-used algorithms which are not detailed below.

2.1.1 Linear Search

For a linear search, all the rules are stored in sequential order by priority. A packet is compared to each rule in turn until a rule is found which the packet matches.

Linear searches are very efficient in terms of storage - very little is needed beyond a single copy of each rule. In addition, updates to the rules can be performed relatively easily. There is no need to perform more than a local update around the modified, inserted, or deleted rule unless a change affecting that rule's position in the list is made.

Of course, as is well known, linear searches perform poorly when time is an issue. The worst case of a linear search is to have to search every rule in the rule set. On average, under the assumption that every rule is equally likely to match a packet header, and that every packet matches a rule, a linear search must search half of its rules to find a match. If no rule matches for a packet, we must do a full linear search to confirm this. For these reasons, this technique scales very poorly in the number of rules it has to classify over [18]. The number of dimensions needed affects this approach only in that there will probably be additional rules in the classifier if there are additional dimensions to classify over, and, depending on how rules are encoded, that each comparison of a packet against a rule may take longer than a comparison on fewer dimensions.

2.1.2 Trie Based Methods

Since packet classification can be seen as trying to find matching rules based upon the prefixes of the various fields on which we are trying to match, the concept of using trie-like structures comes immediately to mind.

A trie is a tree structure where we determine which branch (or branches) to search from a node based upon the prefix of what we are searching for (this description is adapted from [1]). A trie for an English dictionary may have nodes with up to 26 children - one child for each letter in the language. A search for the word "categorize" then, would follow an edge to the "c" node (a child of the root), then the edge to the "ca" node (the "a" child of the "c" node), and so on until the word is found in the trie or a mismatching leaf node is reached.

A trie generally only operates over a single dimension [18]. This leaves no means by which to classify packets over multiple dimensions, at least when using a simple trie structure. Several methods have been proposed to use tries in an attempt to solve problems in packet classification. These include Hierarchical Tries, Set-Pruning Tries and Grid-of-tries.

The simplest of these methods is the Hierarchical Trie [18]. A Hierarchical Trie simply constructs a trie on one particular dimension. Then for each node in that trie, another trie is built on another dimension, containing only those rules that satisfy the node of the original trie the second trie is anchored to. If a matching rule is found in the trie over the final dimension, the classifier knows that this rule properly matches on every dimension, even though the current trie it is searching is only on one dimension. A Hierarchical trie is shown in Figure 2.2, which shows the rule set described in Figure 2.1 split along its two dimensions. Thes ruleset is defined on two dimensions, using prefixes. It is important to note the branch leaving the node for the prefix 0* in the Hierarchical trie. This branch exists because the rules R4 and R3 are defined for prefixes of exactly 0*. That this must be done is one of the major drawbacks of Hierarchical Tries.

While conceptually simple, this algorithm performs poorly in terms of both time and space complexity [18]. The time for finding any particular match is determined by the number of dimensions and the depth of the trie constructed for each dimension. This could get quite large as more dimensions (as well as more rules) are added to the classification scheme. In addition, as a Hierarchical Trie only places rules in sub-tries if they exactly match the prefix held in the node in trie for the previous dimension, multiple tries in later dimensions may need to be searched as more than one node in an earlier dimension may match the packet. The best match in one trie is not guaranteed to lead to a matching rule, and tries under less-constrained nodes on that dimension must be searched to find a rule, or ensure that no rule matches. In terms of space, a trie can take up a lot of memory. Having several levels of multiple tries per level simply makes storage costs worse.

To improve on lookup time a structure called a Set-Pruning trie ([18] which in turn cites an unpublished report by P. Tsuchiya) can be used. This is very similar to a Hierarchical Trie. However, care is taken to ensure that only one branch can be followed at each level. While the Hierarchical Trie only placed rules in the lower dimension tries if the packet properly matched the prefix in the above tries, a Set-Pruning tree includes all rules that could match that prefix - including those which could match another more specific node further down in the trie. This way, the classifier only has to follow one branch in each trie. However, this also means that rules can be duplicated across tries of lower dimensions. This makes the storage space of the algorithm worse than that of Hierarchical tries. A trie built by this method over the ruleset in Figure 2.1 is given as Figure 2.3.

The set-pruning technique can be further improved. A technique known as the grid-of-tries, put forward by Srinivasan et al ([18], [24]), builds structures that are equivalent to the tries built by the original hierarchical approach. However, the grid-of-tries approach maintains additional pointers in the other tries for the same dimension. Information stored in these nodes allows the search to know if it can - and if it should - "jump" from one trie to another. This is conceptually as if we were building a set-pruning trie, but instead of maintaining each trie separately, we merge redundant sub-tries in all the tries on a particular dimension in order to save on storage space. In doing this, only one branch of any trie needs to be searched (eliminating the main deficiency of the Hierarchical method) and rules need not be stored in more than one place (eliminating the main deficiency of the Set-Pruning method). A set pruning tree can be seen as a form of Directed Acyclic Graph. For reasons of space, an example of a Grid-of-Tries is not included, a good example can be found as Figure 5 in [18].

The trie method may not be the best way to construct search trees for the problem of packet classification. Among other failings, it may be prudent to perform classification over dimensions together, not separating them into distinct structures. If the classifier was to look at the structure of

Rule	X	Y
R1	00*	00*
R2	0*	01*
R3	1*	0*
R4	00*	0*
R5	0*	1*
R6	*	1*

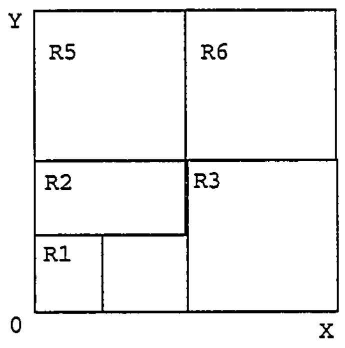

Figure 2.1: The ruleset used for the examples in Section 2.1.2.

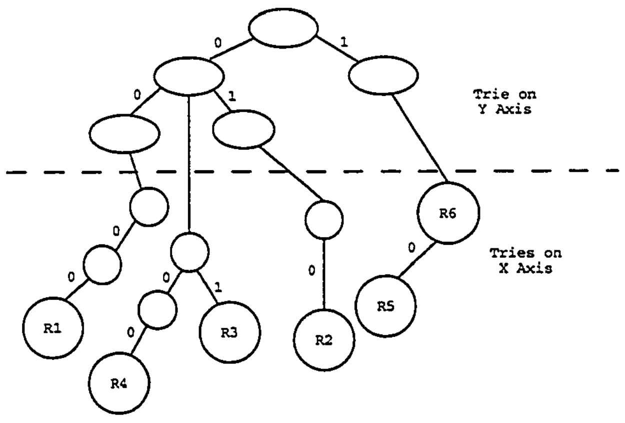

Figure 2.2: An example of a Hierarchical Trie on the ruleset given in Figure 2.1

the rule sets used in building a a search structure, it might be able to exploit this structure to build more efficient packet classifiers [18].

2.1.3 Geometric Approaches

As the search space of a packet classifier can be seen as a d-dimensional geometric space, the applicability of geometric search algorithms and data structures to the problem is readily apparent. In recent years, several geometric approaches have been proposed to help solve the problem of packet classification. While these approaches tend to work very well for 2 dimensional classification, many scale poorly, or worse, cannot be scaled beyond 2 dimensions at all.

In Cross-producting (described along with the aforementioned Grid-of-Tries in [24]) each dimension is divided into small ranges, based upon the ranges of the rules stored in the ruleset. These ranges are then combined into tuples. These tuples are used as references into a cross-product table which would dictate which rule is to be followed when a packet falls into those ranges on every dimension. This is very similar to the Tuple space search outlined below, however, Cross-producting is not limited to prefixes and can compose ranges on more general criteria. Unfortunately, as there

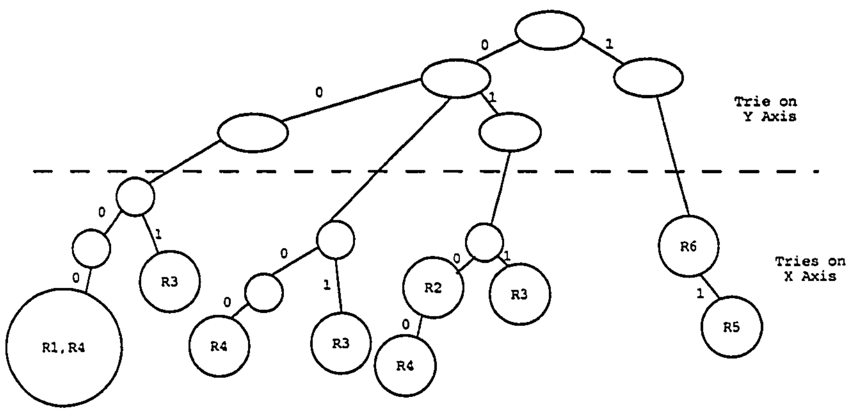

Figure 2.3: An example of a Set-Pruning Trie on the ruleset given in Figure 2.1

can be many ranges generated on each possible dimension, the total size of such a table quickly becomes too large to be practical for large rulesets and/or a large number of dimensions.

Area-based quadtrees, described by Buddhikot et al [5], allow for quick classification of any number of rules in two or fewer dimensions. Every node in a quadtree [12] represents a two-dimensional region of the search space, and has four children, representing the four equal-sized quadrants which the regions can be divided into. Starting with the full space, new nodes are created until each contains only one rule. Such a structure allows for the construction of a tree which is not overly large in terms of the number of rules stored and which is easy to search. However, quadtrees are limited to classification on only two dimensions. That it constructs new nodes based on very strict criteria may also result in larger quadtrees than would be constructed under more flexible criteria.

A segment tree is a balanced binary search tree containing the end points of a large collection S of line segments. The details of such a data structure are given in [12]. Each node in such a search tree represents a certain range in one dimension of the overall search space, each leaf corresponds to a single line segment and every non-leaf is the union of all the ranges of its children. FIS (Fat Inverted Segment) trees were proposed by Feldman and Muthukrishnan [14] to adapt segment trees for use in two-dimensional classifiers. These trees increase the number of children per node (hence 'fat') and follow paths from child to parent in searches (hence 'inverted'). A FIS-tree is created for the first dimension (with each rule's values on that dimension used as a line segment). Each node contains a table of associated rules, which can be searched using the value for the second dimension. A query for a certain point (the packet header value for that dimension) is carried out to find the proper leaf node. Nodes are then searched towards the root of the tree to find if any higher-priority rules exist. Such a technique makes finding nodes quick, and uses little memory. Again, this technique cannot be scaled beyond two dimensions, rendering it useless for more advanced uses of classifiers such as the provision of Quality of Service.

2.1.4 Recursive Flow Classification

For every possible packet header we know that there is either a rule which best matches that packet, or no rule that matches that packet. Therefore, it would seem a good idea to come up with a mapping for every possible packet header to the number of the particular rule that packer header matches.

Unfortunately, this would use a very large amount of memory. If t is the number of bits in the fields of a packet header being used for classification, then there are obviously 2^t different bit strings to be classified in this manner [18]. Even if each packet is classified over a single 32-bit value, the costs of storing this much information in any form of fast storage would be astronomical.

Therefore, it was proposed by Gupta and McKeown (in [16] and summarized in [18]), that this process be done in stages. The required fields of the packet header would be broken up into many independent pieces, and each piece would serve as an index into parallel memory. The results of these lookups could then be combined to provide indices into other memory, which in turn would provide additional indices until finally the index into the rule set itself is computed.

Preprocessing of the rules must be performed to help make two choices [16]. First, the number of iterations (intermediary index lookups) needed to produce the actual ID of the rule we need must be decided. Second, since there are many ways to split and recombine the packet information and the indices returned, a scheme for combining the results of each phase must be determined. Generally, chunks with close correlation are combined first. As many chunks are combined at each stage as is physically possible given memory constraints.

In terms of performance RFC can classify any packet very quickly. In fact the speed of classification is predetermined and never changes unless the rule set is processed again with different parameters. This provides predictable performance, and gives a reliable upper bound on the classifier's worst-case performance. The technique can be expanded to as many different fields in the packet header as the user would like.

However, RFC is costly in terms of memory use. It needs to store all the intermediate results in separate parallel memories. For larger and larger classification rule sets, this scheme becomes almost impossible to implement in a small amount of memory. Also, changes to the rule set may make it necessary to reprocess the entire structure for the algorithm. If rule sets change infrequently, this isn't a problem, however it could be prohibitive if frequent changes are made to the rules a classifier uses.

2.1.5 Use of Caching

Caching of important information is a common strategy in many algorithmic techniques. A significant amount of research has gone into the use of caching for use in classification, and currently, caching is seen by many as the only viable solution for classification in core routers [21]. There are two distinct forms of caching which can be used with classification. The first uses specialized hardware, the second, regular memory.

Ternary CAMs

Ternary CAMs (Ternary Content Addressable Memories, TCAMs) provide what appears, in almost all regards, to be the perfect solution to the problem of packet classification. A ternary CAM provides a means to find an address into memory based upon some particular value (which in this case could be a packet header) [18]. Ternary CAMS are so-named because they implement ternary logic. Each line is composed of trits which can take on the values 1, 0 and "Don't Care."

TCAMs provide several benefits. First, the use of a ternary CAM is trivial - even simpler than a linear search. All one has do do is provide the information to be matched to the TCAM, and the TCAM returns all list of all the rules that match [18]. These can be searched to find the one with the highest priority. Second, Ternary CAMs provide results very quickly. In fact, the results may be returned within 10ns[18]. A Ternary CAM compares the information it is given against every one of its elements in parallel - indicating all the matches on its output lines simultaneously. If many rules match, the classifier must use some metric to determine which rule to apply. Usually, the lowest output line which is active is used, with rules sorted according to priority in the TCAM.

But, as with all classification schemes, there are drawbacks to using Ternary CAMs. The most notable is the cost. Ternary CAMs require more transistors for every trit stored, significantly higher than the number of transistors needed in DRAM, and possibly even SRAM [18]. Also, due to how the hardware operates, the power requirements of a ternary CAM are significantly higher than those of more conventional storage. In addition, as noted in [26], Ternary CAMs may not currently be produced with line widths well suited to storing the rules to be matched.

Should Ternary CAM technology reach the point where it is feasible for large classification schemes, such technology may make other attempts to speed up packet classification obsolete. It is impossible to improve upon storing every rule once (linear storage space), and matching a packet in constant time.

Main Memory Caching

Other work has been performed in the area of caching mostly in standard memory, and not in specialized hardware such as TCAMs. Tuple space search, proposed by Suri et al in [23], decomposes the rules in a classifier's ruleset into several d-tuples (with d being the number of dimensions being classified over). Each of the d entries in the tuple contains the length of the prefix being used for classification. Since these entries have a fixed length (they are fields from strictly-defined packet headers) they can easily be stored in a hash table. Unfortunately, as with TCAM caching, while schemes like this provide very quick lookup of rules corresponding to any given packet header, the number of tuples which can be created is very large, and grows exponentially with the number of dimensions being classified. As well, the classifier is only able to classify based upon prefixes and not on patterns in general.

Despite the memory requirements of the Tuple space search method, which make it rather useless for large rulesets over many dimensions, improvements to it have been suggested. One of the most interesting, put forward by Wang et al [27], alters the Tuple space search method to perform lookahead caching. This method attempts to remove useless information from the hash table in order to speed up classification time and memory use. Memory use can be reduced in this case by choosing a 'default' action and removing all tuples from the hash table which resolve to that action. If no rule can be found for a header, then the default action is taken. This cannot completely control memory use, however, and still has the general deficiency of only being able to classify based upon prefixes. In addition, such a scheme would not work in cases where rules overlap, as removing a default rule from the hash table might allow packets to incorrectly match a rule with another action which had lower priority in the original ruleset. In order to combat this deficiency, rules would have to be split into numerous non-overlapping smaller rules in order to ensure that the removal of some rules does not affect the correctness of the classifier's work. This counter-acts the benefits in reduction of memory seen by removing all rules associated with one action.

In general, main memory caching techniques attempt to store all the rules in a ruleset into a hash table or other form of cache. Just like with TCAM caching, this is useless for large rulesets covering many dimensions. As a result, it should be considered whether caching alone is an appropriate response to the problem of packet classification, or whether it should be combined with other techniques.

2.2 HICuts in Detail

Hierarchical Intelligent Cuttings (HICuts), proposed by Gupta and McKeown in [17] and described in [18], is a form of heuristic-based packet classifier. Like any heuristic-based classifier, HICuts uses a set of heuristics to guide its construction of a search structure which can, hopefully, be searched quickly in order to classify packets at the fastest rate possible. In the case of HICuts, the search structure produced is a search tree. This search tree is described in greater detail below. HICuts' main goal is to produce a search tree which is well-balanced and minimizes the amount of memory used. That HICuts can create such a search tree is confirmed empirically in [17]. Some insight into how HICuts' structure can be proved to do what it claims is given in Chapter 3.

Since rules may overlap in the classifier, there must be some way of determining the precedence of rules in the case where a packet may match more than one rule in the ruleset. HICuts deals with this problem in a straightforward manner. Rather than being an actual set, the ruleset is considered to be an ordered list. The precedence of a rule is determined by its position in the ruleset. Rules which are found closer to the head of the list are considered to have a higher priority than those which come later. As the rulesets may be changed based upon the node in the search tree, it is possible that, at some point, some rules in a ruleset may be identical to each other, or a higher priority rule may completely cover a lower priority rule. In this case, rules with a lower precedence are eliminated completely from the ruleset. This ensures that the termination condition for building the search tree is guaranteed to occur at some point. During the construction of the tree, the region of the search space represented by a node may become so small that rules must be completely covered by rules with higher priority. If these rules are not eliminated, it might be that a node would never have fewer than the maximum number of nodes allowed in a leaf node.

Every node of the HICuts search tree contains a certain ruleset. In the case of the root of the tree, this is the entire ruleset. In nodes further down in the search tree, this ruleset contains fewer rules. Rules are pared out by HICuts at every node when the children of that node are constructed. Every node considers its ruleset based upon four heuristics (described in detail below). Using these heuristics, the node determines two things: the dimension which should be cut by the node and the number of cuts that the node should make. Every node in a HICuts tree divides its ruleset among its children based upon cutting the ruleset on exactly one dimension. This dimension has a certain range (which may be smaller than its original range due to cuts higher in the tree) which is divided *equally* among the children of the node. The provision for equality ensures that the proper child node to search can be found through mathematics. A number of child nodes equal to the number of cuts made plus one are created. The ruleset is then divided up among these children based upon their new ranges (all being subranges of the range of the cut dimension). It is possible that some rules will be inserted into multiple children, as the range of the rule on the dimension being cut may fall into two or more of the ranges given to the new children. This is perfectly acceptable. At this point, the rulesets of the children are inspected to see if there are any rules which are completely covered by a higher-priority rule. If this is the case, then those rules are eliminated from the child's ruleset. The node-splitting work is then performed, recursively, on each of the node's children as long as they do not qualify to be leaves.

Leaves in the search tree are defined as those nodes which have fewer than a pre-set number of rules in them. Any node which has more than this number *must* be further cut until this condition is satisfied. It is easy to see where this may lead to problems, given that HICuts divides the range it is cut on among its children evenly. If all the rules remaining in the node are clustered in one region of the search space, it might take several cuts, and several levels in the search tree to finally pare the ruleset in each node down to below the number of rules permitted in a leaf. The rules stored in a leaf many have any action associated with them. As a result, when a search reaches a leaf, it must search the rules provided in order to find the rule which matches the packet header, and discover the action associated with that rule. HICuts, in order to allow for rule precedence, searches the rule list sequentially, from the first rule to the last. The action associated with the first matching rule is returned as the action for this packet. If no rule in the leaf matches, there is no matching rule in the entire ruleset.

Searches through the non-leaf parts of the tree are straightforward. All searches start at the root. In every node visited, the dimension cut is determined, and the value of the packet header on that dimension is retrieved. At that point, the child node whose range contains that value is selected from the node's list of children. The search continues into that node. If the new node is a leaf, the rules are searched. If the new node is not a leaf, then the process is repeated recursively.

As mentioned above, four heuristics are used to guide HICuts in making the proper selection at each level to produce a balanced search tree which uses as little memory as possible. The first two heuristics work in conjunction with one another. The first attempts to determine the number of cuts which should be made at a node. More cuts might result in a shorter search tree, decreasing search depth. However, more children might also result in the search tree using more memory - especially if there are a number of rules in the ruleset which would make it into multiple children upon being cut. This heuristic tries to find a good number of cuts which will produce several children while dividing the ruleset up evenly and cleanly among them.

The second heuristic determines which dimension should be cut. Here, a good choice is a dimension on which the rules are quite evenly distributed. Since HICuts divides the dimension into equal pieces for the children of a node, a uniform distribution of rules would produce several children with rulesets of roughly the same size. A dimension on which the rules are currently clumped into certain regions of the search space is usually a poor choice for being cut, unless such a cut satisfies the requirements of the other heuristics guiding the construction of the search tree.

The third heuristic considers the possibility of creating child nodes which are common to multiple regions of the search tree. It is possible that cuts in different areas of the search tree may produce children which are identical in terms of the rules which they contain. If this is so, then only one child need be created, and every node which considers that node to be its child can point to that one node, rather than creating a large number of identical nodes. This heuristic looks at the current node and tries to determine if a certain cut could produce a child identical to another child in the tree. If so, that cut should be considered with more weight than it otherwise might be given. Making such a cut would help reduce the amount of memory used by the search structure and improve the usefulness of memory caches. As a result of applying this heuristic, a structure similar to a Directed Acyclic Graph (DAG) may be produced.

The fourth heuristic, like the third, tries to reduce the amount of memory used by the search structure. This heuristic tries to find cuts which would allow rules to be removed from rulesets. Since any rule completely covered by a rule with higher priority will be removed from a ruleset, this heuristic tries to find cuts on dimensions which will allow the maximum number of low-priority rules to be covered and rendered removable. The more rules are removed at each level, the more likely we are to reach a leaf node earlier on. Further, there is less information stored at each level of the search tree, as the rulesets stored at each level will be smaller.

The above four heuristics are expensive to compute, and take a significant amount of time for a large ruleset. However, HICuts only builds the search structure once, as a preprocessing step, so the amount of time needed to build the structure has no bearing on the search time of the algorithm itself. It should be noted, though, that insertions and deletions from the HICuts ruleset might require running the entire algorithm again in order to keep the rules balanced. If this is not done, then a large number of rules inserted in a small region might cause a subtree to grow much larger than it should. This would unbalance the entire tree, and increase the worst-case classification time.

2.2.1 HyperCuts

Singh et al [21] proposed an improvement to HICuts shortly after HICuts itself was first proposed. Their algorithm, named HyperCuts, creates the same overall decision tree structure as HICuts. However, this structure allows more freedom in the choice of dimension to be cut at every stage of search. In particular, it allows for the possibility of multiple dimensions to be cut in every node. This is done through the use of hypercubes created and stored in each node which allow the children of the node to be quickly found, even over multiple ranges. New heuristics ensure that the storage costs of these nodes is kept as small as possible while providing for efficient search times and low update costs. In the case where each node is limited to a one-dimensional hypercube, HyperCuts behaves identically to HICuts. When the hypercube is allowed to span multiple dimensions, however, HyperCuts can perform significantly better than HICuts, while using a lower amount of memory.

It is interesting to note that the authors designed HyperCuts to eliminate the cost of using CAMs in routers. As HICuts and other techniques still tend to use a lot of memory, for many routers CAMs are used to store the entire ruleset and algorithmic approaches are shunned. It is interesting that the authors of HyperCuts consider completely removing the CAM as one of their goals. As discussed above, having a CAM store some rules to save both time and space for searching would seem an obvious solution to many problems. However, while admirably attempting to reduce the time and space needs of classifiers, the authors miss a valuable opportunity when they assume that the goal of such work should be the removal of CAMs from routers. CAMs still work faster than algorithmic techniques, and should not be cast aside so readily.

HyperCuts could be seen as a better choice for the basis of this thesis than HICuts. However, HICuts was chosen mainly because of its relative simplicity. The purpose of this work was to prove that using information learned during classification to improve the search structures used by classifiers could lead to an overall improvement in the speed of classification, and to demonstrate that more than the worst case should be considered when evaluating the performance of classifiers. HICuts is complicated enough to allow many improvements based on learning, while simple enough to implement in a short period of time. Using HyperCuts instead of HICuts would simply have increased the complexity of the original algorithm, while not affording any benefit in helping show the gains that can be made when classifiers can learn from the traffic they see.

Chapter 3

Notation and Metrics

3.1 IDD Notation

A HICuts tree is a form of Interval Decision Diagram (IDD) [25]. While searching a node in the tree, one of its children is chosen based upon intervals along a certain dimension. IDD trees can be created for multiple dimensions, but, like in HICuts, the children of each node must be differentiated only by ranges in a single dimension. Whichever interval matches the value being searched for dictates the child node on which the search continues.

An IDD tree continues to split its nodes into children until a node contains only one associated action. This cannot be done in HICuts due to memory constraints. Several actions might be associated with a leaf in a HICuts tree, and the classifier must search the rules contained in that leaf to figure out which action to take. This is a minor concern, however. The terminating condition on the HICuts tree could be changed to ensure that there is only one action per leaf, at the expense of using up significantly more memory.

Christiansen and Fleury [7] give a notation to describe how a packet might be filtered by a firewall which is based upon an Interval Decision Diagram. This notation will be modified slightly here to allows the description of the path a packet will take through a HICuts search tree to find a matching rule. The use of the notation here is slightly different from that in [7], as there may be multiple paths to the same rule in a HICuts tree. This is not possible in the original IDD notation described in [7]. This new notation gives a very easy way to describe the changes made to a HICuts search tree by the dynamic classifier described in Chapter 4. It also gives a formal way to calculate the amount of work performed by the classifier both for each separate packet classified, and for the classification of every packet seen thus far by the classifier. This is described in Section 3.2.

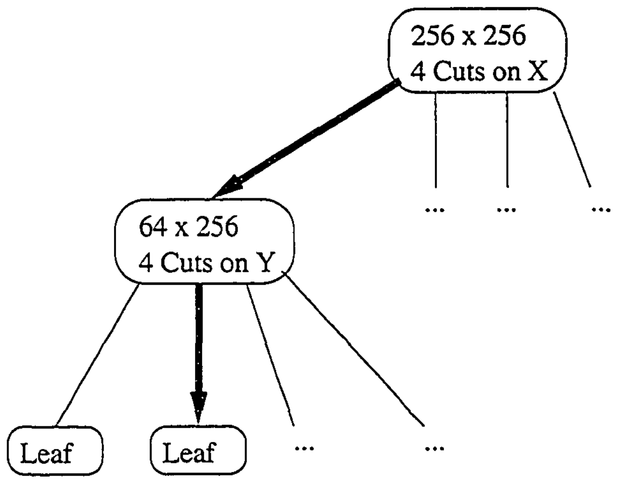

Figure 3.1: A sample path through a small HICuts tree.

Given H, a set of possible headers, and Π, a set of all possible actions the classifier can return, the path p that a single packet will take through the HICuts tree can be defined as:

$$p = (\eta, \pi), \text{ with } \eta \text{ a CNF over ranges in } H \text{ and } \pi \in \Pi$$

Ranges in H refers to a range given over a particular field in the packet header. Such a range acts as a literal in the CNF formula, and will be of the form *lower bound* $\leq$ *dimension* $<$ *upper bound.* Ranges are never combined within clauses, even if contiguous.

In particular, η is a conjunction of several tests over fields in the IP header. Since this notation describes the path of a search through a rooted tree, the operator $\wedge$ is constrained to being left associative, rather than allowing it to be fully associative as it generally is in logic.

Unfortunately, this basic notation still does not quite suffice to fully describe the classifier's search. In particular, because work may be done at a given node to determine which child node to search in, there must be some notation for describing this work. To achieve this, η is altered to be a formula in conjunctive normal form. Each clause is indicative of the considerations made internal to a node, and the overall conjunction charts the course of the search from node to node.

It is important to recognize that this notation puts some semantic strain on the 'or' relation. In particular, 'or' implies that the first boolean tests failed while the last one given in the expression succeeded. While this is not how an 'or' would be used in pure logic, it does match the standard practice of short-circuiting logical expressions or lazy evaluation used in most languages. As an example of the use of this notation, consider the path given as a bold line in Figure 3.1. As this is a standard HICuts tree, the cuts are of equal size, so ranges are not given in the figure. In this case $\eta = (((X < 64)) \wedge ((Y < 64) \vee (64 \leq Y < 128)))$. π will take the value of the action associated with the proper rule in the leaf node. If, in this case, π is *accept*, the path $p = ((((X < 64)) \wedge ((Y < 64) \vee (64 \leq Y < 128))), \textit{accept})$.

3.2 Mathematical Notation

Given the description of the HICuts algorithm provided in Section 2.2, along with the IDD notation described in 3.1 it is now possible to present a number of mathematical functions and operators which will allow description of the amount of work that is needed to classify a single packet. It is also now possible to introduce functions to aid in computing the net change in the amount of work to classify any packet after the HICuts tree has been modified by the techniques outlined in Chapter 4.

The length of a path $p = (\eta, \pi)$, $L(p)$ is the total size of all the clauses in η. It is the count of the number of literals (counting duplicates, if any) in the conjunctive normal form equation η. $L(p)$ gives a rough estimate of how much work the classifier has to perform in order to classify any packet with takes the path p. Counting the size of each of the clauses gives us the number of range checks which had to be performed at any level of the search tree. The search depth, represented in the IDD notation as the number of clauses in the conjunctive normal form formula, need not be considered when calculating the work performed by the classifier as range checks comprise all the work performed in a node while searching.

However, the depth of the search is an important consideration for matters of time and, especially, memory efficiency. Thus, it is important to have a notation for the depth of search. $D(p)$ gives one less than the number of clauses in the η associated with p. This gives an exact count of the depth of search for any given packet being classified. $D(p)$ is one less than the number of clauses as, when calculating depth in a search tree, the root node is given a depth of 0.

A value to express the total amount of work performed by the classifier over all the classifications it performs is also needed. This would, of course, make use of $L(p)$. The value L will be defined as follows: given n packets as input to the classifier, where each packet i follows path p_i through the search structure, $L = \sum_{i=1}^{n} L(p_i)$. The entire focus of this work, then, could be restated as a minimization problem. This work is dedicated to finding a way to minimize the value of L over a very large number of packets classified, while ensuring an upper bound on worst case performance cannot be worse than some preset bound. All of the techniques described in Chapter 4 are specifically designed to produce a net reduction in the value of L over what would be attained by a static classifier.

In order to calculate the net change in the value of L from any changes in the search structure, it must be possible to describe the changes which take place internally in a node when the search structure is altered. To do this, additional notation for finding the children of a node, as well as for extracting statistical information from a node, is given below.

Any node N, which is not a leaf node, must have more than one child. The number of children of node N is given as $|N|$. The children are denoted $C_i, 0 \leq i < |N|$ and numbered in the order that they would be found in the node's child list *at the moment that the calculation is performed.* Since children may be moved around in a node's child list, or completely removed from it, the order of the children may change over time. If the children are moved, their positions, and their corresponding denotations C_i are changed. This means that the value i itself, in this context, can be used to give the position of any child in its parent's child list.

Occasionally, notation other than i will have to be used in order to describe the location of a child node in its parent's range list. It might be that i would be ambiguous in that context, or that another notation had been chosen which might lead to some confusion when combined with the i notation for position. In this case the function $pos(C_x)$ would return the position of the child C_x in the range list of its parent. In the standard context of nodes and children the equation $pos(i) = i$ holds.

As children may be moved by restructuring the search tree, it is important to be able to describe the positions which children held previously in the node's child list. It would be impossible to calculate the net expected change in L for any search if values of C_i were not known. The formula $old_j(C_i)$ gives the location of the child C_i in this node's child list exactly j restructurings ago. If C_i did not exist in this node at that time, $old_j(C_i) = 0$. This ensures that any calculations are able to account for the fact that this child would have added nothing to any searches as it would not have existed in this range list at that time. It is worth noting that $old_0(C_i) = C_i$. For most calculations, the only interesting value is the position of the child exactly one restructuring ago. For that reason, the shorthand $old(C_i) = old_1(C_i)$ will be used to improve the readability of formulas using the *old* function.

Finally, a way to extract the number of times a rule or node has been seen during classification is needed. The exact details on how these are recorded are given in Sections 4.6 and 5.2. For any node N, or any child node C_i, the number of hits currently recorded in the node can be retrieved by the use of the function $hits(N)$. The total number of hits for a node is often needed in order to calculate the relative frequency of hits between children. The value N_h holds this count. N_h is formally defined as the value obtained from $\sum_{i=0}^{|N|-1} hits(C_i)$, where the C_i are the children of the node N. In the actual implementation, in order to simplify matters, each node holds its own value of N_h. This means that the value need not be computed every time it is needed, though more space is used to store this data as HICuts tends to have a branching factor higher than 2.

3.3 Metrics

There are several different measurements which must be taken to ensure that the dynamic enhancements are, in fact, improving the performance of the static classifier, that they do not use an excessive amount of memory, and that they do not violate the classifier's worst-case bound.

Calculations of the performance of the classifier can easily be made using the IDD notation given above. The IDD notation describes the path of a single packet through the search structure, and gives a means for computing L, the total amount of work required to classify any number of packets. This value can be used to give the total amount of work done for classifying a certain number of packets. Given the same packets in a variety of different testing conditions, which conditions speed up classification the most and if classification is faster using the dynamic enhancements can be determined. In order to make the numbers more readable, the value L is usually divided by the number of packets classified to give the average amount of work performed per packet.

The value $L(p)$ is equally useful, as it tells us the total amount of work required to classify any single packet p. This value is important as it is the only way of knowing whether or not the worst case bound is being violated. As will be shown in Section 4.8, the value of $L(p)$ can be computed for all paths in the search tree generated by the dynamic classifier. This information can be used to insure that the changes which have been made will not make the classifier perform worse than the worst-case bound on any search. For the purposes of measuring worst-case performance, the maximum value of $L(p)$ possible in the search tree, as well as the maximum value of $L(p)$ actually searched in the tree for every trace tested are given. These may differ if some regions of the search space are searched never or rarely.

In order to test the memory requirements of the classifier, counts are kept of the size of the search tree, the number of nodes it contains, and the size of each node (in terms or children and rules). In the case where a TCAM is used to aid in searching, the size of the TCAM is not counted in the memory size of the program itself. The size of the search structure is independent of the size of the TCAM and the TCAM size is, itself, invariable. The memory size is reported in total number of nodes in the search tree, and the total number of rules contained in them. As these counts may change as the dynamic classifier is running, the average and maximum values, as well as the standard deviation over all the values will be given for all tests performed on the dynamic classifier.

In addition to the main three metrics given above, a few other metrics are used to keep track of the performance of the search tree.

First, the number of times each possible change to the search tree is performed is recorded. This makes it possible to see what changes are being made as the search tree is modified through use. These measurements are not overly useful for computing the final performance of the search tree. However they are useful to see which improvements are most affecting the speed of search, as well as locate instances where improvements might be working against each other. It is possible that some of the improvements (most notably, node promotions and range combinations) could work on a given iteration of the update process to undo the work performed by the other improvement on a previous iteration. Finding these conflicts and fixing them was a necessity in the early stages of this work.

In addition, whenever a TCAM is used, it is important to keep track both of the number of times that a match was found in the TCAM (the TCAM will always be searched, so there is no need to count that separately) compared to both the number of misses in the TCAM, and the total number of searches. Because of how the TCAM is used in this case, the total number of searches is equal to the number of hits in the TCAM plus the number of misses. A large TCAM may not make use of some of its lines in all cases. It is important to know when this happens so that the proper size of TCAM to use for classification can be predicted.

Chapter 4

The Dynamic Classifier - Design

Many different techniques were used to redesign the HICuts search tree. Several of these make long-term structural changes to the search structure itself. Range reorders (Section 4.1), node promotions (4.2), range combinations (4.3), and enhanced cuts (4.4) were all specifically designed to make use of long-term traffic statistics to move frequently hit regions of the search space closer to the root, and further to the left in the search tree. This reduces the value of $L(\eta)$ for packets which are found in those regions. In addition, these same techniques move infrequently hit regions away from the root, and further to the right in the search tree, increasing the value of $L(\eta)$ for packets in those regions.

Short-term changes in network traffic are dealt with via a Ternary CAM. Using this device allows for the implementation of a form of cache or transposition table which stores recently matched rules for quick lookups. There are several issues involved in storing rules in a TCAM. They are detailed in Section 4.5. It is important to realize that rules can be quite complicated, and representing them via ternary logic might result in a single rule taking up multiple entries in a TCAM.

Having the dynamic classifier remember and forget traffic was essential to the basic form of learning which was used in the enhancements. Because of the dynamic nature of the search structure, forgetting what has been seen before becomes very difficult to do. Section 4.6 outlines the methods used for remembering and forgetting in the dynamic classifier. The forgetting method used, along with the fact that major structural changes cannot continuously be made to our search structure implies that we must have some technique for determining when the search structure should be updated. The techniques for controlling the update frequency of the search structure are outlined in section 4.7.

It is important to ensure that a worst-case bound is strictly adhered to by the dynamic classifier. The techniques which prevent the classifier from exceeding its worst-case bound are given in Section 4.8.

Throughout the rest of the chapter, the term *major restructuring* is used. A major restructuring is simply the stage at which the search tree is evaluated to see if any changes are warranted. If any are, then these changes are performed on the tree to obtain a new tree. The rate of occurrence of these restructurings are described in Section 4.7. All but one of the restructuring techniques described below take place only during major restructurings. This is due to the costs involved in determining if changes need to be made, and in actually making changes to the search structure. It is possible that additional modifications to the search structure could be performed quickly, independent of major restructurings, (just as there are many additional restructurings which could be performed during major restructurings) however these are not considered in this document.

4.1 Range Reorders

HICuts divides the children of a node up equally along a single dimension. The provision for equally-sized divisions was likely to allow one to find the proper child of a node in a reasonable amount of time. However, when one considers that such a technique would require calculating offsets into ranges, it becomes obvious that using a list of children is no more costly unless the list of children for a particular node becomes lengthy. As a result, there is no reason to keep the children of a node cut on ranges of equal size. If linear search is used, there is no reason to keep the children of a node in any particular order.

This fact can be used to help redesign the search structure used for packet classification. Given a node N with $c = |N|$ children, and given that the classifier will search the list of children for the one which matches the current packet, it makes sense to move those children which have been seen more frequently, and which will continue to be seen frequently, closer to the head of the list so that they can be matched quickly. The result of such a move would be that frequently matched regions would have a lower value of $L(\eta)$ than they would otherwise. An unfortunate consequence would be that infrequently matched regions would have a higher value of $L(\eta)$. However, as it is expected that the rules at the head of the list will be matched more frequently than the rules near the tail, the overall expected value of L would be reduced by such a move.

Range reordering is considered for most nodes every time that a major restructuring is performed on the search structure. For every node considered for reordering, a threshold for restructuring, t, is computed. For an explanation of how a node might be excluded from consideration, and how t is calculated, see Section 4.8.

A node is a candidate for range reordering if the number of hits on the node's most frequently visited child is greater than t times the number of hits on its least visited child. Range reordering is only performed should $\max_i hits(C_i) > t \min_i hits(C_i)$, where $1 \leq i \leq |N|$. If this formula doesn't hold for a given node, range reordering is not performed on that node.

Range reordering itself is the simple act of sorting the children of a node by the value of $hits(C_i)$. Those nodes which are more frequently hit are moved forwards in the list, and those which are infrequently hit are moved back. Sorting can be done by any method, however given the small size of the lists, methods which avoid recursive overhead are preferable.

Assuming that no other changes have been made to the search structure and that the classifier

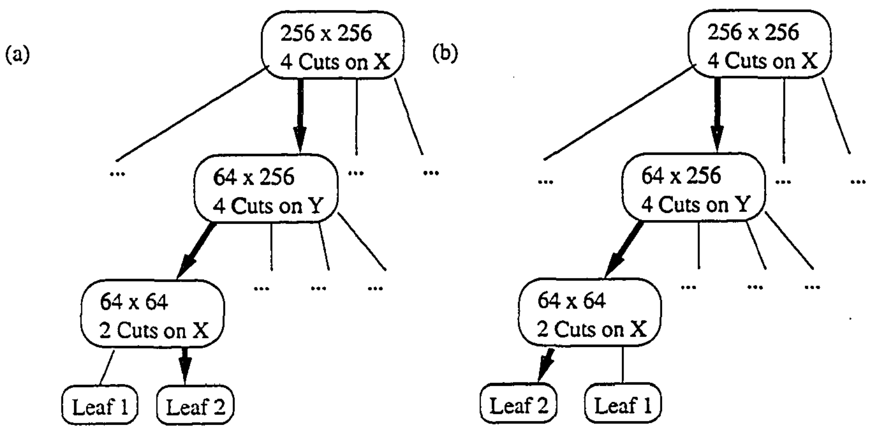

Figure 4.1: A HICuts tree (a) before and (b) after range reordering.

will see essentially the same patterns in traffic it has seen so far, we can calculate the expected net change in the value of $L(\eta)$ as $\sum_{i=1}^{c} \frac{hits(C_i)}{N_h}(i - old(i))$ for all searches through the node which has undergone range reordering. As long as $i - old(i)$ is negative for those values which are frequently hit (ie. have a large $hits(C_i)$), there will be a negative net change in the expected value of $L(\eta)$.

The process of range reordering is displayed in Figure 4.1. In this example the lower node resulting from a cut on X has its children rearranged from its original state to push its second child to the head of the list. In this case the description η of the path a packet would take through the HICuts tree changes from:

$$\eta = (((X < 64) \vee (64 \leq X < 128)) \wedge ((Y < 64)) \wedge ((64 \leq X < 96) \vee (96 \leq X < 128)) \wedge \ldots).$$

to

$$\eta' = (((X < 64) \vee (64 \leq X < 128)) \wedge ((Y < 64)) \wedge ((96 \leq X < 128)) \wedge \ldots).$$

In the example of Figure 4.1, $L(p) - L(p') = 1$. Every search which takes this path contributes 1 less to the overall sum than it would have otherwise.

4.2 Node Promotion

Range reorders dealt with one beneficial result of storing the children of a node in lists. However it doesn't touch on the fact that satisfying the requirement that all children of a node be cut to cover an equal portion of the dimension being cut is no longer necessary. Further improvements in the search structure can be made by allowing certain children of nodes to cover smaller ranges than others. The first of three mechanisms which allow this is node promotion.

HICuts cuts a node on a single dimension. However it often makes cuts on one particular dimension at many levels in the search tree. Like an IDD, a HICuts tree may slice a dimension at one level and then, further down in the tree, determine that the dimension needs to be cut again to satisfy the terminating conditions of the algorithm. This may lead to a situation where, deep in the tree, frequently matched nodes exist which could be pushed higher in the tree to reduce the amount of work required to search for them. Node promotion is the tool which allows the determination of which nodes should be moved up in a tree, to where they should be moved, and what should be done to recreate the information which was maintained by the nodes between the node's original position and its new position.

As with range reorders, node promotions are only considered when the search structure undergoes a major restructuring. Any node which has not been flagged as 'frozen' (see 4.8), and whose parent is cut on the same dimension as a non-parent ancestor, is a candidate for node promotion. As with range reordering, a threshold value t is calculated for each node that the restructuring algorithm considers for promotion.

If a node is a candidate for node promotion, its parent is checked via very similar criteria to range reordering. Requiring the check on the parent excludes the root from promotion. If a child of the parent node exists such that $\max_i hits(C_i) > t \min_i hits(C_i)$, with $1 \leq i \leq |N|$, then that child will be promoted to become the child of its nearest ancestor (other than its parent) cut on the same dimensions as its parent. If multiple children satisfying this formula exist, only one is promoted at a time. Promoting too many nodes at once could create very large child lists higher in the tree. The larger lists that node promotion creates are reduced in size by range combinations (Section 4.3). Range combination can only fix them at the next major restructuring. The interim period might be quite long, resulting in poor performance for an extended period of time, if multiple children are promoted.

Node promotion itself is quite simple. The node in question is simply removed from its parent. The parent itself does not have its range updated as no node matching the node's range will ever reach the parent. At the node designated to be the new parent, the new node is inserted. This means splitting some range in the new parent (as one range would have to contain the range for the promoted node). The result is two (if the promoted range falls to one side of the split range) or three (if the promoted range falls in the middle of the split range) new ranges in the child list. These new children are placed in the list where the child corresponding to the split range was situated, with the promoted child placed ahead of the other child (or children). HICuts is then performed on the promoted node to ensure that it can be properly searched. HICuts may also be performed on the one or two newly created nodes, however it is also safe to leave them pointing to the subtree to which they used to point. This reduces the amount of work performed during restructuring and reduces the amount of redundancy in the search tree.

The process of node promotion is displayed in Figure 4.2. In this example the lower node resulting from a cut on X in the original structure is promoted up to the upper node cut on X to produce a new structure. The exact configuration of nodes under the new cut is dependent on the ruleset and the parameters used by the HICuts algorithm. In this case the path a packet would take through the HICuts tree changes from:

$$\eta = (((X < 64) \vee (64 \leq X < 128)) \wedge ((Y < 64)) \wedge ((64 \leq X < 80) \vee (80 \leq X < 96)) \wedge \ldots).$$

to

$$\eta' = (((X < 64) \vee (64 \leq X < 80) \vee (80 \leq X < 96)) \wedge \ldots).$$

For all searches in the un-promoted part of the tree, the value of $L(\eta)$ remains the same. For all searches in the part of the tree where the node was promoted, the value of $L(\eta)$ will, of course, be affected. For any searches outside of the promoted node's range $L(\eta)$ will either be the same or be increased by 1 or 2 if the search must travel past the new ranges in the parent of the promoted node. For the promoted node itself, however, the value of $L(\eta)$ will drop. Since a much smaller area is now being covered, the size of the HICuts subtree is significantly less than it otherwise would be. Empirical results (see Section 6) show that the savings here tend to be around 2 to 3 on every $L(\eta)$ which searches into the promoted node.

If the expected net savings α for all searches through the promoted node is known, and C_p is the promoted node, the expected net change in the value for all $L(\eta)$ which search through that node's parent is

$$\alpha \frac{hits(C_p)}{h} pos(C_p) + \sum_{i=1, i \neq p}^{|N|} \frac{hits(C_i)}{N_h}(i - old(i))$$

. Given that $i - old(i)$ can only be 0 or positive in this function, the value of α must be negative (and C_p must be relatively frequently hit in comparison to its siblings) to ensure that the overall net change in $L(\eta)$ is negative. Range reordering will help ensure that only infrequently matched ranges have $old(i) > old_0(i)$. As well, range reordering will push the promoted node forward in the list, reducing the value of α.

Because a node promotion can pull a child out of a node completely, while range reorders only manipulate the children of a node, node promotions should always be performed before range reorders. A node which has experienced the promotion of a child may no longer need to be reordered as the child which exceeded the threshold has now been removed. In addition, often after performing a node promotion the ranges in the node's new parent may need reordering to accommodate the fact that the promoted node is probably more likely to be visited than many of its siblings. As node promotions consider only hits and not location in the list, there would be no benefit to performing range reorders before node promotions and nothing is lost by choosing to perform node promotions first.

In the example given in Figure 4.2, one more range check must be performed before the path to follow from the root is found so $L(p) = L(p')$. This may seem to be no improvement. However, since this node is frequently matched, it will either move ahead in the list of ranges during range reorders, or will see some of its siblings eliminated by range combination. After range reordering or range combination, the benefits of node promotions can be seen.

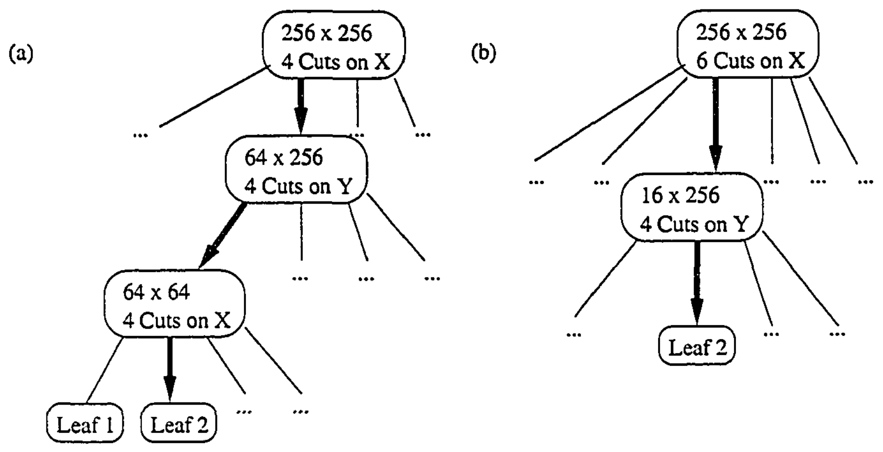

Figure 4.2: A HICuts tree (a) before and (b) after node promotion.

4.3 Range Combination

Like node promotions, range combinations make use of the fact that the children of a node in a HICuts tree do not have to all cover an equal range in the dimension which is to be cut. However, while node promotions move frequently hit nodes up in the search tree, range combinations move infrequently hit nodes further down in the search tree. Node promotion tends to increase the number of children in certain nodes, thus increasing the value of $L(\eta)$ for every search which passes through those nodes. Range combination works to reduce the number of children at a node by combining infrequently-matched nodes together into one node over a larger range.

Range combinations are only performed when the search structure undergoes a major restructuring. Non-frozen nodes with at least four children are candidates for range combination. We limit range combination to large nodes as it increases the work done while searching beneath the node. If there are not large enough savings at the level of the node itself, range combinations are not beneficial. A threshold value of t is calculated for every node which is considered for range combinations.

For any node which is a candidate for range combinations, the number of hits on every child of that node is checked. If there exist more than two and less than $|N| - 1$ children which have fewer than $\max_i(C_i)/t$ hits, then range combination is performed. If there are more than $|N| - 2$ children matching this criteria, range combination is not performed as it would be preferable for the few frequently hit children to be promoted up in the search tree than for all the other children in the node to be combined into one large node. To encourage this behavior, range combinations are performed before node promotions during any major restructuring.

The process of range combination is displayed in Figure 4.3. In this example two children of the root node are combined to produce a new structure. The exact configuration of nodes created under the newly combined range is dependent on the ruleset and the parameters used by the HICuts algorithm. In this case the path a packet would take through the HICuts tree changes from:

$\eta = (((X < 64) \vee (64 \leq X < 128)) \wedge ((Y < 64)) \wedge ((64 \leq X < 80) \vee (80 \leq X < 96)) \wedge \ldots).$

to

$\eta' = (((X < 64) \vee (192 \leq X < 256) \vee (64 \leq X < 192)) \wedge ((Y < 64)) \wedge ((64 \leq X < 80) \vee (80 \leq X < 96)) \ldots).$

due to the nodes representing the ranges $(64 \leq X < 128)$ and $(128 \leq X < 192)$ being combined together in the root node.

Range combination works by taking all the children which satisfy the criteria given above and placing them in one large child. A new node is created to cover all the ranges of the combined nodes (as one single range, meaning that it may also covers ranges not covered by the combined nodes) and all the rules which make up those children are combined into one large ruleset. The HICuts algorithm is then performed on this new node to create a valid search structure underneath it. The new node is permanently anchored at the end of the child list. Since this node has a range which covers more of the search region than its combined rules, it is imperative that it not move ahead of any other nodes in the child list. In addition, it must be part of any further range combinations in that node. There cannot be two large nodes possibly covering overlapping spaces in the child list.

Range combinations can be seen as the opposite of node promotions. While node promotions increase the number of children of a node while moving nodes up in the search structure, range combinations reduce the number of children of a node while moving nodes down in a newly-created search structure. Given β as the expected change in the amount of work needed to search the new subtree, and c as the newly created child node, for any search which goes through the newly-merged node, the expected change in $L(\eta)$ can be calculated as the sum of the changes in the non-merged nodes $\sum_{i=1,i\neq c}^{|N|} \frac{hits(C_i)}{N_h}(i - old(i))$ and the changes in the merged nodes $\sum_i(\frac{hits(C_i)}{h}(pos(c) - old(i) + \beta))$. β will generally be positive, and the non-merged nodes seldom experience any alteration in the length of paths through themselves as they would have been placed at the front of the list by range reordering. As long as there are savings in searching for the merged nodes, a drop in the value of $L(\eta)$ can be expected.

In general, the costs of searches through the combined node should be as much as before, or even slightly worse. Range combinations are performed to reduce the number of children of nodes in order to compensate for the additional children created by node promotion. It is expected (and verified empirically) that the gains from reducing the search time within a particular node afforded by range combinations outweigh the losses from having slightly more work to do while searching the newly merged nodes.

Note that this makes the assumption that the newly combined ranges would split on the same criteria as the smaller range had. This is obviously not always the case in practice. This 'improvement' in fact increased the length of a packet's journey through the search structure - $L(p) - L(p') = -1$.

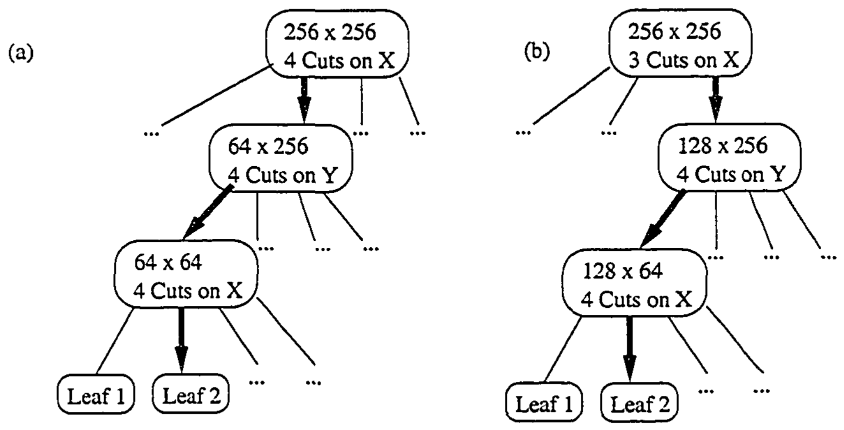

Figure 4.3: A HICuts tree (a) before and (b) after range combination.

Notice, however, that, for all searches on the range $(192 \leq X < 256)$, $L(p) - L(p') = 2$. Since we expect these searches to be more frequent, the gain here more than makes up for the loss on searches in the range $(64 \leq X < 192)$.

4.4 Enhanced Cuts

It can be assumed that, since some areas of the search space will be hit more frequently than others, and since the HICuts tree cuts the search space into contiguous regions, there might be one particular subtree in the entire HICuts search structure which is consistently hit more frequently than the rest of the structure (excluding the nodes and ranges on the path from the root to that subtree). Enhanced cuts recognize that this might happen in a HICuts tree, and work to reduce the work entailed for searches which travel to that region of the search space.

All nodes which are at least at level 3 of the search structure are candidates for being the root of an enhanced cut. Enhanced cuts are not considered until several iterations of major restructuring are performed. This allows the search structure time to move infrequently hit regions away from frequently hit regions. As described below, this is a requirement for an enhanced cut to be performed.

Starting from the root of the search tree, a depth first search is performed to find any node which has more than t times the hits of any of its siblings. The t is the same as the ones calculated for any of the three restructuring techniques described above. If this node matches that criteria, and all its descendants do not match this criteria - in other words, its descendants are all approximately equally likely to be hit in any search which travels through this node - then this node is a candidate for enhanced cuts. If only one candidate can be found, then an enhanced cut is performed. If more than one candidate is found, then either no cuts are performed, or the one which is more frequently hit is chosen.

An enhanced cut creates a new root for the search structure. This root has two children. The first is the root of the subtree selected by the enhanced cuts algorithm. The second is the root of the original HICuts tree. Any search now first is checked to see if it falls in the region defined by the promoted subtree. This involves checking multiple dimensions. The result of this is a decrease in the search time equal to the original level of the subtree minus 2 for any searches which pass through the newly promoted node, and an increase of one in search time for all other searches. As only one range check is now performed at the root, and the subtrees themselves are not changed, the change in levels exactly corresponds to the change in work performed. Since the promoted node must be more frequently matched than its siblings, we expect the overall performance of the classifier to increase.

It should be noted that enhanced cuts can be seen as a slightly more restrictive form of TCAM caching (described immediately below in Section 4.5). In cases where a TCAM is used in conjunction with enhanced cuts, it is required that the subtree contain more rules than can be held in the TCAM. If this is not the case, it can be assumed that most of the rules in the subtree are already in the TCAM, and that enhanced cuts will simply increase the search time of the rest of the tree, with no benefits for the promoted node. The tradeoffs of performing enhanced cuts versus using a TCAM cache in a classifier are described in detail in Chapter 6 on performance. As the TCAM contents change as the dynamic classifier runs, the contents of the TCAM cannot be assumed at any point, and cannot be removed from the tree, or ignored upon building the tree. This means that enhanced cuts cannot assume that the subtree is considering for promotion had no rules overlapping those in the TCAM.

4.5 TCAM Caching

Since Ternary CAMs can find matches against rules in a very short amount of time, such a solution, which runs in constant time regardless of the size of the rule set, is the best which can be hoped for in packet classification. Unfortunately, TCAMs are expensive, both in terms of original cost and in terms of power consumption, and thus cannot be used for large rulesets.

However, TCAMs can still play a role in packet classification over very large rulesets. Rather than storing all the rules in the TCAM, certain rules, either decided a priori or based upon some learning criteria can be stored in the TCAM. The TCAM can be checked first, much like a cache in hardware or a transposition table in software. If a matching rule is found, it is retrieved from the TCAM. Otherwise the search continues in the normal classification algorithm. Since a TCAM is separate hardware from the processor and memory, these two tasks can be carried out in parallel, and nothing is lost if the rule cannot be found in the TCAM.

Such a cache can be used with any packet classifier, not just with a dynamic one as outlined in this chapter. However, the design given below was specifically developed to work well with the dynamic HICuts algorithm.

A TCAM works best to respond to small changes in the locality of traffic. These changes would affect nodes far from the root of the tree in the search structure but would most likely not affect the nodes close to the root. It is known that Internet traffic shows patterns of both spatial and temporal locality [9], and the TCAM helps the classifier exploit this. There are three main criteria to consider when using a TCAM as a cache.

First is the replacement strategy for the TCAM. The strategy used for this work was Least Recently Used. Since the controls for the TCAM are implemented in software as part of the algorithm, it is not difficult to keep track of the order of use of the entries in the TCAM.

The second consideration is the insertion threshold for adding new entries to the TCAM. Every rule keeps track of how many times it has been matched (subject to the rules for forgetting outlined in Section 4.6). When a rule is matched, its hit count is checked. If this count exceeds a preset threshold, then the rule is inserted. The TCAM itself holds pointers to rules, so matches in the TCAM allow rules to increase their hits. If this was not done, rules stored in the TCAM would not maintain proper hit counts and may, if removed from the TCAM, take longer to be reinserted than they should. Determining the proper threshold for insertion is difficult and is discussed in Chapter 6.

Finally, it is important to consider whether to store packet headers or full rules in the TCAM. Storing headers would allow us to use a CAM, which is significantly cheaper and less power consuming than a TCAM. However storing rules allows for the greater likelihood of matching. The full rule, regardless of size, is stored in the TCAM upon exceeding the insertion threshold. After that each TCAM entry for that rule in the TCAM is considered separate for terms of the LRU replacement strategy.

For any rule which can be matched in the TCAM, the value of $L(\eta)$ is 1. Since it is impossible to find a rule in the HICuts tree in that time, and since TCAM caching does not affect the search structure in any way, this always results in a net benefit in terms of search cost. Since we terminate the search in the HICuts tree upon a match in the TCAM, it is important to realize that hit counts on any searches which match rules in the TCAM are not maintained. This is desired. If a rule doesn't spend much time in the TCAM, the effects on the hit counts will be hardly noticeable. Should the rule remain in the TCAM almost permanently, then the lack of hits in the search structure will allow other nodes and rules to move closer to the root at that rule's expense. This further improves the average search time of the classifier.

There is no need to keep an empty TCAM at the beginning of running the classifier. Before accepting traffic, the TCAM should be populated with some rules. As it cannot be known beforehand which rules will frequently be matched, either a set which is believed to be more likely to be matched can be selected, or the TCAM can be filled randomly from the ruleset. All the tests outlined below pre-filled the TCAM randomly, with the order of insertion used to rank the entries for LRU replacement.

4.6 Remembering and Forgetting

A major component of any learning system is a means by which to remember information. There are two main pieces of information which must be remembered in order to make use of the restructuring techniques detailed above.

First, the number of times a range in a node has been matched must be recorded. This involves keeping a counter for every range, which is updated each time that that range is successfully matched. These data are important for all four of the restructuring techniques given in Sections 4.1 through 4.4. Second, the classifier must keep track of the number of times a rule has been matched. This must include both hits in the search structure itself, and hits in the TCAM. This counter is important for determining when a rule should be inserted into the TCAM. As noted above, LRU is used to determine which entries in a TCAM should be sacrificed when needed. As a result, the counts stored in the rule itself are not used for that purpose. Only one count is kept for every rule, regardless of how many leaves contain it in the HICuts tree. This count is only used to determine whether to add the rule to the TCAM, so any hit on this rule, regardless of its location, is incremented in one spot.

In addition, for statistical reasons, other data must be remembered. The number of times a node has been checked, along with the number of times every range has been checked and the number of times every rule has been checked, must be recorded in order to get an accurate measure of the amount of work performed by the dynamic classifier. The sum of these values gives a good approximation of L for the classifier. The sum shows how deeply the classifier searched, how many ranges it checked, and, when the proper leaf node was found, how many rules had to be searched through to find the one which matched the header in question. To this sum the number of searches in the TCAM is added. This number is always identical to the number of packets classified.

Remembering information, however, is only half the battle. In order to properly make use of traffic patterns, forgetting the traffic seen after a certain amount of time is required. Failing to do this will result in a classifier which might, after an extended period of time seeing very similar traffic, be unable to respond to changes in traffic patterns. In order to accommodate for changes in traffic patterns, the classifier must be able to forget what it has seen before. Of course, it should only forget the hit statistics which affect its restructurings, not the statistics which are used solely for measuring the efficiency of the classifier.

A near-perfect solution would be to keep a window of recent searches in memory. When a new search is performed, the rule that matches is added to the front of the window. The rule furthest to the back of the window falls out and the changes made to the counters in the tree when classifying that rule must be forgotten. Unfortunately, this is not particularly feasible. Since the tree can be updated in the time between when a rule is added to the window, and the time the rule falls out of it, the classifier cannot simply find the rule in the tree and reduce counters in a reverse path towards the root.

A possible solution would be to store every range and rule affected by every search, however for any large window or large search structure, that would utilise far too much memory to be practical. Instead periodic forgetting is performed.

At preset intervals every hit counter in the search tree is decremented by one (no counter can ever have a value below 0 at any point). Any range or rule which is infrequently hit will keep a counter close to zero, and those ranges which are frequently hit will build up large counters. This technique has the disadvantage that it is slower to react than the window approach. Forgetting cannot be done at every cycle. However, if the classifier forgets more slowly than it learns, it still runs the risk of not being able to adapt to changes in traffic as quickly as with the window approach. However, as this is the only practicable method of forgetting, its drawbacks are tolerated in exchange for the benefit of being able to forget the data obtained.

Other techniques for forgetting, such as multiplying by a value between 0 and 1 to reduce the number of hits stored for each rule and node may be used as well.

4.7 Update Frequency

Two features of the dynamic classifier are dependent on the rate at which they are performed. All major restructuring is done at a specified rate, as is all forgetting. These two rates are independent of each other and are measured in terms of the number of packets classified between the times at which they are performed. The considerations as to how quickly updates should be performed are based mainly on empirical results which are detailed in Chapter 6. This section outlines the tradeoffs which occur due to changing the update frequencies used by the classifier.

4.7.1 Update Frequency - Forgetting

The rate at which the traffic seen is forgotten affects the speed at which the classifier can respond to changes in the traffic being classified. A very fast rate of forgetting would make it extremely difficult to ever learn anything about the traffic seen. If forgetting occurs at a rate of 1, then no learning occurs, as the contributions to the classifier's knowledge made by a single packet would be immediately wiped out when the classifier forgets what it has learned.

Conversely, forgetting cannot proceed at too slow a pace. Forgetting is required to ensure that the classifier can adapt to traffic as quickly as possible. Consider that, for 5000000 headers, one particular rule has been matched 50% of the time, and that information is forgotten every 5000 packets. This means that for any node on the path to that rule, the number of hits must be at least 2499000. Forgetting has been performed at least 1000 times during this period of time, which explains what the number is not 2500000. The number of hits might be higher depending on what other searches take this path, and at what points forgetting occurs. Now, if the classifier no longer sees any packets taking this path, it should realize that the traffic has changed and that this path should possibly be considered for restructuring in order to lower the value of $L(p)$ for other paths under consideration.

However, these nodes have at least 2499000 hits. Forgetting once for every 5000 packets means that it will take approximately 12.5 billion packets classified, given the assumption that no other packets during that time are classified on this path, to forget the those initial 2500000 packets. This is obviously unacceptable. It is unrealistic to expect the classifier to take that much time to forget traffic it has seen. The result of these two facts, that forgetting cannot take place too quickly or too slowly, means that the classifier must be careful in determining how quickly it forgets what it is seeing.

4.7.2 Update Frequency - Major Restructurings

Major restructurings, as described in the first four sections of this chapter, can be costly to perform. It is important to balance the time spent performing these restructurings with the time spent classifying packets. Further, restructurings cannot be performed faster than they can be completed. In practice, restructuring would be done in a separate thread (possibly on a separate processor) and, once the search structure had been restructured, classifications would be moved to the newly-altered search structure. In the meantime, the old search structure is used. Performing restructurings at a fast rate allows the classifier to react to changes in traffic quickly, and reduce the value of L.

As an example, consider that the classifier is performing major restructurings at a rate of one per every 500000 packets classified. If the time required for a restructuring is 50000 packets, then the rate of restructuring could be increased by up to a factor of 10. In practice, the rate should not be increased by quite that amount, since the time taken for each restructuring and the speed at which packets are classified are variable. Increasing the rate at which restructuring occurs by a factor equal to half the factor between the time required to restructure and the time which currently passes between restructurings is acceptable. By doing this, it is possible to increase the speed of classification while ensuring that the classifier never spends more time restructuring the search structure than is allowed by the rate of restructuring.

If, at any point, the time needed to perform restructurings begins to exceed 75% of the time between iterations of restructuring, then the rate at which restructurings are performed can be slowed slightly. This reduction should be by less than 33% to ensure that we do not immediately halve the time on the next iteration. Since the time needed to alter the search structure should be relatively constant, it can be expected that, given a high enough initial value for the time to perform updates, the update time will be halved repeatedly until a point is reached where updates take one-half to three-quarters of the allotted time. At this point the frequency will likely never be altered again.

Finding the update frequency for reorders by this method allows the classifier to learn how quickly it should reorder itself and guarantees that it will be able to reorder the search structure in a reasonable amount of time and still be able to finish within the time given. However, it is not clear exactly what rate the classifier should start with. Empirical results given in Chapter 6 will detail exactly what happens to the performance of the classifier depending on the initial value given for the update frequency.

4.8 Worst-Case Bound

Given the methods for restructuring the search tree outlined above, an obvious question arises as to the worst-case performance of the new dynamic classifier. Restructuring the HICuts tree could result in extremely poor worst case performance if the worst case is not bound during the restructuring process. Should a single rule never be matched during the period encompassing a large number of iterations of the restructuring algorithm, it is conceivable that it would be pushed further and further away from the root of the search tree. If a sudden burst of packets arrive which match that rule, the classifier might not be able to match them in time, resulting in dropped packets and reduced network performance. Three features of the dynamic tree described in this thesis prevent this from happening during restructuring.

First, all restructuring is performed using the HICuts algorithm. Should a node be promoted, both it, and its newly created sibling(s), will immediately have the HICuts algorithm performed upon them to ensure that they still meet the requirements of a HICuts tree. As each node in a HICuts tree is only allowed to cut on one dimension, a promoted node might lose several levels of cuts which helped define it. These cuts are remade using HICuts. This guarantees that the new subtrees created remain as balanced as possible and prevents newly created subtrees from dangerously unbalancing the tree. Similarly, range combinations, when performed, use HICuts to partition the ruleset in the newly-created node. As a result, these new nodes, while containing more rules than any of the separate nodes used to create them, will remain reasonably balanced.

This alone is not enough. It is conceivable that a long string of alterations to the search structure could be performed along a particular path in the search tree. If each change was to slightly increase the amount of work the classifier is to perform when searching along that path, the cumulative effect could result in worst case performance degrading to unacceptable levels.

To prevent this, restructuring which exceeds a criteria for maximum work to search to any node from the root of the search tree will not be performed. As this cannot be known a priori, once a restructuring has been performed, the tree is quickly tested to see the maximum amount of work done. The maximum depth can be returned by the HICuts algorithm (as it must have reached that depth when restructuring), as can the maximum width of the nodes in the newly-created subtree. These two measures can be used to quickly compute an upper bound on the maximum work used to search the new subtree. The amount of work needed to get to the subtree can be accurately obtained simply by computing the length of the path from the root of the search tree to the root of the new subtree. Since HICuts attempts to make as balanced a search tree as possible, this upper bound is quite tight.

Should the maximum work for the newly created subtree exceed the maximum worst case work allowed, the changes to the search structure are not saved. In this case, the search structure reverts to its original form. A flag is maintained to indicate that restructuring was attempted at this node and failed to satisfy the worst-case bound. This flag is only removed when nodes higher up in the tree are restructured. Until it is removed, restructuring this node is prohibited to avoid wasting time creating a search structure that will have to immediately be thrown away. As any and all violations of the worst case bound are immediately detected and dealt with, the is an absolute guarantee that the classifier will never violate the worst-case bound.

Finally, further down in the tree the degree of difference required to prompt a restructuring of the search tree increases. At the root node, this difference must be a preset factor t. This factor increases deeper in the tree, guided by a monotonically increasing function of depth in the search tree and t. Any function which matches this criteria can be used. This serves the dual purpose of reducing that chances that the search structure will exceed the worst case bound (as it will be far less likely that changes will be made near the worst case bound) and of ensuring that, since locality of traffic becomes more relevant in smaller search areas, changes are not made too often in lower levels of the search structure due to short-lived changes in the locality of traffic.

Chapter 5

The Dynamic Classifier - Implementation

Given the design for the dynamic classifier given in Chapter 4, an actual dynamic classifier needed to be created for experimentation. The dynamic classifier was written in the C programming language and was implemented in two stages. The first stage involved the re-creation of the static HICuts algorithm as described by Gupta and MacKeown in [17]. The details of the implementation are given in Section 5.1. This classifier then had to be modified to take advantage of the dynamic improvements. How this was done is detailed in Section 5.2.

5.1 Implementation of Static HICuts

Figure 5.1 gives the internal details of the node structure used. Pointers were maintained to the list of children, rather than static arrays, as the number of children of a HICuts node can not be known in advance, and might change when the dynamic improvements to HICuts are implemented. It was advantageous to use pointers and dynamically allocatable memory in the design of the nodes themselves. A node also contained a list of ranges (named cuts), which corresponded to the ranges of the node's children on the dimension on which the node was cut. This dimension was stored in the variable dimension. The ruleset associated with the node was stored in the node, as well as the ranges which this node covered on every dimension. Additionally, statistical information, such as the number of total hits, and the current hit count for the node (which most likely would not equal the total hits due to forgetting) were stored in the node.

Ranges for the HICuts search tree were trivially encoded as a small structure containing both an upper value for the range and a lower value for the range. The ranges were considered to be inclusive. A value equal to either the upper or lower value for the range would be considered to be within that range.

Rules were implemented as structures as well. The members of this structure are given as Figure 5.2. The primary component of a rule was the list of ranges associated with that rule, one for each

	range_t **	ranges
	range_t **	cuts
	node_t **	children
struct node_t:	ruleset_t *	ruleset
	unsigned long long int	hit_total
	unsigned long long int	hit_count
	int	dimension

Figure 5.1: The structure of a node.

	range_t **	ranges
	unsigned long long int	hit_total
struct rule_t:	unsigned long long int	hit_count
	int	action
	rule_t **	covers

Figure 5.2: The structure of a rule.

dimension of the packet header being matched, as well as the value containing the action associated with the rule. This action was simply an integer value in this implementation, as it was not of much importance to the experiments. In addition to this, two integer counters for statistical purposes were maintained. Finally, an optional list of rules named covers was provided; the purpose of this list is explained in the discussion of the TCAM in Section 5.2.

Rulesets were created as dynamically-sized arrays of pointers to rules. Each node had its own ruleset, however there was only one copy of every rule. These rules were created with the ruleset for the root node. After that, for every other node, a ruleset was created and the pointers in that ruleset always contained the addresses of the original rules stored in the root. This was the easiest way to ensure that hit counts for rules stayed accurate, and reflected all the traffic that the classifier was seeing.

Searching a HICuts tree is quite simple. Starting at the root of the tree, the dimension that that node was cut on is extracted from the structure. The corresponding field in the packet header is extracted and compared against each of the ranges in the cuts list. This is a linear search, starting with the head of the list, and proceeding to the final element of the list if needed. The HICuts algorithm guarantees that one of the children of the node will match the packet header, so there is no real need to perform bounds checking. When the entry in the list which matched the field of the packet under consideration is found, its index (as the list is embedded in an dynamic array) is used as the index into the list of children of the node. The search then continues into that child.

A node which has no cuts and no children is considered a leaf node. These can be determined by checking to see if the child list has been created, or if it has the value NULL. When the search hits a leaf node, rather than checking for children, the rule list is checked. For each rule, every range is checked. If a rule matches a packet on every range it contains, then the action associated with the rule is returned. If not, then the next rule is checked. There is no guarantee that there will be a rule to match the packet in question, so the classifier has to ensure that it stops checking rules when there are no more to check. A dummy rule was inserted at the end of the rule lists in every leaf, so that the search would know when it has exhausted all the rules in a leaf. If no matching rule is found, there must be no matching rule in the original ruleset, and the classifier returns the action which had been defined for this case.

The heuristics which guided the creation of the tree were based on those provided by Gupta and MacKeown in [17]. The first two heuristics (which determine the dimension to cut, and the number of cuts to make) follow the mathematics set out in [17] exactly. The third and fourth heuristics (which attempt to eliminate identical children and to remove useless rules from nodes) were implemented by having the classifier test the results of the first two heuristics to see which, if any, matched the provisions of the latter two tests. These tests were performed as each node was being created, and the nodes themselves were created in a depth-first manner. No extensive details were given in [17] as to Gupta and MacKeown's implementation of the third and fourth heuristics, and, as a result, a classifier which implements these two heuristics differently might see slightly different performance.

5.2 Implementation of the Dynamic Improvements

For the implementation of the dynamic improvements to the HICuts algorithm, the work performed by the classifier was split into two distinct parts running as separate threads. In the implementation used for the experiments detailed in Chapter 6, a threading API based on the POSIX threads (pthreads) standard [6] was used. The first thread handled all searching performed by the classifier, as well as all immediate changes to the search tree (namely, all changes involved with learning and forgetting), and all changes involving the TCAM. The second thread handled all the aspects of restructuring the search structure itself. The threads shared very little information, outside of the search structure itself, and two condition variables used by the threads to communicate.

The first thread implemented the HICuts search algorithm described in Section 5.1 in order to classify packets. Remembering and forgetting were implemented using a single counter in every node and every rule in the search structure. This counter had a minimum value of 0 and a maximum value of $2^{64} - 1$. Any attempt to reduce the counter below zero, or increase it above the maximum was ignored. As a packet was being classified, every node that it visited and every rule that it searched had its counter incremented. The rate of forgetting r_f was supplied as a parameter to the classifier when it was started. Every time r_f packets had been classified, a quick depth-first search of the search structure was performed. This would guarantee that every node was visited, and all would have their hit counts reduced by one. The rules stored in the ruleset of the root node were all decremented by one as well.

The TCAM was implemented in software, using an array of unsigned 64-bit integers. This limited the size of entries in the TCAM to 64 bits. All experiments were conducted over rulesets of at most two dimensions as a result. Each entry in the TCAM used two of these integers, one to store a pattern, the other to store the mask for that pattern. The classifier compared the bitwise ANDs of the stored pattern and the proper field of the packet header with the mask. If they were equal, they were said to have matched, otherwise, they were said not to have matched. Stored with the pattern and mask was a pointer to the rule's entry in the ruleset in the root node of the search tree, so that the action associated with the rule could be determined when the rule was matched.

In addition to the TCAM itself, a list of the TCAM entries was maintained in regular memory. This allowed the classifier to keep a list of the least frequently matched entries, in order to decide which rule to sacrifice upon the decision to enter another rule into the TCAM. Every time a rule was matched in the TCAM (or was the highest priority of more than one matched rule), it was removed from its place in the LRU list, and placed at the bottom. Rules at the top of the list were the first candidates for sacrificing.

It was possible that two or more rules stored in the TCAM could match a packet. If this was the case, the memory addresses of the pointers retrieved were compared. Since the ruleset was stored in order of priority, those rules which had higher priority would be located further up in the ruleset. Thus, the lowest memory address returned by the TCAM would be taken as the highest-priority rule for the packet matched. Every rule in the root ruleset maintained pointers to other rules, which had higher priority and covered parts of its region. If the rule returned by the TCAM might possibly be covered by higher rules, these needed to be checked. This seldom had to be performed in practice, however, as the rules stored in the TCAM seemed, from empirical results, to cover regions of the search space quite thoroughly, and most rules in a certain region were already in the TCAM and were compared there.

Any time that a rule was matched by the classifier, it was considered for insertion into the TCAM. Every rule had a flag that indicated whether or not it was in the TCAM already. If that flag was not set, and the number of hits on that rule exceed the TCAM threshold given when the classifier was started, then the rule was inserted into the TCAM. This meant that one or more entries were removed from the TCAM to make room. The rule was then encoded into sets of patterns and masks. Several entries might need to be created in the TCAM, as a single ternary logic entry might not suffice to cover the rule completely.

In the case of the experiments detailed in Chapter 6, ranges were used for all rule definitions. When TCAM entries needed to be made from rules, the dimensions were separated, and each was made into the minimum number of ternary expressions needed to represent this range. Then the Cartesian product of these two sets was used to find all the entries needed. Since the two ranges were independent, there was no way to make the number of entries in the TCAM any less than this.

The second thread handled all the matters pertaining to restructuring the search tree itself. The thread waited on a condition variable by default, only being activated when the first thread allowed it. The decision as to when the second thread should be activated was based upon the number of packets that had been classified since the last time the thread was activated, controlled by the parameters described in Section 4.7. The second thread was permitted to ignore activation by the first thread if it had not yet finished restructuring the search tree from the prior activation. When the restructuring had finished, the second thread altered a condition variable for the first thread. This allowed the first thread to estimate how much time (in packets classified) was needed to restructure the search tree, and determine the time which would be alloted to the second thread for another iteration of restructuring.

Reconstructing the search space was done as described in Chapter 4. First, nodes were considered for enhanced cuts. After that, range combinations were performed, then node promotions. Finally, range reordering was performed. These techniques were implemented as described in Chapter 4. Details of the restructuring are given below.

Range reorders were the simplest of the four major techniques to implement. For every node which was being considered for range reordering, a linear scan of the children of the node was performed to determine the most hit, and least hit node. If the difference between these two nodes was greater than the calculated threshold for changing that node, then the children were sorted in descending order based upon number of hits. Both the list of children and the list of cuts were modified simultaneously, to ensure that the ranges stored in the cut list still matched up properly with the children themselves. Since there were seldom more than five or six children of a node in HICuts, insertion sort was used instead of a recursive sorting technique.

Node promotions were slightly more difficult to implement, as the number of hits which should be stored for each newly created node needed to be calculated. If a node N was to be promoted into an ancestor A, the following steps were performed. First the node was removed from the child list of its parent. This involved shifting every element of the list which was to the right of N, once to the left. Then, in A the child which contained the range which included the range N covered on the dimension A was cut on was found. This child was removed from A in the same way N was removed from its parent. One new node was then created and was given the same range as N on the dimension A was cut on, and the same ranges as A on all other dimensions. The proper ruleset (one which matched the ranges) was inserted into this node, and HICuts was performed on it to get the rest of the search tree. One or two other ranges were created to hold the remnants of the range removed from A. Both of these were then set to point to the original child of A. The hit count for the original node N was used as the hit count for the newly promoted node. It was safe to allow the nodes in the subtree rooted at the new node to have values of 0 as this would not affect the restructuring of the new subtree.

A linear scan of the children of a node N was done to determine if it was a candidate for range combination. If a node had been determined to be a candidate for range combination, the nodes which were to be combined were removed from the child list of N. A new node was created, with a range stretching from the smallest lower value of all the removed nodes to the highest upper value for the dimension on which N was cut. The new node had ranges identical to N's ranges on all other dimensions. The rules contained in the nodes to be combined were all added to the ruleset

struct ec_node_t:	range_t **	ranges
	range_t **	cuts
	node_t *	children[2]
	ruleset_t *	ruleset
	unsigned long long int	hit_total
	unsigned long long int	hit_count

Figure 5.3: The structure of an enhanced cuts node.

for the new node. In order to preserve rule priority, the ruleset in the root of the tree was consulted, and rules were added to the new ruleset when seen in a linear scan of the root ruleset. Since the highest priority rules were always at the top of their rulesets, this was a trivial task. HICuts was then performed on this new node to create a new subtree. The sum of the hits on all the nodes added to the new node was given as the hit count for the new node. Nodes in the new subtree were all given hit counts of 0. A number was maintained in N to keep track of the number of non-merged children. When sorting was performed for range reorders, this number was used to ensure that the combined node was always at the end of the list of children. If a node created by range combination was later split by a node promotion, the newly promoted node and the other nodes created remained anchored at the end of the list.

Enhanced cuts were performed as described in Section 4.4, and there is little to detail here in terms of implementation. Searching an enhanced cut node was slightly different than searching a regular node as the classifier must consider the ranges of the first child of the new node on every dimension. As a result, an enhanced cut node had a different structure than a regular node. The structure of an enhanced cut node is given in Figure 5.3. The list *cuts* contained the ranges of the first child on every dimension being classified. If a packet matched every one of these ranges, then the search entered the first child. Otherwise it continued on in the second child. When an enhanced cut node was created, it was given the same hit count as the root node itself. All nodes on the path from the root to the promoted node had their hit counts reduced by the hit count of the promoted node. This ensured that the search tree could quickly accommodate the removal of a subtree.

In order to ensure that the classifier would not violate worst-case bounds, as well as to ensure that significant alterations to the search structure did not interfere with packet classification, all the above changes were not made to the search structure itself. A list of all potential changes was maintained in memory. When the restructuring was finished, and it was confirmed that the new structure did not violate any worst-case bound, then the changes were committed. The primary thread was halted (via a condition variable) at the beginning of a search and the changes were applied. Since there were rarely many changes to make, and since old nodes were replaced with new ones, this process took very little time. At this point, the primary thread was restarted, and would begin its next search using the new search tree. All changes were committed at once to ensure that classification was stopped as few times as possible.

The classifier must, as noted in Section 4.8, ensure that it did not exceed a given worst case bound when making changes to the search structure. The maximum work that would be performed in a search of a newly created subtree can be returned by HICuts. For range combinations and node promotions, it was easy to guarantee that a worst case bound had not been violated. The value returned by HICuts could be added to the length of the path from the altered node to the root to give the amount of work for such a search. For range reorders, things were slightly more complicated, as the total work along some paths might be altered by too much. If a certain path was one less than the worst-case bound, and the classifier moved that path two to the right in a node, the worst case bound would be violated. After a range reorder, a depth-first search of all the subtrees under the node was performed, and the longest path encountered returned. The resulting values, combined with the length of the paths from the root of each subtree to the root of the tree, gave the values for the maximum possible search length for every subtree. As long as all were less than the worst-case bound, the range reorder could proceed. Enhanced cuts also had to probe the search space to ensure that the addition of one node and one range check to the search path did not cause any path in the search tree to exceed the worst case bound. If any change violated the worst case bound, it was eliminated from the list of changes to implement.

Finally, the change threshold at every node in the tree must be known. While the formulas which compute this threshold can be arbitrarily complex, for the purposes of this implementation, a very simple threshold was used. The threshold for any node in the tree was simply $10 + 2d$, where d is the depth of the node in the search tree (with the root having a depth of 0).

Chapter 6

Performance

In order to gauge the performance of the altered HICuts classifier, many tests were run using both the static and dynamic classifiers with several packet traces and two distinct rulesets for classification. For the dynamic classifier the improvements outlined in Chapter 4 were turned on and off. In addition modifications were made in the size of TCAM used, the threshold for a rule's entry into the TCAM, and the rate of forgetting. This was done in order to check how each improvement affected the performance of the classifier, and to determine which improvements contributed the most to the classifier's speedup.

The results of all these tests were then compared to illustrate how the classifier improved the speed of classifying packets while maintaining a worst-case bound and ensuring that memory use did not get out of hand. The structure of the packet traces, rulesets and overall experimental design are given in Section 6.1. Results and commentary on the classifier's performance are then given in Section 6.2.

6.1 Experiment Design

All experiments were conducted by running the HICuts algorithm on a variety of synthetic and real Internet traffic traces. For every trace a large number of runs of the HICuts algorithm were made. The first four enhancements described in Chapter 4 varied between on and off. The TCAM was set to four distinct sizes (0, 10, 50 or 100 entries), set to either store rules or packets, and set to one of five insertion thresholds (0, 10, 100, 1000 or 10000 hits before insertion). Five different rates for forgetting were tested (forgetting after every 100, 200, 300, 400 or 500 packets classified). If the TCAM size was zero, then the other two TCAM parameters were not changed as, without a TCAM, the TCAM parameters are useless. In addition, five different initial rates of restructuring were tested, with restructuring initially occurring once every 100000, 200000, 300000, 400000, or 500000 packets classified. This means that for every packet trace and ruleset combination there were: $3 * 2^5 * 5^3 = 12000$ tests with the TCAM, and $2^4 * 5^2 = 400$ tests without, for a total of 12400 tests.

From these runs, the average depth of search, along with the number of nodes checked, the number of ranges checked, the number of rules searched in leaf nodes, the number of hits in the TCAM, and the number of reorders, promotions and recombinations were all recorded. Of these values, the most important for gauging performance were the number of node hits, the number of range checks and the number of rules searched. These three numbers, added together, gave the final value of $L = \sum_{i=1}^{n} L(p_i)$ for that test. In addition, the amount of memory used was calculated. The maximum value of $L(p)$ for any search was returned, as was the maximum possible $L(p)$, which was used to confirm that the worst-case bound on search time was not violated. Several additional statistics were generated and returned, however, they did little to describe the overall performance of the classifier, and were used for bug-fixing and confirmation of the classifier's activities. As a result, they will not be discussed further in this thesis.

6.1.1 Rulesets

For the purposes of the initial experiments, a ruleset which was composed of two dimensions of size 2^{16}, split into equal sized squares was used. Each of these squares corresponded to a single rule in the overall ruleset, did not overlap any other rule in the ruleset, and had sides of length 2^{12}. This ruleset was named the 'checkerboard' ruleset and was used, despite its artificial nature, so that the packet headers used for some of the tests could be created easily to test the effects of locality on packet headers. Given this ruleset, it was easy to derive synthetic packet traces which could test a large variety of patterns in the frequency at which certain rules were hit in comparison to other rules. This allowed for the testing of boundary cases, as well as for the testing of other interesting cases to see the effects of the dynamic classifier under very tightly controlled situations.

In addition to this artificial checkerboard ruleset, a ruleset for testing was derived from the ruleset for the intrusion detection program Snort [22]. Snort has a large number of rules, which allows for the testing of large rulesets, and also the fine-tuned testing of smaller, hand-chosen rulesets. In addition, since the Snort rules are defined to aid in intrusion detection, they can be tested with any Internet trace. Using firewalls or routers as the basis for a real-world ruleset instead of Snort would require traces specific for the device they were taken from for testing. Using rulesets from a publicly available source like Snort should also aid others who wish to duplicate this or similar work. Finally, with a large ruleset which needs to be searched quickly, Snort is a program which could benefit from any improvements which could be made to packet classification. Since Snort is an intrusion detection system, classifying over source and destination addresses seemed quite useless. Instead, a medium-sized subset (300 rules) of the Snort ruleset was used, and classification was performed using both the source and destination port in each header. This meant that classification was over two dimensions, each of size 2^{16}. This corresponds nicely with the dimensions of the checkerboard ruleset. In fact, the checkerboard ruleset was reduced in dimension (though not in number of rules) to match the Snort ruleset.

6.1.2 Packet Traces

Several real-life traces used by the Communications Networks group at the University of Alberta were used as data for the tests. For reasons related to the method by which CNS names its traces, these are referred to as 'Telus' traces in the results given below. This naming is not meant to signify any participation by TELUS Communications Inc. in this research.

In addition, several synthetic traces were used. The synthetic traces were produced in four different ways. The first, uniform, produced packet headers with each field chosen uniformly (and independently) from the range of the dimensions in the classifier (in this case $[0, 2^{16} - 1]$). This trace therefore had no patterns which the dynamic classifier could exploit. The second method, single, simply produced the same packer header the desired number of times. As contrasted to the uniform trace, the single trace should be easily exploited by the dynamic classifier. Neither of these first two methods produced traces a packet classifier would see in real life. However, these two cases allowed the evaluation of the performance of the classifier in extreme situations. In particular, the single and uniform traces give indications of what to expect when the first few iterations of restructuring were all that was needed to perfect the dynamic classifier, and when there was little to no structure in the packet headers which can be exploited.

A third method generated between 10 and 20 random headers. Then, with a preset probability (either 10% or 1%) it chose to create a completely random packet header (as in the uniform technique above). Otherwise it produced a packet header which was very similar (within a range of 64 for either dimension) to one of the pre-calculated headers. This was called the 'clumped' method. Clumped traces are referred to as 'clumped1' for the 1% random case, and 'clumped10' for the 10% random case in the results given in Section 6.2.

Finally a last method, setsource, generated a single random packet header. Then, with probabilities as for the clumped method, it either generated a random packet, or a packet very similar (within a range of 64) to the single pre-generated packet on one of its two dimensions, and uniformly distributed on the other. This was an attempt to match the actions of a classifier in front of a small subnet. Most packets will have that subnet's prefix as either source or destination, but there will be some random traffic, and the classifier cannot control the source of packets coming in, or the destination of packets going out. In Section 6.2, these traces are referred to as 'setsource1' and 'setsource10'.

6.2 Results and Commentary

6.2.1 Overall Improvements in Classification

Table 6.1 gives the best path length for all combinations of ruleset and packet trace. For each of these tests, all improvements were used, the TCAM had 100 entries and an insertion threshold of 10000. The results clearly show the benefits of using dynamic classification over static classification. It is

Ruleset	Trace	Static	Best Dynamic
Checkerboard	Clumped1	14.216	1.036
Checkerboard	Clumped10	14.795	1.629
Checkerboard	SetSource1	15.102	1.044
Checkerboard	SetSource10	12.861	1.862
Checkerboard	Single	10	1.000
Checkerboard	Telus	9.894	1.003
Checkerboard	Uniform	15.201	9.655
Snort	Clumped1	12.362	7.270
Snort	Clumped10	11.637	8.259
Snort	SetSource1	13.255	4.592
Snort	SetSource10	11.505	3.834
Snort	Single	16	8.160
Snort	Telus	16.067	3.878
Snort	Uniform	12.112	6.617

Table 6.1: Best path length results in terms of L.

Ruleset	Trace	Static	Average	2 times Std.Dev.
Checkerboard	Clumped1	14.216	9.049	13.606
Checkerboard	Clumped10	14.795	9.883	12.981
Checkerboard	SetSource1	15.102	10.038	12.468
Checkerboard	SetSource10	12.861	9.504	10.223
Checkerboard	Single	10	1.282	3.248
Checkerboard	Telus	9.894	4.911	5.294
Checkerboard	Uniform	15.201	14.817	4.604
Snort	Clumped1	12.362	11.353	2.209
Snort	Clumped10	11.637	10.877	2.139
Snort	SetSource1	13.255	9.429	8.046
Snort	SetSource10	11.505	8.286	6.862
Snort	Single	16	10.811	4.696
Snort	Telus	16.067	8.021	4.628
Snort	Uniform	12.112	10.496	3.581

Table 6.2: The average value of L, and 2 times the standard deviation for all tests.

interesting to note that the Snort ruleset did not see the same improvement that the checkerboard ruleset saw. This is because the Snort ruleset has overlapping rules which make performing some of the improvements more difficult. Also, the Snort ruleset does not cover the entire search space, which means that packets which fall into the regions not covered by rules do not contribute as much information to the classifier as those which match rules in the ruleset.

Table 6.2 gives the average value of $L(p)$ and 2 times the standard deviation for all 12400 tests for every combination of ruleset and packet trace. Also given is the average value of $L(p)$ for that combination of packet trace and ruleset run on a static HICuts classifier. The numbers include those tests in which very few of the enhancements are turned on, as well as those in which all of the enhancements are in use. They also include those tests in which only one of the pair of mutually beneficial enhancements (node promotions and range combinations) is in use. As discussed above, in such situations, it is expected that the dynamic classifier might perform worse than the static classifier, as range reorders and node promotions compensate for each other's actions.

As shown in Table 6.2, the dynamic classifier performed better on average than the static classifier in every combination of ruleset and packet trace. For the purposes of discussing these results, attention will be paid primarily to the Telus trace, as well as the two extreme traces, Single and Uniform.

The Telus trace, on both the checkerboard and the Snort ruleset, showed the full benefits of the dynamic classifier. This trace had locality of traffic which was exploited, and the dynamic classifier performed quite well as compared to the static classifier. In fact, the classifier performed better on the Telus trace than on the single trace with the Snort ruleset. Since the Snort ruleset was not complete, the TCAM was not used in those tests involving the single trace. It was used with the Telus trace, however, and this allowed the classifier to perform better on this trace than the single trace on this ruleset. As with the single trace, a few instances of tests using the Telus trace performed worse with the dynamic classifier than with the static classifier. These instances were the result of inopportune mixing of enhancements in the dynamic classifier. This would not be seen in real-life, as such combinations of parameters would never be used.

The single trace showed, as expected, the full benefits of the dynamic classifier, especially on the checkerboard ruleset. The dynamic classifier performed nearly an order of magnitude better than the static classifier on average. This was, of course, due to the construction of the single trace. As only one rule was matched during the entire run of any test using the Single trace, the dynamic classifier continued to make changes to the search structure which were beneficial to searching for that one node. Nothing which the dynamic classifier did to other regions of the search space had any effect whatsoever on the overall performance of the classifier. In other words, practically every change made to the classifier was beneficial. The only possible way that classification could be made worse was when node promotions were performed without the aid of range reorders or range combinations to clean up the node into which a promotion was made. The end result of such a promotion might be a slight increase in the average value of $L(p)$.

The Uniform trace showed that, even when there was nothing to be learned from the traffic that the classifier was seeing, the dynamic classifier could still perform better than the static classifier. Generally, the performance of the classifier was improved in this situation because of the TCAM. Occasionally rules were inserted into the TCAM which were matched before they were removed from it. In the case where there were thresholds for entry into the TCAM, it was less likely for a rule to be inserted, but when a rule was, it was also very unlikely that the rule would be removed from the TCAM. In such cases, any packet which matched that rule would be matched in $L(p) = 1$ step, improving the overall performance of the classifier. As it turned out, the vast majority of tests on the Uniform trace performed equally well on the dynamic classifier as on the static classifier. As the dynamic classifier could not learn from the traffic it was seeing, this was the performance that

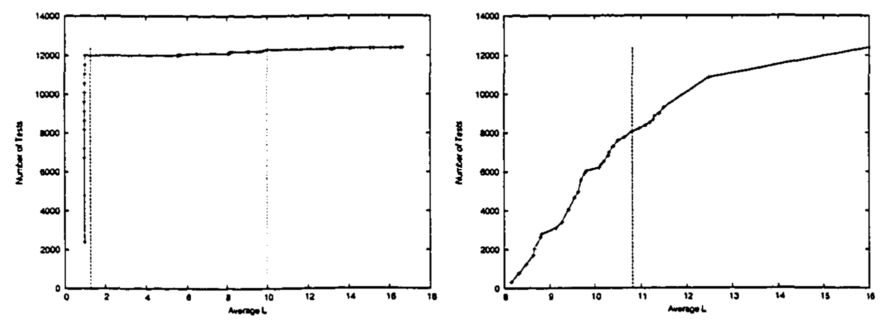

Figure 6.1: Average values of L for Single

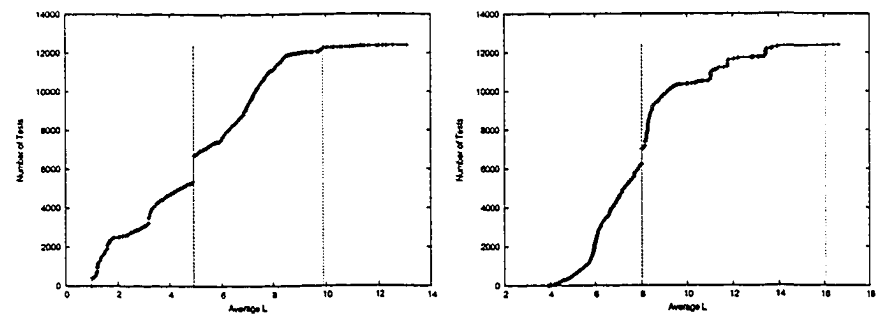

Figure 6.2: Average values of L for Telus

was expected. It is very encouraging to see that the dynamic classifier did not perform worse than the static classifier, even in cases where there was nothing which could be learned from the traffic the classifier was seeing.

The graphs given as Figures 6.1 through 6.3 show the cumulative number of runs of the classifier which provided an average value of $L(p)$ equal to or lower than any given value of $L(p)$. Additional graphs for other traces can be found in the appendix as Figures A.1 through A.4. The two vertical lines on the graph show the average overall performance of the dynamic classifier and the average value of $L(p)$ seen in the static classifier for each combination of ruleset and packet trace. These graphs are provided to show the distribution of the results which lead to the numbers

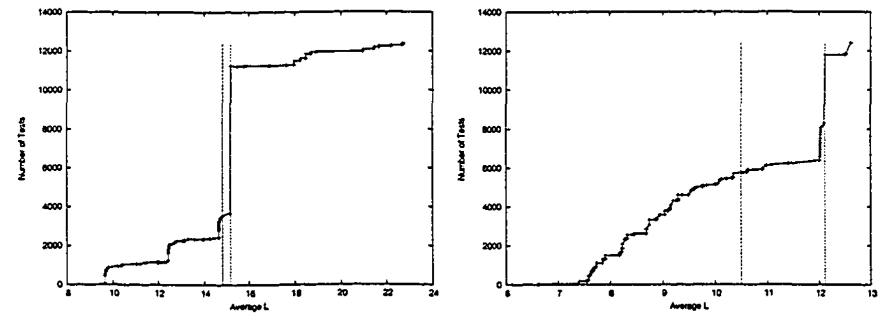

Figure 6.3: Average values of L for Uniform

Ruleset	Trace	Static	Average	2 times Std.Dev.
Checkerboard	Clumped1	21	29.052	18.609
Checkerboard	Clumped10	21	29.078	18.754
Checkerboard	SetSource1	21	22.267	7.137
Checkerboard	SetSource10	21	28.212	17.789
Checkerboard	Single	21	22.753	7.375
Checkerboard	Telus	21	30.598	21.295
Checkerboard	Uniform	21	22.765	10.622
Snort	Clumped1	35	32.076	6.563
Snort	Clumped10	35	35.014	2.604
Snort	SetSource1	35	34.916	1.659
Snort	SetSource10	35	33.777	4.659
Snort	Single	35	36.175	2.355
Snort	Telus	35	33.125	7.214
Snort	Uniform	35	33.839	4.744

Table 6.3: Actual worst case results for all tests.

given in Table 6.2. More importantly, however, they show that, given the proper combination of the improvements in the dynamic classifier, and the proper tuning of the parameters used to control those enhancements, the dynamic classifier can significantly outperform the static classifier.

In every combination of ruleset and packet trace, those test runs which had all, or almost all, of the dynamic enhancements in use performed significantly better than the other tests shown in the graph. Those tests which occurred near the average value for the tests, as well as near the value generated by the static classifier, were generally those tests which had most of the enhancements turned off, but did have the enhancements which complement each other (range combinations and node promotions) either both turned on, or both turned off. Those tests which did perform as badly as or worse than the dynamic classifier were those in which one of node promotions or range combinations, but not both, was turned on. In these cases, the negative impact of these improvements could be compensated for, and the overall performance of the dynamic classifier suffered as a result.

From these results, we conclude that any combination of enhancements might not positively affect the performance of the classifier. Fortunately, as the results for these tests showed, simply turning on all the enhancements was enough to ensure that the dynamic classifier outperformed the static classifier. There is no need for any convoluted or complicated method of figuring out which enhancements need be turned on, and which need be turned off to deal with certain types of traffic. In real cases, as well as in the extreme synthetic cases, using all the enhancements lead to the best performance in the dynamic classifier.

6.2.2 Worst-Case Performance

For the purposes of the experiments performed, the maximum worst case bound was set at $L(p) = 50$. This number was chosen as it was slightly more than twice the worst case seen in the checkerboard ruleset. With the worst-case size set to this value, the dynamic classifier could not make every

Ruleset	Trace	Static	Average	2 times Std.Dev.
Checkerboard	Clumped1	21	31.407	23.318
Checkerboard	Clumped10	21	32.026	25.189
Checkerboard	SetSource1	21	23.245	11.532
Checkerboard	SetSource10	21	30.437	23.135
Checkerboard	Single	21	22.917	7.978
Checkerboard	Telus	21	32.686	25.732
Checkerboard	Uniform	21	23.215	13.312
Snort	Clumped1	35	35.75	0.866
Snort	Clumped10	35	35.943	2.620
Snort	SetSource1	35	35.379	1.274
Snort	SetSource10	35	35.566	1.471
Snort	Single	35	36.513	2.842
Snort	Telus	35	36.027	6.368
Snort	Uniform	35	35.399	1.268

Table 6.4: Maximum worst case results for all tests.

change it would have liked to make to the search structure. At the same time, such a bound allowed some changes to be made to the structure. If the worst-case bound had been set at exactly the static HICuts level, then many useful enhancements would have been ignored simply because they increased the worst-case by a small amount.

It should be noted that the numbers for the worst-case given here differ from the numbers given in [2]. This occurred for three reasons. First, the value of L calculated here included the time needed to find the rule in the ruleset of a leaf node. Second, the HICuts algorithm itself was modified slightly between the tests used for [2] and the tests used for this thesis. Finally, since the packet traces are not fixed but are generated due to certain criteria, the traces used for these experiments, while having the same overall properties of the traces used in [2], did not contain exactly the same packets, altering the actual worst-case performance slightly.

Table 6.3 gives the average actual worst-case (the worst-case actually seen during the test, rather than the worst possible case for the test) for each of the seven packet traces and both of the rulesets. In addition, a value equal to 2 times the standard deviation is given. From these results we conclude that the improvements do not greatly affect the worst-case performance of the classifier. On average the dynamic classifier had only a slightly worse actual worst case than the static classifier. This shows that the benefits provided by the dynamic classifier did not come at the expense of significantly worsening the worst-case performance of the classifier. It is interesting to note that that standard deviation for the checkerboard ruleset is generally larger than that for the Snort ruleset. This is due to the fact that there are many more 'useless' rules in the checkerboard ruleset than in the Snort ruleset. In any test which hit only a certain few checkerboard rules, many more Snort rules were hit. This made it possible, given the right combination of enhancements, for more dramatic changes to be made to the search structure derived from the checkerboard ruleset and resulted in a much greater variation in the results for the actual worst-case.

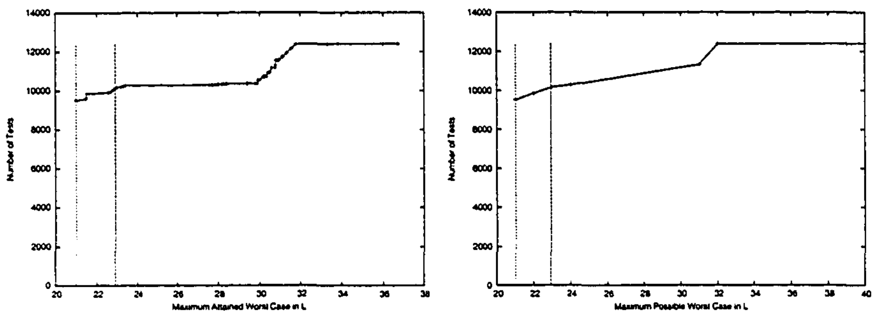

Figure 6.4: (a) Actual and (b) maximum worst-case results for Single on Checkerboard

Table 6.4 gives the average worst possible value of the worst case for each of the seven packet traces and both of the rulesets. Here, even the worst possible cases in the dynamic classifier were, in general, not much worse than the worst cases in the static classifier. In fact, for the Snort ruleset, the difference was negligible. Again, this shows that the dynamic classifier was capable of improving the overall speed of classification while keeping its worst case bounded. In fact, in most cases, it improved the speed of classification while keeping its worst case very close to the static classifier's worst case.

The Snort ruleset has a much smaller standard deviation than the checkerboard ruleset mainly because fewer improvements were performed on it than on the Checkerboard ruleset. Since the Snort rules overlapped, and did not cover the entire search region, fewer changes could be made in many cases. As a result, it was difficult for the search tree based on the Snort ruleset to vary as far from the static worst case as for the Checkerboard ruleset.

The worst possible case only rarely approached the value of 50 set for the program. There are two reasons for this. First, many enhancements changed the value of $max(L(p))$ by more than 1. If these enhancements were attempted when the value of $max(L(p))$ was already getting close to the bound, they would be rejected. This kept the maximum worst-case path in the tests shorter than the bound that had been set. Second, since the enhancements were set to counter-act each other's effects on the worst case (i.e. range combinations will help clean up the increases in $L(p)$ caused by node promotions), it was to be expected that the worst possible case in the dynamic classifier would not stray too far from the worst case in the static classifier. In fact, those few cases where the worst case in the dynamic classifier ended up much larger than the worst case in the static classifier were those in which only one of node promotions or range combinations was turned on.

Figures 6.4 though 6.6 show the cumulative worst case values for all the tests on the checkerboard ruleset, as do Figures A.5 through A.8. The two vertical lines show the worst-case performance of the static classifier and the average value for the worst-case in question (actual or maximum) for the packet trace in question. The monotonically increasing line gives the number of tests, for every measured worst-case value of L, which had that worst-case value, or had a lower worst-case value. The majority of tests show that the dynamic classifier had better worst-cases, or similar

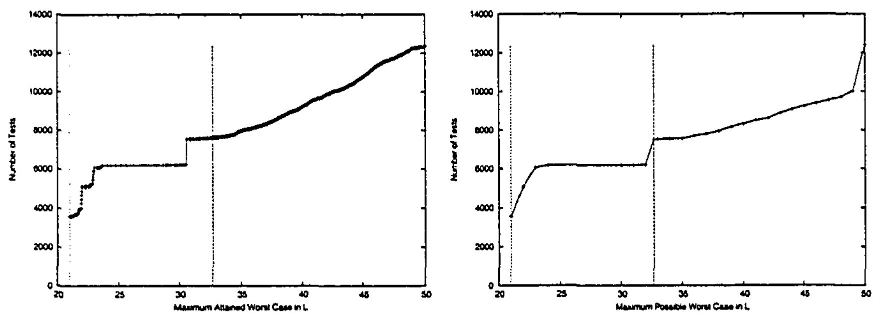

Figure 6.5: (a) Actual and (b) maximum worst-case results for Telus on Checkerboard

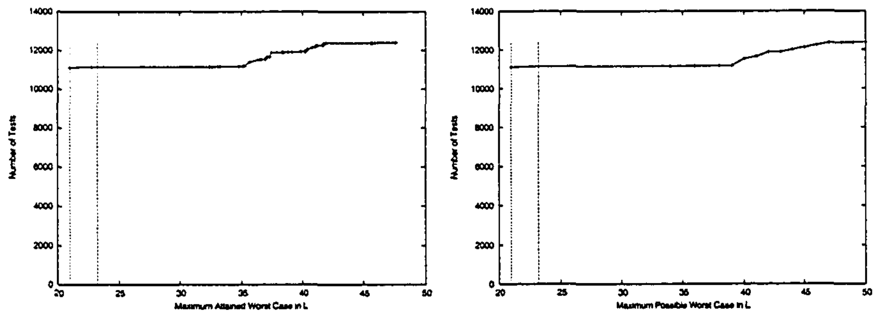

Figure 6.6: (a) Actual and (b) maximum worst-case results for Uniform on Checkerboard

worst-cases, to the static classifier. The cases which had an extremely high worst-case tended to be few in number compared to those below or near the static classifier's worst-case.

Figures A.9 though A.15 show the same data, however, the Snort ruleset was used for the experiments, rather than the checkerboard ruleset. These graphs show essentially the same information as the graphs for the checkerboard ruleset, and will not be discussed further.

6.2.3 Effects of the Individual Improvements

To explore the effects of each individual improvement on overall classification, only the results using the checkerboard ruleset are discussed below. The results for the Snort ruleset are essentially the same. For each trace and for each improvement, four numbers are given. The first two represent the average performance when that improvement was being used. The second two numbers are for when that improvement is turned off.

The overall purpose of this section is not to claim that one improvement allows for greater enhancement in performance than the others, but rather to demonstrate that all the enhancements work together in order to provide the performance boost seen in Section 6.2.1.

As Table 6.5 shows, node promotion had a tendency to increase the average value of L over those tests when it was turned off. The only case where this did not happen was with the clumped10 trace. Once the standard deviation is considered, though, it can easily be seen that the classifier performed essentially the same with or without node promotion. This may seem a strange result. Node

Trace	On Average	2 times S.D.	Off Average	2 times S.D.
Clumped1	9.847	14.659	8.291	12.334
Clumped10	8.992	13.411	9.105	13.796
SetSource1	10.126	12.668	9.952	12.266
SetSource10	10.275	11.170	8.952	9.333
Single	1.348	3.917	1.216	2.393
Telus	4.980	5.401	4.855	5.204
Uniform	15.259	5.452	14.377	3.342

Table 6.5: Effects of Node Promotion on classification

Trace	On Average	2 times S.D.	Off Average	2 times S.D.
Clumped1	8.992	13.411	9.105	13.796
Clumped10	9.875	13.117	9.891	12.856
SetSource1	9.988	12.306	10.089	12.628
SetSource10	9.246	9.622	9.765	10.772
Single	1.255	2.918	1.308	3.547
Telus	5.064	5.419	4.764	5.154
Uniform	14.814	4.591	14.820	4.618

Table 6.6: Effects of Range Reorders on classification

promotions were supposed to improve the performance of the classifier, not hinder it. However, it is important to remember that node promotion was designed to work in tandem with other improvements. In many of the cases where node promotion was enabled, range combinations and range reorders were not. In these cases, node promotion would make changes to the search tree which introduced deficiencies which could not later be fixed up. In the cases where node promotion was turned off, it could not negatively affect the performance of the search. Of course, in these cases, the effects of other improvements could not be altered and fixed through the process of node promotion.

As with node promotions, there was little difference to be seen between those runs which used range reorders and those runs which did not (as shown in Table 6.6). Once the standard deviation is considered, the results were essentially statistically identical. Again, this shows that range reorders on their own did not greatly affect the performance of the classifier. Whether range reorders were turned on or off, little change in the overall performance of the classifier could be seen. In this case, this could be expected as range reorders compensate for themselves. They can be expected to provide a slight benefit as they adapt the search structure to the traffic being seen and then adapt it further if the traffic the classifier was seeing changed. The effect of range reorders themselves on the classifier was slight, however, as range reorders were local to nodes, and did not create wide-ranging changes to the search structure.

Range combinations showed, as demonstrated in Table 6.7, the same behavior as node promotions and range reorders. The difference in the effects of this improvement being turned off versus it being turned on were negligible. As with node promotions, this was due to the fact that range

Trace	On Average	2 times S.D.	Off Average	2 times S.D.
Clumped1	9.070	13.595	9.028	13.617
Clumped10	9.877	12.900	9.889	13.065
SetSource1	10.042	12.467	10.034	12.471
SetSource10	9.507	10.215	9.500	10.232
Single	1.282	3.248	1.282	3.248
Telus	4.849	5.254	4.972	5.332
Uniform	14.817	4.604	14.816	4.604

Table 6.7: Effects of Range Combinations on classification

Trace	On Average	2 times S.D.	Off Average	2 times S.D.
Clumped1	8.833	13.623	15.353	2.644
Clumped10	9.681	12.987	16.023	2.861
SetSource1	9.864	12.523	15.247	1.594
SetSource10	9.364	10.267	13.635	2.727
Single	1.004	0.014	9.607	6.376
Telus	4.725	4.979	10.254	1.763
Uniform	14.789	4.640	15.650	2.916

Table 6.8: Effects of the TCAM on classification

combinations worked in tandem with other improvements to provide the benefits that could been seen in the dynamic classifier. As a result, when range combinations were disabled, the classifier performed worse when node promotions were enabled. And when range combinations were on, the classifier performed worse when node promotions were off. In both cases, there existed situations where the classifier performed more poorly than it otherwise should have, and little difference could be seen in the classifier's performance when only the state of range combinations was considered.

Typically, one enhancement does not contribute significantly to the performance of the classifier on its own. However, this is not the case for the TCAM, as shown in Table 6.8. The TCAM contributed significantly to the performance of the classifier in every case in which there was some pattern to the traffic in question which could be exploited. Whether that pattern was due to the rules used to generate a synthetic trace, or the locality of traffic in a real Internet trace was unimportant. While the other improvements used in isolation made only minor changes to the value of $L(p)$, the TCAM reduced that value to 1, regardless of how much work might be required to search for the rule in the HICuts tree. It is also important to note that the average performance of those tests which did not employ the TCAM was still significantly better than the performance of the static classifier. In other words, while the TCAM certainly aided the performance of the classifier by a large margin, it was also certainly not the only improvement which allowed the dynamic classifier to outperform the static one. As the TCAM did affect the performance of the classifier to such a large extent, it will be discussed in more detail in Section 6.2.5.

Table 6.9 shows the effect of enhanced cuts on the performance of the classifier. Again, there was

Trace	On Average	2 times S.D.	Off Average	2 times S.D.
Clumped1	9.077	13.583	9.021	13.629
Clumped10	9.871	12.962	9.896	13.002
SetSource1	10.054	12.467	10.023	12.471
SetSource10	9.512	10.245	9.496	10.202
Single	1.212	2.346	1.351	3.945
Telus	4.936	5.321	4.886	5.267
Uniform	14.832	4.683	14.801	4.524

Table 6.9: Effects of Enhanced Cuts on classification

Trace	NP & RC Average	2 times S.D.	Neither Average	2 times S.D.
Clumped1	7.270	12.345	9.791	14.292
Clumped10	7.127	11.726	9.796	12.101
SetSource1	7.938	12.279	10.024	12.355
SetSource10	6.952	9.335	9.667	9.912
Single	1.238	2.637	1.316	3.536
Telus	3.057	5.092	5.455	4.702
Uniform	12.377	3.343	15.254	5.432

Table 6.10: Effects of Node Promotion and Range Combination on classification

little difference between the performance of the classifier with and without enhanced cuts, showing that this improvement, by itself, did not improve the performance of the classifier. Rather, it must be used in conjuction with the other enhancements. It is important to note, however, that, while enhanced cuts worked vaguely like a TCAM, the benefits the TCAM afforded were not seen with enhanced cuts. This was simply due to the fact that enhanced cuts could not reduce the search time for any packet down to 1 as a TCAM could, and that enhanced cuts increased by 1 the value of $L(p)$ for all searches not in the enhanced cut. The TCAM did not do this. In fact, the placement of rules in the TCAM allowed the classifier to further reduce the value of $L(p)$ in the HICuts tree. In any case where a TCAM can not be used, enhanced cuts can not be counted on to provide the same benefit to the classifier.

It has repeatedly been mentioned that node promotions and range combinations must work together in order to see the benefits of both. Table 6.10 shows the effect of the node promotion and range combination together versus those cases where only one or the other is in use. In combination with Tables 6.5 and 6.7, this table shows just how beneficial the combination of node promotions and range reorders is.

6.2.4 Memory Use

While memory use was not the main concern of this research it should still be explored. Classification has, like many other endeavors in computing science, a pronounced tradeoff between memory use and time performance [14]. Increased use of memory, and use of better-performing memory,

Ruleset	Trace	Static	Average	2 times Std.Dev.
Checkerboard	Clumped1	1676	4141.213	6255.209
Checkerboard	Clumped10	1676	3605.685	5428.760
Checkerboard	SetSource1	1676	2036.950	2667.689
Checkerboard	SetSource10	1676	4012.184	6323.594
Checkerboard	Single	1676	2037.623	1756.081
Checkerboard	Telus	1676	3675.103	6329.198
Checkerboard	Uniform	1676	2255.930	3665.282
Snort	Clumped1	8043	7052.779	4975.884
Snort	Clumped10	8043	7016.354	2841.146
Snort	SetSource1	8043	7281.260	2758.892
Snort	SetSource10	8043	6720.710	3146.998
Snort	Single	8043	6842.176	3346.934
Snort	Telus	8043	7327.067	8208.435
Snort	Uniform	8043	11846.013	19133.795

Table 6.11: Average memory use results for all tests.

such as storing the entire ruleset in a TCAM, leads to much shorter classification times. As a speed-up in classification times with the dynamic approach can be seen, it can be assumed that the classifier may be using more memory to store the search structure. As classifiers must work within the limits of the memory they are given, it is important to keep track of the use of memory in the dynamic classifier to make sure that the improvements in classification time are not coming at the expense of too much additional memory use.

For the purposes of this research, memory use was calculated by summing the number of nodes in the search tree with the size of the ruleset contained in each node. While this didn't provide an exact measure of the size of the search structure, it was enough for keeping track of how memory use changed during the run-time of the dynamic classifier.

Table 6.11 gives the average memory use of the classifier over all tests. For every test run, the amount of memory the classifier was using was calculated immediately after every iteration of restructuring. The average value over all these iterations was then calculated. All the average values for every run were grouped by the ruleset and packet trace used, and another average was calculated. The results in the table show that, on average, the average amount of memory being used by the dynamic classifier after any given iteration was not that much different than the amount of memory used by the static classifier. In fact, in many of the tests involving the Snort ruleset, the average size of the search structure was less than that of the static classifier. As with the worst-case results, the vast majority of tests which resulted in significantly higher memory use were those in which only one of node promotions and range combinations was used. In these cases, the additional nodes created by one improvement during one restructuring could not be later reclaimed by the corresponding improvement during another restructuring. This resulted in far more memory being used than would otherwise be necessary.

For every test run for this thesis, the absolute worst-case use of memory was also obtained.

Ruleset	Trace	Static	Average	2 times Std.Dev.
Checkerboard	Clumped1	1676	5495.639	8946.593
Checkerboard	Clumped10	1676	4884/979	8252.481
Checkerboard	SetSource1	1676	2466.022	4669.655
Checkerboard	SetSource10	1676	5393.535	9217.998
Checkerboard	Single	1676	2116.152	1999.778
Checkerboard	Telus	1676	5543.764	9588.154
Checkerboard	Uniform	1676	2547.881	5211.307
Snort	Clumped1	8043	20909	25733.038
Snort	Clumped10	8043	8374.377	2103.018
Snort	SetSource1	8043	8047.500	63.003
Snort	SetSource10	8043	8500.783	2487.931
Snort	Single	8043	8057.077	51.595
Snort	Telus	8043	14393.382	19280.177
Snort	Uniform	8043	16331.508	28423.322

Table 6.12: Worst case memory use results for all tests.

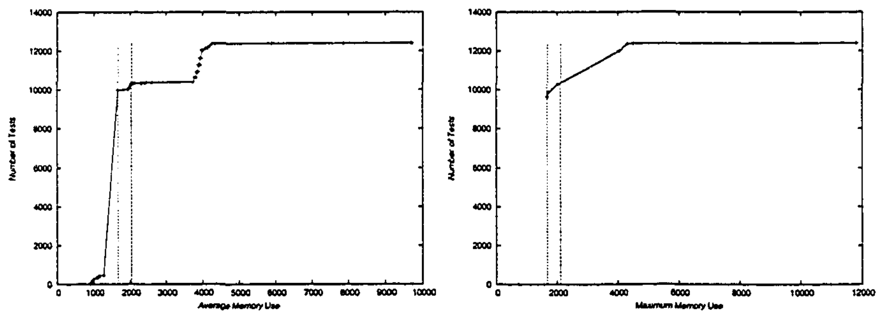

Figure 6.7: (a) Average and (b) maximum memory use results for Single on Checkerboard

Since memory use can only change after a restructuring, this involved keeping track of the largest value encountered while maintaining the data to calculate the average size of the search structure. Table 6.12 gives the average worst-case memory size for every combination of ruleset and packet trace tested. Here it is clear that, while the average case memory use of the classifier was often slightly more or slightly less than the memory use of the static classifier, the maximum use of memory could be significantly greater than the memory use of static HICuts. This was especially true in the case of the Snort ruleset. Since the Snort ruleset contains many rules which overlap each other, changes to the search structure could easily create very large subtrees with a large number of partially overlapping rules in them. These newly created subtrees would be reduced in size on later iterations of the restructuring algorithm. However, the fact that the search tree could occasionally grow to several times larger than its average size might be of some concern in practice. However, it would not be difficult to extend the technique which prevented the worst-case bound from being violated to also ensure a space bound was obeyed.

Figures 6.7 though 6.9 (as well as Figures A.16 through A.19) show the cumulative memory use values for all the tests on the checkerboard ruleset. The two vertical lines show the memory use of

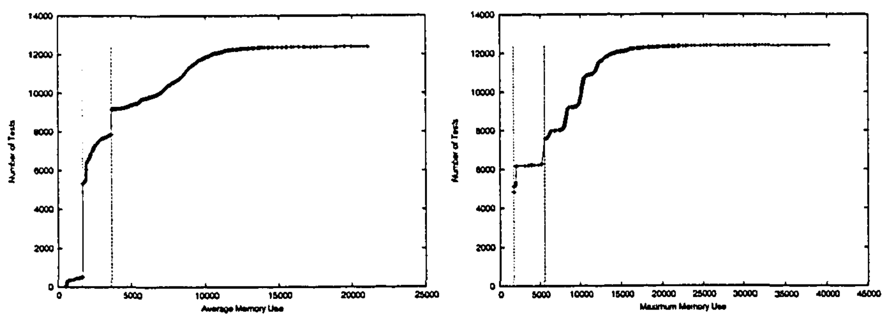

Figure 6.8: (a) Average and (b) maximum memory use results for Telus on Checkerboard

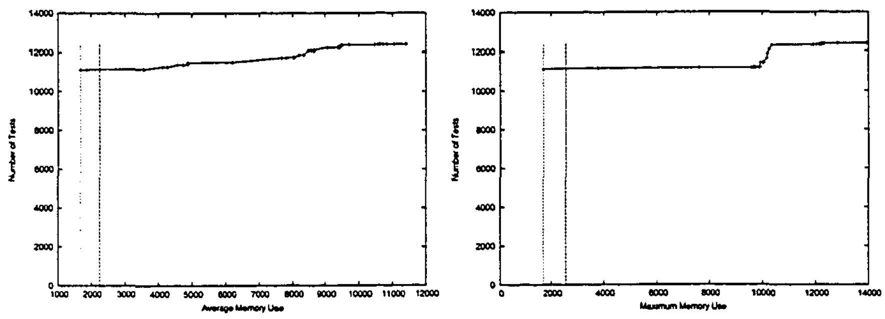

Figure 6.9: (a) Average and (b) maximum memory use results for Uniform on Checkerboard

the static classifier and the average value of memory use for the packet trace and metric (average memory use or worst-case memory use) in question. The monotonically increasing line gives the number of tests, for every measured value of memory use, which used at most that much memory. The vast majority of tests of the dynamic classifier used an amount of memory close to the average and close to the memory use of the static classifier. There were some outliers, but they were few in number, and all involved those cases in which the ability of the search structure to fix the changes it had made had been compromised through the disabling of some of the enhancements used by the dynamic classifier.

Figures A.23 though A.22 show the same information as the graphs mentioned above. In this case, however, the Snort ruleset was used for the experiments, rather than the checkerboard ruleset. The data here are not different enough from that of the checkerboard tests to warrant further comment.

6.2.5 TCAM Hits - Locality of Traffic

It was to be expected that the TCAM would prove to be useful in situations where there is locality of traffic. The results described in Section 6.2.3 confirmed this. In other words, the classifier should have been able to make good use of the TCAM in the real-world trace, as well as most of the synthetic traces. The uniform trace should have made much less use of the TCAM as it was not constructed with locality in mind. However, there were cases where the TCAM was useful even

Ruleset	Trace	Average	2 times Std.Dev.
Checkerboard	Clumped1	43.9%	93.7%
Checkerboard	Clumped10	40.9%	85.3%
Checkerboard	SetSource1	36.5%	87.1%
Checkerboard	SetSource10	33.7%	80.7%
Checkerboard	Single	96.7%	35.3%
Checkerboard	Telus	97.2%	93.5%
Checkerboard	Uniform	5.78%	23.5%
Snort	Clumped1	0.0596%	0.197%
Snort	Clumped10	0.00536%	0.0328%
Snort	SetSource1	0.0459%	0.197%
Snort	SetSource10	0.0533%	0.219%
Snort	Single	0.000%	0.000%
Snort	Telus	50.389%	50.887%
Snort	Uniform	0.146%	0.458%

Table 6.13: The average number of hits in the TCAM, as a percentage of packets classified.

when there was no explicitly constructed structure to exploit. These were fortunate occurrences because they allowed the classifier to make quick matches in the TCAM.

Table 6.13 shows the average number of hits in the TCAM for every packet trace on each of the two rulesets. These results show the great benefit that the TCAM afforded the classifier when it saw a single repeated packet header. In particular, the tests performed on the Telus trace clearly show the benefits of using a TCAM. Since there is locality in Internet traffic, the TCAM was able to exploit that locality and store rules which matched the patterns in the traffic, giving the classifier the ability to short-cut every search which fell in the same area of the search space. On average, the Telus trace performed as well as the single trace on the checkerboard ruleset, albeit with a much greater standard deviation. The Telus trace also performed significantly better than all other traces on the Snort ruleset. On both the checkerboard ruleset and the Snort ruleset the locality of Internet traffic in the Telus trace allowed the TCAM to perform exactly as was expected. Rules which could be expected to be matched again in a short period of time were stored and were matched quickly in the TCAM, rather than in the much slower tree structure created by HICuts.

The single trace was 5000000 headers in length, but there were not always 4999999 hits in the TCAM simply because many of the tests involved thresholds on the entry of the rule into the TCAM. This also explains the high standard deviation. Some of the tests which used a high threshold on the entry of a rule into the TCAM affected the performance of the TCAM enough that the deviation in the results grew to reflect this.

The number of TCAM hits for the Snort ruleset were not nearly as high as those for the checkerboard ruleset mainly because there were regions in the Snort search space with no assigned rules. When these regions were 'matched', there was no rule to be promoted into the TCAM to aid in later searches. This is most explicitly shown by the single trace on the Snort ruleset. Here, because the packets generated always managed to miss all of the rules in the Snort ruleset, there were absolutely

Rate	Mean	2 times Std.Dev.	Rate	Mean	2 times Std.Dev.
100	1.281	3.243	100	10.811	4.697
200	1.282	3.250	200	10.811	4.697
300	1.282	3.250	300	10.811	4.697
400	1.282	3.250	400	10.811	4.697
500	1.282	3.250	500	10.811	4.697

Table 6.14: *The effects of the rate of forgetting on the value of L for the Single trace on the (a) Checkerboard and (b) Snort ruleset.*

Rate	Mean	2 times Std.Dev.	Rate	Mean	2 times Std.Dev.
100	5.243	4.994	100	7.499	4.881
200	4.966	5.311	200	8.160	4.468
300	4.858	5.301	300	8.203	4.546
400	4.681	5.390	400	8.182	4.556
500	4.797	5.408	500	8.102	4.496

Table 6.15: The effects of the rate of forgetting on the value of L for the Telus trace on the (a) Checkerboard and (b) Snort ruleset.

no hits in the TCAM. This suggests that the use of a TCAM for packet classification should be done with rulesets which cover the entire search space. Since rules in rulesets can be prioritized, a rule to cover the entire search space and with the lowest priority can be created. If such a rule had been created for the Snort ruleset then the Single trace would have experienced identical performance with the Snort ruleset as it did when tested with the checkerboard ruleset. It is important to note, however, that creating such an all-encompassing rule could result in a larger search tree. Such a rule would exist in every node of the HICuts tree and could detrimentally affect HICuts' terminating condition.

In the case of the uniform trace on the checkerboard ruleset, there are only a very few hits in the TCAM. Those that we see are artifacts of the random method used to generate the trace. Every now and then packet headers will be produced in the same region of the ruleset within the time interval needed to register a hit in the TCAM, or to push the rule's hits high enough to get it over the insertion threshold. Should the second case occur, it is almost guaranteed that the inserted rule would never leave the TCAM, and would match every additional packet in that rule's region. To have rules hit often enough to be inserted in the TCAM in spite of forgetting was quite rare with the uniform trace, but it did happen and this had a very positive effect on the overall performance of the classifier.

The other traces will not be explicitly discussed, but are included in Table 6.13 for completeness.

6.2.6 Effects of the Rate of Forgetting on Learning

As mentioned in Section 4.6, it is possible that the rate at which the classifier forgets the data it sees can affect the performance of the classifier in real-world use. In order to explore this possibility, the classifier tests were all run using five different rates of forgetting. The classifier could forget once

Rate	Mean	2 times Std.Dev.	Rate	Mean	2 times Std.Dev.
100	17.106	5.606	100	10.436	3.923
200	14.667	2.754	200	10.431	3.433
300	14.153	3.562	300	10.117	3.976
400	14.098	3.709	400	10.406	3.718
500	14.089	3.737	500	11.103	2.244

Table 6.16: The effects of the rate of forgetting on the value of L for the Uniform trace on the (a) Checkerboard and (b) Snort ruleset.

every 100 packets classified, or forget every 200, 300, 400 or 500 packets classified. While this may seem a rather course granularity between rates of forgetting, it was quickly discovered that, for the most part, the rate of forgetting had little to no impact on the performance of the classifier. This is a very encouraging result, as it means that forgetting can be tuned in order to maximize processor time used for classification rather than in order to prevent undesirable effects in the classifier due to it improperly forgetting the traffic it has seen.

The test results detailed in Tables 6.14 through 6.16 (as well as in Tables A.1 through A.4) all show essentially the same thing. Namely, that the rate at which the classifier forgot the traffic it had seen had very little effect on its overall performance. While there were slight differences in the overall average performance of the classifier under different rates of forgetting, they were too small to be of consequence, especially when the standard deviation was considered. The optimal value for forgetting cannot be determined from these graphs, and, judging from them, is likely dependent on both ruleset and traffic. The effects of the rate of forgetting on the performance of the dynamic classifier cannot be ignored completely. Two reasons why this rate must still be considered, including one which leads directly from test results, are presented below.

Despite the general performance of the classifier remaining almost constant regardless of the rate of forgetting, the overall possible effects of the rate of forgetting on the dynamic classifier's performance cannot be ignored. In particular, Table 6.16a shows that, in some cases, the rate of forgetting had a dramatic effect on the overall performance of the classifier. With a high rate of forgetting, in this case forgetting once every 100 packets classified, there was a sharp drop in the overall performance of the dynamic classifier. In this case, the classifier was losing information too quickly. As a result, it did not retain enough information to be able to ascertain that the traffic it was seeing was uniform. Rather, it believed that there were patterns in the traffic. The classifier attempted to perform 'optimizations' which were not, in fact, optimizations. The optimizations improved the classifier's performance on those packets which it believed were being seen more frequently than others. This degraded the classifier's performance on all other packets. But, since the packets were uniformly distributed, this led to an overall performance degradation.

6.3 Summary of Results

The results detailed in this chapter show that it is possible to design and implement a dynamic classifier which learns from the traffic it sees and can adapt its search structure to make it better suited to the traffic that it sees. While it certainly is true that some specific cases made the performance of the classifier slightly worse than that of the static classifier, these are all cases in which only some of the improvements were in use, and the classifier could not deal with changes in traffic, nor with the undesirable effects of some of the improvements. When all enhancements are used, the dynamic classifier performs significantly better than the static classifier, while obeying a bound on its worst case classification time and not using much more memory than the static classifier.

Chapter 7

Conclusions

7.1 Conclusions

Conventional wisdom in packet classification states (most works cite [20] as the source of this claim) that, when considering the time performance of a classifier, only the worst case performance should be considered. In addition, many in the field have suggested that learning from traffic cannot benefit classification in the least or, even worse, that learning would be detrimental to classification. However, there is no recorded proof of such a statement anywhere in the literature. The general argument against learning from traffic seems to stem from the belief that the worst-case performance of structures designed by learning algorithms cannot be controlled. The primary intent of this thesis was to demonstrate that packet classification can, in fact, be improved by learning from traffic, and that learning does not necessarily imply that worst-case bounds will be violated by any such algorithm.

This thesis presented a dynamic form of the HICuts algorithm which, using information gathered from the traffic that it is seeing, is able to periodically update itself in an attempt to better match its search structure to the traffic due to both short-term locality ([10], [19]) and long-term trends in traffic. This dynamic version of the HICuts algorithm incorporates several restructuring techniques: node promotion, range reorders and range combinations, and extended cuts. These techniques capitalize on discrepancies in the number of hits certain regions of the search space receive, and, as such, take into consideration long-term trends in traffic. In addition, a TCAM was used as a form of transposition table (a concept described for chess-playing software in [15]) to better exploit the locality of traffic. Since locality is generally short lived, the periodic restructuring performed by the first four techniques cannot make use of it. The TCAM compensates for this deficiency.

The new classifier was then put through a barrage of tests, using both synthetic and real-life packet traces and rulesets. The results from the real-world traces are very encouraging. On average, over all the tests, the dynamic classifier required significantly less computation to classify the same number of packets as the static classifier. Despite the fact that the dynamic rearrangements might be expected to make the wrong choices from time to time, as it occasionally did with the uniform packet trace, in general, this does not occur. Or, at the very least, the poor restructuring choices that the classifier made weren't enough to cause it to perform worse than the original static classifier would have. These results are extremely encouraging, and give strong incentive to go ahead with several other optimizations and general improvements to make the dynamic classifier even more responsive to the traffic it is seeing.

The synthetic traces were used to cover special cases which would never be seen in real life, but which allow for the evaluation of the performance of the classifier in extreme cases. The results from the synthetic traces show that the new dynamic classifier can significantly outperform the static classifier when there is structure in the traffic it sees. The dynamic classifier never performs worse than the static classifier by more than a slight degree. This performance degradation happens when the dynamic classifier makes an 'improvement' which in fact does not improve the overall performance of the classifier. In these cases, the dynamic classifier is able to correct its mistake at the next iteration, so only a slight performance degradation over the static classifier is seen. However, this degradation is so slight, and its cause is so easily corrected that, in the long run, the dynamic classifier performs at least as well, and in the vast majority of cases, much better than a static classifier. Many of the poorly-performing tests also had no way of fixing the mistakes introduced by the classifier, especially those in which only one of node promotions and range reorders was performed.

In addition, the worst-case performance, both the absolute possible worst-case possible in the new search structure, and the actual worst-case path followed by the classifier during the tests, was recorded for each test. The results show that it is possible to bound the worst-case performance of the dynamic classifier. In other words, the classifier's average-case performance can be improved while the worst-case is held to some pre-set limit to ensure that all packets can be classified in a reasonable amount of time.

7.2 Future Work

There is still much to be done in the way of improving packet classification in order to better utilize information which can be obtained through the analysis of Internet traffic.

First, any IDD based classification system, such as HICuts, could benefit from the IDD minimization techniques outlined in [7] and [8]. HICuts does not always find the minimum possible path (and, as a result, neither do the improvements outlined herein). Using techniques to shrink IDDs down to more manageable sizes, one could start with a smaller initial search tree, reduce the size of the altered search trees during actual classification, and compensate for the negative effects of some of the update techniques.

In addition, there are still several parameters which can guide the restructuring of the classifier which have yet to be manipulated. The choice of formula used when determining when to perform node promotions, range reorders and range combinations was, to a degree, arbitrary. It may be that different formulas here may, in fact, improve performance and that there may be a way to determine the optimal value for this threshold from the traffic the classifier is seeing.

There are several other packet classification algorithms (many are outlined in [18]) which may also benefit from the techniques described in this paper. In particular, several of the Trie techniques ([18], [24]) and other heuristic-based classifiers (such as Recursive Flow Classification [16]) might benefit from learning. Since other heuristic classifiers use search structures which are very different from HICuts, it is possible that the techniques here cannot be used directly. This leads to two fairly distinct areas of future research.

First, and the more ambitious of the two, is the question as to whether or not general learning techniques can be created which could be applied to virtually any type of packet classifier, regardless of the search structure and search technique which it used. Certainly, having a set of general techniques is far more useful than having to create unique learning techniques for every possible classifier. Second is the question of whether techniques be developed for other classification algorithms so that they, too, can learn from traffic in order to improve their search behavior. It seems highly unlikely that only HICuts would benefit from learning from traffic, however, the results in this thesis do not preclude, at the moment, that from being a possibility. Recent work in the literature ([4]) shows that some learning techniques can be used to improve other forms of packet classification and lends credence to the idea that learning is not applicable only for a few types of classifier, but for all classifiers in general.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] K. Andrusky and M. H. MacGregor. Improving packet classification: Learning from traffic. In *Internet Technologies and Applications*, September 2005.

[3] Florin Baboecsu and George Varghese. Scalable packet classification. In *Proceedings of ACM Sigcomm*, pages 199–201, August 2001.

[4] A. Bergamini and L. Kencl. Network of shortcuts: An adaptive data structure for tree-based search methods. In *Networking*, pages 523–535, May 2005.

[5] M. M. Buddhikot, S. Suri, and M. Waldvogel. Space decomposition techniques for fast layer-four switching. In *Conference on Protocols for High Speed Networks*, pages 25–41, August 1999.

[6] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1st edition, 1997.

[7] M. Christansen and E. Fleury. An interval decision diagram based firewall. In *3rd International Conference on Networking*, February 2004.

[8] M. Christiansen and E. Fleury. An MTIDD based firewall using decision diagrams for packet filtering. *Telecommunication Systems*, 27(2-4):297–319, October 2004. Kluwer Academic Publishers.

[9] I. Chvets and M.H. MacGregor. Multi-zone caches for accelerating IP routing table lookups. In *High Performance Switching and Routing (HPSR)*, pages 121–126, 2002.

[10] K. Claffy, G. C. Polyzos, and H. W. Braun. Traffic charateristics of the T1 NSFNET backbone. In *IEEE INFOCOM 93*, volume 3, pages 885–892, 1993.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw Hill, second edition, 2001.

[12] M. de Berg, M. van Krevald, and M. Overmans. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second revised edition, 2000.

[13] Antoine de Saint-Exupery. Richard Howard, Trans. *The Little Prince*. Harcourt, 1943.

[14] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. *IEEE Infocomm*, pages 1193–1202, 2000.

[15] R. D. Greenblatt, D. E. Eastlake, and S.D. Crocker. The greenblatt chess program. In *Fall Joint Computer Conference*, volume 31, pages 801–810, 1967.

[16] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proc. Sigcomm, Computer Communication Review*, volume 29, pages 147–160, September 1999.

[17] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. *IEEE Micro*, 20(3):34–41, January/February 2000.

[18] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network Special Issue*, 15(2):24–32, March/April 2001.

[19] R. Jain. Characteristics of destination addreses locality in computer networks: a comparison of caching schemes. *Computer Networks and ISDN systems*, 18:243–54, May 1990.

[20] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *ACM Computer Communication Review*, 28(4):203–214, September 1998.

[21] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. Technical Report CS2003-0736, UCSD, 2003.

[22] Snort: The open source network intrusion detection system. URL: http://www.snort.org/.

[23] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM Sigcomm*, pages 135–146, September 1999.

[24] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM Sigcomm*, pages 203–214, September 1998.

[25] K. Strehl and L. Thiele. Symbolic model checking using interval decision diagrams. Technical Report 40, Computer Engineering and Networks Lab, Swiss Federal Institute of Technology, 1998.

[26] Masanori Uga, Masaaki Omotani, and Kohei Shiomoto. A high-speed packet classification using TCAM. *IEICE Trans. Communication*, E85-B(9):1766–1773, September 2002.

[27] P. Wang, C. Chan, S. Hu, and C. Lee. Performance improvement of packet classification by using lookahead caching. *IEICE Transactions on Communications*, E87-B(2):377–379, February 2004.

Appendix A

Additional Results

The following pages contain all the data collected for results not explicitly mentioned in Chapter 6. Chapter 6 only contains graphs for the single, Telus and uniform traces, and only for the checkerboard ruleset. The graphs and charts given here show the details of the data collected for the other traces, which are often mentioned in passing in the charts in Chapter 6. This data is not necessary for description of the results of this work, but is included here for completeness.

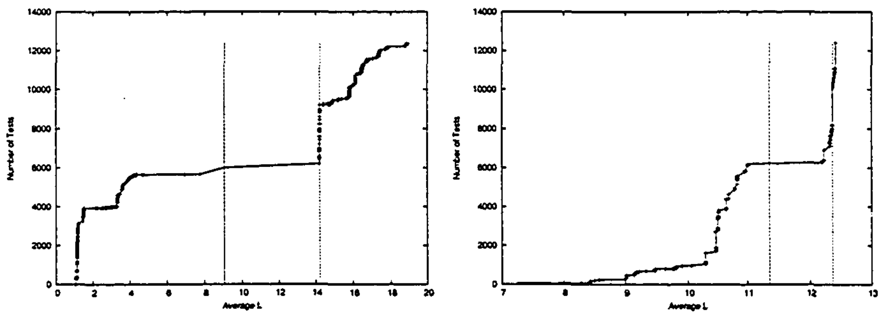

Figure A.1: Average values of L for Clumped1

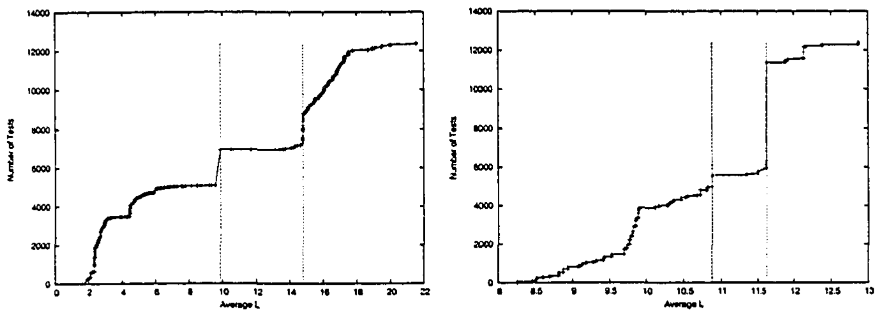

Figure A.2: Average values of L for Clumped10

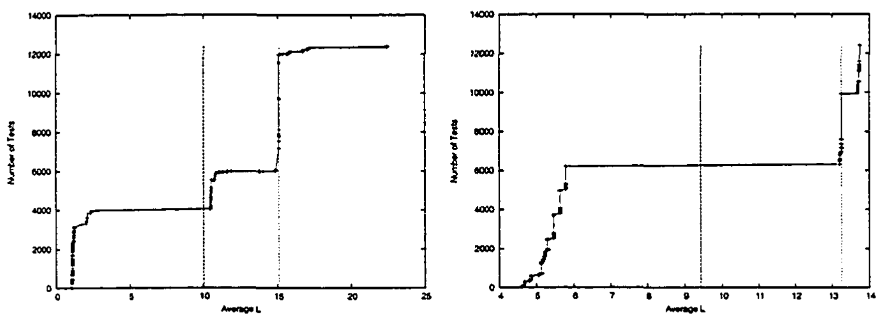

Figure A.3: Average values of L for SetSource1

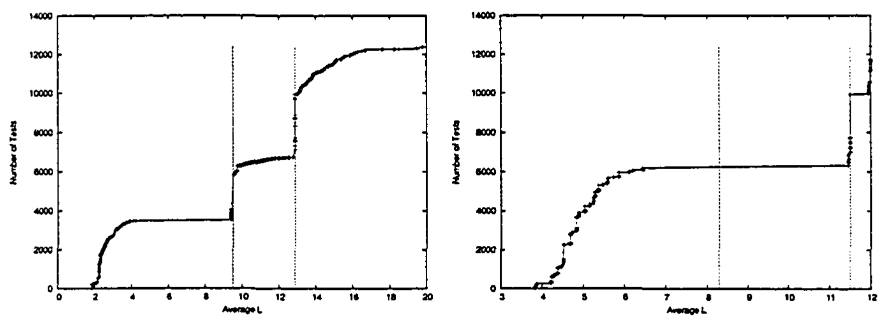

Figure A.4: Average values of L for SetSource10

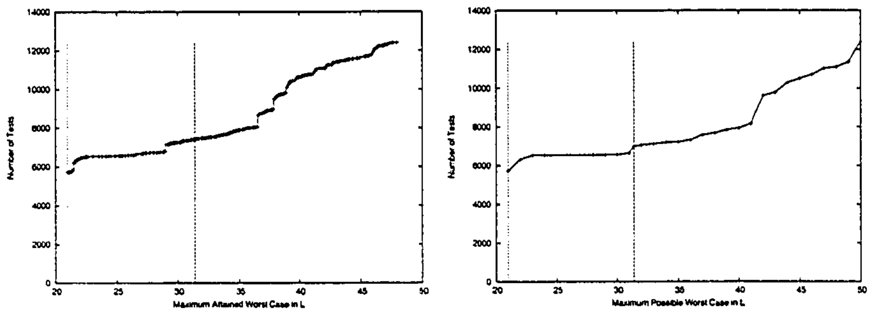

Figure A.5: (a) Actual and (b) maximum worst-case results for Clumped1 on Checkerboard

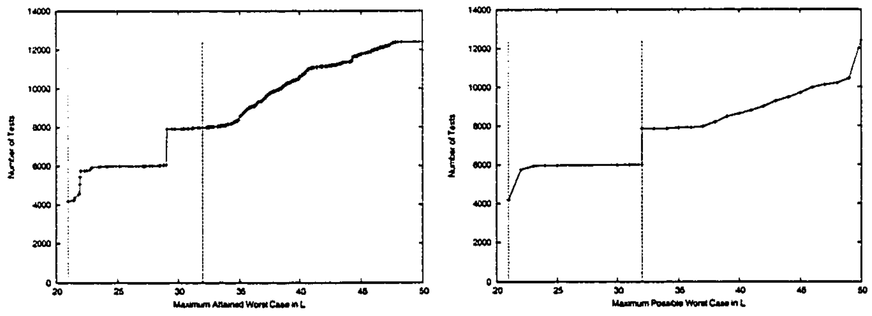

Figure A.6: (a) Actual and (b) maximum worst-case results for Clumped10 on Checkerboard

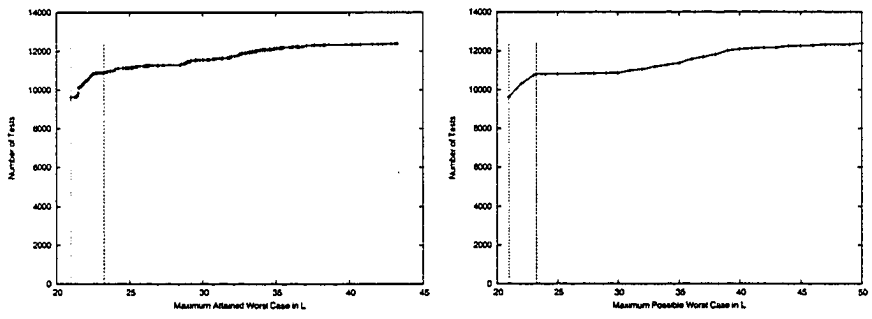

Figure A.7: (a) Actual and (b) maximum worst-case results for SetSource1 on Checkerboard

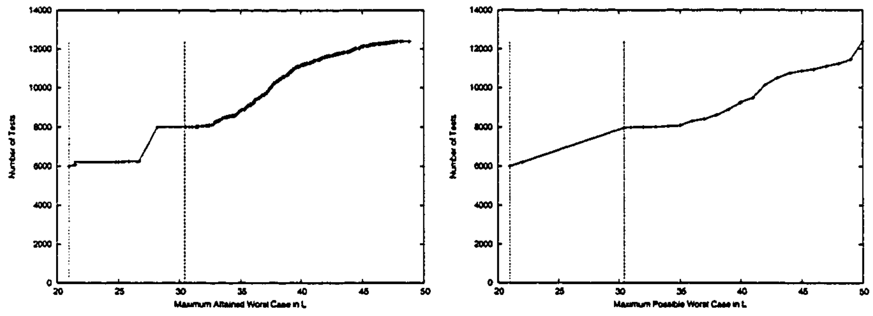

Figure A.8: (a) Actual and (b) maximum worst-case results for SetSource10 on Checkerboard

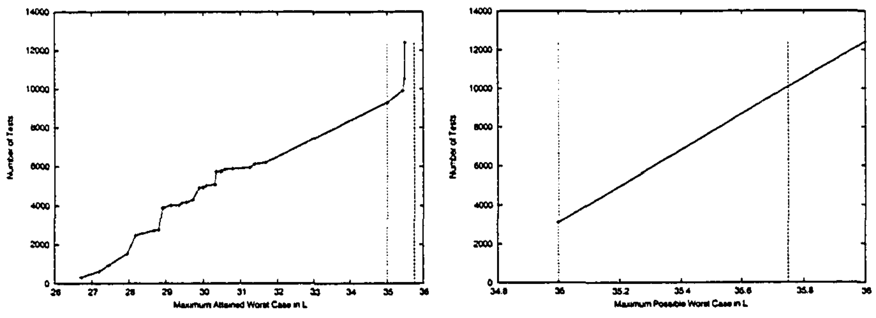

Figure A.9: (a) Actual and (b) maximum worst-case results for Clumped1 on Snort

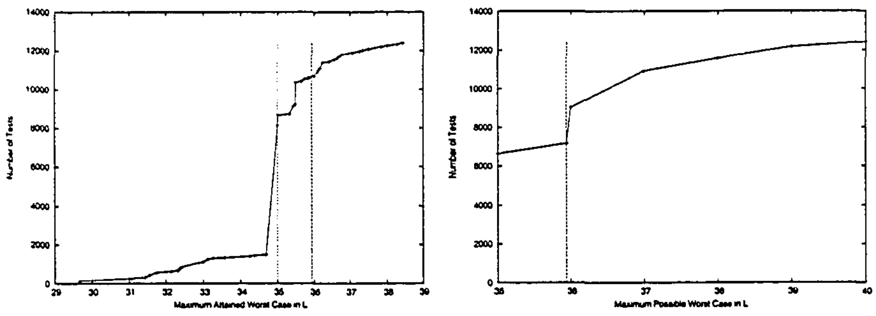

Figure A.10: (a) Actual and (b) maximum worst-case results for Clumped10 on Snort

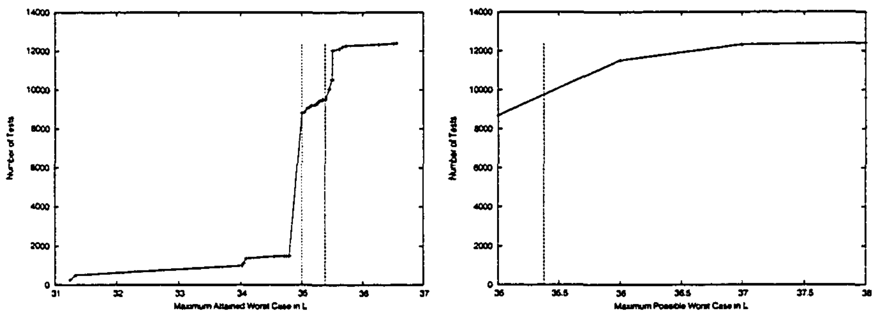

Figure A.11: (a) Actual and (b) maximum worst-case results for SetSource1 on Snort

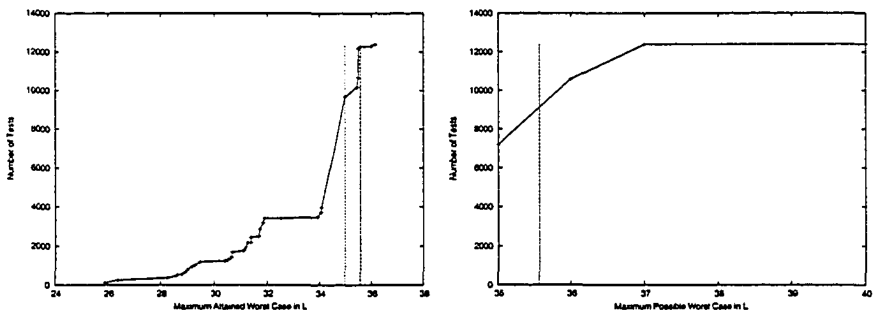

Figure A.12: (a) Actual and (b) maximum worst-case results for SetSource10 on Snort

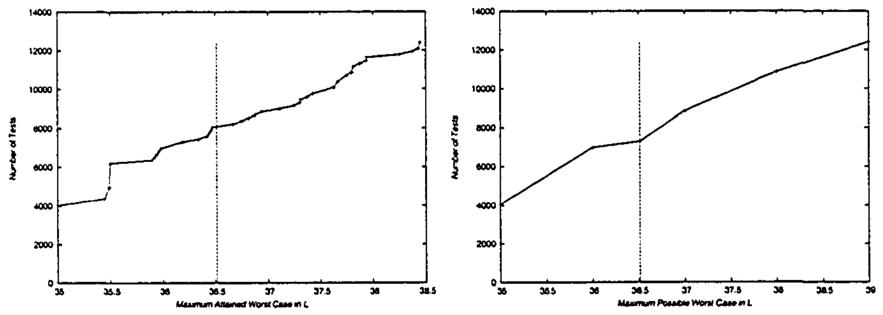

Figure A.13: (a) Actual and (b) maximum worst-case results for Single on Snort

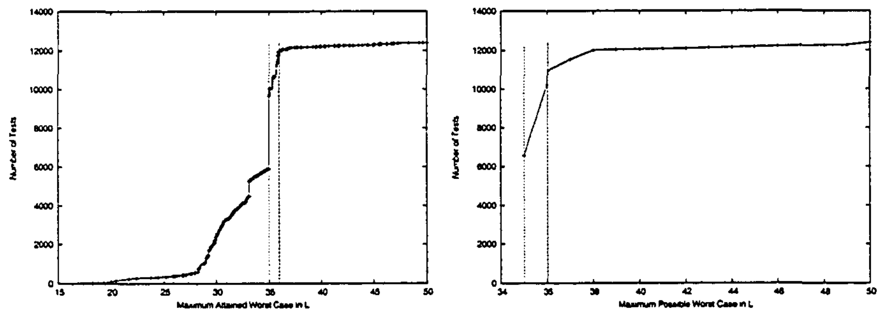

Figure A.14: (a) Actual and (b) maximum worst-case results for Telus on Snort

g=

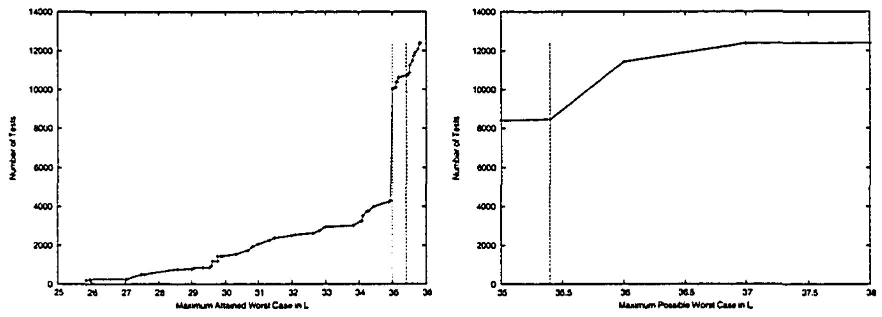

Figure A.15: (a) Actual and (b) maximum worst-case results for Uniform on Snort

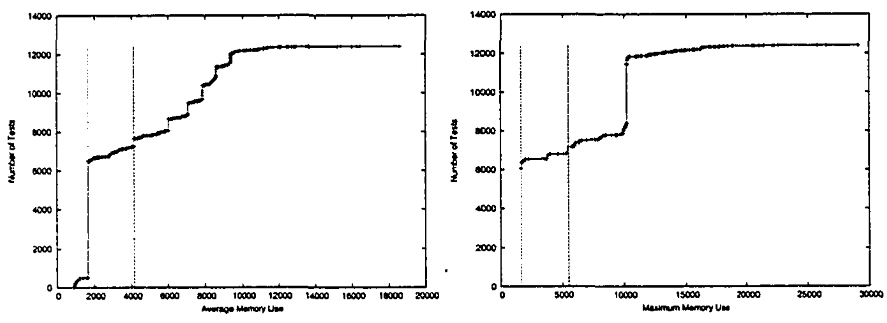

Figure A.16: (a) Average and (b) maximum memory use results for Clumped1 on Checkerboard

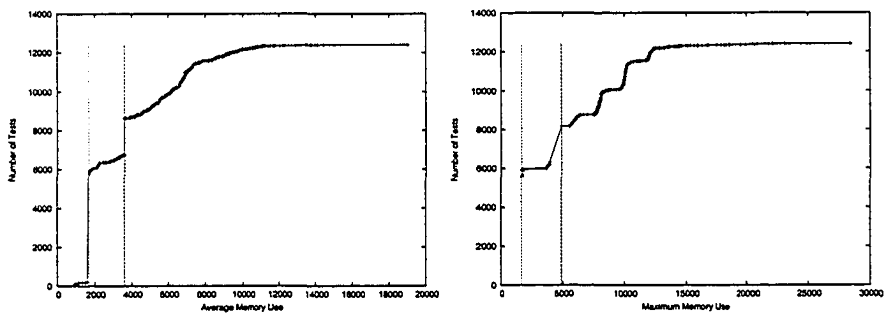

Figure A.17: (a) Average and (b) maximum memory use results for Clumped10 on Checkerboard

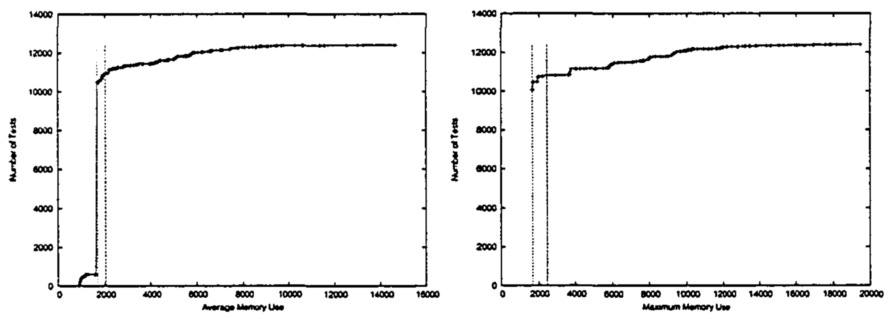

Figure A.18: (a) Average and (b) maximum memory use results for SetSource1 on Checkerboard

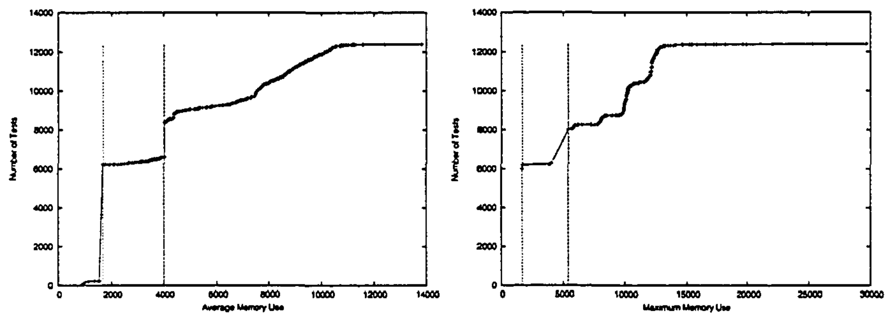

Figure A.19: (a) Average and (b) maximum memory use results for SetSource10 on Checkerboard

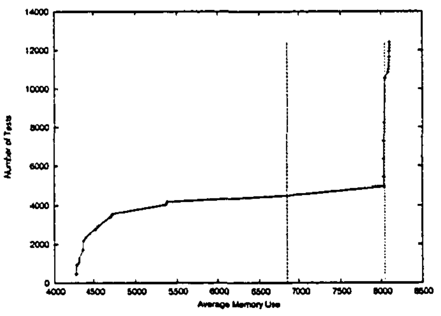

Figure A.20: (a) Average and (b) maximum memory use results for Single on Snort

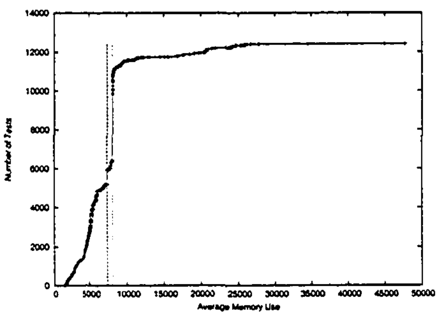

Figure A.21: (a) Average and (b) maximum memory use results for Telus on Snort

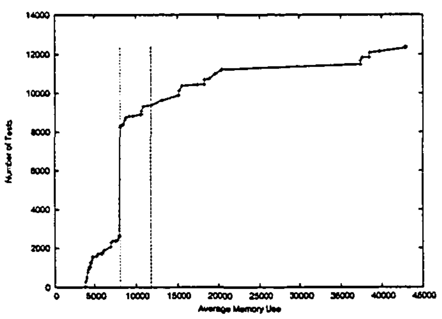

Figure A.22: (a) Average and (b) maximum memory use results for Uniform on Snort

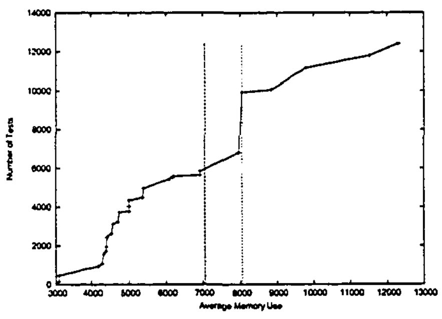

Figure A.23: (a) Average and (b) maximum memory use results for Clumped1 on Snort

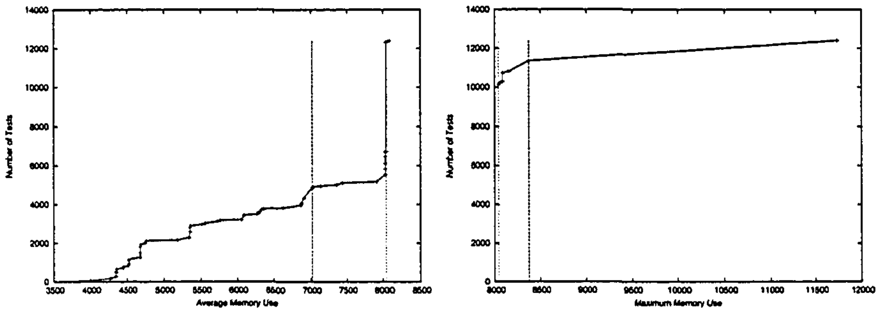

Figure A.24: (a) Average and (b) maximum memory use results for Clumped10 on Snort

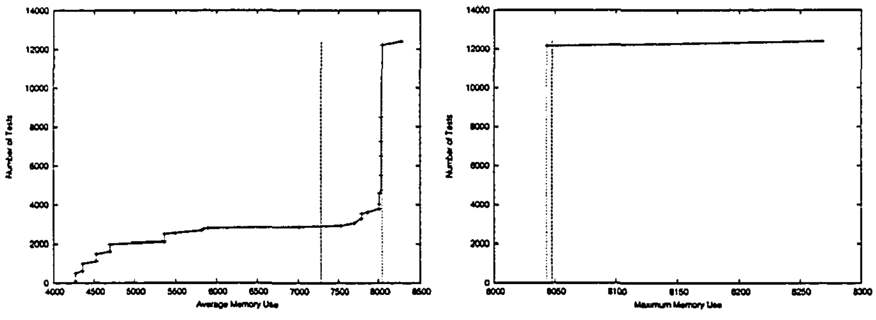

Figure A.25: (a) Average and (b) maximum memory use results for SetSource1 on Snort

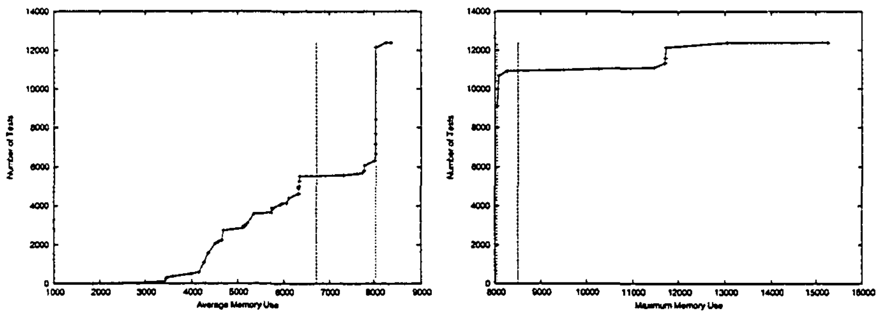

Figure A.26: (a) Average and (b) maximum memory use results for SetSource10 on Snort

Rate	Mean	2 times Std.Dev.	Rate	Mean	2 times Std.Dev.
100	8.975	13.581	100	11.260	2.428
200	9.178	13.778	200	11.376	2.150
300	9.052	13.340	300	11.375	2.162
400	9.058	13.667	400	11.373	2.158
500	8.985	13.658	500	11.379	2.126

Table A.1: The effects of the rate of forgetting on the value of L for the Clumped1 trace on the (a) Checkerboard and (b) Snort ruleset.

Rate	Mean	2 times Std.Dev.	Rate	Mean	2 times Std.Dev.
100	9.805	12.890	100	10.587	2.311
200	9.645	12.869	200	10.602	2.189
300	10.515	12.841	300	11.194	2.116
400	9.475	12.9871	400	11.071	1.824
500	10.375	13.238	500	10.975	1.911

Table A.2: The effects of the rate of forgetting on the value of L for the Clumped10 trace on the (a) Checkerboard and (b) Snort ruleset.

Rate	Mean	2 times Std.Dev.	Rate	Mean	2 times Std.Dev.
100	9.959	12.233	100	9.447	8.108
200	9.950	12.269	200	9.333	7.849
300	10.389	13.251	300	9.457	8.096
400	9.957	12.2738	400	9.455	8.093
500	9.944	12.285	500	9.455	8.084

Table A.3: The effects of the rate of forgetting on the value of L for the SetSource1 trace on the (a) Checkerboard and (b) Snort ruleset.

Rate	Mean	2 times Std.Dev.	Rate	Mean	2 times Std.Dev.
100	9.432	10.466	100	8.114	7.278
200	9.638	10.442	200	7.995	7.024
300	10.049	9.947	300	8.533	6.502
400	9.647	10.001	400	8.298	6.904
500	8.522	9.926	500	8.492	6.510

Table A.4: The effects of the rate of forgetting on the value of L for the SetSource10 trace on the (a) Checkerboard and (b) Snort ruleset.