Adaptive Gaussian-credit Probing Sequence for Packet Classification in Computer Communication Networks

By

Mohamed H. Jayeh

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Supervisor: Dr. Kui Wu

ABSTRACT

The task of classifying and routing packets is a constant challenge in designing network routers. This task involves parsing packet headers and sequentially probing the memory for a best match amongst pre-existing entries in a routing table or a classifier. The entries are a set of filters or rules. The router relies on these rules to decide about the destination of the packet. Once the best matching rule is found the rule is applied to the packet. This task becomes challenging if the size or the number of filters to be probed in the classifier is large. In this thesis, we introduce a new adaptive probing sequence to probe such classifiers. When routers are trained for a period, they become adaptively capable of capturing packet header statistics as seen by the classifier. Routers can then utilize these statistics to dynamically devise future probing sequences. Performance evaluation demonstrates that finding a matching rule in one memory probe is attainable, if a router is trained according to the proposed probing technique.

Table of Contents

ABSTRACT i
Table of Contents ii
List of Tables iv
List of Figures v
Acronyms vii
Acknowledgments ix
Dedication x
1 Introduction 1
1.1 Motivation and Contributions 5
2 Background 8
2.1 Problem Formulation 8
2.2 Structure-Based Techniques 8
2.2.1 Overlapping Filters 9
2.2.2 Geometry-Based Techniques: Cross-Producting 10
2.2.3 Hash-Based Techniques: Tuple Space Search 10
2.2.4 Heuristic Techniques: Hierarchical Intelligent cuttings 12
2.3 Traffic-Based Techniques 13
2.3.1 Network Traffic Locality 13
2.3.2 Temporal Locality 13
2.3.3 Spatial Locality 14
2.3.4 Locality Measurements 14
2.4 The Cache Referencing Technique 15
2.4.1 Models for Cache Reference Behavior 16
2.4.2 Cache Replacement Algorithms 17
2.4.3 Cache Organization 18
3 Linear Probing 19
3.1 Linear Probing in Static Classifiers 19
3.2 Behavior and Analysis of LP in Static Classifiers 20
3.3 Linear Probing in Dynamic Classifiers 27
3.4 Behavior and Analysis of LP in Dynamic Classifiers 28
3.5 Concluding Remarks 31

4 The Adaptive Gaussian-credit Probing Sequence 33
4.1 AGPS in Static Classifiers 33
4.2 Initial Values for iPMF 35
4.3 Indexed-credit Update Mechanism for iPMF 38
4.4 The *Indexed* Gaussian-credit Updates for iPMF 42
4.5 The Gaussian-credit Updates for iPMF 47
4.6 Behavior and Analysis of AGPS in Static Classifiers 50
4.7 AGPS in Dynamic Classifiers 55
4.7.1 iPMF Values Update upon Filter Insertion 55
4.7.2 iPMF Values Update upon Filter Deletion 56
4.7.3 iPMF Values Update upon Filter Matching 56
4.7.4 Behavior and Analysis of AGPS in Dynamic Classifiers 57
5 Experiments and Results 58
5.1 Filter Generation 59
5.2 Packet Header Label Generation 61
5.3 Simulation Bench for Static Classifiers 63
5.4 iPMF Initialization 66
5.5 Matching Labels and *iPMF* Update 67
5.6 Referring to Linear Probing 68
5.7 Results and Analysis for Static Classifiers 68
5.7.1 Search Time 69
5.7.2 Throughput 75
5.8 Simulation Bench for Dynamic Classifiers 78
5.8.1 Search Time 81
5.8.2 Post Deletion Reference Period Analysis in AGP 86
5.8.3 Throughput 88
6 Conclusion and Future work 89
6.1 Conclusion 89
6.2 Future Work 90
Appendices 95
Appendix A AGPS Working Formulas 95
Appendix B AGPS Performance Tables 100
References 105

List of Tables

Table 1.1: Services provided by a given ISP. 2
Table 1.2: Traffic flows as classified by the router at interface eth2 3
Table B.1: AGPS versus LP. N=100 to N=900 101
Table B.2: AGPS versus LP. N=1000 to N=9000 102
Table B.3: AGPS versus LP. N=10,000 to N=90,0000 103
Table B.4: AGPS versus LP. N=100,000 to N=500,000 104

List of Figures

Figure 1.1: Example of an ISP connected to three client networks 2
Figure 1.2: Example of a Classifier. 4
Figure 3.1: A transition diagram modeling the LP technique 20
Figure 3.2: The matching distribution as exhibited by LP 24
Figure 3.3: A transition diagram representing LP in DC 30
Figure 4.1: The initial *iPMF* values assigned using the IHG assignment method. 36
Figure 4.2: The iPMF updates upon filter matching (N=10) 40
Figure 4.3: The iPMF updates upon filter matching (N=15) 41
Figure 4.4: Credit accumulation for filter N with and without Gaussian credit 45
Figure 4.5: Credit accumulation for a matching filter using AGPS 49
Figure 4.6: Step size for credit and β using AGPS 49
Figure 4.7: A transition diagram representing the desired probing technique. 51
Figure 4.8: A transition diagram of AGPS superimposed over the transition diagram of the desired probing technique. 53
Figure 5.1: Filter generation 60
Figure 5.2: Simulation bench for static classifiers 64
Figure 5.3: PHL generation and locality simulations 65
Figure 5.4: Example of simulation response. 68
Figure 5.5: Search time comparison between AIGPS and AGPS, N=100 69
Figure 5.6: Search time of LP, N=100 70
Figure 5.7: Search time comparison between AGPS and LP, N=100 71
Figure 5.8: Search time comparison between AGPS and LP, N=1000 72
Figure 5.9: Histogram of classification search times, (a) LP, (b) AGPS. N=1000 72
Figure 5.10: Search time comparison between AGPS and LP. N=10,000 73
Figure 5.11: Histogram of classification search times, (a) LP, (b)AGPS. N=10,000 74
Figure 5.12: Histogram of throughput, (a) LP, (b) AGPS. N=100 75

Figure 5.13: Histogram of throughput, (a) LP, (b) AGPS. N=1000 76
Figure 5.14: Histogram of throughput, (a) LP, (b) AGPS. N=10,000 77
Figure 5.15: Simulation bench for dynamic classifiers 80
Figure 5.16: GP combined search time comparison between AGPS and LP in DC. N=100, B=1K 81
Figure 5.17: GP search time comparison between AGPS and LP in DC. N=100 B=1K 82
Figure 5.18: GP histogram of search times in DC, (a) LP, (b) AGPS. N=100, B=1K..... 83
Figure 5.19: GP search time difference between AGPS and LP in DC. N=100, B=1K..... 83
Figure 5.20: AGP search time comparison between AGPS and LP in DC. N=100, B=1K... 84
Figure 5.21: AGP histogram of search times in DC, (a) LP. (b) AGPS. N=100, B=1K..... 85
Figure 5.22: AGP search time difference between AGPS and LP in DC. N=100, B=1K 86
Figure 5.23: AGP histogram comparing PDRP of AGPS and LP in DC. N=100, B=1K. 87
Figure 5.24: Histogram of throughput in DC, (a) LP. (b) AGPS. N=100, B=1K. 88
Figure B.1: Experimental and theoretical performances of AGPS and LP 100

Acronyms

AGP	After-Growth-Period
AGPS	Adaptive Gaussian-credit Probing Sequence
AIGPS	Adaptive Indexed Gaussian-credit Probing Sequence
AIPS	Adaptive Indexed-credit Probing Sequence
BARRNET	Bay Area Regional Research Network
BSL	Bit String Length
DC	Dynamic Classifier
DF	Dynamic Filter
DFTE	Default-Filter Temporary Elimination
DoS	Denial-of-Service
GP	Growth-Period
HiCuts	Hierarchical Intelligent Cuttings
IHG	Inverted Hyper-Geometric
IDS	Intrusion Detection System
iPMF	instantaneous Probability Mass Function
IRM	Independent Reference Model
ISP	Internet Service Provider
LP	Linear Probing
LRUSM	Least Recently Used Stack Model
MTU	Maximum Transmission Units
NFR	Network Flight Recorder
NSFNET	National Science Foundation Network

PDRP	Post Deletion Reference Period
PHL	Packet Header Label
PHLS	Packet-Header-Label sub-Space
PHLSS	Packet-Header-Label Super Space
QoS	Quality of Service
RNG	Random Number Generator
SC	Static Classifier
SDF	Set of Dominant Filters
SE	Search Engine
SF	Static Filter
SURANET	Southeastern Universities Research Association Network
T1	Trunk-level 1
TCAM	Ternary-Content Addressable Memory
TSS	Tuple Space Search
VAS	Value Added Services
VPN	Virtual Private Network
WS	Working Set model

Acknowledgments

I would like to express my appreciation to *Dr. Kui Wu* for his support, guidance, and ideal supervision from the introduction of the problem to the completion of this thesis.

I would like to thank *Dr. Eric Manning and Dr. Ali Shoja* for their willingness to be on this committee, and for their participation and supervision of *PANDA* lab's weekly presentations, which helped refine this work. I am grateful to Dr. Eric Manning for introducing me to Dr. Kui Wu. I would like to thank *Dr. Fayez Gebali* for his time and the fruitful discussions that we had.

I would like to thank Uvic friends and office mates, *Jeff Hornsberger, Eric Gowland, and Glenn Mahoney.*

I would like to thank my friend *Dr. Watheq El-Kharashi* for the good advice.

I would like to thank my *mother, father, brothers, and friends* for their patience and support during the hard times.

Thank you.

Dedication

This thesis is dedicated to my father, *Dr. Hamada H. Jayeh* and mother *F. M. Aiad.*

1 Introduction

In its general sense, packet classification is the task of categorizing streams of packets into different flows based on a given criterion. Routers rely on packet classification as an essential step towards supporting Quality of Service *(QoS)* applications, access control[1], resource reservation, Virtual Private Networks *(VPNs)*, accounting and billing, and other Value Added Services *(VAS)*.

A router uses a classifier to accomplish the packet classification task. A classifier is a set of rules. In general, each rule[2] contains one or more fields, the matching priority of the filter amongst other filters, and a target or action. Typically, the fields correspond to a regular expression of the TCP/IP header components, such as source and destination IP addresses[3], layer 4 source and destination port numbers (or ranges), layer-4 protocol identifiers, input/output interface, and possibly other fields. The number of fields determines the dimension of the classifier. An inbound packet is said to match a certain filter if and only if each field in the packet header matches its corresponding field in a filter. Once a match occurs, the action associated with the matching filter is executed. Consider the following example that shows how an Internet Service Provider *(ISP)* can use packet classification to provide different services. The *ISP* is connected to three client networks Net_0, Net_1, and Net_2 through interfaces eth_0, eth_1, and eth_2 respectively as shown in Figure 1.1.

---

[1] Example: Firewalls

[2] Hereinafter referred to as a "filter"

[3] IP address fields can also be of the form address/mask

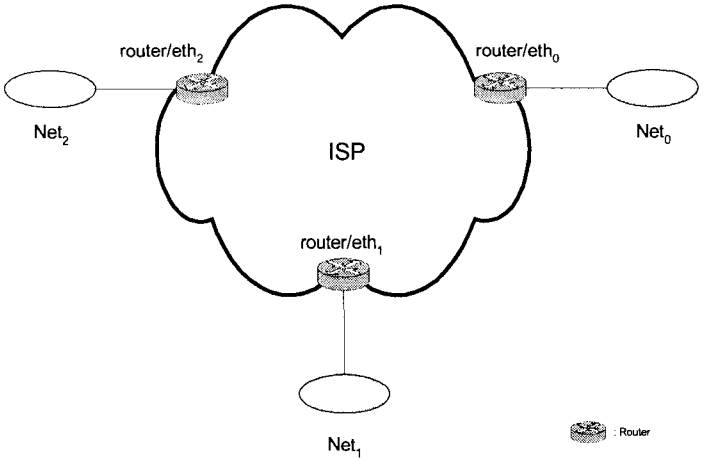


Figure 1.1: Example of an ISP connected to three client networks

The *ISP* can provide various services to its clients as shown in Table 1.1, each router (i.e., eth_0, eth_1, eth_2) classifies packets into different flows based on the services required.

Table 1.1: Services provided by a given ISP.

Service provided by the ISP	**Example of service**
Accounting and Billing	Perform accounting for traffic sent from Net_0 to Net_1 and assign it highest priority.
Policy Routing	Send all delay-sensitive traffic arriving from Net_1 to Net_2 via a separate network.
Packet Filtering	Accept only X Mbps of WWW Traffic from Net_2
Traffic Rate Limiting	Accept no more than Y Mbps of total traffic coming from Net_2

Table 1.2 shows the traffic flows that should be classified by the router at interface eth_2 and the packet header fields that the router needs to investigate to achieve this classification.

Table 1.2: Traffic flows as classified by the router at interface eth2

Flow	**Packet header fields to be investigated by a Filter**
WWW traffic from Net_2	Source network prefix. Layer-4 Source port number.
Traffic from Net_2	Source network prefix.
From Net_1 to Net_2	Source network prefix. Destination network prefix.
All other flows	Apply default filter

Note that the first and second flows are overlapping. Therefore, a strict priority needs to be assigned to each filter. In practice, filters listed at the top of the classifier are assigned higher search priority than those closer to the bottom of the classifier. The default filter being listed last in the classifier is assigned the lowest priority. It is usual for a given packet to produce a match with more than one filter in a given classifier. Therefore, during the classification process, we not only have to find a matching filter among all the filters, but also have to verify that this matching filter has the highest matching priority.

In addition to the description of classifiers above, where filters are permanently inserted in the classifier (Static Classifiers), new types of classifiers that support *stateful* packet classification were introduced. In this type of classifiers, the size of the classifier changes dynamically as filters can be inserted and/or deleted dynamically from the classifier. For

example, when a UDP request is sent to the output interface, a filter can be dynamically inserted to allow the anticipated response to bypass a firewall or any other application. Similarly, a match with a pre-existing TCP-specific filter can result in the insertion of another TCP-connection-specific filter. A termination of the established TCP connection is followed by the deletion of the filter, originally inserted to support that TCP connection. Stateful packet classification requires the router to keep track of all communication channels' states. We refer to pre-existing filters, which are typically inserted manually by network operators as *Static Filters (SF)*, and hence belong to a *Static Classifier (SC)*. We refer to dynamically inserted/deleted filters as *Dynamic Filters (DF)*, and these belong to a *Dynamic Classifier (DC)*.

Index	S_IP	D_IP	Protocol	In	Out	S_Port	D_Port	State	ACTION
1	1101*	1100*	tcp	eth0	eth1	1024:65535	*	RELATED	ACCEPT
2	1101*	110*	*	*	*	*	22	ESTABLISHED	DROP
3	*	1100*	*	*	*	*	*	*	ACCEPT
4	11*	101*	*	*	*	*	*	*	ACCEPT
5	*	10*	udp	*	eth0	1024:65535	*	*	DROP
⋮									
N	*	*	*	*	*	*	*	*	DROP

Figure 1.2: Example of a Classifier.

Figure 1.2 shows an example of a classifier. Filters are ordered according to increasing order of their indices (alternatively, we can say that the filters are ordered in decreasing order of their matching priorities). In Table 1.3, eight fields are specified for each filter. A field specified for an IP address can be specified as a full IP address or as a network prefix with a wild card "*". Fields specified for layer-4 port numbers can be either a range of the form x:y or an exact port number. Fields can also be specified to track the

state of a connection, which, for example, can be done by checking a layer-4 protocol flag; specifically, those fields specified to inspect the interface that a packet came through. The last column of the classifier is the action to be taken when a match occurs with the filter associated with the action. The last row of the classifier is where the filter with the lowest priority is placed. This filter is associated with the default action policy *DROP*.

Notably, the packet classification process that provides a supportive platform for a given service or application does not only involve extensive logical operations for comparison purposes, but also has to consider the matching priority constraints. This is in addition to time constraints (processing time) and other resource constraints.

To present these constraints, we will formally define the classification problem and review previous research on this problem in the following chapter.

1.1 Motivation and Contributions

Based on our study of packet classification techniques, we realized that almost all the techniques were strongly influenced by, and/or based on the combinational structure of the filter classifier. Moreover, though these techniques were different in their approaches, they shared a common attribute, memory-speed tradeoffs. We also noticed that most, if not all, of the proposed algorithms ignored the exploitation of packet header statistics as seen by the classifier. In addition, no research efforts had been made to develop a solution that is *probing–sequence* based. Instead, the evolution of the probing sequence within any given search algorithm relied on the search order adopted by the algorithm. For these reasons, we were motivated to develop a statistical packet classification technique.

This thesis makes the following contributions:

First, in the first stages of development, we analyzed the behavior of the simplest form of probing in a classifier, namely *linear probing*. We modeled linear probing as a Markov chain and showed that the search process exhibited by a linear probing technique follows a hypergeometric distribution. By analyzing the behavior of linear probing in both static and dynamic classifiers, we showed that, using this probing technique, the matching probability in one memory probe is very low, and the matching probability only increases with more probing.

Second, based on our analysis of linear probing, we developed a statistical probing technique called the Adaptive Gaussian-credit Probing Sequence *(AGPS)*. This technique is described and analyzed in Chapter 4. The idea was to derive statistical information about the inbound packet headers as seen by the filters in the classifier *online*. These statistics were used to derive a probing sequence that maximizes the probability of producing a match in one memory probe. The goal was achieved by designing an adaptive credit system using two formulas to exploit traffic locality. If packet header statistics change, either instantaneously or gradually, the classifier dynamically adapts to the changes and derives another probing sequence. *AGPS* was also designed to support dynamically changing classifiers. We analyzed *AGPS* in a manner similar to that used for analyzing linear probing.

Third, in Chapter 5, we performed a simulation study to compare the performance of *AGPS* to that of linear probing. We tested the performance of both techniques in two types of classifiers, static and dynamic. Two main metrics, search time and throughput, were examined.

Results showed that in small to large sized static classifiers, the search time of *AGPS* was 50% to 80% better than linear probing, with a low memory requirement of 16 bytes/filter. Search time results also showed that linear probing spent more than 50% of its time in searching more than 50% of the classifier. *AGPS*, on the other hand, spent about 80% of the time searching 10% of the classifier. Results of testing the throughput of *AGPS* compared to linear probing showed *AGPS* has a significant advantage over linear probing, especially in large classifiers. Moreover, the adaptive feature of *AGPS* enabled it to achieve throughputs that could never be reached by linear probing for the same classifier.

Our results were different with dynamic classifiers. *AGPS* was successful in utilizing the adaptive tool. However, the advantage of *AGPS* over linear probing in dynamic classifiers was not as significant as its advantages over linear probing in static classifiers. Our search time analysis showed that *AGPS* outperformed linear probing, by at most 33.40%. Throughput analysis showed that in dynamic classifiers, *AGPS* has an average advantage of about 12% over linear probing, which was not as significant as throughput improvements achieved by *AGPS* in static classifiers.

Finally, Chapter 6, we propose future development considerations for *AGPS*, and highlight some suggested applications that can benefit from using *AGPS* as a statistical tool for their operation.

2 Background

Packet classification techniques can be divided into two categories. The first one includes techniques that analyze the structure of a given classifier to create a more effective filter structure for the problem at hand. We refer to these types of techniques as *structure-based* solutions. The second category of techniques exploit certain characteristics of the network traffic, like network traffic localities, and are hence referred to as *traffic-based* solutions. A review of representatives from each category is presented after we define the packet classification problem in the following section.

2.1 Problem Formulation

Given a packet P with k header fields ($P[1], P[2], \ldots P[k]$) to be filtered, and a classifier of N filters ($F_1, F_2, \ldots, F_N$) each with k fields ($F[1], F[2], \ldots F[k]$) specified as an expression on the i^{th} field of the packet header, find a filter F with highest priority among the set of N filters of the classifier such that $\forall i$, the i^{th} field of the packet header satisfies the expression $F[i]$.

2.2 Structure-Based Techniques

In the literature, structure-based techniques are usually classified into three main groups: geometry-based, hash-based, and heuristics. The time and space complexities for structure based-techniques vary, depending on the way the problem is addressed. Before we review a representative from each group, we first provide background on the overlapping filters case, and discuss complexity bounds for the packet classification problem using structure-based techniques.

2.2.1 Overlapping Filters

In the example of Table 1.2, we encountered a case where the first and second filters overlapped. In general, a given packet can produce a match with more than one filter in a given classifier, at which case the pre-assigned priorities of the filters provide a way for arbitration.

Consider the case of the 2-dimensional filters, F_1=*(11*, 11*)* and F_2=*(11*,1*)*, a packet *P* with a bit string (11100..., 11011...) will produce a match with both filters. Unless a priority is assigned to each filter, a decision about the packet will not be clear.

With structure-based techniques, the packet classification problem is frequently mapped to a standard problems from the field of *computational geometry*, namely, *the point location problem in k-dimensional space*. The point location problem requires finding the region that encloses a *point* given a set of *non-overlapping* regions. For *N* non-overlapping regions in a $k>3$ dimensional space, the problem can be solved in $O(\lg N)$ time and requires $O(N^k)$ storage space[17][28][44]. If we trade time for less space requirements, the problem can be solved in $O(\lg^{k-1} N)$ time and $O(N)$ space[17][28][44]. Consider an example of a 4-dimensional classifier with $N=100$ non-overlapping filters, N^k space uses about 100M memory units to achieve classification in two memory accesses. The complexity of the packet classification problem, however, exceeds the bounds introduced by the point location problem, since in our case overlapping is possible, which implies that the packet classification problem is a hard problem.

2.2.2 Geometry-Based Techniques: Cross-Producting

The *cross-producting* technique [14] divides the k dimensional classifier into a table of k columns. Each column i stores the number of unique prefixes, ranges, or wildcards in the i^{th} field of the classifier. The entries of the table are then used to build a *cross-product* table, which includes all possible combinations of the *k-column* table. Next, the filter with highest priority that matches each cross product is pre-computed. Hence, given a packet P, a best match prefix process is done using the entries in the k-column table for each i^{th} field of the packet. The result can then be concatenated to build a specific cross product. Thus, the pre-computed highest priority filter that matches the specific *cross-product* is in fact the desired matching filter.

Clearly, the cross-producting technique depends strongly on the structure of the classifier. Therefore, a single update (insertion or deletion of a filter) requires a rebuild of the *cross-product* table. Unfortunately, this technique suffers from a worst-case space complexity due to the aggressive derivation of all possible cross products of the prefixes as described above. To optimize for space, an *on demand cross-producting* (a caching scheme) is proposed, where the cross-product table is incrementally built when needed. Cross products that are not used are deleted. The time complexity for the cross-producting technique is $O(kw)$, where k is the number of fields and w is the maximum length of the fields, and the space complexity is $O(N^k)$, where N is the number of filters[45].

2.2.3 Hash-Based Techniques: Tuple Space Search

V. Srinivasan et al. proposed a hash-based packet classification technique called the Tuple Space Search *(TSS)* [14]. *TSS* is based on the observation that, while a classifier contains different prefixes and ranges, the *number* of prefix lengths is small. Thus, the number of unique combinations of prefix lengths should also be small. Therefore, for all N filters, each set of filters of the same prefix length in each field can be grouped in a tuple out of M possible tuples. Linearly hashing the *tuple space* of M tuples $(M<N)$ is

therefore faster than linearly searching through N filters. A tuple is looked at as a vector of k fields, with each i^{th} field specifying the number of bits a filter must have in its i^{th} field in order to belong to a particular tuple. That is, a filter F belongs to tuple T, if for $\forall i$, the number of bits in the i^{th} field of the filter is exactly the same as the number of bits specified by the i^{th} field of tuple T. To elaborate more on how filters are grouped in their respective tuples, consider the following classifier example of five 2-dimensional *(k=2)* filters F_1, F_2, F_3, F_4, *and* F_5, where each filter is of the form *F=(source IP prefix, destination IP prefix).*

F_1*=(110*, 1010*)*

F_2*=(1*, 011*)*

F_3*= (001*, 10*)*

F_4*=(011*, 1010*)*

F_5*=(011*, 10*)*

Filter F_1 has 3 bits specified in its first field and 4 bits in the second field. Thus, we can say that filter F_1 belongs to tuple T_a*={3, 4}*, and filter F_4 can be grouped together with filter F_1 in the same tuple, since the number of bits specified in its fields satisfies the number of bits specified in tuple T_a. Similarly, filters F_3 and F_5 belong to tuple T_b*={3, 2}* and filter F_2 belongs to a *single-filter-tuple* T_c*={1, 3}*. Therefore, the classifier of size *N=5,* is compressed to a *tuple space* of size *M=3* tuples. Next, a hash table is built for every tuple, and a query in the tuple space would require searching the tuple space linearly to find the best matching filter. Clearly, since $M<N$, this mechanism is successful in improving its average search time. Notice, however that in the worst case a classifier of N filters can result in a tuple space of size $M=N$.

The update mechanism in *TSS* requires computing the number of bits in each field of the inserted filter and then inserting it in the appropriate tuple where it should belong.

Although *TSS* is a simple technique, the authors however did not provide any information about the hash function to be used, especially for arbitrary sizes of classifiers.

The time and space complexity for *TSS* is *O(N)* for both complexities, where N is the number of filters in the classifier[45].

2.2.4 Heuristic Techniques: Hierarchical Intelligent cuttings

Gaputa et al. [17] proposed a tree-based packet classification technique *(HiCuts)*. *HiCuts* works by carefully examining the structure of a given classifier and then building a decision tree, where the root of the tree is a representation of the k dimensions of the classifier. The root node is then partitioned into smaller sub-spaces by cutting through each dimension. Each sub-space or child is then recursively partitioned until each leaf node of the tree carries no more than a pre-specified number of filters called a *binth*. The maximum number of filters in each node and the decisions to be made as the search traverses the decision tree is pre-computed in a long pre-processing period [17]. The search through the decision tree is carried out by linearly searching the number of filters in each node. If a match is not found in a given node, the local decisions stored in the node is used to decide as to which node of the tree the search should proceed. The time complexity for *HiCuts* is *O(k)*, where k is the dimension of the classifier, and the space complexity is $O(N^k)$ [45].

To this end, we have reviewed three techniques that were different in the way the packet classification problem was addressed and solved. They are all, however, *structure-based.* There are other packet classification techniques that are geared towards hardware implementations, such as the Lucent bit vector scheme [28], and implantations that use Ternary-Content Addressable Memory *(TCAM)* to provide parallel processing.

Next, we review techniques that use the network traffic's inherent characteristics, such as *traffic locality*, to solve the packet classification problem.

2.3 Traffic-Based Techniques

Techniques that use the locality of network traffic to speed up the packet classification process usually depend on cache memories to cache frequently referenced packet headers. Before reviewing previous research work, we first present background on the locality of network traffic, in the following section.

2.3.1 Network Traffic Locality

In computer systems, *virtual memory* for memory management in an operating system was one of the first applications that used the concept of locality. Pages that are frequently used are kept in a cache memory for faster referencing [34]. The same strategy is also applied to accelerate IP routing table lookups [2]. The IP destination addresses that are referenced the most are cached for faster lookup. In the following sections, we define 2 prominent network traffic localities, namely, the *temporal* and *spatial* localities of network traffic, and then we review some studies under different networking environments to investigate such localities.

2.3.2 Temporal Locality

Temporal locality is a phenomenon that a given destination/source[1] IP address is referenced many times in a given period. This is because data is usually fragmented (segmented) when transmitted closely in time, or when traversing different networks with different Maximum Transmission Units *(MTU)*[29]. Consequently, *a train of packets*

destined to the same address is created [35]. This particular phenomenon is also caused by the *browsing preferences* of network users. Consider an example sequence such as {5, 5, 5, 9, 5}. It has a high degree of temporal locality since "5" is referenced 4 times out of 5. Therefore, temporal locality means that when a given IP address is referenced, it is very likely to be referenced again within a short period.

2.3.3 Spatial Locality

Another interesting phenomenon is the *spatial locality* of the network traffic [36]. When traffic is directed towards a subnet, the destination IPs can be mapped to a set of addresses with the same prefix. A sequence of destination IPs, such as *192.168.64.1, 192.168.64.2, 192.168.64.3, and 192.168.64.10* has a high degree of spatial locality since all addresses belong to subnet *192.168.64/24.* Thus, spatial locality implies that when a given IP address is referenced, there is a high probability of referencing other addresses with the same prefix.

2.3.4 Locality Measurements

Many studies have been done to investigate and quantify the locality of network traffic in both Wide and Local Area Networks. Claffy *et al.* [39] measured the National Science Foundation Network *(NSFNET)* T1 backbone traffic. During the one-month period of study, measurements were made of the average packet size on the network, most popular sources-destination site pairs, of traffic locality, as well as the international distribution of traffic. *NSFNET* included the transcontinental backbone, mid-level networks like Bay Area Regional Research Network *(BARRNET)* and Southeastern Universities Research Association Network *(SURANET)*, and the campus networks. The backbone also supports international connections to national backbones of other countries.

---

[1] Also referred to as *source locality* **[35]**.

The backbone carried traffic of about 980 billion bytes to and from 4254 networks. Over 50% of the traffic was generated by 0.7% of the networks. More than 50% of the traffic was destined to 2.8% of the most popular destinations and about 45% of the traffic was exchanged between 1500 out of 560,000 site-pairs (0.28%). Measurements were also made to investigate the *favorite-site* trend of 2 mid-level networks as well as NSF's local networks. For the first mid-level network, 90% of the traffic went to 6.7% of the most favorite sites, for the second mid-level network, 90% of the traffic went to 6.6% of the favorite sites. The same percentage of traffic was generated by NSF's local networks to 13% of the sites.

Another measurement was conducted at the Massachusetts Institute of Technology *(M.I.T)* on a token ring network [35] that connected 33 computers, 7 gateways, and a number of servers. Measurements showed that a packet originating from a given source was followed by a packet coming from the same source about 30% of the time. In addition, the probability that a packet from a source S to a destination D is followed by a packet from D to S is about 31%. The authors referred to this type of locality as *source locality,* and modeled it by what they called *the tandem trailer model.*

Next, we review previous research work that utilizes the traffic localities to accelerate the packet classification process.

2.4 The Cache Referencing Technique

The cache referencing technique works as follows. Given a cache memory with a pre-determined limited storage space of size C and a classifier of size N, a given packet header P with k fields relevant to classification is compared sequentially to the N filters to find the best matching filter. The cache memory is initially empty. When a match is found, the entire k fields of the packet header and the associated action of the matching

filter are cached in memory. Based on the assumption that traffic locality exists there is a high probability that the next packet header is the same as the one already cached. Hence, caching the most recently referenced entries can speed up the classification process as the search is only required in the cache memory, which has a size $C<N$.

Most of the research based in this technique focused on three areas: developing and testing models for cache-reference behavior [36][41][42], cache replacement algorithms, and cache organization design [2]. A review of each of these areas is presented next.

2.4.1 Models for Cache Reference Behavior

A number of cache-referencing models have been investigated widely in the literature. The main motivation is to simulate the address access patterns in a given address trace, being a fundamental step towards designing better caching scheme. Amongst these were two representative models: the Working Set model *(WS)* and the Least Recently Used Stack Model *(LRUSM)*.

In the *WS* set model [41], an interval *W* referred to as the *working set window size* is defined, and it is assumed that the number of unique addresses referenced in this interval (referred to as the *working set size*) are likely to be referenced. The *WS* model, therefore, assumes the availability of traffic locality. The ratio of the *working set size* to the *working set window size* reflects the degree of locality. A smaller ratio implies a high degree of locality.

The other model is the *LRUSM* [42], which is extensively analyzed in the literature. In *LRUSM*, the referenced addresses are arranged in a stack with the least recently referenced address placed at the last position of the stack. Each time an address is

referenced, it is pushed to the top of the stack[1]. Therefore, assuming that traffic locality exists, the probability that the address at the top of the stack is to be referenced again is high. In general, we can say that the probability of referencing an address at position *n* is a decreasing function of the stack position.

2.4.2 Cache Replacement Algorithms

A *cache hit* occurs when a search in the cache memory produces a match. However, the situation is different when a match is not available, being referred to as a *cache miss*. When a cache miss occurs, the search turns to the classifier for a match. When a match is found, the entire *k* fields of the packet header and the associated action of the matched filter are then cached in memory. When the cache memory is already full and there is no space available for the new packet header to be inserted, an entry in the cache memory is sacrificed to create space for the new entry. Which entry should be sacrificed motivated the development of *replacement algorithms*. In the following sections, we review some *replacement algorithms* that are frequently used in the literature, namely, First-in-First-out *(FIFO)*, Random *(RAND)*, Least Recently Used *(LRU)*, and Optimum *(OPT)* [43].

In *FIFO*, the address that enters the cache memory first is sacrificed and the new entry is inserted. That is, the addresses are evicted according to the order of their insertion. For example, consider a cache memory that can only hold five entries, and entries such as *4,7,5,3,6* already exist, if entry *15* needs to be inserted, entry *4* is sacrificed and entry *15* is inserted, thus, the entries would be *7,5,3,6,15*.

In *RAND*, as the name implies, entries are selected at random. The selected entry is replaced with the new entry.

---

[1] Note that the referenced address is not gradually pushed towards the top.

The most successful replacement algorithm to date is the *LRU* algorithm and its variants. In *LRU*, the entry that was least recently referenced is sacrificed for the new entry. For example, consider the same example that we used above for *FIFO*, when entry *15* needs to be inserted, entry *6* is sacrificed if its reference time is the least, and *15* is inserted in the top position. Each time an entry is referenced, it moves to the top position, thus pending its eviction in the future. This will result in a continuous shuffling of the entries, with *least recently used* at the position at which an entry is sacrificed.

The theoretical optimum replacement algorithm *OPT* or *MIN* was developed by Belady [36][43]. *OPT* assumes that knowledge about the future address reference pattern is readily available. Therefore, *OPT* is capable of deciding precisely which cache entry should be replaced. Although, this algorithm is yet to be practically implemented, it is used as an optimum reference for practical algorithms.

2.4.3 Cache Organization

Chvets et al. [2], exploited both the spatial and temporal locality of traffic. In this method, memory was divided into zones and entries that have the same length of network prefix were placed in each zone. The idea was to allocate most of the available space to the mostly used portion of the address space. It was reported that 95% of the traffic came from less than 50% of the address space found in a traffic trace. The *LRU* replacement algorithm was used. The study reported that using 2-zone caches produced miss ratios that were half those of no-zone caches.

3 Linear Probing

3.1 Linear Probing in Static Classifiers

In a Linear Probing technique *(LP)*, an inbound packet is compared sequentially to the N filters of the *Static Classifier (SC)*. The packet is initially compared to the filter with highest priority, which is the first filter. If the packet matches the first filter then we have a match after 1 memory probe. The *search engine (SE)* then transits to an idle state. If the packet does not match the first filter, it proceeds to the second filter requiring an additional memory probe. If a match occurs, then we have a match with a cost of two memory probes. Then the *SE* goes back to an idle state waiting for the next inbound packet. In general, if a packet matches filter m then we have a match with a cost of m memory probes.

Note that the packet can proceed with no matches until it reaches the default filter with index N. Hence, we have a match with a cost of N memory probes. In fact, for a given *SC* of size N, the maximum possible matching cost is N, in which case the *LP* technique will require N memory probes.

In the following section, we analyze the behavior and performance of *LP* in *SC*.

3.2 Behavior and Analysis of LP in Static Classifiers

We use a Markov chain transition diagram to model the behavior of *LP* and analyze its performance. Given a classifier with N filters, we assume the following:

1. The filters are placed in ascending order based on the values of their indices. This is in agreement with the ordering convention introduced in the previous chapters.
2. Filters are probed sequentially according to the order of their priority [14] starting with the filter of index 1. If this filter does not match, the next filter is probed and so on, until a match is found, otherwise the default filter N is applied [31].
3. After a match occurs, *LP* goes to an idle state.
4. Since *LP* does not utilize any historical information regarding traffic locality, *LP* works in a fashion that assumes no relation between inbound packets. As such, it is assumed that each filter is equally likely to match a given packet on a long-term observation.

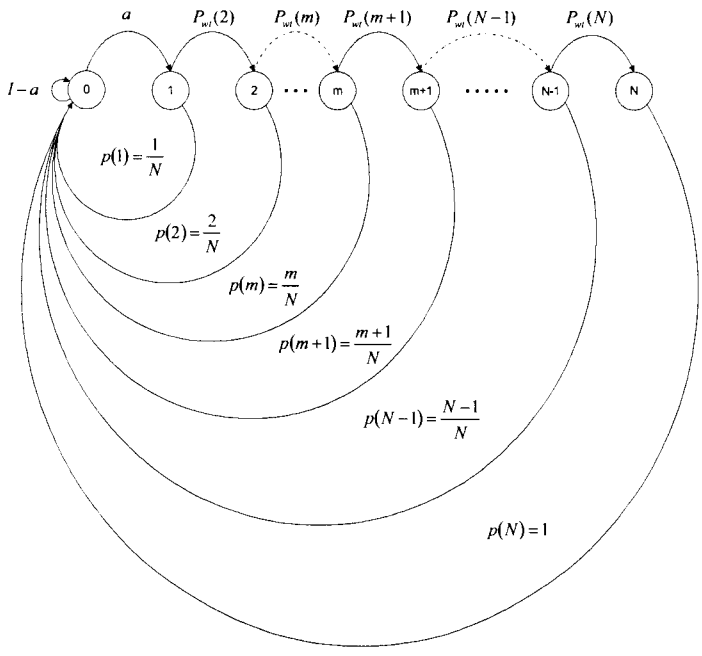


Figure 3.1: A transition diagram modeling the LP technique

To map the behavior of *LP* on a transition diagram, we first need to identify our states [1], define a hold time or sampling period, during which the system resides in any of the available states, and find the transition probabilities between states. One way of evaluating the performance of a given search-technique is by looking at the number of memory probes made to reach a match. Thus, we choose each state to represent the number of memory probes required by *LP* to find the matching filter as shown in Figure 3.1, where State 0 is the idle state.

At the end of each memory probe, *LP* chooses between two mutually exclusive events: match and no-match. Therefore, we define the hold time as the time elapsed since the transition to the current state occurred, and the decision of match or no-match is taken. We call this period the processing time T_P.

We now proceed to find the transition probabilities that govern the transition of the system from one state to the other.

According to assumption 4 stated earlier, we assumed that filters are equally likely to match a given packet according to a long-term observation period. However, according to assumption 2, when a filter mismatches a packet, the search proceeds to the next filters and the mismatching filter is *not included again* in the search space. This short-term observation maps the behavior of *LP* to a *sampling without replacement* case. For example, consider a game of chance where a player is allowed m trials to pick a box out of N boxes. Only k boxes contain a prize, while the rest is empty. The player starts the game by checking one box at a time.

If we define a random variable x that assumes a value equal to the number of successful attempts in the m trials, then the random variable x is said to be hypergeometrically distributed with parameters N, m and k, and has the following probability mass function [33]

$$p(x;N,m,k) = \frac{\binom{k}{x}\binom{N-k}{m-1}}{\binom{N}{m}} \tag{3.1}$$

Therefore, if the player is allowed one trial to pick a box out of 10 *(N=10)* boxes given that only one box has a prize *(k=1)*, the probability of finding one box *(x=1)* in one attempt *(m=1)* is

$$p(1;10,1,1) = \frac{\binom{1}{1}\binom{9}{0}}{\binom{10}{1}} = \frac{1}{10}$$

Alternatively, assuming that these boxes are equally likely to contain the prize, we can reach the same result. That is, if we asked what is the probability of picking the only winning box out of 10 boxes? The answer is 1/10.

Now, if the player were allowed 5 attempts to find the only winning box, the probability of finding this box would be

$$p(1;10,5,1) = \frac{\binom{1}{1}\binom{9}{4}}{\binom{10}{5}} = \frac{5}{10}$$

So in general, if the player was allowed m attempts to find the only winning box out of N boxes, the probability of finding this box would be

$$p(1;N,m,1) = \frac{\binom{1}{1}\binom{N-1}{m-1}}{\binom{N}{m}} = \frac{m}{N} \tag{3.2}$$

We now map the example described above to the case at hand, where we have only one filter to be applied to a given packet and a classifier of N filters. According to Equation (3.2) above, the probability of finding the matching filter in one memory probe *(m=1)* is *1/N*, and the probability of finding this filter in 2 probes is *2/N*, and so on. Therefore, the probability of finding the matching filter in m memory probes is *m/N*. Moreover, according to assumptions number 1, 2 and 3, the first filter should be probed first on the first memory probe. If no match is found, the second filter should be probed on the second memory probe, and the m^{th} filter on the m^{th} memory probe. Therefore, the probability *p(m)* of finding a match with filter number m and hence transiting from state m to the idle state is given by

$$p(m) = \frac{m}{N} \tag{3.3}$$

As shown in Figure 3.1 above, where *"a"* is the probability that a packet arrives for classification. The following example summarizes the statements made above.

Example 1

In a classifier of N filters, the probability *p(1)* of finding a match with the first filter, and hence transiting from state 1 to the idle state is

$$p(1) = \frac{1!(N-1)!}{N!} = \frac{1}{N}$$

The probability *p(5)* of finding a match with filter number 5 and hence transiting from state 5 to the idle state is

$$p(5) = \frac{5!(N-1)!}{N!.4!} = \frac{5}{N}$$

The probability *p(N)* of finding a match with filter number N and hence transiting from state N to the idle state is

$$p(N) = \frac{N!(N-1)!}{N!.(N-1)!} = 1$$

Figure 3.2 illustrates the matching probabilities as exhibited by *LP* where $N = 10$.

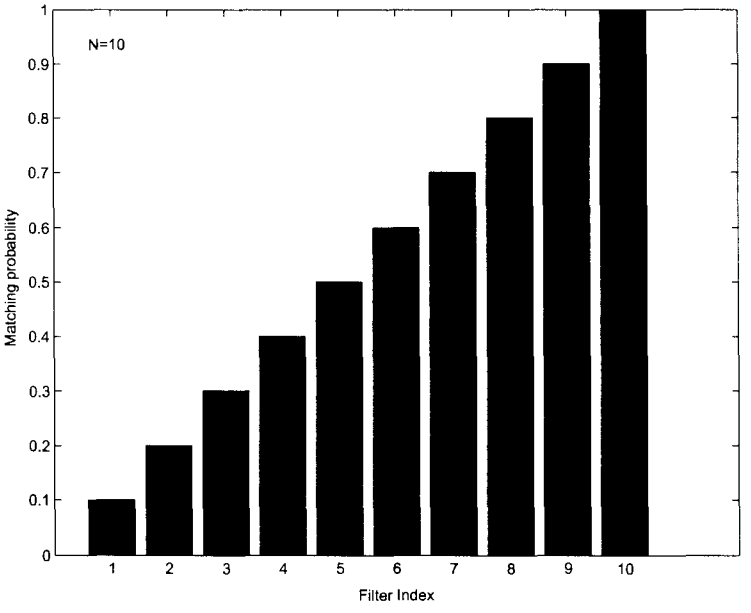


Figure 3.2: The matching distribution as exhibited by LP

As shown in the figure, the probability of producing a match increases as the system executes more memory probes.

Now that we have found the transition probabilities to the idle state (matching probabilities), we need to find the transition probabilities from one state to the next. With

reference to assumption 2 above, in the event of *no-match*, the system carries out an additional memory probe and transits from state m to state $m+1$. Since each additional memory probe constitutes wasted processing time, we name this transition probability, the probability of wasting time transiting from state m to state $m+1$ denoted by $p_{wt}(m+1)$.

On Figure 3.1, since the probability of making a transition from state m to state $m+1$ depends on the probability of producing a match at state m, the transition probability $p_{wt}(m+1)$ from state m to state $m+1$is given by

$$p_{wt}(m+1) = 1 - p(m) \tag{3.4}$$

The following is an example of the transition probability mentioned above.

Example 2

The probability of making a transition from state 1 to state 2 is

$$p_{wt}(2) = 1 - p(1) = 1 - \frac{1}{N}$$

The probability of making a transition from state 5 to state 6 is

$$p_{wt}(6) = 1 - p(5) = 1 - \frac{5}{N}$$

The probability of making a transition from state $N-1$ to state N is

$$p_{wt}(N) = 1 - p(N-1) = 1 - \frac{N-1}{N}$$

Note that the probability of making a transition and hence wasting time decreases with more memory probes.

The probability transition matrix P of size $(N+1)\times(N+1)$ shows all the possible transition probabilities relevant to LP technique.

$$P=\begin{bmatrix} 1-a & p(1) & \cdots & p(m) & p(m+1) & \cdots & p(N-1) & p(N) \\ a & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 \\ \vdots & p_{wt}(2) & \cdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \ddots & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & p_{wt}(m+1) & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & p_{wt}(m+2) & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & p_{wt}(N) & 0 \end{bmatrix}$$

After modeling LP and defining all the system parameters, we analyze its performance as follows. We define the throughput of LP for a given packet as the number of packets classified per one memory probe. For example, if 1 packet was classified in 1 memory probe, then the throughput is 1 packet per memory probe. Similarly, if the packet was classified in 2 memory probes, then the throughput is ½ a packet per memory probe. In general, if a packet is classified in m memory probes then $1/m$ packet was classified in each memory probe, hence, the throughput is $1/m$ packets/memory probe. Therefore, the average number of packets processed per one memory probe or the average throughput of LP in a classifier of N filters is given by

$$Th_{LP}=\frac{N}{\sum_{m=1}^{N} m}\left[1\times p(1)+\frac{1}{2}\times p(2)+\cdots+\frac{1}{m}\times p(m)+\cdots+\frac{1}{N}\times p(N)\right]$$

$$=\frac{2}{N+1} \qquad \text{packets/time step} \tag{3.5}$$

Where the probabilities where normalized by dividing by the term $\frac{\sum_{m=1}^{N} m}{N}$ to satisfy the equation

$$\sum_{m=1}^{N} p(m)=1 \tag{3.6}$$

Similarly, we define the search time to classify one packet as the number of memory probes required to classify a packet. Therefore, the average search time of *LP* in a classifier of N filters is given by

$$S_{LP} = \frac{N}{\sum_{m=1}^{N} m}\left[1 \times p(1) + 2 \times p(2) + \cdots + m \times p(m) + \cdots + N \times p(N)\right]$$

$$= \frac{2\sum_{m=1}^{N} m^2}{N^2 + N} \quad \text{memory probes} \tag{3.7}$$

In the next section, we model and analyze the behavior of *LP* in *DC*.

3.3 Linear Probing in Dynamic Classifiers

As we stated previously in Chapter 1, in addition to *SC*, there are Dynamic Classifiers *(DC)*, where filters may be inserted to account for an anticipated flow. Similarly, a filter may be deleted when the reason for which it existed does not apply any further. Thus, the size of the classifier changes dynamically as required. Practically, a Dynamic Filter (*DF)* is assigned a higher priority than a Static Filter (*SF)* [31].

Since deletion and insertion are independent events, any given time step the size of the *DC* can increase or decrease depending on the rates at which filters are inserted or deleted.

In *SC*, the indices of the filters represent a cost. This cost influences our search priority. In *DC* however, the indices of the filter can represent the insertion order. For example, the first inserted filter will be assigned index number one, the second inserted filter will

be assigned index number two, and so on. In general, the most recently inserted filter is assigned index *Z*. Where the value of *Z* reflects the number of filters currently in the classifier. When the *DC* reaches the maximum allowed space, and a filter needs to be inserted, the first inserted filter is deleted.

Intuitively, we would prefer to probe the most recently inserted filter, as the probability that this filter will be used shortly is higher than the other *Z-1* filters. Moreover, in case probing filter with index *Z* produces no match, we would prefer to probe the other *Z-1* filters in a decreasing order of their indices until a match is found, otherwise the search turns to a *SC* for a match.

In the next section, we model and analyze the behavior of *LP* in *DC*.

3.4 Behavior and Analysis of LP in Dynamic Classifiers

We use a Markov chain transition diagram to describe and analyze the behavior of *LP* in a *DC*. We define ρ_i and ρ_d as the average rates of insertion and deletion respectively in units of filters/time step. We select the time step T relative to the maximum rate of insertion δ_i.

$$T = \frac{1}{\delta_i} \tag{3.8}$$

Thus, at a given time step, at most one filter can be deleted and/or inserted. The probability *"a"* of inserting a filter in a given time step is given by [1]

$$a = \frac{\rho_i}{\delta_i} \tag{3.9}$$

The probability *"d"* of deleting a filter in a time step is given by

$$d = \frac{\rho_d}{\delta_i} \tag{3.10}$$

Alternatively, The probability of no insertion *"b"* of a filter in a time step is given by

$$b = 1 - a \tag{3.11}$$

and the probability of no deletion *"c"* of a filter in a time step is given by

$$c = 1 - d \tag{3.12}$$

We assume the following:

1. The size of the classifier is initially zero (contains no filters).
2. At any time step, the system resides in any given state. The states represent the size of the classifier.
3. When a filter is inserted at time instant *n*, it is assigned index *Z(n)*, which is equivalent to the new size of the classifier.
4. Filters with larger indices are of higher priority.
5. The maximum allowed size for *DC* is *B*. When a filter needs to be inserted while the classifier is already at its maximum size, filter at index 1 is deleted.
6. When a packet arrives at time instant *n*, filters are probed sequentially according to the order of their priority starting with filter *Z(n)*. If this filter does not match, filter *Z(n)-1* is probed and so on, until a match is found otherwise the search turns to the *SC*.
7. After a match occurs, *LP* goes to an idle state.

Figure 3.3, shows a transition diagram with states representing the size of the *DC* for a given time step. The diagram shows a birth-death process [1].

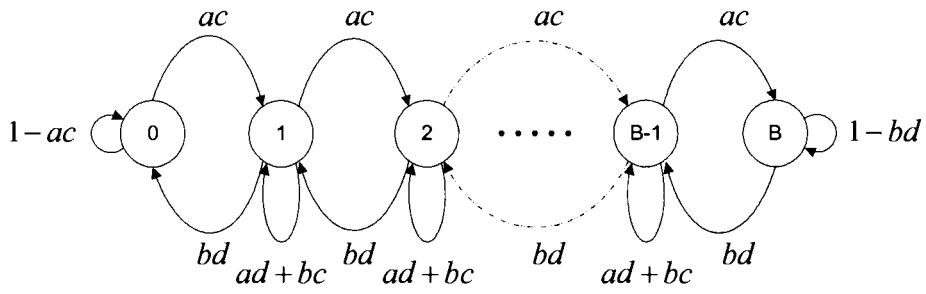


Figure 3.3: A transition diagram representing LP in DC

The transition between states or the change of size of the *DC* at a given time step is governed by the transition matrix P of size *(B+1) x (B+1).*

$$P = \begin{bmatrix} 1-ac & bd & 0 & \cdots & 0 & 0 \\ ac & ad+bc & bd & \cdots & 0 & 0 \\ 0 & ac & ad+bc & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & ad+bc & bd \\ 0 & 0 & 0 & \cdots & ac & 1-bd \end{bmatrix} \tag{3.13}$$

Initially the size of the classifier is zero. Therefore, the initial distribution vector ***s(n)*** at time step $n = 0$ is given by

$$\boldsymbol{s}(0) = [\, 1\ 0 \cdots 0\,]^t \tag{3.14}$$

Where the superscript t denotes a transposed vector. Since the transition probabilities are not a function of time as $n \to \infty$, the system reaches its steady state. We need to find the steady state distribution vector to proceed with our analyses. There are many ways to find the steady state distribution vector. If we assume that the values of the transition matrix can be expressed numerically, then the eigenvector of the transition matrix, which corresponds to an eigenvalue $\lambda = 1$ is the distribution vector $\boldsymbol{s}$ at steady state [1], that is

$$\mathrm{S} = eig(\mathrm{P} \mid \lambda = 1) \tag{3.15}$$

Thus,

$$\boldsymbol{s} = [s_0 \quad s_1 \quad \cdots \quad s_B]^t \tag{3.16}$$

Where s_B denotes the probability that the size of the classifier is B. Therefore, the average size of the DC is given by

$$Q = 0 \times s_0 + 1 \times s_1 + 2 \times s_2 + \cdots + B \times s_B$$

$$= \sum_{i=0}^{B} i\, s_i \quad \text{filters} \tag{3.17}$$

Therefore, the average throughput for the LP technique in a DC as defined above is

$$Th_{LP} = \frac{Q}{\sum_{m=0}^{Q} m}$$

$$= \frac{2}{Q+1} \quad \text{packets/time step} \tag{3.18}$$

3.5 Concluding Remarks

The analysis was based on the assumption that filters are equally likely to match a packet. This is true only for a long-term period. During a short-term period, traffic locality will render inbound packets matching the filters with different probabilities.

The analysis results only reflected the average performance of LP. During a short-term period, the performance may be much better than the average (e.g., inbound packets can always match the first filter) or worse than the average (e.g., inbound packets can always match the last filter). Referring to the previous discussion, and analysis of LP behavior and performance in static and dynamic classifiers, we summarize our conclusions as follows:

1. *LP* is not capable of exploiting traffic locality and works in a fashion that assumes no relation between inbound packets. This behavior is similar to *IRM* described in Section 2.4.1 above.
2. The probability of producing a match increases as mismatches occur.
3. Ideally, we would like the probability of needing more memory probes (mismatch) to be as low as possible and as early as possible, however, in *LP* the probability of needing more memory probes or the probability of wasting time only decreases as more memory probes are made and not as early as the first probe.
4. The performance is highly affected by the size of the classifier.
5. In a *DC*, when a filter needs to be inserted, the first inserted filter is deleted.

In summary, *LP* is not flexible with respect to changes in traffic locality, which degrades its performance. Motivated by the above conclusions, we introduce an adaptive approach, which exploits the temporal and spatial characteristics exhibited by the network traffic in, the following chapter.

4 The Adaptive Gaussian-credit Probing Sequence

4.1 AGPS in Static Classifiers

In this chapter, we introduce a probing technique, the Adaptive Gaussian-credit Probing Sequence *(AGPS)* in an attempt to eliminate unnecessary memory probes, and thus, accelerate the search and processing time needed by the Search Engine (*SE)*. We use *AGPS* for both *SC* and *DC* to exploit the temporal and spatial characteristics of network traffic described in Chapter 2.

Given a *SC* of size *N*, we want to search this classifier for an appropriate match to a given inbound packet. While undertaking this task, we have to search the *SC* for a matching filter according to the search preferences dictated by the indices of the filters. In other words, if we have two possible matching filters with indices m and n, where $m < n$, then we elect the filter with index m as a matching filter.

As discussed in the previous chapter, probing the *SC* according to an increasing order of filter indices produces a *linear probing* behavior. This behavior results in a number of unnecessary memory probes, influenced by the statistics of the inbound packet headers as seen by the classifier.

According to the definition of temporal and spatial characteristics of network traffic described in Section 2.3, there is a set of frequently matching filter(s) within a given time period. If there is a probability mass function *(PMF)* associated with this time period or traffic burst that describes the statistics or matching frequency as seen by the *N* filters, then the *PMF* values for this instant will show a bias towards the set of the frequently

matching filter(s). We identify this set of filters as the Set of Dominant Filters (*SDF)* of size D where $1 \leq D << N$ [1]. We name this particular *PMF* the instantaneous *PMF (iPMF)* associated with this specific burst.

By definition, the values of the *iPMF* can provide a lot of valuable information about the current burst. Therefore, providing these values to the search engine in advance will help us save a considerable number of unnecessary memory probes for a given burst. In fact, the values of the *iPMF,* when updated appropriately before each upcoming packet, will be a contiguous representation of the randomness of the packet header statistics as seen by the classifier for each and every burst. The search engine will simply choose to start the search starting from the filter with the maximum *iPMF* value. If no match is found, the engine proceeds to the filter with the next to maximum *iPMF* value and so on. Therefore, a search engine relying on *iPMF* values as a probing criterion should be able to reach a matching filter after very few memory probes.

In case there is more than one matching filter for a given packet, pointers are used to link all filters that can be a potential match for one given packet. The pointer carries appropriate information about filters with higher priorities that match the same packet. Thus, if a given filter was found to be a dominant filter based on its *iPMF* value, but another matching filter exists with less *iPMF* value but of a lower index value, the search engine switches to the filter with a lower index and applies the associated rule. Eventually, the most *frequently used* filter will gain enough credit to be dominant.

---

[1] Similar to the WS window size defined in Chapter 2

Since the values of the *iPMF* are initially unknown, we have to assume these values. The following section proposes the initial values of *iPMF* as well as the formula used to update these values prior to the next upcoming packet.

4.2 Initial Values for iPMF

As discussed earlier, the search engine will rely on the values of the *iPMF* as a search criterion as opposed to the indices of the filter. Our criterion in choosing initial *iPMF* values is as follows.

The sum of all *iPMF* values at time instant $n=0$ should be 1, that is

$$\sum_{j=1}^{N} iPMF_j(0) = 1 \tag{4.1}$$

Where $iPMF_j(0)$ is the initial *iPMF* value of filter with index j.

We propose three different methods to assign the initial *iPMF* values: the *uniform* method, the *random* method, and the *Inverted Hyper-Geometric* method *(IHG)*. In the uniform method, as the name implies, filters are assigned equal *iPMF* values. Therefore, for a classifier of N filters, each filter is assigned an *iPMF* value of $1/N$. This satisfies Equation (4.1).

In the second method, we assign each filter a value randomly in the range [0:1]. Each random value is then normalized by dividing it by the sum of all the randomly generated values, hence, satisfying Equation (4.1).

In the last method, the values assigned are a normalized inverted version of the distribution exhibited by the *LP* technique, as shown in Chapter 3. That is, the maximum *iPMF* value is assigned to filter with index 1, the next to maximum *iPMF* value is assigned to the filter with index 2, and so on, until the minimum *iPMF* value is assigned to the filter with index N. Figure 4.1 shows the initial *iPMF* values assigned to the search engine using the *Inverted Hyper-Geometric (IHG)* assignment method.

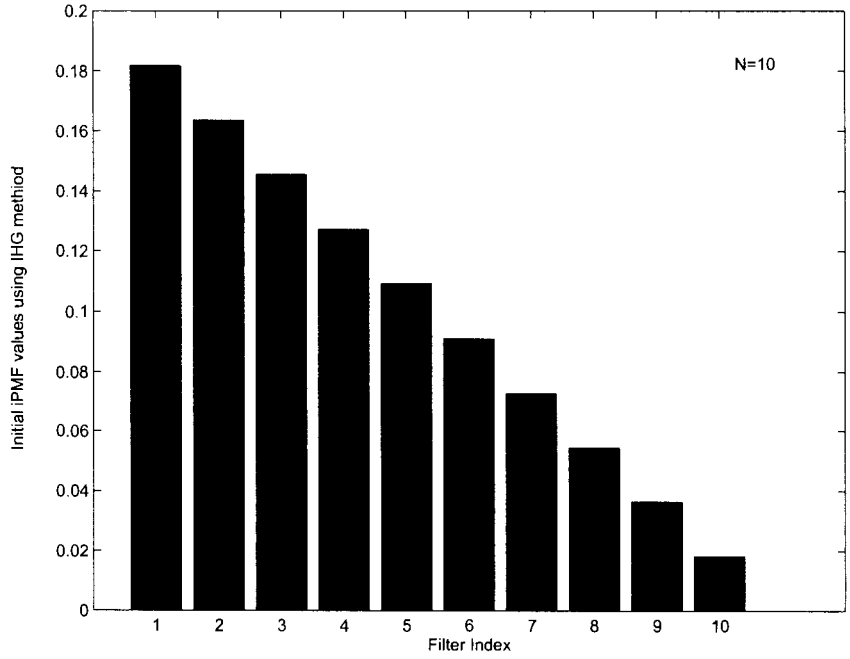


Figure 4.1: The initial iPMF values assigned using the IHG assignment method.

The initial probabilities for the N filters are assigned as follows.

$$p_1(0) = \frac{N}{\sum_{m=1}^{N} m} = \frac{2}{N+1}$$

$$p_2(0) = \frac{N-1}{\sum_{m=1}^{N} m} = \frac{2(N-1)}{N^2+N}$$

$$\vdots$$

$$p_m(0) = \frac{N-m+1}{\sum_{m=1}^{N} m} = \frac{2(N-m+1)}{N^2+N} \tag{4.2}$$

$$\vdots$$

$$p_N(0) = \frac{1}{\sum_{m=1}^{N} m} = \frac{2}{N^2+N}$$

Where m represents the index of the filter.

Thus, the search engine will try to match a given packet to the first filter, and then the second, and so on, until an appropriate match is reached at filter m. This method also satisfies Equation (4.1).

Notably, there are other assignment methods for the initial *iPMF* values. Nevertheless, the behavior of *AGPS* does not depend on the initial *iPMF* values. Those values will be dynamically adjusted to reflect the packet header statistics as seen by the classifier at a given time instant. That is, after *AGPS* finds a matching filter m at time instant n, the *iPMF* values are updated as a preparation step for the upcoming packet. The *iPMF* values can be seen as credit values, where a matching filter gains credit and a non-matching filter loses credit. In the following section, we propose an update mechanism.

4.3 Indexed-credit Update Mechanism for iPMF

The following constitutes the criteria for the Indexed-Credit Update mechanism.

1. Matching filters with higher priority should be granted more credit.
2. At time step n, a matching filter m is granted credit. The additional credit should be added to its previous credit $(iPMF_m(n-1))$.
3. The credit for the matching filter *should* be < 1 for all n.
4. At the end of the update mechanism, the following equation should be satisfied.

$$\sum_{j=1}^{N} iPMF_j(n) = 1, \text{ for any } n \tag{4.3}$$

Note that there may exist many update mechanisms that can satisfy these criteria. In the sequel, we propose three different update mechanisms based on our evaluation of the current ones.

To update the previous credit of a matching filter m as stated by the criteria above, we propose the following formula, where the $iPMF_m(\xi)$ of a matching filter m at time instant ξ is denoted by $p_m(\xi)$.

$$p_m(n) = \frac{p_m(n-1)(1+\frac{1}{m})}{1+\frac{p_m(n-1)}{m}} \tag{4.4}$$

$$= \frac{p_m(n-1)(1+m)}{p_m(n-1)+m}$$

We update the $iPMF$ values of the remaining $N-1$ filters using the following equation

$$p_i(n) = \beta\, p_i(n-1), \text{ for all } i \neq m \tag{4.5}$$

Where

$$\beta = \frac{1 - p_m(n)}{\sum_{i \neq m}^{\wp} p_i(n-1)}$$

$$= \frac{1 - p_m(n)}{1 - p_m(n-1)} \qquad (4.6)$$

We prove in the following that our proposed mechanism satisfies the criteria for the indexed-credit update mechanism.

1. Using Equation (4.4), a credit with a value of $iPMF(n-1)/m$ is added to the previous credit of the matching filter with index m, hence, the credit gained by the matching filters with lower indices will always be more than the credit gained by matching filters with higher indices for a given *iPMF*. The denominator of the equation guarantees that the resulting credit is <1. This satisfies requirements 1, 2, and 3 of the update criteria.
2. Equations (4.5) and (4.6) satisfies Equation (4.3) and therefore satisfies the fourth requirement of the criteria.[1]

We refer to a probing technique using this update mechanism as the Adaptive Indexed-credit Probing Sequence *(AIPS)*.

For the purpose of illustration, Figure 4.2 is an example of the *iPMF* updates or training period for *AIPS* upon filter matching. For simplicity, we assume a *SC* of size $N = 10$, three dominant filters, and a continuous matching case where all the inbound packets are dedicated to one of the dominant filters at a time.

---

[1] Please refer to Appendix A for proof.

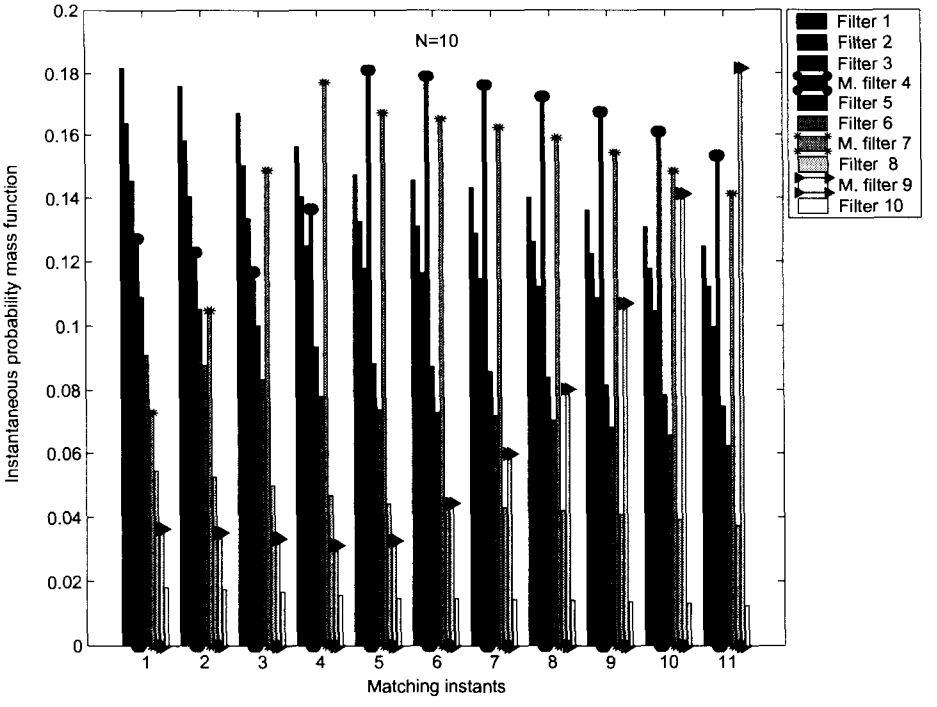


Figure 4.2: The iPMF updates upon filter matching (N=10)

As shown in the first matching instant of Figure 4.2, initial *iPMF* values are assigned according to Equation (4.2). Assume that from the second to the third matching instant, filter 7 is matched with inbound packets. The *iPMF* value of filter 7 increases gradually according to Equation (4.4). All other *iPMF* values of the nine non-matching filters decrease according to Equations (4.5) and (4.6). On the fourth and the fifth matching instant assume that it is filter 4 that matches the inbound packets. Filter 4 starts gaining credit while working its way towards dominance until it reaches the fifth matching instant where it becomes the dominant filter with maximum *iPMF* value. On the sixth matching instant and until the last matching instant in the same figure, assume inbound packets match filter 9. In this case, both filters 7 and 4 start to lose credit together with all other filters, except for filter 9, which starts to gain credit, indicating that most of the inbound packets are matching with it. Note that the matching filters claim their dominance gradually by acquiring credit and the dominant maintain their relative *iPMF* values even when they start losing credit.

As stated before the *initial iPMF* values are of no statistical value or meaning. During the training period, however, the *iPMF* values are adjusted dynamically to represent the packet header statistics on the input for a finite period. If the packet header statistics on the input link changes, *AIPS* will adapt the *iPMF* values for the new burst of packet headers.

Figure 4.3 shows another example of a *SC* of size $N = 15$ and three dominant filters: 5, 10, and 15. Initial *iPMF* values are assigned to all filters on the first instant. On the second matching instant, filter 5 shows a potential for dominance by an increase in its *iPMF* value.

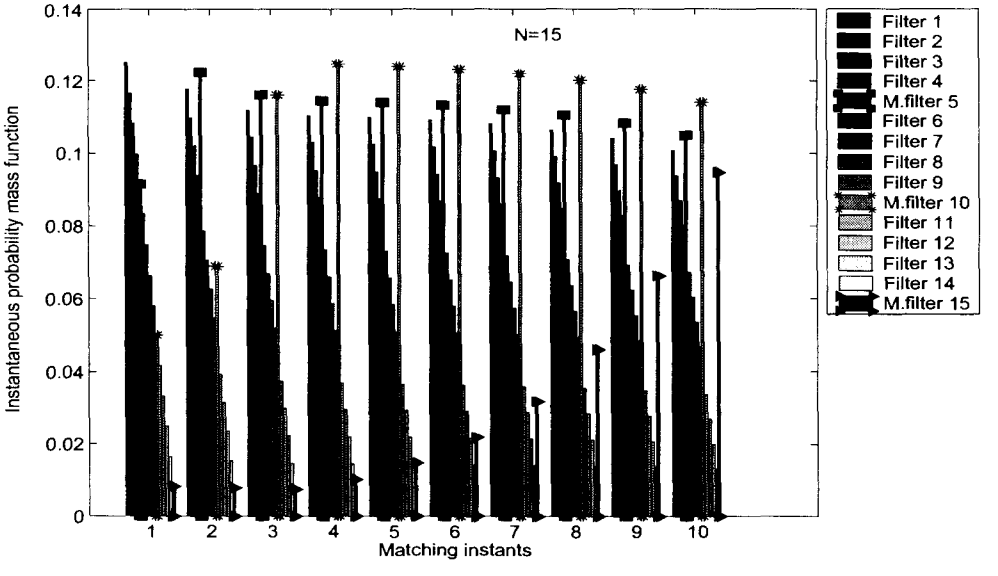


Figure 4.3: The iPMF updates upon filter matching (N=15)

This value however starts to decrease on the third matching instant because filter 10 begins to gain credit at this instant and becomes dominant on the fourth matching instant. After the fifth matching instant, all filters including filters 5 and 10 start to lose their credit to filter 15, which gradually builds up its credit as a response to the packet header statistics of the input link.

Equation (4.4) uses the indices of the filters to assign a credit. Compared with a filter with a lower index, a filter with a relatively higher index will require more matches to prove its dominance. For example, in Figure 4.3, filter 15 shows potential for dominance. Filter 15 needs more matching instants to build up the necessary credit to be dominant, and hence slowly converges to the ideal one-memory probe. On the other hand, a filter with relatively lower index as filter 5, requires fewer matches to achieve the same rank. Assume we have a burst with specific packet header statistics biased towards a filter with a relatively higher index. This filter will require many more matches to be labeled dominant. Consequently, a longer period may elapse before the search engine starts benefiting from this technique. Similarly, the burst period may be too short, giving the filter with relatively higher index insufficient time to build the required credit. In this situation, unnecessary memory probes are required because this filter cannot be fast enough to build its credit. Thus, the information carried by the previous *iPMF* values are not utilized in an efficient way because the granted credit was influenced by the index of a given matching filter.

To alleviate the drawbacks stated above, we propose a way to help filters with relatively higher indices to accumulate credit faster, requiring less matches, and without overriding any of the requirement of the update criteria above. The following section discusses the proposed method.

4.4 The *Indexed* Gaussian-credit Updates for iPMF

As stated in the previous section, using Equation (4.4), filters with relatively higher index have difficulty in building up their credit. We attempt to alleviate this difficulty by offering *additional* credit to matching filters. Moreover, to allow for enhanced utilization of the information contained in the previous credit values, the additional credit is a function of the previous credit of a matching filter. This is accomplished without

violating the first requirement of the update criteria stated in the previous section. Thus, in addition to the 4 requirements of the update criteria outlined for *AIPS*, we add the following requirements:

1. Additional credit should be given to a matching filter with filter N granted the maximum *additional* credit. Alternatively, filters with relatively lower indices should not gain as additional credit.
2. The additional credit should be influenced by the values of the previous credit.

One way of satisfying these requirements is by using a Gaussian function. The additive credit is based on the Gaussian distribution function. Note that other functions can be used if they satisfy the requirements stated above.

A random variable φ is called Gaussian if its density function has the form

$$f(\varphi) = \frac{1}{\sqrt{2\pi\delta^2}} e^{\frac{-(\varphi-\mu)^2}{2\delta^2}} \tag{4.7}$$

This function has its maximum value $1/\sqrt{2\pi\delta^2}$ at $\varphi = \mu$, where μ and δ^2 are the mean and variance of φ respectively. The spread of the function around its maximum at point $\varphi = \mu$ is controlled by δ^2.

Based on the brief background of the Gaussian function, the additional credit is constructed as follows.

For a matching filter *m*, an additional credit is added to the value expressed in the nominator of Equation (4.4), and the update formula is given by

$$p_m(n) = \frac{p_m(n-1) + \frac{p_m(n-1)}{m} + G(m)\sum_{k=2}^{\infty} p_m^k(n-1)}{1 + \frac{p_m(n-1)}{m} + G(m)\sum_{k=2}^{\infty} p_m^k(n-1)} \tag{4.8}$$

Where

$$G(m) = \frac{2}{\sqrt{2\pi\delta^2}} e^{\frac{-(m-N)^2}{2\delta^2}} \tag{4.9}$$

The Gaussian function is doubled since m takes integer values > 0. Given that $p_m(n-1)<1$, we can rewrite Equation (4.8) as

$$p_m(n) = \frac{p_m(n-1) + \frac{p_m(n-1)}{m} + G(m)\left[\frac{p_m^2(n-1)}{1-p_m(n-1)}\right]}{1 + \frac{p_m(n-1)}{m} + G(m)\left[\frac{p_m^2(n-1)}{1-p_m(n-1)}\right]} \tag{4.10}$$

$$= \frac{(1+m)p_m(n-1) + (m\,G(m) - 1 - m)p^2{}_m(n-1)}{(1-m)p_m(n-1) + (m\,G(m) - 1)p^2{}_m(n-1) + m} \tag{4.11}$$

Equation (4.11) differs from Equation (4.4) in the added Gaussian-credit. There are two cases, however, where Equation (4.11) will behave similar to Equation (4.4). The first case is when the matching filter has a very small index compared to N. In this case Equation (4.11) will approximate to Equation (4.4) since the value $(m-N)$ is relatively large, this is achieved by assigning $\mu = N$ in the Gaussian function. Hence, the maximum value of the additive Gaussian term will always be assigned to the filter with the largest index, which is the filter with index N, and the minimum value of the additive Gaussian term will be assigned to the filter with minimum index, which is filter with index 1. Consequently, only filters with higher indices will gain significant benefit from this additive Gaussian term. The second case is when the *iPMF* value of the matching filter at time $(n-1)$ is too small, indicating that the filter has no potential for dominance for the period of this burst. Therefore, the additive Gaussian credit will have a negligible value, this is achieved by granting the additional credit as a *decaying polynomial* in $p_m(n-1)$.

The value of δ is a tunable parameter that determines the spread of the additive Gaussian credit over the N filters. For all other non-matching filters, Equations (4.5) and (4.6) are used.

We refer to a probing technique utilizing this update mechanism, as the Adaptive Indexed Gaussian-credit Probing Sequence *(AIGPS)*.

Figure 4.4 shows a comparison between the accumulated credit for matching a filter with index 5000 using *AIPS* and *AIGPS* in a *SC* of size $N = 5000$. This way, we are examining the gains achieved with and without using the additive Gaussian credit for the filter with highest index in a relatively large *SC*.

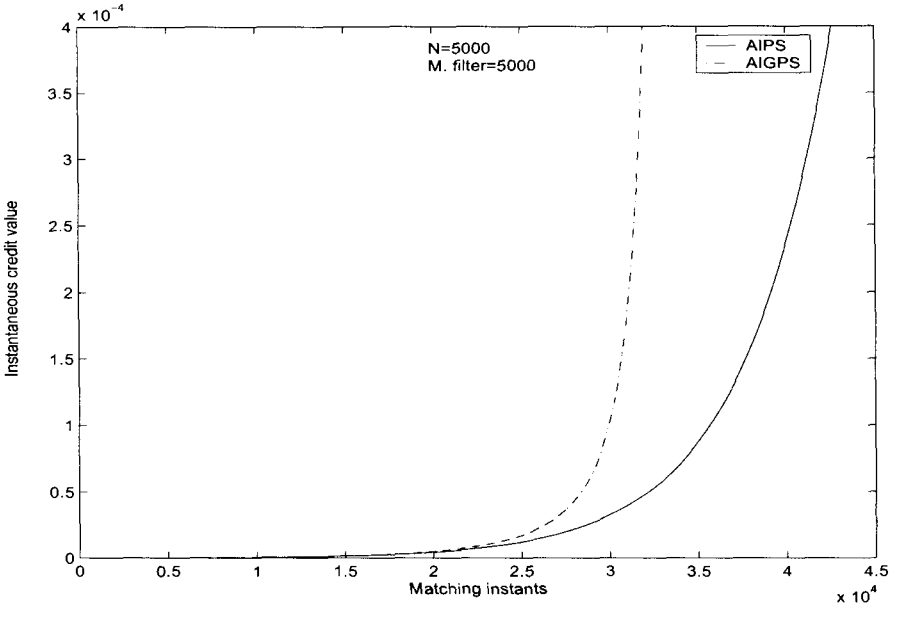


Figure 4.4: Credit accumulation for filter N with and without Gaussian credit

For this example, the *IHG* initialization method is used. Using Equations (4.4) and (4.11), the matching filter acquires *the same* credit slowly and gradually until matching instant 2.2×10^4. After this particular matching instant and using Equation (4.11), the Gaussian-

credit gains a considerable additive value since the matching filter has shown potential for dominance as judged by the amount of credit successfully accumulated so far. Note that this matching filter of maximum index and minimum initial *iPMF* value reached a credit of $N/\sum_{m=1}^{N} m$ after 3×10^4 matching instants using the additive Gaussian-credit. On the other hand, using Equation (4.4), the filter needed more than 4×10^4 matching instants to accumulate the same credit. That is, the filter saved a time equivalent to more than 10 000 matching instants and yet reached the same rank.

As a conclusion, Equation (4.11) demonstrates a different behavior from Equation (4.4) only when needed. That is, if we have a filter with a relatively high index and this filter has shown relative potential for dominance by accumulating credit as judged by its *iPMF* value at time instant $(n-1)$, the added Gaussian-credit is granted. If the same filter did not show any potential for dominance, the Gaussian-credit will have negligible value. On the other hand, if a filter with relatively low index, shows potential for dominance or not, the Gaussian-credit will not be of any significant value, simply because a filter with relatively low index does not need this extra credit. Hence, the new requirements are satisfied.

Although *AIGPS* proved more effective than *AIPS,* there are 3 concerns. First, the value of the parameter δ, even though tunable, is static. A small value for this parameter may deprive filters with relatively high index from the benefit of the Gaussian-credit, and a large value defeats the purpose of the mechanism. Therefore, an optimum value for this parameter is not easily defined as it imposes credit trade-offs.

The second concern is regarding the use of pointers. As stated earlier in this chapter, when more than one filter matches a given packet header, pointers link to higher priority filters that can be a potential match. If such pointers exist, the search engine switches to a

higher priority filter and checks for a match. If a match is found, the associated action is applied and the filter gets the credit. Otherwise, the current matching filter is the only possible match. If a pointer to a higher priority filter does not exist, then the action associated with the first matching filter is applied. In case a pointer exists, memory probes to other filters have to be made *anyway* to check for a match. Thus, indexing the granted credit is a limitation to the update mechanism.

The third concern arises due to the possible existence of *strong traffic* locality. For example, a *Denial-of-Service* attack *(DoS)* can produce instability in the update mechanism. This instability is due to the following. In such a situation, a given filter can accumulate credit rapidly resulting in an *iPMF* value of 1, at which case, the update values of all non-matching filters will be undefined, and even if the *strong locality* situation is over, any new matching filter will not be able to accumulate credit since its *iPMF(n-1)* value is 0 or undefined. These concerns motivated the design of a new update mechanism, which is discussed in the following section.

4.5 The Gaussian-credit Updates for iPMF

In previous sections, we proposed two credit-update formulas that took the index of the filters into account. Mechanisms using these formulas showed effectiveness. However, 3 concerns were raised as discussed previously. Therefore, we propose a new formula to update the credit of a matching filter based on the following criteria.

1. The granted credit should *not* be influenced by the index of a matching filter. Instead, it should be influenced entirely by the values of the previous credit.
2. The credit for the matching filter should be ≤ 1 for any n.
3. The update mechanism should be stable for all values of *iPMF(n)*, where $0 \leq iPMF(n) \leq 1, for\ any\ n$
4. Static parameters are not permitted.

5. At the end of the update mechanism, Equation (4.3) should be satisfied.

To satisfy these requirements we propose the following formula to update the credit value of matching filter *m*.

$$p_m(n) = \frac{p_m(n-1) + e^{-(1-p_m(n-1))^2}}{1 + e^{-(1-p_m(n-1))^2}} \tag{4.12}$$

We use Equations (4.5) and (4.6) to update all other non-matching filters. Except for the case when $p_m(n) = p_m(n-1)$, Equation (4.6) is assigned a value of 1.

Note the formula does not require pre-specified parameters similar to the spread factor of Equation (4.11), and the granted credit is not influenced by the index of the matching filter but rather is determined by the *negative* exponent $(1- p_m(n-1))^2$. In addition, a strong-locality situation as described in the previous section is supported, if the new credit value reaches a value of 1, a filter can still build credit starting from $p_m(n-1)=0$. At which case, the new credit will be $p_m(n) = 1/1+e \approx 0.27$.

A probing sequence utilizing this update formula is referred to as the Adaptive Gaussian-credit Probing Sequence *(AGPS)*.

Figure 4.5 shows an example for the behavior of *AGPS*. A matching filter with any given index *m* is assigned an initial credit of 0. Assuming a continuous match, the filter starts building its credit rapidly and reaches a credit of 0.27 and 0.54 on the second and the third matching instants respectively. At matching instant 56 the credit reaches a value of 1 and stays at that value. Factor β of Equation (4.6) is represented by the dotted line. This factor acquires a value of 0.73 on the second matching instant, then starts decaying to 0.5 and settles around this value until matching instant 56 where it drops to 0, indicating that the new credit value reached 1. On instant 57, the credit value is computed

as 1 according to the update formula, which is equal to the credit value at the previous instant, therefore, β is assigned a value of 1.

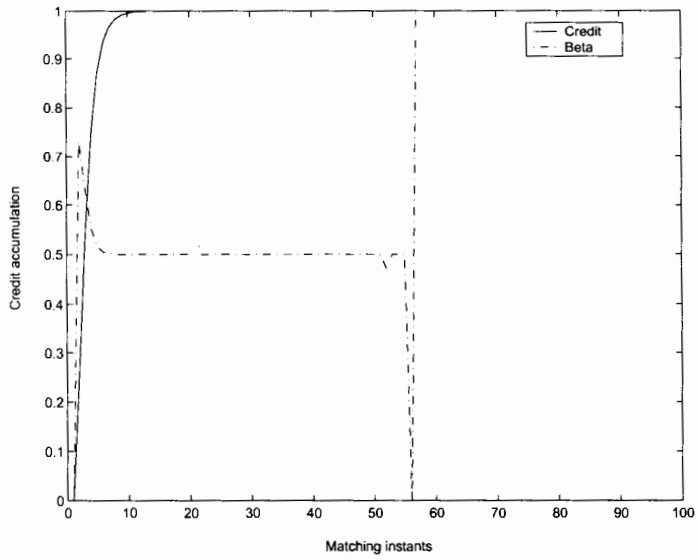


Figure 4.5: Credit accumulation for a matching filter using AGPS

Figure 4.6 shows the step sizes for the accumulated credit and factor β. The step size is defined as the difference between the current value and the previous value.

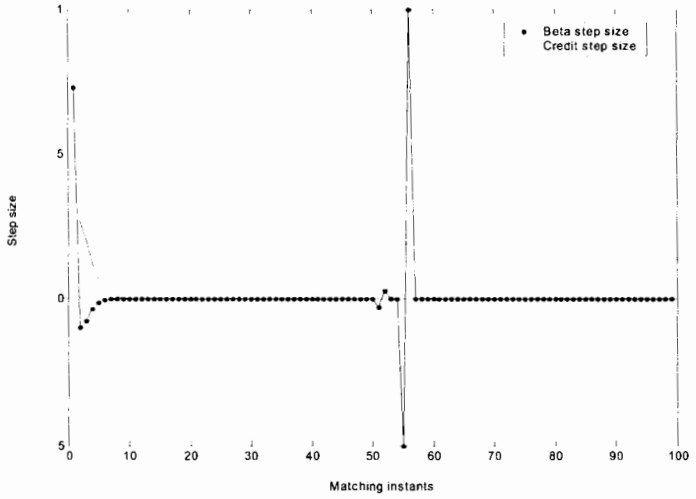


Figure 4.6: Step size for credit and β using AGPS

Note the step size of the acquired credit starts high and then decays as the credit value approaches 1.00. Afterwards, the step size settles at 0 since the previous credit and the current credit values are locked at 1. The step size of β starts high then undershoots until it settles to a value of 0. On instant 56, the step size of β drops to -0.5 indicating that the current credit value has reached 1.00. On the following instant, the current and the previous credit values are equal, thus, β is assigned a value of 1. Therefore, its step size goes to 1, then settles to 0.

The credit value of 1 can represent a special traffic behavior. This representation can be used for certain applications as presented in Chapter 6. In the following section, we analyze the performance of the probing sequence introduced in this chapter.

4.6 Behavior and Analysis of AGPS in Static Classifiers

We use two transition diagrams to model the behavior of *AGPS* and analyze its performance. Given a classifier with N filters, we assume the following for a *desired probing technique* represented in the transition diagram of Figure 4.7, where *"c"* is the probability a packet arrives for classification.

1. Each state of the diagram represents the number of accumulated memory probes made by a given probing technique to find the matching filter.
2. A *desired* probing technique is used. The transition probabilities, referred to as the *target/desired transition probabilities* $p_t(m)$, take values as described in Equation (4.2). A *desired transition probability* from state m to state 0, represents the desired probability of producing a match in m memory probes. Therefore, these probabilities are assigned values using the normalized inverted version of the matching distribution exhibited by *LP* technique, as shown in Chapter 3. That is, we *desire* a match to occur in one memory probe with the highest probability.

Alternatively, the probability of making an additional memory probe p_{wf} decreases with additional memory probes.

3. Filters are probed according to their $p_t(m)$ values starting with the filter with maximum $p_t(m)$. If this filter does not match, the filter with next to maximum $p_t(m)$ is probed and so on, until a match is found, otherwise the default filter is applied.

4. We define the hold time as the processing time T_P.

5. The system goes to the idle state numbered 0, after a match occurs.

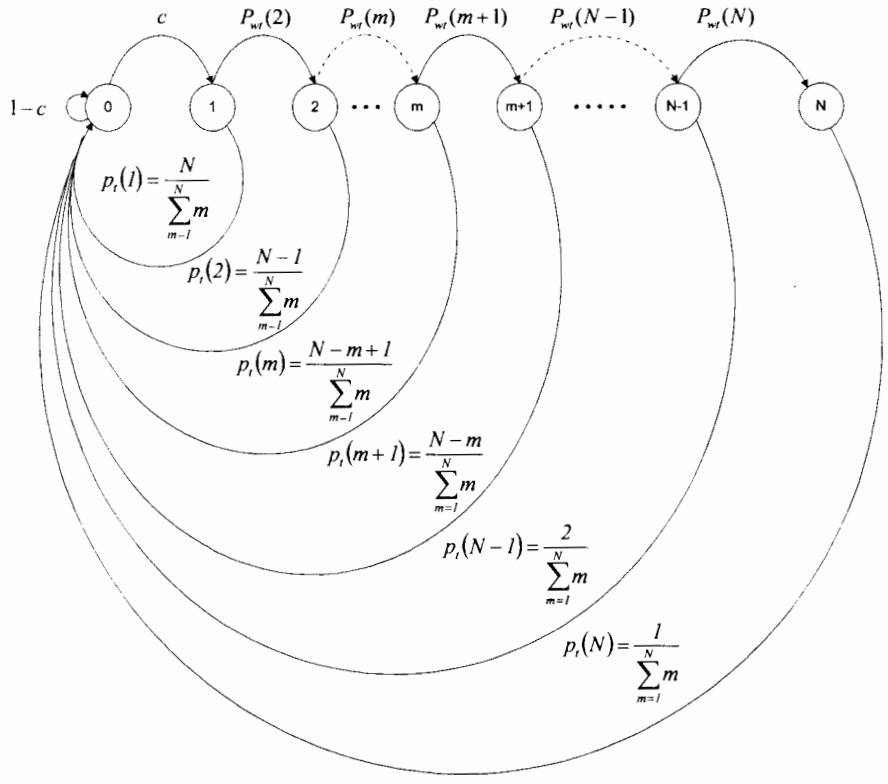


Figure 4.7: A transition diagram representing the desired probing technique.

The *AGPS* technique, described in this chapter, can be mapped to the desired probing technique. Figure 4.8 presents the behavior of *AGPS*, as a dashed line, superimposed over the transition diagram of the desired probing technique. For the behavior of *AGPS*, we assume the following:

1. Each state on the diagram represents the *sequence number* according to which a given filter is probed.
2. The transition probability $p_h(m)$ identifies the probability a filter acquires a probing sequence number $< m$ when it *matches,* this is therefore equivalent to the matching probability. Similarly, a non-matching filter with a probing sequence number j is assigned a new probing sequence number $> j$, with probability equivalent to p_{wt}.
3. We define the hold time as the processing time T_p.
4. *AGPS* is initialized with any of the initial assignment methods described in Section (4.2).
5. Filters are probed in a sequence based on their *iPMF* values starting with the filter of maximum *iPMF*[1]. If this filter does not match, the filter with next to maximum *iPMF* value is probed and so on, until a match is found, otherwise the default filter is applied.
6. If filter with index N has the maximum *iPMF*, the filter with next to maximum *iPMF* is probed and so on, otherwise the default filter is applied.
7. Assuming the existence of traffic locality, all *iPMF* values are updated according to Equations (4.5) and (4.12). The sequence number of the matching filter is therefore decremented, and *AGPS* goes to the idle state.

---

[1] If *iPMF* values are equal, filters are probed in increasing order of their indices.

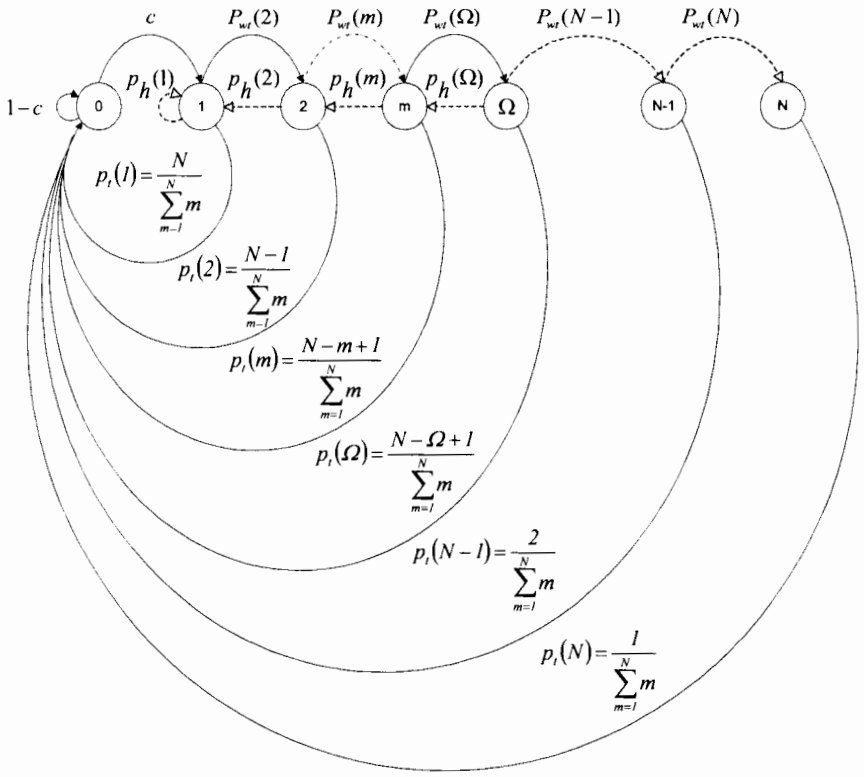


Figure 4.8: A transition diagram of AGPS superimposed over the transition diagram of the desired probing technique.

In Figure 4.8, filters are initially probed according to their initial *iPMF* values according to an initial probing sequence 1, 2, 3 ... Ω. Assume a given packet produces a match with a given filter F in Ω memory probes with assumed desired matching probability $p_t(\Omega)$. Based on assumptions 5 and 7 above, *iPMF* values are then updated, therefore, the matching filter acquires a probing sequence number $m < \Omega$ with probability $p_h(\Omega) = p_t(\Omega)$. Thus, according to the new probing sequence 1, 2, 3 … m, the filter produces a match in $m < \Omega$ memory probes with probability $p_t(m) > p_t(\Omega)$ on the following matching instant. Assuming the matching filter stays dominant, it will eventually acquire a probing sequence number $1 << \Omega$, and a match can be produced in

$1 << \Omega$ memory probe with probability $p_t(1) >> p_t(\Omega)$ according to a probing sequence 1.

The average number of packets processed per one memory probe or the average throughput (as defined in Chapter 3) of *AGPS* is given by

$$Th_{AGPS} = \left[1 \times p_t(1) + \frac{1}{2} \times p_t(2) + \cdots + \frac{1}{m} \times p_t(m) + \cdots + \frac{1}{N} \times p_t(N)\right]$$

$$= \frac{\sum_{m=1}^{N} \frac{N-m+1}{m}}{\sum_{m=1}^{N} m}$$

$$= \frac{2\sum_{m=1}^{N} \frac{N-m+1}{m}}{N^2+N} \quad \text{packets/memory probe} \tag{4.13}$$

where $1 \le m \le N$.

Similarly, the average number of memory probes required by *AGPS* to classify one packet in a classifier of N filters is given by

$$S_{AGPS} = \left[1 \times p_t(1) + 2 \times p_t(2) + \cdots + 3 \times p_t(m) + \cdots + N \times p_t(N)\right]$$

$$= \frac{\sum_{m=1}^{N} m(N-m+1)}{\sum_{m=1}^{N} m}$$

$$= \frac{2\sum_{m=1}^{N} m(N-m+1)}{N^2+N} \quad \text{memory probes} \tag{4.14}$$

4.7 AGPS in Dynamic Classifiers

As established in Chapter 3, in a *DC*, filters can be dynamically inserted or deleted. Adopting the same criteria that we used to probe a *SC*, we propose to probe a *DC* in a similar fashion. However, the *iPMF* values of the filters must be updated upon filter insertion, deletion, and upon filter matching without altering the relative *iPMF* values. Moreover, Equation (4.3) must be satisfied. In the following sections, we present *iPMF* updates upon filter insertion, deletion, and matching.

4.7.1 iPMF Values Update upon Filter Insertion

Using *LP* technique in *DC*, filters were probed in a decreasing order of their indices, since it was assumed that the inserted filter would be used in the near future. Using *AGPS,* filters are probed according to their *iPMF* values. In a *SC*, initial *iPMF* values were pre-assigned. However, in a *DC*, these initial values must be assigned dynamically when a filter is inserted. A good method, therefore, is to always assign the maximum *iPMF* value to the inserted filter. We use the following formula to assign an initial *iPMF* value $p_{z(n)}$ to an inserted filter with index *Z(n)*.

$$p_{Z(n)} = \frac{Z(n)}{\sum_{i=1}^{Z(n)} i} \tag{4.15}$$

Where $Z(n)$ is the size of the *DC* at the current time step n.

For all other *Z(n)-1* filters, we evaluate their *iPMF* values using

$$p_i(n) = \beta\, p_i(n-1), \text{ for all } i \neq Z(n) \tag{4.16}$$

where $p_i(n)$ is the *iPMF* value of the i^{th} filter at time instant n, and

$$\beta = 1 - p_{Z(n)} \tag{4.17}$$

Equation (4.17) is the same as Equation (4.6), except that the previous *iPMF* value of the inserted filter is 0.

4.7.2 iPMF Values Update upon Filter Deletion

In a *DC,* one or more filters can be deleted because either they are no longer needed or the maximum allowed size B of the classifier is reached and a filter needs to be inserted. In this case, the *iPMF* values must be updated to reflect this change without affecting the relative credit values of the other filters. Thus, to delete a set of filters S_D of size D filters at time step n, where the size of the classifier is $Z(n)$, we distribute the sum of the *iPMF* values of the D deleted filters amongst all remaining $Z(n)-D$ filters as in

$$p_i(n) = \frac{\sum_{d \in S_D} p_d(n)}{Z(n)-D} + p_i(n), \text{ for all } i \notin S_D \tag{4.18}$$

Where $p_d(n)$ is the *iPMF* value of filter with index d at instant n.

4.7.3 iPMF Values Update upon Filter Matching

We use Equation (4.12) to update the *iPMF* value for any matching filter m at time step n, where $1 \leq m \leq Z(n)$. We use Equations (4.5) and (4.6) to update the *iPMF* values of all other $Z(n)$-1 filters.

In the following section, we analyze the performance of *AGPS* in *DC*.

4.7.4 Behavior and Analysis of AGPS in Dynamic Classifiers

We use the same method, described in Section 3.4, to describe and analyze the behavior of *AGPS* in a *DC*. The only differences however are the following assumptions.

1. The maximum allowed size for the *DC* is *B*. When a filter needs to be inserted while the classifier is already at its maximum size, the filter with minimum *iPMF* value is deleted.
2. When a packet arrives, filters are probed according to their *iPMF* values starting with filter of maximum *iPMF* value. If this filter does not match, the filter with next to maximum *iPMF* value is probed and so on, until a match is found otherwise the search turns to the *SC*.

The average size of the classifier is computed using the same Markov chain analysis presented in Section 3.3. The average throughput of *AGPS* in *DC* is therefore,

$$Th_{AGPS} = \frac{\sum_{m=1}^{Q} \frac{Q-m+1}{m}}{\sum_{m=1}^{Q} m}$$

$$= \frac{2\sum_{m=1}^{Q} \frac{Q-m+1}{m}}{Q^2+Q} \quad \text{packets/memory probe} \quad (4.19)$$

where $1 \le m \le N$ and Q is the average size of the *DC*.

In the following section, we introduce an experimental performance comparison between *LP* and *AGPS* in both static and dynamic classifiers.

5 Experiments and Results

The effectiveness and adaptability of *AGPS* is demonstrated in this chapter. We show the performance of *AGPS* with various sizes of classifiers and varying degrees of traffic locality.

Due to the limited availability of real life classifiers of different structures and arbitrary sizes, and since *AGPS* is independent of the structure of the classifier, we generated virtual classifiers of various sizes. Many researchers adopted this approach due to the limited availability of real classifiers [13][14][15][16][17]. The use of an implemented or non-industrial classifier should not however affect the performance of *AGPS*, which does not propose a structure-dependent solution, but rather proposes a statistical solution for the packet classification problem. Therefore, the only purpose for the generated classifiers was to compute the number of packets that are a possible match to a given filter. This number was then mapped to a unique discrete range on the number line. Each unit on this number line is refereed to as a Packet Header Label *(PHL)*. In doing this, we mapped each filter to a deterministic range of labels. This type of mapping is widely used in the literature, where researchers convert fields that are not expressed in ranges to prefixes [14], or convert fields that are not expressed in prefixes to ranges [12] [46] as a preprocessing step that suits their proposed packet classification techniques. This technique is based on defining a filter as *the set of all packets that could match it* [16]. The definition is in agreement with the function assigned to each filter as a flow classifier.

PHLs were generated randomly from within the mapped ranges described above. Generating *PHLs* was controlled to accommodate varying locality degrees of network traffic. The randomness of *PHLs* generation process was determined by a random number generator driven with a heavy tailed-distribution function to simulate, long-range dependence, and self-similarity of network traffic [1][6][37][47].

Simulations were performed using the Statistical Tool Box of MATLAB® 6.1. The structures of real life classifiers were taken from *iptables*, the Linux[1] firewall administration program.

The following sections explain the process of generating virtual classifiers, *PHLs*, and the structure of a simulation bench.

5.1 Filter Generation

We want to generate a classifier of a given size *N-1* and *k* dimensions[2], where *N* increases in size according to the order 100, 200, 300..., etc. We generate classifiers with two dimensions, namely the Source IP network prefix and the Destination IP network prefix. Nevertheless, the performance of *AGPS* is not influenced by the dimension or structure of a given classifier, and therefore the results are applicable to the *k*-dimensional case.

A uniform random number generator is used to generate a 2-dimensional filter at a time. For each dimension of the filter, we randomly select a prefix length. For a chosen prefix length, we randomly generate 4 binary octets. The octets are then concatenated to form a field.

---

[1] RedHat Linux 7.3-kernel 2.4, *iptables 1.2.6a.*

[2] The default filter need not be generated, as it has a zero length for all dimensions.

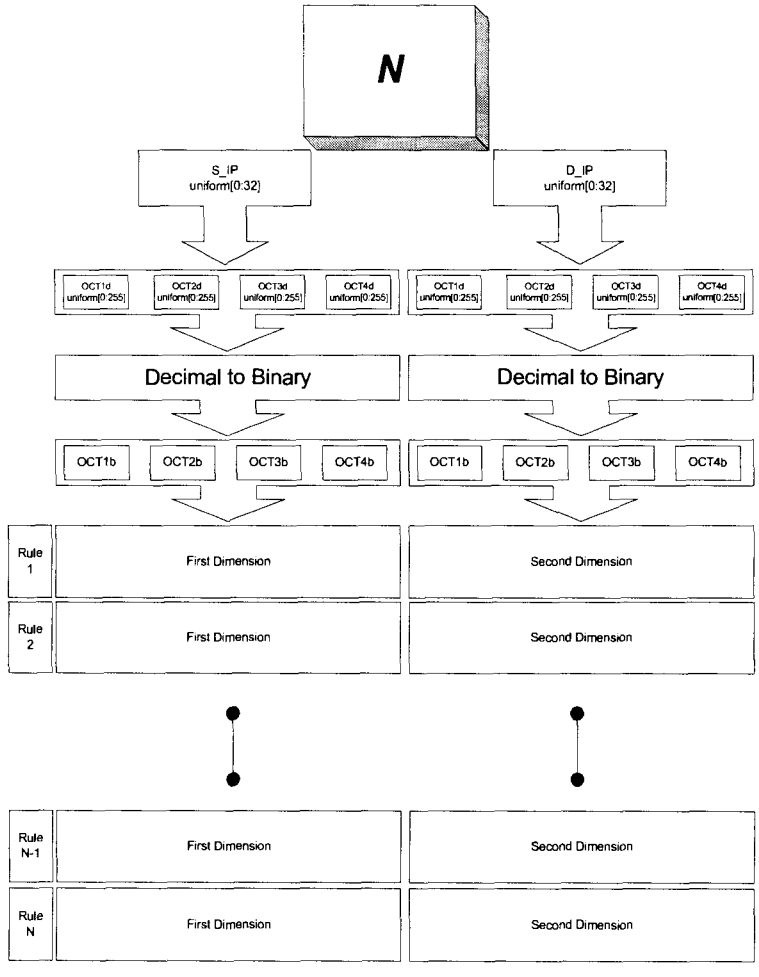


Figure 5.1: Filter generation

Figure 5.1 shows the classifier generation process. The process starts with a given desired size *N* for the classifier to be generated. A uniform random number generator seed is set based on the system clock.

The length of one dimension of a given filter is then chosen randomly using a uniform random number generator in the range [0:32]. Once the length of the specific dimension is decided, we uniformly generate up to 4 decimal numbers from the range [0:255] depending on the chosen length from the decimal range. Each decimal number is then converted to a binary octet. The 4 octets are then concatenated to form one dimension.

The process is repeated for the other dimension. A 2-dimensional filter is then created. The entire process is repeated again to generate another filter and so on, until filter *N-1* is generated.

To assign a matching priority for each filter, the bit string length of the first dimension of filter *m*, denoted $BSL(m)_1$, is added to $BSL(m)_2$, the resulting value total-*BSL(m)* decides the priority of filter *m*. Filters with larger total-*BSL* are assigned a higher priority.

In order to solve for overlapping filters, a bit-wise comparison is used. A vector *PTR(m)* is associated with each filter *m,* where *PTR(m)* stores the index of a higher priority filter *x* that overlaps with *m*. If more than one filter overlaps with *m*, all indices of the overlapping filters are stored in *PTR(m)* in order of their priority as in

$$PTR(m) = [x_1\ x_2 \cdots x_v],\ where\ x_1 < x_2 < \cdots x_v \qquad (5.1)$$

In [13] and [45] a number of moderate to large size classifiers were analyzed to determine the maximum number of filters that a given packet can match. The authors observed that a given packet could at most match 4 filters. Their observations agreed with a study made in [16], which concluded that the number of overlapping filters in a given classifier is low. The same conclusion was drawn in [17]. Hence, in our experiments, we controlled the size of vector *PTR(m)* to vary from $4 \sim 10$.

5.2 Packet Header Label Generation

Once the classifier is built, we computed the range of labels for each filter *m* based on the *BSL(m)* of all dimensions. Each range was recorded in vector *Labels(m),* where the size of *Labels(m)* is

$$Size\_Labels(m) = \prod_{i=1}^{k} 2^{32-BSL(m)_i}, \text{ where } 1 \le m \le N-1 \qquad (5.2)$$

and k is the dimension of the classifier. Part of the size of the labels assigned to the default filter is assumed equivalent to *max [Size_Labels(m)]*. We distinguish these N ranges as the set of *pre-determined* labels. The maximum value for a pre-determined label is P_α, where

$$P_\alpha = \sum_{1}^{N} Size\_Lables(m), \text{ where } 1 \le m \le N \qquad (5.3)$$

Ranges of all N filters were recorded in a matrix called *Packet-Header-Label Super Space (PHLSS)* with size $N \times 2$, where 2 is the dimension of the classifier.

At a given sampling instant *i,* a portion of up to 0.1% is taken from *Size_ Labels(m)* for all $1 \le m \le N$. This produces a sample range $S_m(i)$ with size $Size\_S_m(i)$. Sampling *Labels(m)* and hence *PHLSS* is intended to reduce the simulation processing time and accommodate memory resources. The sampling process however is repeated many times, until the following condition is met.

$$\frac{\sum_{i=1}^{K} P_\beta(i)}{P_\alpha} \ge 0.01 \qquad (5.4)$$

Where

$$P_\beta(i) = \sum_{m=1}^{N} Size\_S_m(i), \text{ at sampling instant i, where } 1 \le i \le K \qquad (5.5)$$

and K is a variable that represents the number of sampling instants. The 1% ratio in Equation (5.4) was chosen to accommodate resources. It can however be changed without affecting the results.

We refer to the resulting range of labels after sampling *PHLSS* at instant i as *Packet-Header-Label sub-Space (PLHS(i))* of size $N \times 2$, and maximum range value $P_\beta(i)$.

5.3 Simulation Bench for Static Classifiers

This section describes the simulator setup. Random number generators were used extensively to impose variable working conditions on the proposed technique and to demonstrate its capability to adapt and optimize its performance independent of the varying environment variables. The random number generators as well as their respective parameters were based on results of previous research work that will be referenced in this section where appropriate.

Evaluating the performance of *AGPS* constitutes testing under traffic flows with localities that vary from weak to strong traffic localities, as described in Section 2.3.1. Flows come in traffic bursts that are self-similar [1][37], thus, self-similar processes were widely used to model forms of traffic as Ethernet traffic[49], VBR traffic[47], and recently for wide area traffic [48][50]. A self-similar process is typically associated with a long-range dependence phenomenon, in the sense that the process has an autocorrelation function that is hyperbolically decaying [37].

A widely adopted modeling method for bursty-self-similar traffic that demonstrates long-range dependence used a heavy-tailed distribution [47][37][6][1]. The Pareto distribution is well acknowledged as a good fit for most self-similar traffic traces, therefore, it was usually used as a form of a heavy-tailed distribution [1][6][37][47]. The shape parameter of the Pareto distribution however, must be adjusted to satisfy the characteristics of a heavy-tailed distribution such as a high variance[1].

Based on the brief introduction above, we demonstrate the process of generating synthetic traffic.

Figure 5.2 is a block diagram representing a simulation bench. Given the size of a classifier, N filters are generated as described in Section 5.1. P_α and P_β are computed as described in Section 5.2.

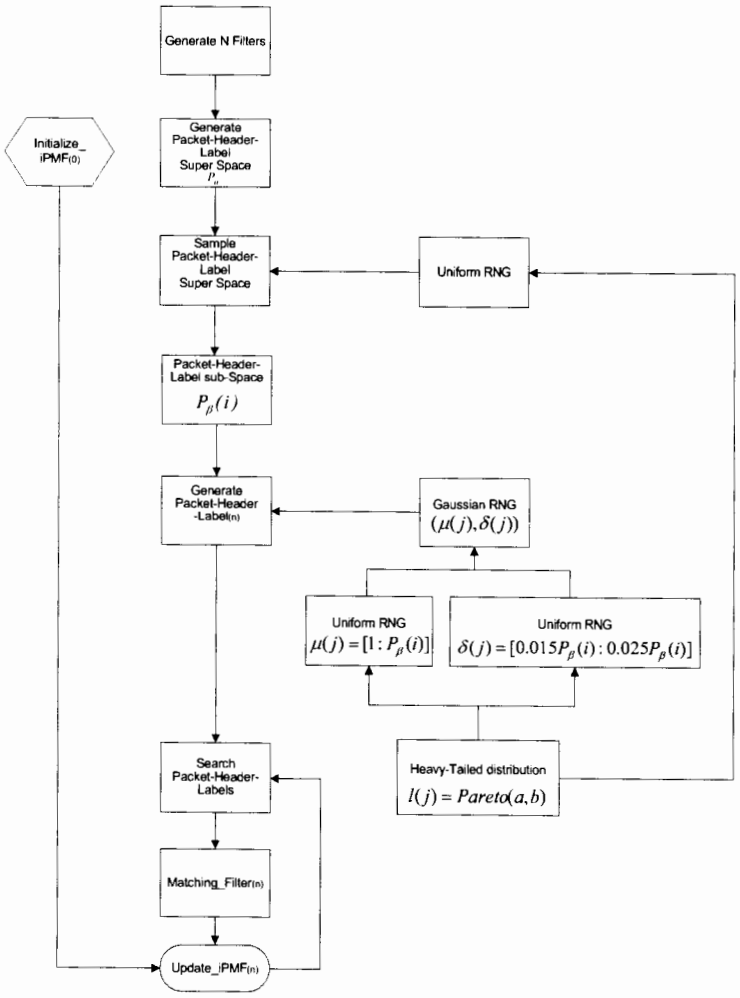


Figure 5.2: Simulation bench for static classifiers

As shown in the figure, based on a Gaussian distribution, we start by generating Packet Header Labels *(PHLs)* from a discrete range $[1:+\infty]$. The parameters of the random number generator are used only for a certain period of time *l(j)*, after which different

parameters are generated using a uniform random number generator for each time instant j, where $\mu(j)$ is chosen in the range $[1 : P_\beta(i)]$ and $\delta(j)$ is chosen from the range $[0.015\,P_\beta(\mathrm{i}) : 0.025\,P_\beta(\mathrm{i})]$. We refer to [2][35][36] and [39] for our choice of $\delta(j)$. These studies almost agreed that around 50% of the traffic is directed to 3% of the filters. We however included up to 15% of the traffic during the label generation process [33]. This produced spatial localities that are 5 times weaker than reported in literature.

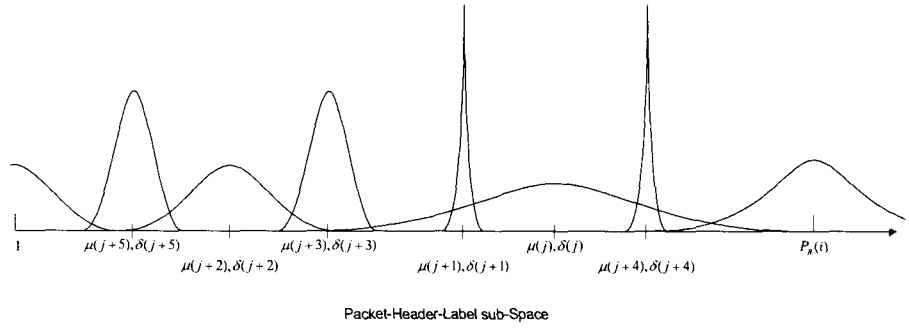


Figure 5.3: PHL generation and locality simulations

It has been shown that the distribution of burst sizes in general is heavy-tailed [37]. Hence, as stated earlier we chose to use the Pareto distribution as a heavy-tailed distribution to decide about the time period *l(j)*, after which we uniformly generate new Gaussian parameters $\mu(j+1)$ *and* $\delta(j+1)$. That is, the time value *l(j)* of a particular Gaussian period at time instant j (we refer to this period as the burst period) is a random variable that is heavy-tailed distributed with a *PDF* given by

$$p(l(j)) = \frac{ba^b}{l(j)^{b+1}}, where \;\; a \le l(j) < \infty \tag{5.6}$$

where $1 \le j \le P_\beta(i)$, a is the minimum burst period of value $a=1$, and b is the shape parameter which influences the variance of the distribution. For the case at hand, where we require a bursty-self-similar traffic, b is chosen uniformly in the range [1.4:1.6][1], thus we achieve a high variance for the Pareto distribution.

As illustrated in Figure 5.3, the time varying choice of $\mu(j)$ represents the *PHLs* of high frequency of reuse. Depending on the spread factor $\delta(j)$, for a given time instant j, other labels are generated with different probabilities that are inversely proportional to their distance from $\mu(j)$ [32].

The following sections describe the method of assigning the initial values for *iPMF*, the process of filter matching and updating the *iPMF* values at each time instant n. From this point forward, we use n to denote a global time index for a given experiment.

5.4 iPMF Initialization

Experiments were run using uniform, *IHG*, and random initialization of the *iPMF*. The effect of choosing one over the other was observed to be insignificant. Hence, the initial values of the *iPMF* should not affect the values of the *iPMF* on the long run. Using uniform initial values can reduce the length of the very first training period, but this virtue will have no effect afterwards. In contrast, using *Inverted Hyper-Geometric* initial values can lengthen the first training period, especially if the first dominant filter happens to be of a larger index, this drawback however will not affect the overall performance on the long run. Random-initialization can possibly result in a long initial training period, it can also result in a short training period with equal probability.

Although the choice of the initialization method does not affect the results, it is however, an essential step. Thus, we chose to uniformly initialize the *iPMF* based on the observations stated above.

5.5 Matching Labels and *iPMF* Update

For each instant j, a Gaussian combination $\mu(j)$ *and* $\delta(j)$ is chosen and is in effect for a burst length *l(j)*. During this period, *l(j)* possible *PHLs* can be picked from the range $[1:\infty]$. Based on the value of a given *PHL* at instant n (*PHL(n)*), the label is examined sequentially amongst the ranges of the N filters based on *iPMF(n)* values. Once a matching filter m is found, *PTR(m)* is examined. If *PTR(m)* is empty then m is the only matching filter in the classifier. Otherwise, the labels associated with the elements stored in *PTR(m)* are checked according to their order. Consider Equation (5.1), if *PHL(n)* matches $Labels(x_2)$, then filter with index x_2 is considered the matching filter, otherwise the search proceeds to $Labels(x_3)$ and so on until the matching filter is found and *iPMF(n)* is updated accordingly, as described in Chapter 4. The number of memory probes required for this search is recorded as the number of elements of *PTR(m)* that needed to be examined to reach a matching filter in addition to 1 memory probe for probing *Labels(m)*. If *PHL(n)* does not belong to any of the filters in *PTR(m)*, then m is the only matching filter and its *iPMF(n)* is updated accordingly. In this case, the number of memory probes required for this search is recorded as the number of all v elements in *PTR(m)* in addition to 1 memory probe for probing *Labels(m)*.

For the case when the default filter is the matching filter, the *iPMF* values are updated and the number of memory probes for this search is recorded as N.

In general, once the matching filter is found at instant n, the *iPMF* values of all filters are updated and fed back to the search engine to guide a probing sequence for another search at time instant $n+1$.

5.6 Referring to Linear Probing

Linear probing was used as a reference to evaluate our adaptive probing technique. All experiments were run twice. Once using *AIGPS* or *AGPS*, and another time using *LP*. Seeds for random number generators used were strictly maintained to assure that all probing techniques were tested in the same conditions and variables. For a given *PHL(n)*, *Labels(m)* is examined for a match for all $1 \leq m \leq N$ in a linear fashion according to the decreasing order of filter indices. The number of memory probes required for a given search is recorded as the index of the matching filter *m*.

5.7 Results and Analysis for Static Classifiers

In this section, we present the results of our simulations in *SC*. We compared *AIGPS* to *AGPS* to validate our choice as to which technique should be used for the rest of the simulations.

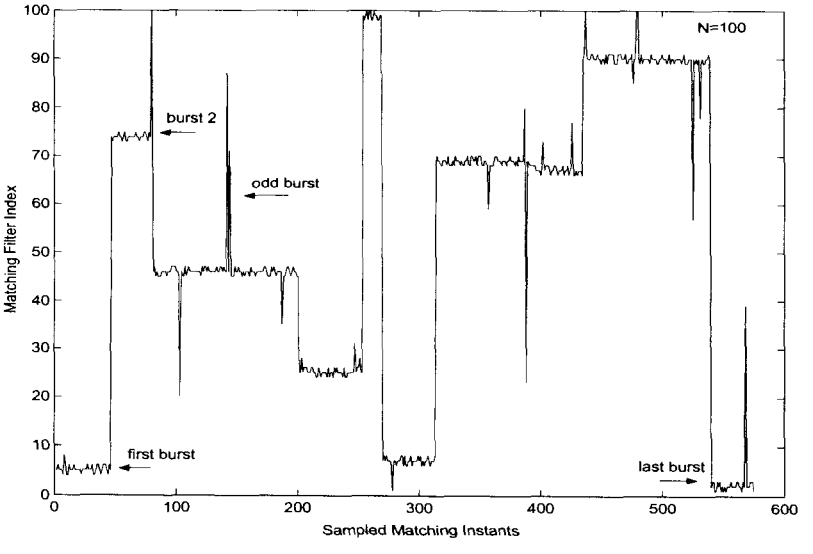


Figure 5.4: Example of simulation response.

Figure 5.4 shows an example of matching instants corresponding to the indices of the matching filters. The first burst is indicated with an arrow. The burst is consistently matching a set of filters *(SDF)* for a period of time until the flow switches to burst 2, which is biased towards matching filters with indices around 75. A burst that occurs only for a very short period and matches a filter that does not belong to the current burst's *SDF* is referred to as an odd burst or an anomaly.

5.7.1 Search Time

We conducted one experiment to compare the performance of *AIGPS* versus *AGPS* in terms of their search time. A classifier of size *N=100* was used.

Search time for N=100

For a classifier of size *N=100*, we generate 10 classifiers with the method presented in Section 5.1.

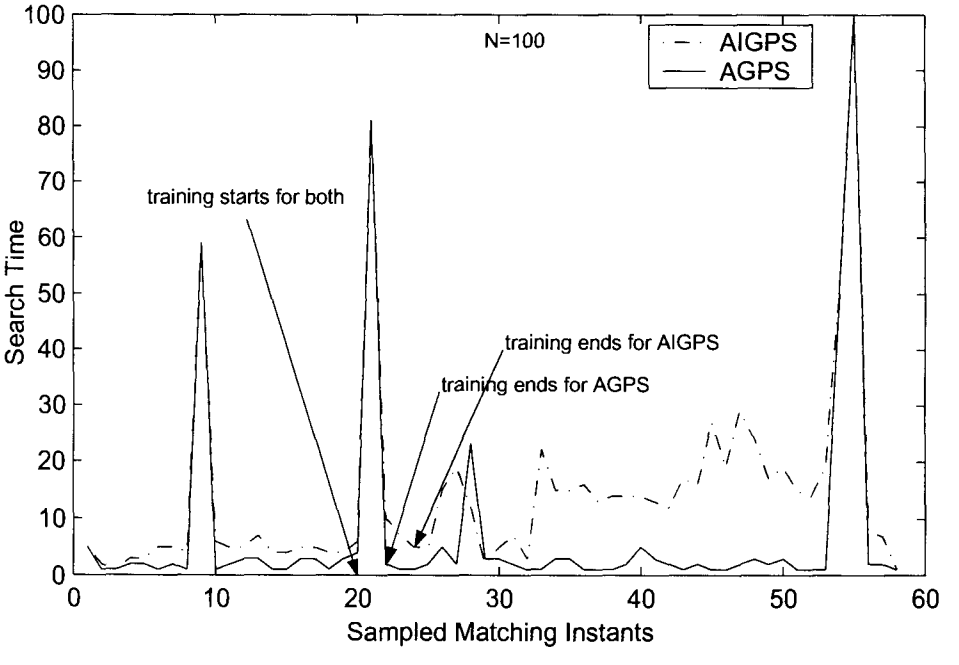


Figure 5.5: Search time comparison between AIGPS and AGPS, N=100

Figure 5.5 shows a comparison between the search time of *AGPS* and *AIGPS*. Both mechanisms adapt to varying bursts. We can observe however that *AGPS* has a shorter training period as it grants credit with no regard to the index of the matching filter, while *AIGPS* takes the index of the matching filter into account. Thus, the search time of *AGPS* is bounded by the search time of *AIGPS* as shown in the figure.

If the same experiment was applied to *LP*, the search time of *LP* at a given instant of time should coincide with Figure 5.4, since the search time of *LP* depends on the index of the matching filter at a specific instant.

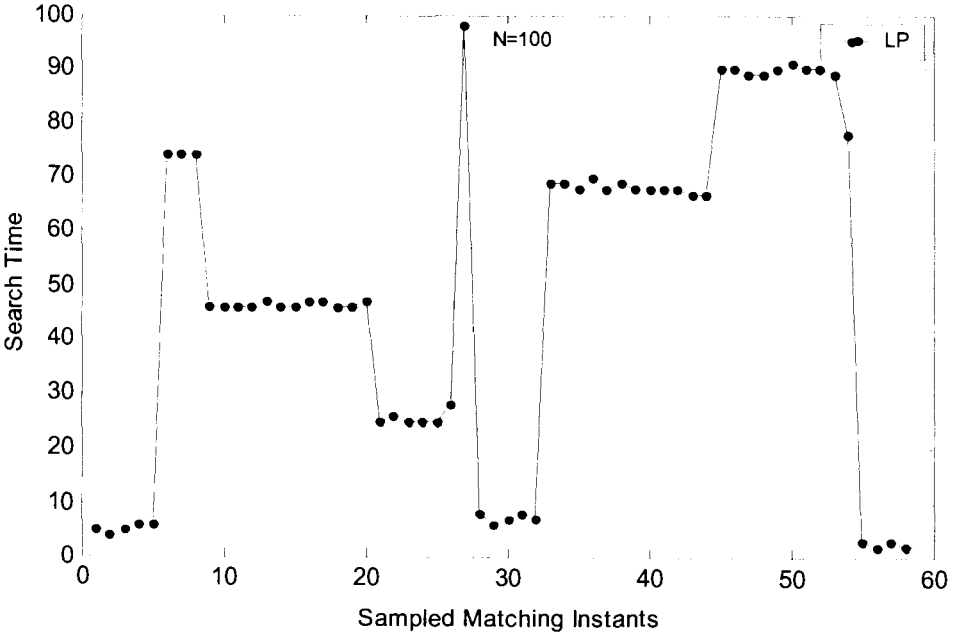


Figure 5.6: Search time of LP, N=100

In Figure 5.6, the search time of *LP* follows the indices of the matching filters, which can be observed as a replica of Figure 5.4.

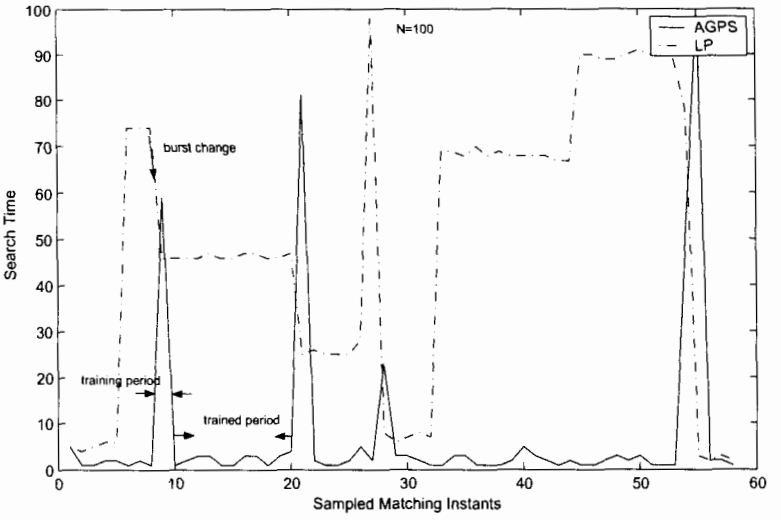


Figure 5.7: Search time comparison between AGPS and LP, N=100

Figure 5.7 shows a comparison between the search time of *AGPS* and *LP* for the same experiment. Search time of *LP* depends on the index of the matching filter, while the search time for *AGPS* is independent of the index of the matching filter. When a burst-change occurs, *AGPS* may have high search time consequently. The high search time gradually decreases during the adaptation or training period. After the training period is over, a probing sequence is derived, and the search time goes to its minimum.

Search time for N=1000

As an example for large-size classifiers, we present experimental results for a classifier of size *N=1000*. Simulation results of Figure 5.5 above, demonstrated that *AGPS* is more efficient than *AIGPS*. Hence, we only present our experimental results for *AGPS* versus *LP*. Figure 5.8 shows a sample comparison between *AGPS* and *LP* for a classifier of size *N=1000*.

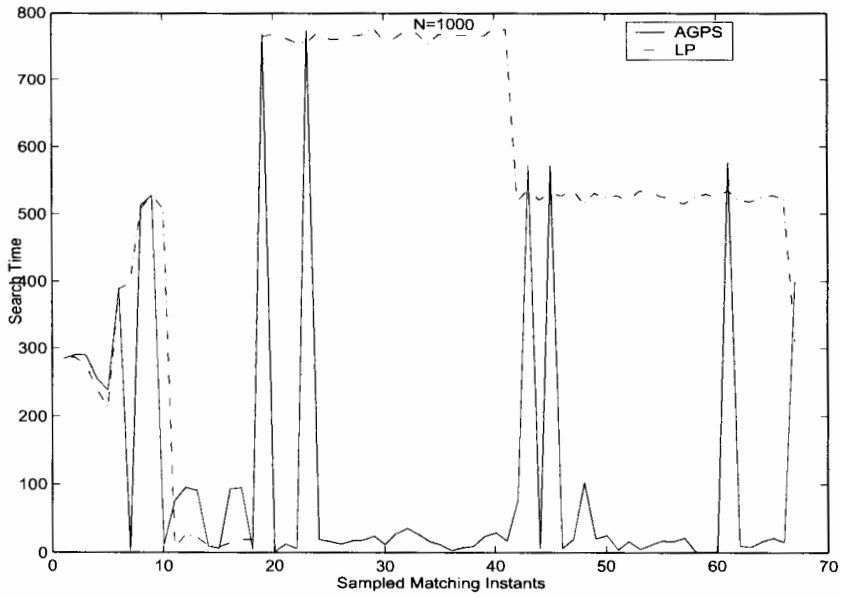
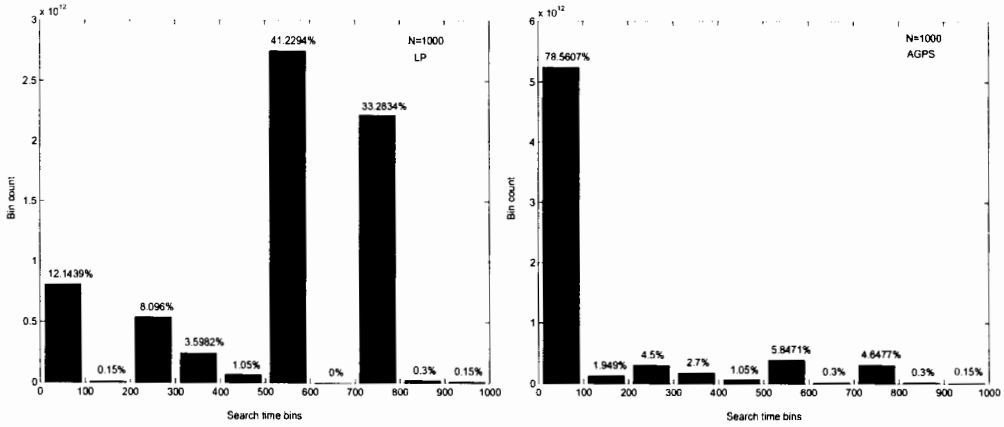


Figure 5.8: Search time comparison between AGPS and LP, N=1000

LP has search times that are proportional to the size of the classifier, *AGPS*, however has only limited training periods, which are proportional to the size of the classifier. After the training period(s), *AGPS* demonstrates search times that are independent of the size of the classifier and can achieve search times comparable to search times in a smaller classifier. In fact, *AGPS* can reach search times that can hardly be reached by *LP* for this size of classifiers.



(a) *(b)*

Figure 5.9: Histogram of classification search times, (a) LP, (b) AGPS. N=1000

Figure 5.9 *(a)* and *(b)* show a histogram of the classification search times produced by *LP* and *AGPS* respectively. We can observe that *AGPS* has 78.5% of its search times in the 1-100 bin, while *LP* has 41.2% of its search times in the 500-600 bin, and 33.28% in the 700-800 bin. The next section verifies this phenomenon further, by comparing the performance of *AGPS* and *LP* in a larger classifier.

Search time for N=10,000

We examine the performance of *AGPS* and *LP* in a large classifier of size *N=10,000*. Figure 5.10 and Figure 5.11 show a sample comparison between *AGPS* and *LP* for a classifier of size *N=10,000*.

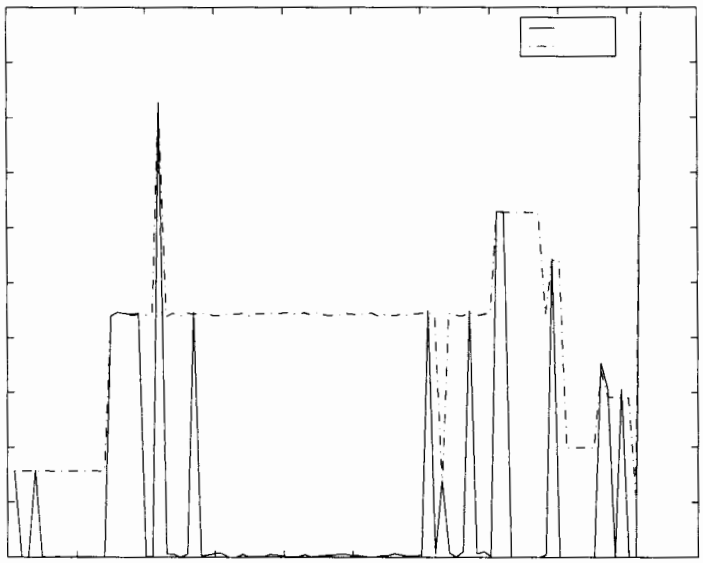
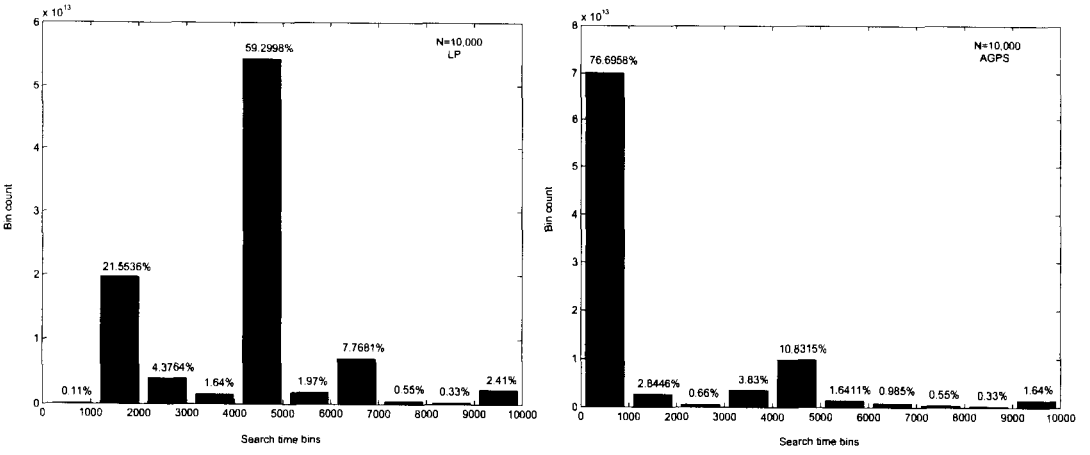


Figure 5.10: Search time comparison between AGPS and LP. N=10,000



(a) *(b)*

Figure 5.11: Histogram of classification search times, (a) LP, (b)AGPS. N=10,000

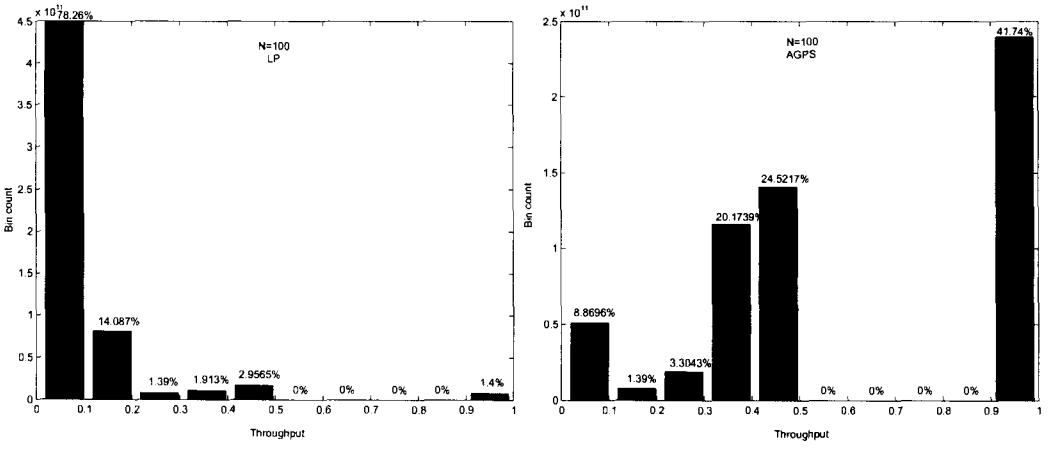
From Figure 5.11 *(a)* and *(b)*, we can observe that *AGPS* has 76.7% of its search times in the 1-1000 bin, while *LP* has spent the same period of time in 1000-5000 bins, that is 21.5% of its search times in the 1000-2000 bin and 59.3% in the 4000-5000 bin.

5.7.2 Throughput

Classifier throughput, which is the average number of packets processed per one memory probe as defined in Chapter 4, is another metric that can be used to analyze the performance of *AGPS* and to compare classification techniques. This sub-section presents an analysis for the throughput of the 3 classifiers mentioned above.

Throughput for N=100

We examine the throughput of the classifier of size *N=100* for both *AGPS* and *LP*.



(a) *(b)*

Figure 5.12: Histogram of throughput, (a) LP, (b) AGPS. N=100

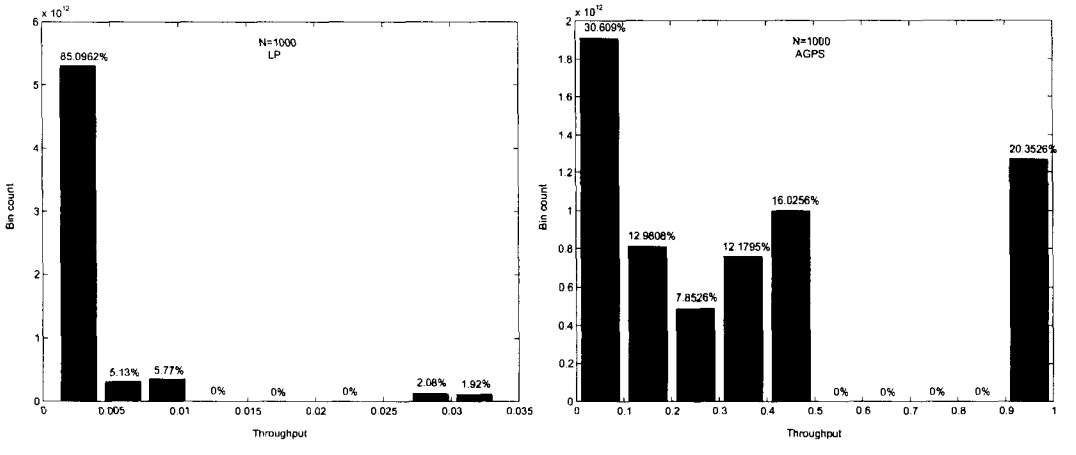
Figure 5.12 *(a)* and *(b)* show a histogram of the throughput for *LP* and *AGPS* respectively. We can see that for around 40% of the time, *AGPS* can classify 1 packet in one memory probe. On the other hand, *LP* can achieve the same maximum throughput only 1.4% of the time.

Approximately, 45% of the time, *AGPS* can classify 0.3 to 0.5 of a packet in one memory probe, while *LP* operates in the same throughput range for only 5% of the time.

AGPS produces its worst throughput in the range less than 0.1 for around 8.9% of the time. *LP* operates in the same range for most of the time (78%).

Throughput for N=1000

Figure 5.13 *(a)* and *(b)* show a histogram of the throughput for *LP* and *AGPS* respectively in the classifier of size *N=1000.*



(a) *(b)*

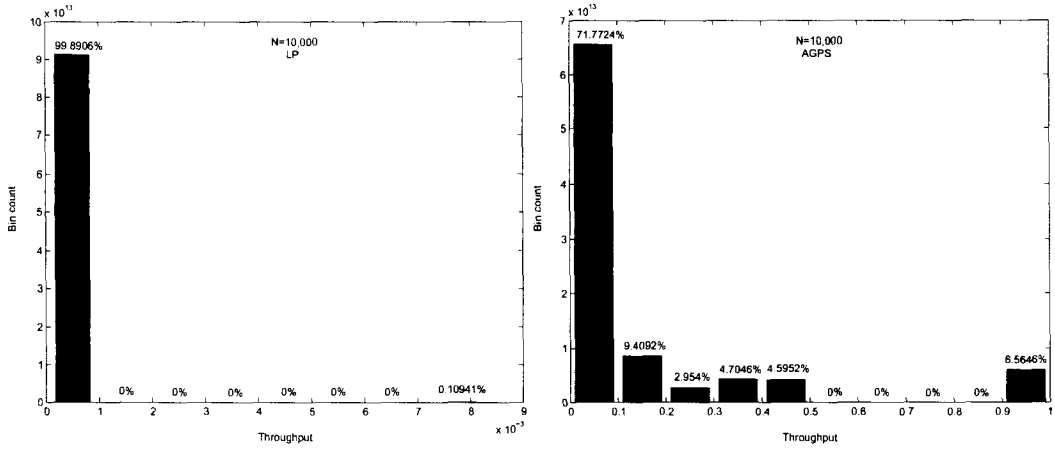
Figure 5.13: Histogram of throughput, (a) LP, (b) AGPS. N=1000

We can see that for around 20% of the time, *AGPS* can classify 1 packet in one memory probe. On the other hand, *LP* can never achieve this maximum throughput all the time. The maximum throughput achieved by *LP* was 0.03 packets per memory probe.

While *LP* operated below 0.005 for 85% of the time, *AGPS* classified 0.1 to 0.5 of a packet per memory probe for about 50% of the time. *AGPS* produced its worst throughput in the range less than 0.1 for around 30% of the time.

Throughput for N=10,000

Figure 5.14 *(a)* and *(b)* show a histogram of the throughput for *LP* and *AGPS* respectively in the classifier of size *N=10,000.*



(a) *(b)*

Figure 5.14: Histogram of throughput, (a) LP, (b) AGPS. N=10,000

Although the throughput of both *AGPS* and *LP* deteriorated in comparison to the throughput of smaller classifiers examined above, *AGPS* showed that it could still challenge the increased size (10 folds) of the classifier and reach a maximum throughput of 1 for 6.5% of the time, and 0.5-0.1 for about 21% of the time. *LP*, which had no access to the statistical techniques exploited by *AGPS*, operated poorly below 0.001 packets per memory probe for about 99% of the time. *AGPS* produced its worst throughput below 0.1 for about 72% of the time. Although *AGPS* operated in the worst range for most of the time, affected by the large size of the classifier, it is noticeable that *LP* never achieved this throughput.

5.8 Simulation Bench for Dynamic Classifiers

Our objective in this section is to test the performance of the statistical approach proposed by *AGPS* in a dynamic classifier, where the size of the classifier is initially zero. In this case, the values of the *iPMF* have to be dynamically derived and maintained online to account for insertion and deletion events, as explained in Chapter 4. The *iPMF* values are then used to efficiently probe the classifier, as was the case with a *SC*.

Dynamic filters are inserted on top of static filters implying that they have a higher priority, since the inserted filters are more specific. Thus, for a given inbound packet, the dynamic classifier is probed first for the best match [31][27]. If no match is found in the dynamic classifier, the static classifier is probed. We treat *DC* as an independent classifier from *SC* when computing *iPMF* values. We however maintain the order in which they should be probed.

We assume the following:

1- We assume the availability of an insertion/deletion mechanism[25][26].

2- Priorities of the filters are directly proportional to their indices.

3- The size of the *DC* at $n = 0$ is $Z(0) = 0$.

4- The maximum allowed size for the dynamic classifier is B

5- The size of the *SC* is N for all n.

6- We assume extreme conditions:

- The size of the *DC* is increasing most of the time, in other words we assume high insertion rate and the probability of inserting a filter at a given time step is $a \to 1$.
- We assume a low rate of deletion and hence, the probability of deleting a filter at a given time step is $d \to 0$.

7- When the *DC* reaches its maximum allowed size *B* and a filter needs to be inserted, the dynamic filter with the lowest *iPMF* value is deleted. For linear probing, when the maximum allowed size of the *DC* is reached, we delete the filter with the lowest index.

We start the experiment as described in Section 5.3. We made some modifications however to the simulation setup illustrated in Figure 5.2 to accommodate our experiments on dynamic classifiers. Figure 5.15 shows the modified setup.

Since *DC* initially contains no filters at time $n = 0$, we start comparing *PHL(n)* with the *SC*. We assume that the action associated with the matching filter is applied and as a consequence a dynamic filter with index *Z(n)* is inserted with probability *a,* where *Z(n)* also represents the size of the *DC* at time step n . Since dynamic filters are typically inserted to support specific traffic flows[14], the dynamic filter is inserted in the form of a label equal to *PHL(n)* that resulted in its insertion. Hence,

$$Labels(Z(n)) = PHL(n) \tag{5.7}$$

This type of insertion is referred to as *full-IP* [54]. An initial *iPMF* value is computed for the inserted filter *Z(n)* as described in Equation (4.15). The *iPMF(n)* values are also recomputed for all other *Z(n)-1* filters, if any.

Since we assumed a high probability of insertion and a low probability of deletion, the size of the *DC* is growing for most of the time. Hence, when the size of the *DC* reaches $Z(n) = B$, we choose the dynamic filter with the minimum *iPMF(n)* to be deleted, in an attempt to avoid the deletion of a frequently used filter, which may result in delaying the associated traffic flow, while searching the *SC* for a match to produce a new insertion in the *DC*.

Once the dynamic filter of minimum *iPMF(n)* is decided, its *iPMF(n)* value is distributed amongst all other *Z(n)-1* filters using Equation (4.18).

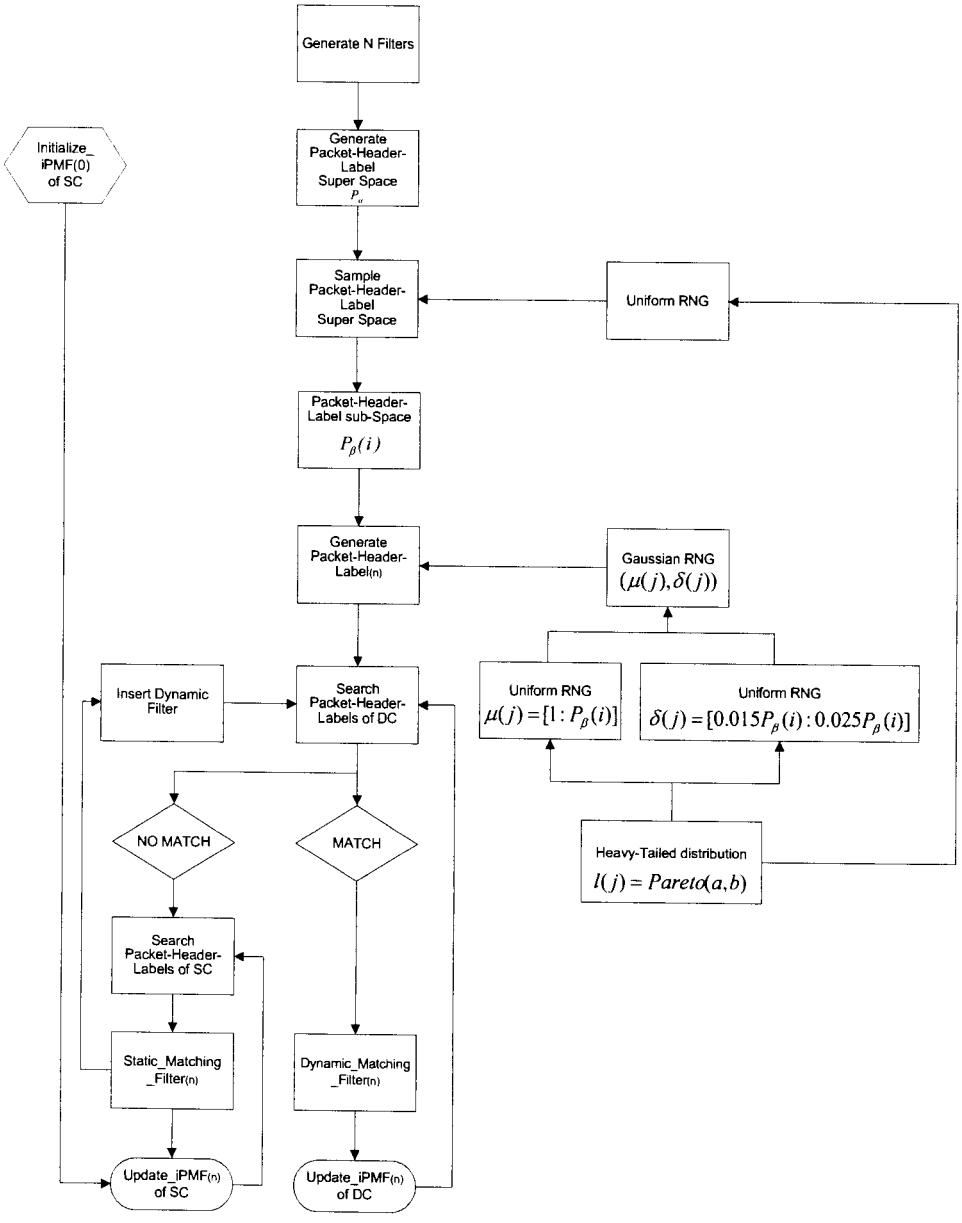


Figure 5.15: Simulation bench for dynamic classifiers

We tested the performance of *AGPS* in *DC* using the setup of Figure 5.15 where the size of the *SC* is *N=100*, and the maximum allowed size of the *DC* is *B=1000*.

5.8.1 Search Time

We first consider the combined[1] search time comparison analysis between *AGPS* versus *LP*. We start our analysis only in the Growth-Period (*GP*), which starts at the first time instant and until the size of *DC* reaches its maximum *B*. Figure 5.16 shows the combined search time comparison during *GP* for *N=100* and *B=1000*.

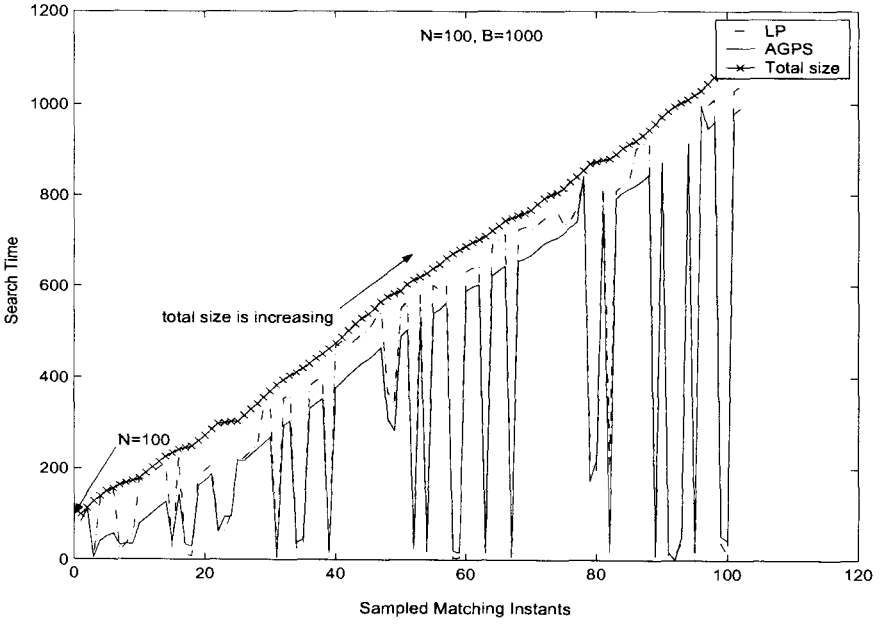


Figure 5.16: GP combined search time comparison between AGPS and LP in DC. N=100, B=1K

Since the *DC* has an initial size of zero, the total size of the classifiers (*SC* and *DC*) is initially 100. As new dynamic filters are inserted almost every time step, the total size of the classifiers starts increasing until it reaches its maximum limit of *B*. We can see that the search time for *LP* represented by the dotted line is comparable to that of *AGPS* represented by the solid line. For some instants, the access times of *LP* are almost the same as the total size of the classifiers. Only at these instants, one can say that *AGPS* has better performance than *LP*.

---

[1] Search time in SC and DC together.

Further, Figure 5.17 shows the previous search time comparison in *DC* only. As shown in the figure, *AGPS* is performing better than *LP* for some instants, and *LP* has better performance for other instants. Average time analysis for this period showed that *AGPS* needed 49.603 memory probes on the average to reach the matching filter, while *LP* needed 54.75 memory probes on the average to reach the same filter. Hence, *AGPS* outperformed *LP* by only 9.40% in *DC*.

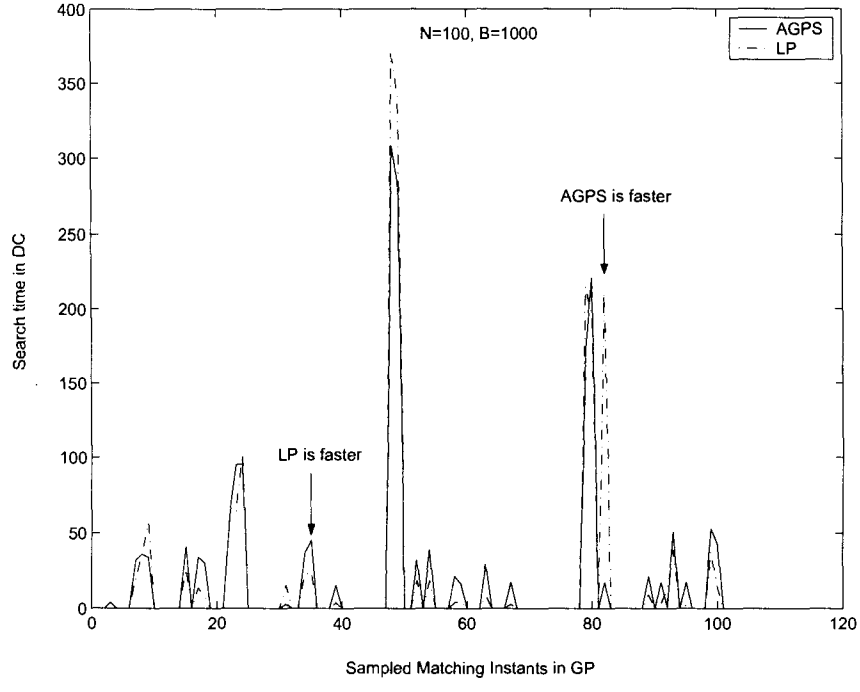
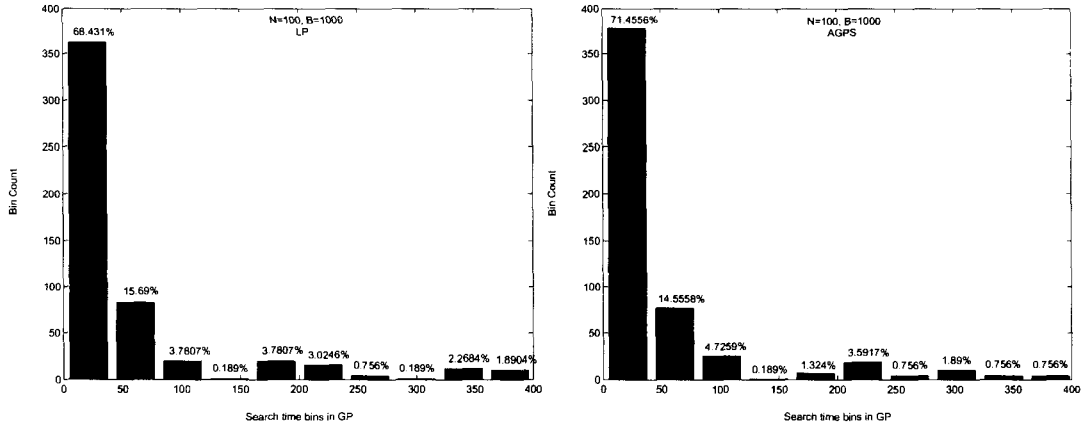


Figure 5.17: GP search time comparison between AGPS and LP in DC. N=100 B=1K

Figure 5.18 *(a)* and *(b)* show a histogram of classification search times in *DC* during *GP*. There is no significant difference in search time between *AGPS* and *LP* as can be seen on histograms. However, we can observe that for about 71% of the time, *AGPS* performs in the 1-50 search time bin, while *LP* performs for about 68 % of the time for the same bin. *AGPS* also performs in the last search time bin (350-400) for about 0.75614%, which is less than the time spent by *LP* (1.8904%) in the same bin by more than 50%.



(a) *(b)*

Figure 5.18: GP histogram of search times in DC, (a) LP, (b) AGPS. N=100, B=1K

To obtain more insight into what Figure 5.18 shows, we compute the search time difference between *AGPS* and *LP*, Figure 5.19 shows a plot of the time difference (search time of *LP* minus search time of *AGPS*).

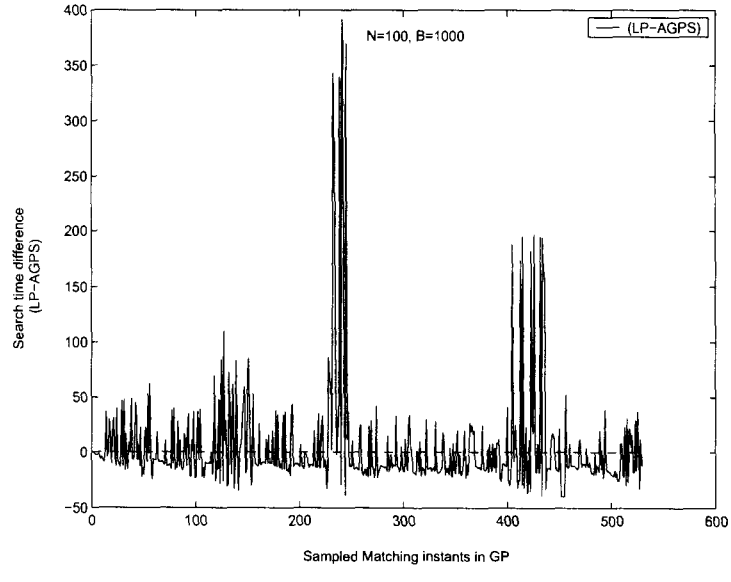


Figure 5.19: GP search time difference between AGPS and LP in DC. N=100, B=1K

We observe that except for 2 periods of time (100-250 and 400-450) where *AGPS* significantly outperforms *LP*, the plot is semi-symmetrical around the horizontal axis of

origin (zero axis) represented by the dotted line. This implies that *AGPS* and *LP* are behaving almost the same except for the instants specified above.

We now conduct the same analysis for the After-Growth-Period *(AGP)*, which is the period that starts with the first action to delete a dynamic filter when the maximum allowed size of the *DC* is reached.

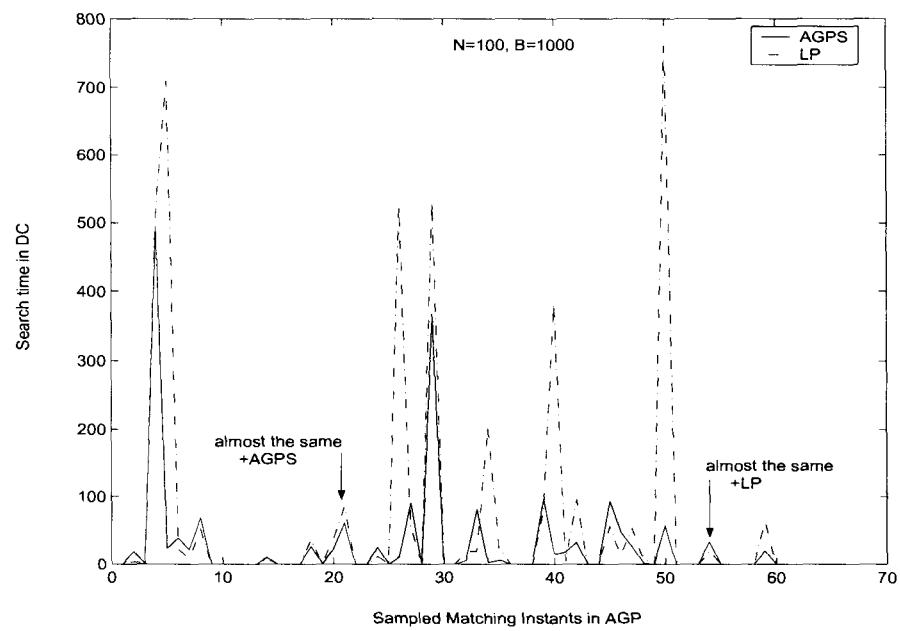
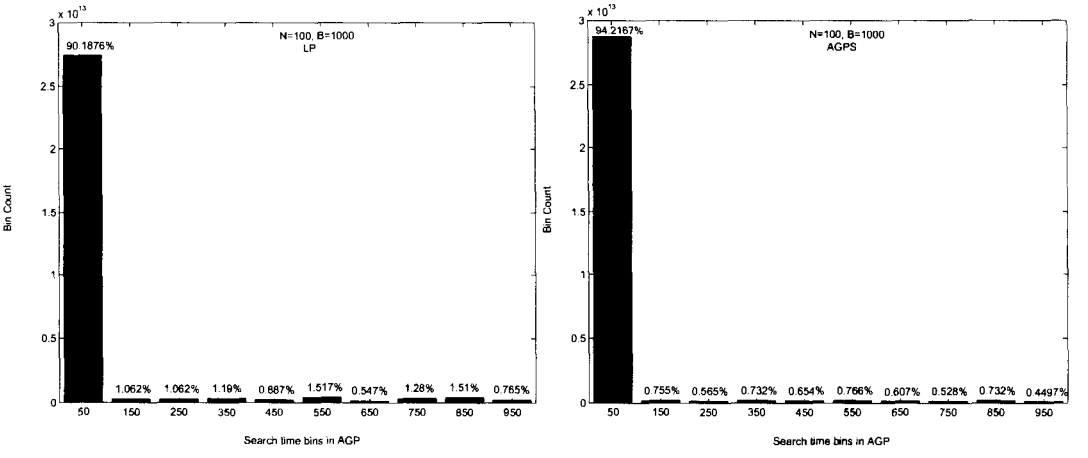


Figure 5.20: AGP search time comparison between AGPS and LP in DC. N=100, B=1K

As shown in Figure 5.20, *AGPS* performs better than *LP* for most of the time, except for few instants (indicated by the arrows) where the performance of both techniques is marginally the same. Average time analysis shows that in *AGP, AGPS* needed 44.471 memory probes on the average to reach the matching filter, while *LP* needed 66.763 memory probes on the average to reach the same filter. Hence, *AGPS* outperformed *LP* by 33.389 % in *AGP* as opposed to 9.40 % only in *GP*. This indicates

Figure 5.21 *(a)* and *(b)* show a histogram of classification search times in *DC* during *AGP*. It is noticeable that for about 94.2167% of the time, *AGPS* performs in the 1-50

search time bin while *LP* performs in the same time bin for about 90.1876% of the time. In addition, *AGPS* spends less time performing in all other corresponding time bins than *LP*.



(a) *(b)*

Figure 5.21: AGP histogram of search times in DC, (a) LP. (b) AGPS. N=100, B=1K

Figure 5.22 shows the search time difference between *AGPS* and *LP*. A positive value for a given instant indicates that *AGPS* is faster than *LP*, and a negative value indicates that *LP* is faster than *AGPS*.

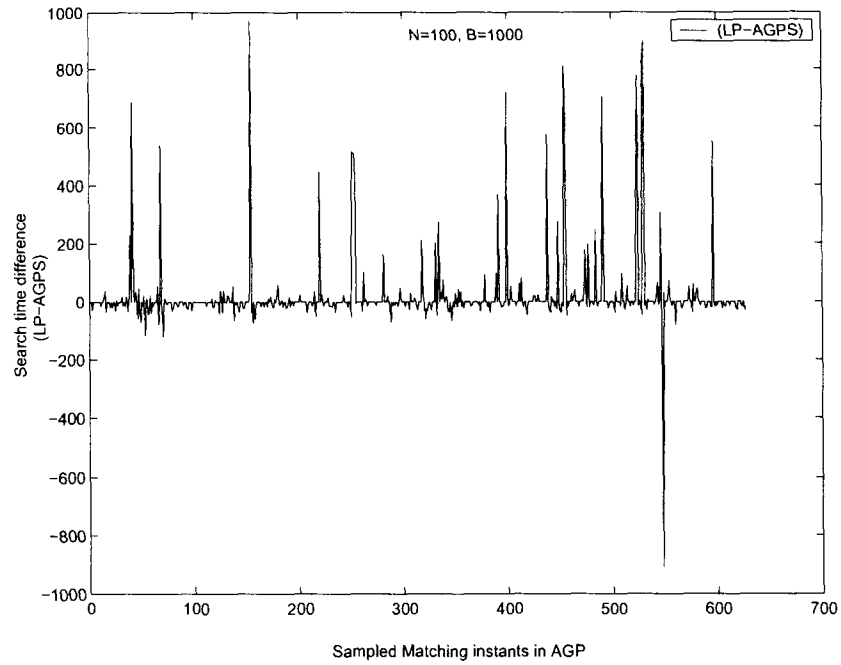


Figure 5.22: AGP search time difference between AGPS and LP in DC. N=100, B=1K

Referring to the search time difference analysis in the *GP*, we can conclude that *AGPS* outperformed *LP* in *AGP*, while both techniques performed marginally the same in *GP*.

5.8.2 Post Deletion Reference Period Analysis in AGP

Another metric that we considered testing in *AGP* is the Post Deletion Reference Period *(PDRP)*. We define this metric as the period between the instant of deleting a dynamic filter to create space, and the instant when a received packet matches the deleted dynamic filter. Thus, *PDRP* is the period between deleting a dynamic filter and referencing it. If a given packet label does not result in a match with any of the filters in the *DC*, it is then compared to a vector carrying information about deleted filters. If a match with any of the deleted filters occurs, we count this period as a miss and the packet is forwarded to the *SC* for a match with a static filter. Testing this metric evaluates the deletion policy

adopted by *AGPS* or *LP* in case *DC* reaches it maximum limit and a new filter needs to be deleted.

Ideally, we would like a large *PDRP*. A large value for this metric corresponds to a good deletion policy and a smaller value corresponds to a poor deletion policy.

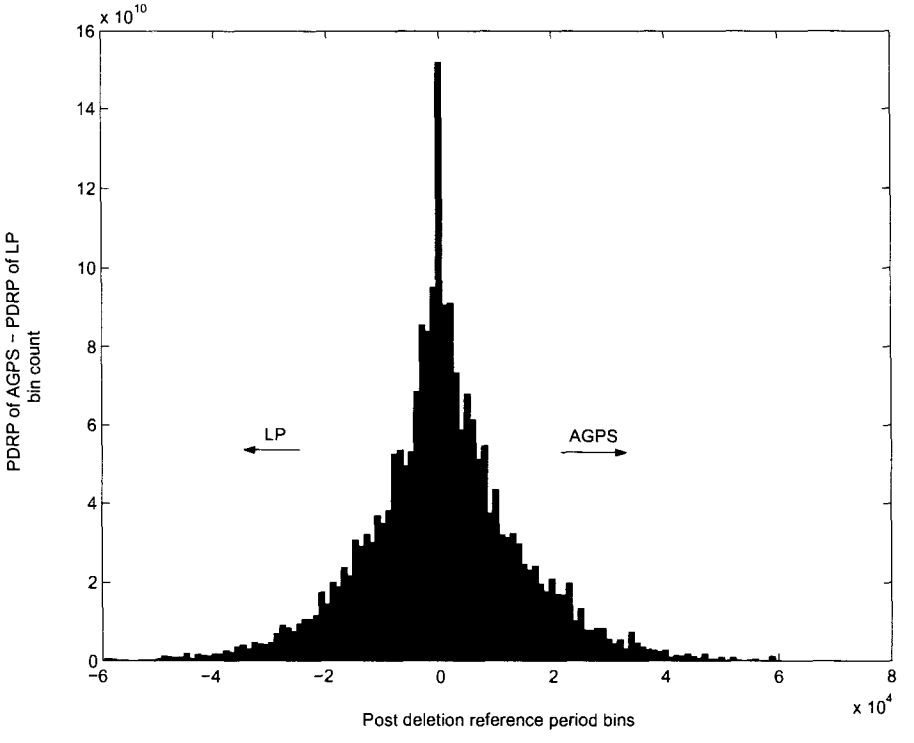
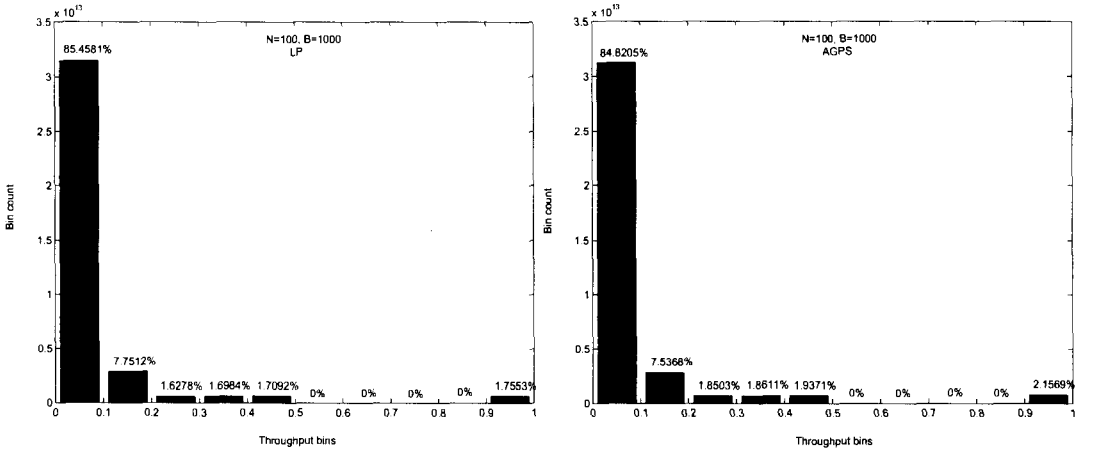


Figure 5.23: AGP histogram comparing PDRP of AGPS and LP in DC. N=100, B=1K.

Figure 5.23 shows a histogram comparing the difference between *PDRP* of *AGPS* and *PDRP* of *LP*. We can see that for most of the time, the difference in *PDRP* of both techniques is the same *(PDRP of AGPS – PDRP of LP=0)*. The histogram is semi-symmetric around the origin, which shows that neither *AGPS* nor *LP* had an advantage over the other as far as this metric is concerned. Both techniques showed a miss ratio of about 30%. This is due to the assumptions made about the insertion and deletion policies of *LP*, which is practically the same as those of *AGPS*.

5.8.3 Throughput



(a) *(b)*

Figure 5.24: Histogram of throughput in DC, (a) LP. (b) AGPS. N=100, B=1K.

Figure 5.24 *(a)* and *(b)* show throughput of *LP* and *AGPS* respectively. We can see that *AGPS* shows its best performance (in the 0.9-1 bin) for 2.1569% of the time while *LP* performs in the same bin for 1.7553% of the time. We can also see that *AGPS* shows its worst performance (0-0.1 bin) for 84.8205% and *LP* operates in the same bin for 85.4581% of the time. The average throughput of *AGPS* is 0.040545-packets/memory probe, while the average throughput of *LP* is 0.036226-packets/memory probe. This is an enhancement of about 11.922%.

A tabulated comparison between *AGPS* and *LP* is presented in Appendix *B*.

6 Conclusion and Future work

6.1 Conclusion

To summarize, *AGPS* was tested using simulations in static and dynamic classifiers. *LP* was used as a reference for comparison. Both techniques were tested in a neutral environment. Two main metrics, search time and throughput, were measured.

Results showed that, in small to large sized static classifiers, the search time of *AGPS* was 50% to 80% better than *LP*, with a low memory requirement of 0.015625-Kbytes/filter. Search time results also showed that *LP* spent more than 50% of the time searching more than 50% of the classifier, *AGPS* on the other hand, spent more than 50% of the time searching around 10% of the classifier. Results of testing the throughput of *AGPS* compared to *LP* showed that *AGPS* has a quite significant advantage over *LP*, especially in large classifiers. This is because the throughput of *LP* is highly affected by the size of the classifier, while the adaptive feature of *AGPS* assists in reaching throughput values that can never be reached by *LP* for the same classifier.

In dynamic classifiers, our results were different. We were interested in stressing the ability of *AGPS* to derive, maintain, and update the statistical values online, as filters were inserted or deleted. *AGPS* was successful in utilizing the adaptive tool. However, the advantage of *AGPS* over *LP* in dynamic classifiers, though existed, was not as significant as its advantages over *LP* in static classifiers. This is due to the assumption that in *SC*, *AGPS* works on range matching as a form of prefix matching, while in *DC*, we assumed that the entire IP address represents one filter. This difference in assumptions makes the performance of *AGPS* in *SC* more tolerant to locality changes. Since filter

insertion or update occurs less frequent than filter look-ups ($1:10^7$) [12] this difference is acceptable. This argument is in agreement with a comparison study between *IP-prefix* and *full-IP-address* caching schemes [54]. Our search time analysis showed that in the Growth-Period, *AGPS* outperformed *LP* by only 9.40%. In the After-Growth-Period, however, *AGPS* outperformed *LP* by 33.40%.

We used *PDRP* in Section 5.8.2 to test the performance of the assumed deletion policy of *AGPS*. In comparison with *LP*, *AGPS* showed the same performance as far as this metric is concerned. This implied that assumptions made about the deletion policies of both techniques are logically the same.

Throughput analysis showed that in dynamic classifiers, *AGPS* has an advantage of about 12% over *LP*, which is again not as significant as throughput measures showed by *AGPS* in static classifiers. We note however, that this difference in the advantage of *AGPS* over *LP* in *SC* verses *DC*, is not due to a difference in the performance of *AGPS*, but rather, due to significant difference in the behavior of *LP* in *DC*, where the order at which filters are inserted are considered when probing the classifier.

6.2 Future Work

There are many environments where *AGPS* can be tested: for example, *AGPS* could be tested and compared to traffic-filtration techniques currently used for access control (firewalls), *VPNs*, and other forms of network security applications. It could also be tested as a replacement for caching schemes currently used for IP-Lookup. In addition, examining the effectiveness of *AGPS* to support functions such as *QoS*, resource reservation, and other Value Added Services *(VAS)* is of future research interest. We note that the underlying concept of *AGPS* can be generalized and used in many possible applications that require online approximation for a pre-specified statistic, feature, or

environment variable. An example of these applications can be intrusion detection, network traffic measurements, and IP lookup caching schemes in network processors.

Adaptive Update Formulas

In Chapter 4, we placed the fundamental criteria in choosing the update formula in *AGPS*. There are other considerations, however, which would be interesting to address. For example, the step size of the credit value evaluated by Equation (4.12), and the step size of factor β evaluated by Equation (4.6) can be adaptively changed based on the strength of traffic locality in a particular burst or number of packets. Ideally, the instantaneous credit value of every filter should be exactly equivalent to the percentage of packets that matched the filter in this burst. One way of accomplishing this desired adaptability is by changing the value of the step size of the update formula based on the three factors mentioned above.

Default-Filter Temporary Elimination

Since classifiers must have a finite size, they cannot account for all the possible combinations of packet headers. Thus, default filters are placed to reflect the default classification policy. Unfortunately, the default policy is only used after sequential examination against every filter in the classifier. Consequently, valuable processing time is wasted each time a given packet header is destined to the default policy. From this perspective, the role of the default filter, albeit intrinsic, is regarded as a time-inefficient element in the system.

If the role of the default filter is fulfilled without the necessity of examining all preceding filters, time is saved. One way of satisfying this goal is by *Default-Filter Temporary Elimination (DFTE)*. If inbound traffic is biased towards the default filter, then *iPMF* values of the default filter can be used to indicate this bias, then a dynamic filter with the

default policy and fields exactly matching the biased traffic is temporarily inserted in the highest priority position. When the traffic bias stops, the inserted filter is deleted. Thus, using the indicators provided within *AGPS* can help save time that was usually wasted because of unaccounted-for-traffic.

Denial-of-Service Attacks

One class of attacks is the *denial-of-service* attack [51], whose objective is to bring a system to an extremely busy state whereby its main resources (memory, time, allocated bandwidth, etc.) are consumed, and even friendly service requests cannot be acknowledged. Kinds of *DoS* attacks include but are not limited to *TCP SYN* flooding, *ping* flooding, *UDP* flooding, *ICMP redirect bombs,* and *FTP memory attack.* Different kinds of this class of attacks are being recognized. A common feature between them is persistence and repetition [31].

It is difficult for a network security administrator to provide a filter for every possible attack. Thus, the job is left to the default filter, which as we described above, becomes an extremely time-inefficient element if exposed to the persistence and the repetition that comes with *DoS* attacks.

To alleviate some of the consequences of a *DoS* attack, such as the wasted processing time, *DFTE* can be used, which reduces the time processing time consumed by the attack to *1/N*, where *N* is the number of filters in a packet classifier.

Intrusion Detection

Providing network and system security is a hard task since the development of various types of attacks is an ongoing process. Therefore, enhancing the ability of the system to be adaptively sensitive to attacks and intrusions is a desired objective. This compensates

for the inability to account for every possible attack. For this purpose, Intrusion Detection Systems *(IDS)* are constantly being developed. *IDS* include auditing tools, vulnerability scanners, integrity checkers, and attack detectors [31].

These systems monitor the local as well as the external environment of the system. Their main task is to identify, log, and notify unexpected or abnormal changes to some indicators such as the system's log files (odd, deleted entries, or size), configuration (open ports), and performance (speed, network traffic, disk space usage, load averages). In order to identify what is abnormal, we need first to be consistent in identifying what is normal. For example, we can identify what is normal by logging and checking system-and-neighborhood information relevant to frequent security checks.

The more information we obtain, the stronger our ability to detect intrusion. However, keeping track and analyzing all the desired information can be both memory and time-consuming. Thus, a scaleable system is needed that monitors as many system elements as possible without exhausting system resources. We can develop a technique that can extract statistical data about the environment elements by utilizing the method used by *AGPS* for packet classification. For example, we can define a random process that describes the time varying size of a routing table. The *iPMF* values can be retrieved at specific sampling instants to provide an estimation of a certain statistic, such as the expected value of the element, its variance, or its autocorrelation. Changes in these statistics can be a reference to identify a hostile activity. This way, we can refer to single system statistic to provide a valuable view of the elements monitored for security, and without over logging or exhausting system resources.

Enhancing IP Lookup Caching Schemes in Network Processors

A study made by Partridge [55] on the hit rate of a backbone-router cache using a 5-Mpackets trace of FIX WEST, showed that a cache of one entry could achieve a hit rate of 1.8%. In other words, packets are classified in one memory access for about 1.8% of the time, while in a classifier of 100, 1000, 10,000 entries, *AGPS* can produce a match in one memory access for 42%, 20%, 7% of the time respectively. In addition, results of Chapter 5 showed that in classifiers of 100, 1000, and 10,000 entries, *AGPS* searched only 10% of the classifier for about 80% of time to reach a match.

Network processors *(NP)* usually use cache memory to speed up IP lookup. A cache memory typically copies a portion of the entries in the IP table, which is stored in main memory. Full Associative or Set Associative caches can be used. If a cache miss occurs, an entry in the cache is replaced by a matching entry in the routing table [54]. Therefore, a miss in the cache memory implies a search in the cache memory plus a search in the original IP table. Thus, cache misses can degrade the performance since the lookup process is off chip [56]. To compensate for the delays resulting from cache misses, *AGPS* can be used for faster lookup from the routing table. Moreover, if the Set Associative caches are used where entries of the cache are divided into sets, adaptive probing of *AGPS* can be used for faster search through the sets.

In summary, *AGPS* can be used in many ways for different environments. Generally, it can be a simple tool, which approximates a model for any random process believed to exist. Approximating first and second order density functions can be easily achieved by sampling the values of the *iPMF* at a variable or fixed rate. As a result, valuable statistical measurements can be derived and referenced to boost efficiency. In light of this wide spectrum of possible applications, we have established a corporation with the explicit mission to examine and develop *AGPS* in different systems and applications.

Appendices

Appendix A AGPS Working Formulas

This appendix lists all *AGPS* formulas required.

A.1 Condition for Operation

The following condition must be satisfied

$$\sum_{i=1}^{N} p_i(n) = 1, \text{ for any } n \qquad \text{(A.1)}$$

Where N is the number of filters and $p_i(n)$ denotes the *iPMF* value of filter with index i at time instant n.

A.2 Static Filters

A.2.1 Static Filters Credit Initialization

As mentioned in Section 4.2, any initialization method can be used if Equation (A.1) is satisfied. *Uniform*, *random*, or *Inverted Hyper-Geometric* initialization methods are suggested. The first method is, however, favorable.

A.2.2 Matching Static Filter Credit Update

The following formula is proposed to update *iPMF* value of matching filter m at time instant n.

$$p_m(n) = \frac{p_m(n-1) + e^{-(1-p_m(n-1))^2}}{1 + e^{-(1-p_m(n-1))^2}} \tag{A.2}$$

Other formulas can be developed conditioned that $p_m(n) \leq 1$, *for any m and n.*

A.2.3 Non-Matching Static Filters Credit Update

We update the *iPMF* values of non-matching filters with indices $i \neq m$ at time instant n using

$$p_i(n) = \beta\, p_i(n-1), \text{ for all } i \neq m \tag{A.3}$$

Where m is the index of the matching filter and

$$\beta = \frac{1 - p_m(n)}{\sum_{i \neq m}^{\wp} p_i(n-1)}$$

$$= \frac{1 - p_m(n)}{1 - p_m(n-1)} \tag{A.4}$$

If $p_m(n) = p_m(n-1) = 1$, β is assigned a value of 1. Equations (A.2) and (A.3) satisfy (A.1). A proof is provided at the end of this appendix.

A.3 Dynamic Filters

A.3.1 Inserted Dynamic Filter Credit Update

We use the following formula to assign an initial *iPMF* value $p_{z(n)}$ to an inserted filter with index *Z(n)*.

$$p_{Z(n)} = \frac{Z(n)}{\sum_{i=1}^{Z(n)} i} \leq 1, \text{ for any } n \qquad \text{(A.5)}$$

Where $Z(n)$ is the size of the *DC* at insertion time step n. Equation (A.3) is used for all other filters and Equation (A.1) is satisfied.

A.3.2 Deleted Dynamic Filter(s) Credit Update

To delete a set of filters S_D of size D filters at time step n, where the number of dynamic filters is *Z(n)*, the following formula is used

$$p_i(n) = \frac{\sum_{d \in S_D} p_d(n)}{Z(n) - D} + p_i(n), \text{ for all } i \notin S_D \qquad \text{(A.6)}$$

Where $p_d(n)$ is the *iPMF* of filter with index d at instant n, which satisfies Equation (A.1).

A.3.3 Matching Dynamic Filter Credit Update

We use Equation (A.2) to update *iPMF* for any matching filter m at time step n, where $1 \leq m \leq Z(n)$. We use Equation (A.3) to update *iPMF* of all other non-matching dynamic filters.

Proof

From Equation (A.2)

$$p_m(n) \geq p_m(n-1) \tag{A.7}$$

and

$$\sum_{i=1}^{N} p_i(n-1) = p_m(n-1) + \sum_{i \neq m}^{N} p_i(n-1) = 1 \tag{A.8}$$

We would like to prove that for a matching filter m at instant n Equations (A.2) and (A.3) satisfy

$$\sum_{i=1}^{N} p_i(n) = p_m(n) + \sum_{i \neq m}^{N} p_i(n) = 1, \text{ for any } n \tag{A.9}$$

From Equation (A.7)

$$1 - p_m(n) \leq 1 - p_m(n-1) \tag{A.10}$$

then

$$1 - p_m(n) = \beta[1 - p_m(n-1)] \tag{A.11}$$

and

$$p_m(n) = 1 - \beta[1 - p_m(n-1)] \tag{A.12}$$

From Equation (A.3), since

$$p_i(n) = \beta\ p_i(n-1), \text{ for all } i \neq m \tag{A.13}$$

then

$$\sum_{i \neq m} p_i(n) = \beta \sum_{i \neq m} p_i(n-1) \tag{A.14}$$

Substituting Equation (A.12) and (A.14) in (A.9), we get

$$1-\beta[1-p_m(n-1)]+\beta\sum_{i\neq m}p_i(n-1) \qquad \text{(A.15)}$$

but,

$$\sum_{i\neq m}^{N}p_i(n-1)=1-p_m(n-1) \qquad \text{(A.16)}$$

and

$$\beta=\frac{1-p_m(n)}{1-p_m(n-1)} \qquad \text{(A.17)}$$

from Equations (A.8) and (A.11) respectively. Hence, substituting Equations (A.16) and (A.17) in (A.15) we get

$$1-[1-p_m(n)]+1-p_m(n)=1 \qquad \text{(A.18)}$$

Done.

Appendix B AGPS Performance Tables

Figure B.1 shows a 2 dimensional plot of the performances of both *AGPS* and *LP* according to the results of Chapter 5. The plot also shows the performances of both *AGPS* and *LP* according to the theoretical arguments established in Chapter 4.

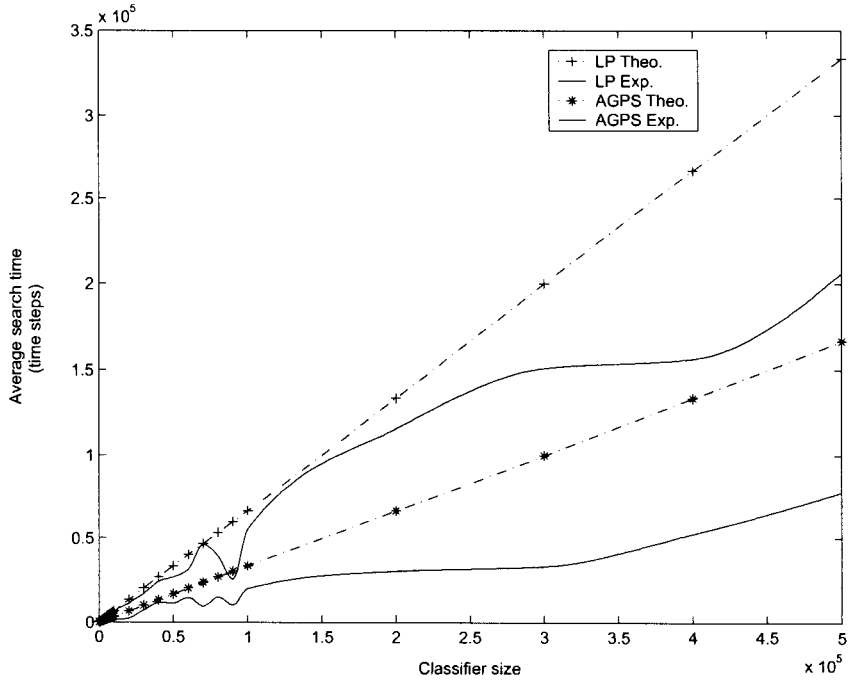


Figure B.1: Experimental and theoretical performances of AGPS and LP

Tables B.1 to B.4 below, show a detailed comparison between *AGPS* and *LP* regarding the average search time and average throughput. Classifier sizes vary from 100 to 500,000.

Table B.1: AGPS versus LP. N=100 to N=900

Number of filters	Avg. search time (Time step)			Avg. Throughput (Packets/time step)			Memory required (Kbytes)
	LP	AGPS	Enhancement (%)	LP	AGPS	Enhancement (%)	
100	58.404	8.3109	85.77	0.031974	0.47636	1389.8	1.5625
200	72.734	13.961	80.805	0.01931	0.4928	2452.1	3.1250
300	131.41	24.724	81.185	0.0090318	0.4506	4889	4.6875
400	187.28	55.591	70.317	0.017439	0.35699	1947.1	6.2500
500	264.14	48.375	81.686	0.009218	0.39891	4227.5	7.8125
600	320.25	75.148	76.534	0.004134	0.41526	9944.8	9.3750
700	444.08	105.73	76.191	0.0025401	0.35534	13889	10.9375
800	429.25	118.34	72.43	0.0045194	0.282	6139.7	12.5000
900	564.4	158.33	71.947	0.0020497	0.27466	13300	14.0625

Table B.2: AGPS versus LP. N=1000 to N=9000

Number of filters	Avg. search time (Time steps)			Avg. Throughput (Packets/time step)			Memory required (Kbytes)
	LP	AGPS	Enhancement (%)	LP	AGPS	Enhancement (%) x104	
1000	559.25	126.44	77.391	0.0031	0.2921	0.9314	15.6250
2000	867.03	249.22	71.256	0.0084	0.3186	0.3692	31.2500
3000	1207	432.99	64.128	0.0040	0.3030	0.7510	46.8750
4000	1712.9	290.64	83.032	0.0008	0.3040	3.8059	62.5000
5000	2381.1	896.39	62.354	0.0007	0.2953	4.3702	78.1250
6000	3141.8	1040	66.899	0.0011	0.3038	2.8519	93.7500
7000	2924.3	722.89	75.28	0.0004	0.2788	6.2850	109.3750
8000	5501.7	1213.7	77.939	0.0004	0.2550	6.1162	125.0000
9000	4426.4	1712.4	61.313	0.0004	0.1881	5.1998	140.6250

Table B.3: AGPS versus LP. N=10,000 to N=90,0000

Number of filters	Avg. search time (Time steps)			Avg. Throughput (Packets/time step)			Memory required (Kbytes) $x10^3$
	LP	AGPS	Enhancement (%)	LP $x10^{-3}$	AGPS ---	Enhancement (%) $x10^5$	
10000	6398.1	1673.8	73.839	0.1964	0.2225	1.1322	0.1563
20000	11183	2402.1	78.52	0.1101	0.1271	1.1527	0.3125
30000	16848	7220.9	57.142	0.0933	0.3002	3.2186	0.4688
40000	23879	11538	51.681	0.0705	0.2330	3.3060	0.6250
50000	26596	11345	57.341	0.0457	0.3619	7.9186	0.7813
60000	30917	14628	52.688	0.0877	0.3419	3.8967	0.9375
70000	46173	9683.7	79.027	0.0307	0.2629	8.5593	1.0938
80000	38874	14856	61.786	0.2322	0.3429	1.4756	1.2500
90000	24984	10335	58.635	0.1425	0.1332	0.9342	1.4063

Table B.4: AGPS versus LP. N=100,000 to N=500,000

Number of filters	Avg. search time (Time steps x 103)			Avg. Throughput (Packets/time step)			Memory required (Kbytes) x103
	LP	AGPS	Enhancement (%)	LP x10-4	AGPS ---	Enhancement (%) x106	
100000	55.35	19.814	64.202	0.2858	0.1997	0.6986	1.5625
200000	155	30.071	73.966	0.1081	0.1240	1.1469	3.1250
300000	150.99	33	78.145	0.9248	0.3237	0.3499	4.6875
400000	156.36	52.705	66.293	0.1189	0.2155	1.8126	6.2500
500000	206.36	78.125	62.141	0.0425	0.4010	9.4338	7.8125

References

[1] F. Gebali. "Computer Communication Networks Analysis and Design". Second Edition, 2001.

[2] I. Chvets and M MacGregor. "Multi-Zone Caches for Accelerating IP Routing Table Lookups". Proc. High Performance Switching and Routing, 2002.

[3] W. Shi, M. MacGregor, and P. Gburzynski. "Synthetic Trace Generation for the Internet". Proc. Workshop on Workload Characterization, 2001.

[4] W. Shi and M. MacGregor. "Cache Reference Behavior of Three IP Routing Table Lookup Algorithms". Proc. SCI'2001.

[5] W. Shi, M. MacGregor, and P. Gburzynski. "On Temporal Locality in IP Address Sequences". IEICE Transactions, 2003.

[6] W. Shi, M. MacGregor, and P. Gburzynski. "Traffic Locality Characteristics in a Parallel Forwarding System". International Journal of Communication Systems, 2003.

[7] J. Xu, M. Singhal, and J. Degroat. "Novel Cache Architecture to Support Layer-Four Packet Classification at Memory Access Speeds". Proc. INFOCOM, 2000.

[8] K. Li, F. Chang, D. Berger, and W. Feng. "Architectures for Packet Classification Caching". Proc. IEEE International Conference on Networks, 2003.

[9] F. Chang, W. Feng, and K. Li. "Approximate Caches for Packet Classification". ACM SIGCOMM (poster session), 2003.

[10] P. Warkhede, S. Suri, and G. Varghese. "Fast Packet Classification for Two-Dimensional Conflict-Free Filters". INFOCOM, 2001.

[11] L. Qiu, G. Varghese, and S. Suri. "Fast Firewall Implementations for Software-Based and Hardware-Based Routers". ACM SIGMETRICS Performance Evaluation Review, June 2001.

[12] T. Lakshman, D. Stiliadis. "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching". SIGCOMM, 1998.

[13] F. Baboescu and G. Varghese. "Scalable Packet Classification". Proc. SIGCOMM, 2001.

[14] V. Srinivasan, G. Varghese, M. Waldvogel, and S. Suri. "Fast and Scalable Layer Four Switching". Proc. ACM SIGCOMM, 1998.

[15] V. Srinivasan, S. Suri, and G. Varghese. "Packet Classification Using Tuple Space Search. Proc". ACM SIGCOMM, 1999.

[16] P. Gupta and N. McKeown. "Packet Classification on Multiple Fields". Proc. ACM SIGCOMM, 1998.

[17] P. Gupta and N. McKeown. "Packet Classification Using Hierarchical Intelligent Cuttings". Proc. Interconnects VII, 1999.

[18] P. Gupta and N. McKeown. "Algorithms for Packet Classification". IEEE Network Special Issue, 2001.

[19] V. Srinivasan and G. Varghese. "Fast address Lookup Using Controlled Prefix Expansion". Trans. ACM CS, 1999.

[20] Y. Gao. "Expansion-Based Tuple Space Search for Packet Classification". Course Project, University of Alberta, 2000.

[21] A. Balamash and M. Krunz. "Application of Multifractals in the Characterization of WWW Traffic". In proc. ICC, 2002.

[22] A. Feldmann and S. Muthukrishnan. "Tradeoffs for Packet Classification". In proc. IEEE INFOCOM, 2000.

[23] M. Mitzenmacher and A. Broder. " Using Multiple Hash Functions to Improve IP Lookups". In proc. IEEE INFOCOM, 2001.

[24] W. Shi, M. MacGregor, and P. Gburzynski. "Effects of a Hash-Based Scheduler on Cache Performance in a Parallel Forwarding System". In proc. CNDS, 2003.

[25] V. Srinivasan. "A packet Classification and Filter Management System". In Proc. INFOCOM, 2001.

[26] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. "Scalable High Speed IP Address Lookups". In proc. SIGCOMM, 1997.

[27] P. Gupta and N. McKeown, "Dynamic Algorithms with Worst-Case Performance for Packet Classification". In proc. IFIP Networking, 2000.

[28] T. Lakshman and D. Stiliadis. "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching". In Proc. ACM SIGCOMM, 1998.

[29] D. Comer. "Computer Networks and Internets with Internet Applications". Third Edition, 2001.

[30] A. Tanenbaum. "Computer Networks". Third Edition, 1996.

[31] R. Ziegler. "Linux Firewalls". Second Edition, 2001.

[32] P. Peebles, JR. "Probability, Random Variables and Random Signal Principles". Fourth Edition, 2001.

[33] J. Devore. "Probability and Statistics for Engineering and Sciences". Fourth Edition, 1995.

[34] R. Bunt and J. Murphy. "The Measurement of Locality and the Behavior of Programs". The Computer Journal, 1984.

[35] R. Jain and S. Routhier. "Packet Trains - Measurements and a New Model for Computer Network Traffic". IEEE JSAC, 1986.

[36] R. Jain. "Characteristics of Destination Address Locality in Computer Networks: A Comparison of Caching Schemes". Computer Networks and ISDN Systems, 1989.

[37] V Paxson and S. Floyd. "Wide-Area Traffic: The Failure of Poisson Modeling" In trans. IEEE ACM on Networking, 1995.

[38] T. Wolf and M. Franklin. "Locality-Aware Predictive Scheduling for Network Processors". In proc. IEEE ISPASS, 2001.

[39] K. Claffy, H. Braun, and G. Polyzos. "Traffic Characteristics of the T1 NSFNET Backbone". In proc. INFOCOM, 1993.

[40] V. Paxson. "Fast, Approximate Synthesis of Fractional Gaussian Noise for Generating Self-Similar Network Traffic". Computer Communications Review, 1997.

[41] P. Denning. "The Working Set Model for Program Behavior". In Proc. ACM SOSP, 1967.

[42] C. Wood, E. Fernandez, and T. Lang. "Minimization of Demand Paging for the LRU Stack Model of Program Behavior". Information Processing Letters, 1983.

[43] L. Belady. "A study of Replacement Algorithms for Virtual Storage Computers". IBM Systems Journal, 1966.

[44] M. Overmars and A. Stappen. "Range Searching and Point Location Among Fat Objects". Journal of Algorithms, 1996.

[45] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. Campbell. "Directions in Packet Classification for Network Processors". In Proc. 2nd Workshop on Network Processors, 2003.

[46] S. Hazelhurst. "A Proposal for Dynamic Access Lists for TCP/IP Packet Filtering" In proc. The South African Institute of Computer Scientists and Information Technologists, 2001.

[47] M. Garrett and W. Willinger. "Analysis, Modeling and Generation of Self-Similar VBR Video Traffic". In Proc. ACM SIGCOMM, 1994.

[48] K. Claffy, H. Braun, and G. Polyzos. "Long-Term Traffic Aspects of the NSFNET". In Proc. INET, 1993.

[49] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. "On the Self-similar Nature of Ethernet Traffic (extended version)" In trans. IEEE ACM on networking, 1994.

[50] M. Crovella and A. Bestavros. "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes". In trans IEEE ACM on networking, 1997.

[51] W. Cheswick and S. Bellovin. "Firewalls and Internet Security: Repelling the Wily Hacker". Addison Wesley Publishing, 1994.

[52] The CERT Coordination Center, Software Engineering Institute, Carnegie Mellon University, "UNIX Security Checklist v2.0", http://www.cert.org, 2001.

[53] K. Claffy. "Measuring the Internet". In proc. IEEE Internet Computing, 2000.

[54] H. Liu. "Routing Prefix Caching in Network Processor Design". In proc. International Conference on Computer Communications and Networks, 2001.

[55] C. Partridge. "Locality and Route Caches". In Proc. NSF Workshop on Internet Statistics Measurement and Analysis, 1999.

[56] B. Liljeqvist. "Visions and Facts: A Survey of Network Processors". Master Thesis, Department of Computer Engineering, Chalmers University of Technology, 2003.

[57] A. Antoniou. "Digital Filters, Analysis, Design, and Applications", Second Edition. 1993.